Querying and extracting heterogeneous graphs from structured data and unstrutured content

Rania Soussi

ECOLE CENTRALE PARIS

ECOLE NATIONALE DES SCIENCES DE L' INFORMATIQUE

PHD THESIS

to obtain the title of

Doctor of Computer Science

Defended by

Rania Soussi

Querying and Extracting Heterogeneous Graphs from Structured Data and Unstructured Content

prepared at Ecole Centrale Paris, MAS Laboratory, and Ecole Nationale des Sciences de l'Informatique, RIADI Laboratory
defended on 2012, June 22nd

Jury :

Advisors:	Henda Ben Ghezala: Professor at Ecole Nationale des Sciences de l'Informatique
	Marie-Aude Aufaure: Professor at Ecole Centrale Paris
President :	Bernd Amann: Professor at Université Pierre et Marie Curie (Paris 6)
Reviewers:	Alexander Löser: Professor at Technische Universität Berlin,Germany
	Faiez Gargouri: Professor at Institut Supérieur d'Informatique et de Multimedia de Sfax
Examinator:	Bénédicte Le Grand: HDR at Université Pierre et Marie Curie (Paris 6)

Acknowledgments

You'll meet more angels on a winding path than on a straight one.
Terri Guillemet

It is a pleasure to thank many people who made this thesis possible. I wish to express my deepest gratitude to my advisors at Ecole Centrale Paris and Ecole Nationale des Sciences de l'Informatique.

I would really like to thank my advisor Prof. **Marie-Aude Aufaure**, Professor at Ecole Centrale Paris, for her gentleness, guidance, inspiration and patience throughout these years. I am grateful for having been given the opportunity to work in the ARSA project. By working in such a project, I had the chance to gain experience in a variety of aspects related to research and team work.

I am also grateful to Prof. **Henda Ben Ghezala**, Professor at Ecole Nationale des Sciences de l'Informatique, for accepting me as a member of RIADI Laboratory. I would like to thank her for her gentleness, encouragement and confidence during my thesis and my master.

I am especially thankful to Prof. **Alexander Löser**, Professor at Technische Universität Berlin, and Prof. **Faiez Gargouri**, Professor at ISIMS, for accepting to review my PhD thesis. I would also like to thank Dr. **Bénédicte Le Grand**, HDR at Université Pierre et Marie Curie, who had accepted to be my jury examinator. I express my thorough gratitude to Prof. **Bernd Amann**, Professor at Université Pierre et Marie Curie, for the honor he made me by accepting to be president of the jury.

I would like to thank Dr. Yves Lechevallier for his precious comments and remarks about my work. I would like also to thank Dr. Hajer Baazaoui for her help and advice during the first year of my thesis.

My appreciation likewise to whole BI team and MAS Laboratory, my second family, particularly to Etienne Cuvelier for the research advises and for labeling my graph model as SPIDER-Graph, and also to Nesrine Ben Mustapha, Casio Mello, Carlos Mello, Hichem Bannour , Nizar Messai for advises and supporting me in the most difficult moments; to Micheline Elyes , Raphael thollot and Nicolas Kuchmann-Beauger for the productive discussions. I would like to thank the new BI members: Yves Vanrompay, Mario Cataldi, Alexandre Mikheev, Baptiste Thillay and Ilaria Tiddi for their help during the final preparations. I thank also Sylvie, Annie and Danny for all the administrative and technical help in ECP.

During this work I have collaborated with many colleagues for whom I have great regard, and I wish to extend my warmest thanks to all those who have helped me with my work in ECP or ENSI and I forgot to mention their names.

The permanent support of my family was extremely precious to me. Thank you, my aunts Salwa and Naziha , my grandmother Habiba, and my brother Mohamed

to whom I owe so much.
I wish to thank Hassen for the support that he has provided me and for believing in me that I can finish my thesis on time.

Lastly, and most importantly, I wish to thank my parents, Zakia and Mhamed. They bore me, raised me, supported me, taught me, and loved me. To them I dedicate this thesis.

Abstract

Nowadays, a huge volume of data is collected and stored daily in enterprises. To efficiently extract and manage this valuable knowledge helpful in the decision-making process is a hard task. Firstly, because the data can be stored in different ways: in a very structured way like in databases but also in unstructured repositories. Moreover the description of a sole object, person or process can be disseminated in several sources with several structures. Managing these different data models is difficult and makes the extraction of information process not very efficient.
Classical query techniques permit to retrieve the set of data which match with specific criteria under a specific model, but richer results could be provided if a unified representation of the disparate data of the enterprise and of their interactions and links could be defined. Graphs can be used for this unified data model, and facilitate the information search as well as the extraction of objects interaction.
The present work introduces a set of solutions to extract graphs from enterprise data and facilitate the process of information search on these graphs. First of all we have defined a new graph model called the SPIDER-Graph, which models complex objects and permits to define heterogeneous graphs. Furthermore, we have developed a set of algorithms to extract the content of a database from an enterprise and to represent it in this new model. This latter representation allows us to discover relations that exist in the data but are hidden due to their poor compatibility with the classical relational model. Moreover, in order to unify the representation of all the data of the enterprise, we have developed a second approach which extracts from unstructured data an enterprise's ontology containing the most important concepts and relations that can be found in a given enterprise. Having extracted the graphs from the relational databases and documents using the enterprise ontology, we propose an approach which allows the users to extract an interaction graph between a set of chosen enterprise objects (for example between customers and products or even the enterprise social network). This approach is based on a set of relations patterns extracted from the graph and the enterprise ontology concepts and relations. Finally, information retrieval is facilitated using a new visual graph query language called GraphVQL, which allows users to query graphs by drawing a pattern visually for the query. This language covers different query types from the simple selection and aggregation queries to social network analysis queries. All these approaches and methods have been developed and evaluated using real enterprise data.

Keywords: Graph Model, SPIDER-Graph relational database, entreprise ontology, visual query language

Résumé

La quantité des données stockées et collectées au sein des entreprises ne cessent d'augmenter, cependant, l'extraction et la gestion des connaissances gisant au sein de tels puits de données peut savérer difficile, et ce alors même que ces même connaissances sont très précieuses dans les processus de décision de lentreprise. La première de ces difficultés vient du fait que ces données peuvent être stockées sous diverses formes: sous forme structurées, et donc dans des bases de données relationnelles, ou sous forme non structurées, cest-à-dire dans des documents, des e-mails, . En outre, autre difficulté, la description d'une entité, objet, personne ou un processus peut être éparpillée dans plusieurs structures. Si les techniques classiques d'interrogation des données permettent de chercher l'ensemble de données qui correspondent à des critères précis et spécifiques dans un modèle de données spécifique, un résultat plus riche ne peut être obtenu quen modélisant les données dispersées de l'entreprise d'une façon unifiée. Un moyen naturel pour représenter et modéliser ces données disparates en structures est dutiliser les graphes. Ces derniers peuvent être utilisés comme un modèle unifié de données et faciliter la recherche d'information. . L'avantage de ce type de modèle réside dans ses aspects dynamiques et ses capacités à représenter les relations simples, ainsi que ses facilités d'interrogation de données appartenant à des sources hétérogènes, mais aussi ses capacités à découvrir des relations et des informations non explicites sur les différents objets de l'entreprise modélisés. La capacité des graphes à modéliser les interactions entre les objets hétérogènes (ex. clients et produits, produits et des projets, l'interaction entre des personnes tel que les réseaux sociaux, etc) est aussi un avantage non-négligeable. Lutilisation de ces représentations en graphes peut grandement aider les entreprises dans les processus de prise de décision, comme suggérer quelles recommandations envoyer à un client (en utilisant le graphe des produits et des clients) ou trouver lexpert le plus adéquat sur un sujet précis (en utilisant le réseau social). Ce travail introduit un ensemble de solutions pour extraire des graphes à partir des données de l'entreprise et pour aussi faciliter le processus de recherche d'information dans ces graphes. Premièrement, nous avons défini un nouveau modèle de données appelé SPIDER-Graph permettant de modéliser des objets complexes et de définir des graphs hétérogènes. Puis, nous avons développé un ensemble d'algorithmes pour extraire le contenu des bases de données de l'entreprise et les transformer suivant ce nouveau modèle de graphe. Cette représentation permet de mettre à jour des relations non explicites entre objets, relations existantes mais non visibles dans le modèle relationnel. Par ailleurs, pour unifier la représentation de toutes les données dans l'entreprise, nous avons développé, dans une deuxième approche, une méthode de constitution d une ontologie d'entreprise contenant les concepts et les relations les plus importantes d'une entreprise, et ceci, à partir de lextraction des données non structurés de cette même entreprise.
Ensuite, après le processus d'extraction des différents graphes de données l'entreprise, nous avons proposé une approche qui permettent d'extraire des graphes d'interactions entre des objets hétérogènes modélisant l'entreprise. Cette approche

permet d'extraire des graphes de réseaux sociaux ou des graphes d'interactions en se basant sur le processus suivant : premièrement, l'utilisateur choisit les objets dont il veut voir les interactions à partir des concepts de l'ontologie, ce qui permet à un processus d'identification d'objets didentifier les nuds correspondant à ces concepts dans le graphe extrait à partir de la base de données relationnelle. Ensuite, en se basant sur les relations de l'ontologie et un ensemble de patrons de relations construit à partir de la base relationnelle, un processus d'extraction de relations crée les relations entre ces objets.

Extraire, à partir des données (structurées ou pas) dune entreprise, de la connaissance sous forme de graphes est sans grand intérêt si lon ne peut interroger et interagir cette connaissance. Pour faciliter la recherche d'information, nous avons proposé un nouveau langage d'interrogation visuel appelé GraphVQL (Graph Visual Query Langauge) qui permet aux utilisateurs non experts de poser leurs requêtes visuellement sous forme de patron de graphe. Ce langage propose plusieurs types de requêtes de la simple sélection et agrégation jusqu'à l'analyse des réseaux sociaux. Il permet aussi d'interroger différent type de graphes SPIDER-Graph, RDF ou GraphML en se basant sur des algorithmes de pattern matching ou de translation des requêtes sous forme de SPARQL. Lévaluation de toutes ces approches et méthodes a été réalisée en utilisant des jeux de données réelles d'entreprises.

Mots clés: base de données, graphe, SPIDER-Graph, Langage d'interrogation visuel, ontologie d'entreprise.

List of Personal Publications

Chapters in Books

1. Rania Soussi, Etienne Cuvelier, Marie-Aude Aufaure, Amine Louati, and Yves Lechevallier : DB2SNA: an All-in-one Tool for Extraction and Aggregation of underlying Social Networks from Relational Databases, book chapter in "Social Network Analysis and Mining" ,eds.Tansel Ozyer, Springer LNSN, 2011(to appear),

2. Rania Soussi, Marie-Aude Aufaure and Hajer Baazaoui (2011) Graph Database For collaborative Communities, In: Community-Built Databases:Research and Development, Pardede, Eric (Ed.) 1st Edition., 2011, 400 p., Hardcover ISBN: 978-3-642-19046-9, Springer,Due: May 2011

International conferences and workshops

1. Rania Soussi (2012) SPIDER-Graph: A model for heterogeneous graphs extracted from a relational database"In 31st International Conference on Conceptual Modeling (ER 2012) -PhD Symposium.

2. Rania Soussi, Marie-Aude Aufaure and Hajer Baazaoui (2010) Towards a Social Network Extraction using Graph Databases, The Second International Conference on Advances in Databases, Knowledge, and Data Applications (DBKDA 2010), April 11-16, 2010 - Menuires, France.

3. Rania Soussi, Hajer Baazaoui Zghal, Marie-Aude Aufaure (2009) Toward a social network extraction approach from relational database, Workshop ARS'09 Social Network Analysis: Models and Met and Methods for Relational Data, 13-14 July 2009, Fisciano (SA), Italy.

4. Rania Soussi, Hajer Baazaoui, Marie-Aude Aufaure, Henda Ben Ghezala. Social Networks Extraction: State of the Art,ICWIT'09,juillet12-14. KerKEnah .Tunisie

National conferences

1. Amine Louati, Rania Soussi, Marie-Aude Aufaure, Hejer Baazaoui and Yves Lechevallier: Recherche de classes dans des réseaux sociaux,SFC 2011, Orléans,Février 2011.

2. Rania Soussi, Amine Louati, Marie-Aude Aufaure, Hajer Baazaoui Zghal, Yves Lechevallier, Henda Ben Ghézala: Extraction et analyse de réseaux sociaux issus de bases de données relationnelles. EGC 2011: 371-376

Poster

1. Rania Soussi: Extraction of interaction graphs from a relational database using the SPIDER-Graph model, ebiss 2011, http://cs.ulb.ac.be/conferences/ebiss2011/files/soussi.pdf

Technical Reports (in the ARSA project)

1. Rania Soussi.Survey on Graph query languages. Ecole centrale Paris. 06-2010
2. Rania Soussi. Visual Graph Query Language(global analysis algorithms).Ecole centrale Paris. 07- 2011

Contents

List of Figures

Chapter 1

Introduction

Contents

1.1 Context and Motivation

Decision making is a main task for managing a business at every level, from operational to strategic one. It requires to choose a path among all the alternatives taking into account all the influencing parameters in order to produce the desired result. To take the good decision, it is important to have the right information at the right time. Information technologies have contributed on the field of transforming data into information and delivering it through technologies and techniques. The methods, techniques and their application used to provide this kind of support are included under the term of Business Intelligence. Business intelligence can be defined as "the process of turning data into information and then into knowledge" [Golfarelli 2004].

Historically, Business Intelligence (BI) term is used broadly from early 90s' after it was coined from Howard Dresner. He proposed this term "as an umbrella term to describe concepts and methods to improve business decision making by using fact-based support systems" [Power 2007]. In the 60s', the investigation on the business intelligence systems has started by computerizing quantitative models to assist in decision making planning" [Buchanan 2006]. Thus a BI system can be called a decision support system (DSS).

Until late 80s', Data Warehouse [Inmon 1992] notion was introduced. Data Warehouses are large repositories of historical data, organized according to the multidimensional model, that are directly accessed by users (i.e. the managers) through interfaces that allow them to carry out very detailed analyses. All the resources are integrated on a Data Warehouse using ETL (Extraction, transformation and loading) processes. In the middle of the 90's, the CUBE operator has been introduced. Then, it was possible to conduct multidimensional analysis and use OLAP (Online Analytical Processing) techniques that seem now essential for every BI Suite. On that stage, data are collected from different sources which can be relational

databases and flat files. Those can be external to the company or internal (feedback from business processes).
However, the new data streams created under the notion of Web 2.0 [O'Reilly 2005] (information sharing, interoperability, user-centered design, and collaboration on the Web) have led enterprises to deal with big data from structured databases and unstructured content (emails, documents, social networks, etc). Moreover, these data are often distributed and highly dynamic. Social Media and mobile technologies have changed the way to access information, facilitating communication and data exchange. All these evolutions afford BI to move to Business Intelligence 2.0 which integrates BI elements with elements from both Web 2.0 and the Semantic Web (semantic integration through shared ontologies to enable easier exchange of data). These evolutions of the data sources and the increase of the competitions have conducted the business users to expect immediate feedback and want to find rather than searching for.
In this context, having the interactions and links between several objects (e.g. products and sites, customers and products, social network...) is a precious mean to permit a good understanding of a lot of situations in the enterprise context. For example, by analyzing the interaction between the products and the customers, the project manager can discover new high-potential products, markets, needs, clients and features in order to adjust new product development line.
For instance, the enterprise social networks (the interactions between the enterprise persons) can be used to improve the productivity and the collaborations inside the enterprise. By analyzing these interactions using the social network measures like the centrality measure [Freeman 1977], the importance and influence of actors, groups key actors can be identified. These types of analysis are useful for the internal enterprise management. Identifying the most central groups helps finding groups with a good communication, which are important elements of cohesion. The similarity between groups can be computed in order to analyze groups with positive properties, apply observation on a group to a similar one or explain a behavior within a structure. We can also search persons who are not well integrated, and find the best group for these persons. Another application is to be able to constitute efficient teams, in order to enable innovation and to reach group cohesion.
The increasing quantities and the evolution of data in enterprise can also be challenging to manage. These different data types are complementary. The description of a sole object, person or process can be disseminated in several sources with several structures. For example, the description of a project (start-date, budget, people and etc.) can be stored in a database and the analysis of its impact can be detailed in a document. Efficiently extract and manage all this valuable knowledge located in different sources and helpful in the decision-making process is a very hard task. Firstly, classical queries techniques are not appropriate to search for information in different data sources. Information retrieval [Kowalski 2010] or enterprise search [Balog 2007] approaches can help finding information on unstructured data. Other techniques like semantic data search [Tran 2011] and keyword search are useful to extract information from semi-structured or structured data [Agrawal 2002]. Be-

cause of this, the user loses important information searching for one type of data at the same time. Secondly, in order to find the right information, the user should know its location (in a database or a document) and how it is named (for example a product can be referred too as merchandise). Finally, the user should have the technical knowledge to use many of these approaches. In fact, to query relational databases, the user should know the SQL query language. Thus, a unified representation of the disparate data of the enterprise and of their interactions and links could provide richer results than classical queries techniques.
To fulfill the previous objectives, we need methods to model with unified manner the heterogeneous enterprise data and enterprise objects interactions, to analyze, to query them and to visualize results in an efficient and intuitive way. Graphs can be used as a unified data model for heterogeneous data source. Graphs are a natural way of representing and modeling these interactions and to facilitate their querying. The main advantage of such structure relies on its dynamic aspect and its capability to represent relations, even multiple ones, between objects. It also facilitates data query using graph operations and algorithms. On the other hand, extracting and modeling such heterogeneous graphs, with heterogeneous objects and relations, are outside of the classical graph models capabilities, moreover when each node contains a set of values. Whereas, which graph model can be adapted to model heterogeneous data? How can this graph model improve the information analysis in the enterprise?

1.2 Proposed Approach and Contributions

The present thesis introduces a set of solutions to extract object interaction graphs from enterprise data and facilitate their querying. We can summarize the main objectives and propositions in what follows:
Model enterprise data with graph model:
In a business context, important expertise information is stored in files, databases and especially relational databases. Relational database pervades almost all businesses. Many kinds of data, from e-mails and contact information to financial data and sales records, are stored in databases. Thus, it is important to integrate relational database in our process to model enterprise data with graph model and extract from it object interaction. In this context, we have defined a new graph model called SPIDER-Graph (Structure Providing Information for Data with Edge or Relations), which models complex objects and permits to define heterogeneous graphs.
Then, we have developed a set of algorithms to transform the relational database model to a SPIDER-Graph model. The latter representation allows us to discover explicitly the existing relations between the objects in the database.
In order to model the unstructured data as graph model, we have developed a second approach which extracts from unstructured data enterprise ontology containing the most important concepts and relations that can be found in the enterprise.
Extract enterprise object interaction graph from heterogeneous data

sources:
Having the SPIDER-Graph extracted from the relational databases and the enterprise ontology from the unstructured data, we propose an approach which allows the users to extract an interaction graph of a set of chosen enterprise objects (for example between customers and products or even the enterprise social network). This approach is based on two main steps: (1) an object identification process of the chosen objects in the SPIDER-Graph using the enterprise ontology and (2) relations extraction process from both the SPIDER-Graph and the ontology.
Propose an adapted querying approach:
the analysis and the querying of the extracted graphs and the other existing enterprise graph are facilitated using a new visual graph query language called GraphVQL, which allows users to query graphs visually by drawing a query. This language covers different query types from the simple selection and aggregation queries to social network analysis queries.

Then, the proposed solution can be summarized in the architecture depicted in Figure 1.1.

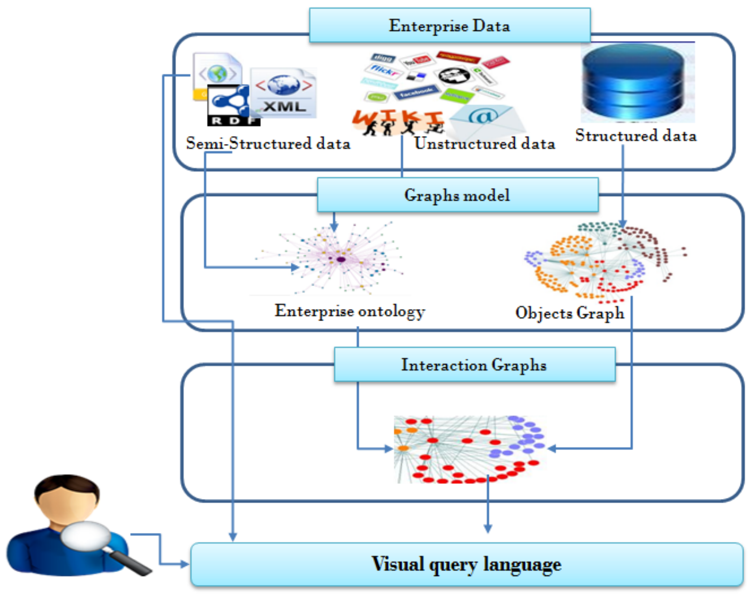


Figure 1.1: Architecture of the Proposed Solution.

1.3 Thesis Organization

The thesis is organized in seven chapters: an introduction chapter, a conclusion and perspectives chapter and five chapters for the related works and the main work.
Chapter 2: Graphs and Relational databases: describes the state of the art of graph models and graph manipulation techniques. It details also the different techniques used to transform relational database into a graph like data.
Chapter 3: The enterprise Ontology Learning approach: presents some aspects of

the ontology learning state of the art. Then, it presents our enterprise ontology learning approach from semi-structured and unstructured data.
Chapter 4: Object Interaction Graph extraction Approach. This chapter details the extraction of interaction graphs. It starts by presenting our graph model SPIDER-Graph. Then, it describes the transformation of the relational database to a SPIDER-Graph model. Finally, the interaction graph extraction using the enterprise ontology and the relation patterns is detailed.
Chapter 5: GraphVQL: Visual Graph Query Language. This chapter presents the graph visual query language (GraphVQL) for heterogeneous graphs that supports different type of queries. GraphVQL is a query language for SPIDER-Graph model and also for heterogeneous graphs modeled with RDF or Graphml.
Chapter 6: Implementation and evaluation. In this chapter, we present the evaluation of the different approaches presented in the previous chapters.

Chapter 2
State-of-the Art

Contents

In this chapter the state of the art techniques related to our proposal are presented. In the first section 2.1, we survey the different graph models: the basic graphs definition, the graph database models and the graph-like structure (XML and RDF). The different graph data models are manipulated by specific techniques like the graph matching, graph query languages and graph transformation which are presented in section 2.2.
We then review the different approaches allowing transforming structured data (relational database) to graph models and querying relational databases with graph techniques (section 2.3).

2.1 Graph Models

2.1.1 Basic Definitions

Graphs are used in many areas and model with natural way entities interconnectivity or topology like social network, geographic network, hypertext, etc. Graphs have the advantage of being able to keep all the information about an entity in a single node and show related information by arcs connected to it [Paredaens 1995]. In

this section, we present the different graph definitions from basic graphs to more complex graph models.
A graph is defined as follows:

Definition 1 (Basic Graph) *A graph $G = (V, E)$ consists of a set V of vertices (also called nodes), a set E of edges where $E \subseteq V \times V$.*

This definition can be extended as in [Ehrig 2006a].

Definition 2 (Graph) *A graph $G = (V, E, s, t)$ consists of a set V of vertices (also called nodes), a set E of edges, and two functions s,t:E $\longrightarrow V$, the source and target functions.*

Adding directions to the graph edges will transform a graph to a directed graph (see figure 2.1(a)).

Definition 3 (Directed and Undirected Graph) *Edges are said to be undirected when they have no direction, and a graph G containing only such types of graphs is called undirected. When all edges have directions and therefore (u, v) and (v, u) can be distinguished, the graph is said to be directed. Usually, the term arc is used when the graph is directed, and the term edge is used when it is undirected. [Chartrand 1985]*

Graph vertices and edges can also contain information. When this information is a simple label the graph is called a labeled graph(see figure 2.1(b)).

Definition 4 (Labeled Graph) *Given two alphabets Σ_V and Σ_E a labeled graph is a triple $LG = (V, E, l_V)$ where V is a finite set of nodes, $E \subseteq V \times \Sigma_E \times V$ is a ternary relation describing the edges (including the labeling of the edges) and $l_V : V \mapsto \Sigma_V$ is a function describing the labeling of the nodes.*

Other times, vertices and edges can contain attributes. In this case, the graph is called an attributed graph(see figure 2.1(c)).

Definition 5 (Attributed Graphs) *An attributed graph (over $\Sigma = (S, OP)$ with S is the set of and a set of operation symbols OP) is a pair $AG = \langle G, A\rangle$ of a graph G and a $\Sigma -$ algebra A such that $|A| \subseteq G_V$, where $|A|$ is the disjoint union of the carrier sets A_s of$A$, for all $s \in S$, and such that $\forall e \in G_E : src(e) \notin |A|$. Let $Attr(AG) = \{e \in G_E\, |tar(e) \in |A|\}$, $Graph(AG) = G\ (|A| + Attr(AG))$ and Alg(AG) =A.*

In the figure 2.1, we present some examples of the presented graphs.

Graphs are related by graph morphisms, which map the nodes and edges of a graph to those of another one, preserving the source and target of each edge.

Definition 6 (Graph Morphism) *Given graphs G1,G2 whith $Gi = (Vi, Ei, si, ti)$ for i=1,2 a graph morphism f:$G1 \longrightarrow G2$,$f = (f_v, f_E)$ consists of two functions $fv : V1 \longrightarrow V2$ and $f_E : E1 \longrightarrow E2$ that preserve the source and the target functions, i.e $f_v os1$=$s2of_E$ and $f_v ot1$=$t2of_E$*

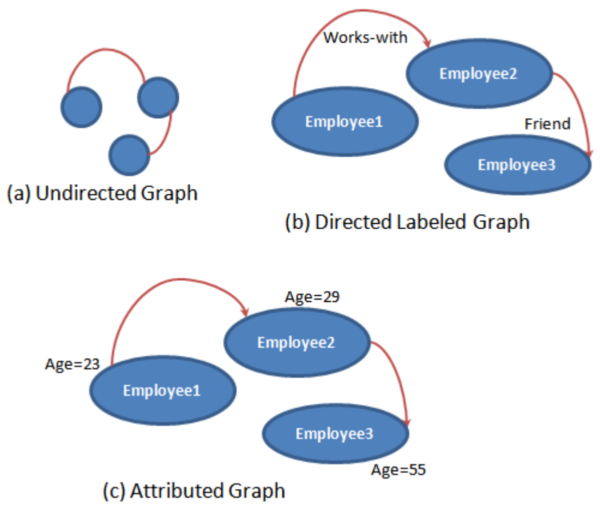


Figure 2.1: Examples of the Presented Graphs.

Another important graph type is the typed graph. A typed graph defines a set of types, which can be used to assign a type to the nodes and edges of the graph [Ehrig 2006a].

Definition 7 (Typed Graph) *A type graph is a distinguished graph* $TG = (V_{TG}, E_{TG}, s_{TG}, t_{TG})$. V_{TG} *and* E_{TG} *are called the vertex and the edge type alphabets,respectivly. A tuple (G,type) of a graph G together with a graph morphism type:G* $\longrightarrow$ *TG is then called a typed graph.*

A graph can be characterized by one or many graph type. Indeed, a graph can be in the same time typed attributed labeled directed graph.
Simple graphs are not sufficient to model real life applications. Moreover, they cannot represent heterogeneous graphs based on complex objects. The basic structure of a graph (nodes and edges) is complemented with the use of hypernodes and hypergraphs extensions that provide support for nested structures.

Definition 8 (Hypernode) *A hypernode [Levene 1995] is an equation of the form* $H = (N, E)$ *where* $H \in L$ *is termed the defining label (or simply the label) of the hypernode and (N, E) is a digraph such that* $N \subset (P \cup L)$ *(P is the set of primitive node). (N,E) is termed the digraph of the hypernode (or simply the digraph corresponding to H).*

A hypergraph is a generalized notion of graph where the notion of edge is extended to hyperedge, which relates to an arbitrary set of nodes.

Definition 9 (Hypergraph) *An hypergraph [Berge 1985] G is a tuple* (V, E, μ), *where V is a finite set of nodes, E is a finite set of edges,* $\mu : E \longrightarrow V^*$ *is a connection function where* V^* *means multiple nodes.*

2.1.2 Graph Database Model

In order to model more complex structures and connections using the different proposed graph aspects, a graph database model has been proposed. A graph database is defined [Angles 2008] as a *"database where the data structures for the schema and/or instances are modeled as a (labeled) (directed) graph, or generalizations of the graph data structure, where data manipulation is expressed by graph-oriented operations and type constructors, and has integrity constraints appropriate for the graph structure"*. More formally, a graph database is defined as follow:

Definition 10 (Graph Database) *Graph database schema is in the form of a graph $Gdb = (V, E, L, \psi, \lambda)$ where: V is a set of nodes and E is a set of edges; L is a set of labels;ψ is a labeling function from $V \cup E$ into L and $\lambda$is an incidence function from E into $V \times V$.*

There is a variety of models for a graph database (for more details see [Angles 2008]). All these models have their formal foundation as variations of the basic mathematical definition of a graph. The structure used for modeling entities and relations influences the way that data is queried and visualized. In this section, some models are presented and classified according to the data structure used to model entities and relations.

2.1.2.1 Models based on Simple Node

Data are represented in the models by a (directed or undirected) graph with simple nodes and edges. Most of the models (GOOD [Gyssens 1990a], GMOD [Andries 1992], etc.) represent both schema and instance database as a labeled directed graph. Moreover, LDM [Kuper 1993] represents the graph schema as a directed graph where leaves represent data and whose internal nodes represent connections between the data. LDM instances consist of two-column tables, one for each node of the schema.
Entities in these models are represented by nodes labeled with type name and also with a type value or an object identifier (in the case of instance graph). Some models have nodes for explicit representation of tuples and sets (PaMaL [Gemis 1993], GDM [Hidders 2002]), and n-ary relations (GDM).
Relations (attributes, relations between entities) are generally represented in these models by means of labeled edges. LDM and PaMaL use tuple nodes to describe a set of attributes which are used to define an entity. GOOD defines edges to distinguish between mono-valued (functional edge) and multi-valued attributes (nonfunctional edge). Nevertheless, these models do not allow the presentation of nested relations and are not very well suited for complex object modeling.

2.1.2.2 Models based on Complex Node

In models based on complex node, the basic structure of a graph (node and edge) and the presentation of entities and relations are based on hypernodes or hypergraphs. Hypernodes can be used to represent simple (flat) and complex objects (hierarchical, composite, and cyclic) as well as mappings and records. The Hypernode Model [Levene 1995] and GGL [Graves 1995] emphasize the use of hypernodes for representing nested complex objects. GROOVY [Levene 1991] is focused on the use of hypergraphs. The hypernode model uses nested graphs at the schema and instance levels. GGL introduces, in addition to its support for hypernodes (called Master-nodes), the notion of Master-edge for the encapsulation of paths. It uses hypernodes as an abstraction mechanism consisting in packaging other graphs as an encapsulated vertex. Whereas, the Hypernode model uses hypernodes to represent other abstractions like complex objects and relations. Most models have explicit labels on edges. In the hypernode model and GROOVY, labeling can be obtained by encapsulating edges that represent the same relation, within one hypernode (or hyperedge) labeled with the relation name.

2.1.3 Graph-Like Structure

Apart from the reviewed models, there are other proposals that present graph-like features, although not explicitly designed to model the structure and connectivity of the information.

2.1.3.1 The Resource Description Framework (RDF)

The Resource Description Framework (RDF) [Klyne 2004] represents the data model of the Semantic Web. RDF is a recommendation of the W3C, originally designed to represent metadata. RDF allows structured and semi-structured data to be mixed, exposed, and shared across different applications.
Formally, RDF is a set of triples. RDF triple is a triple $(s, p, o) \in (I \cup B) \times I \times (I \cup B \cup L)$ where I, B, and L are sets that represent IRIs [Duerst 2005], Blank nodes, and Literals, respectively. In this triple, s is the subject, p the predicate, and o the object. The subject s can be any resource identifiable by an URI, while the predicate p defines a property of a resource and is identified with an URI itself. The object o, on the other hand, represents the value of that property and can be either a Literal or a Resource identified again by an URI.
The RDF triplet can be seen as a graph(see Figure 2.2). The predicate connects the subject with the object and therefore a triple can as well be represented as a node-arc-node link. Thus, RDF models information with graph-like structure, where basic notions of graph theory like node, edge, path, neighborhood, connectivity, distance, degree, and so on play a central role.

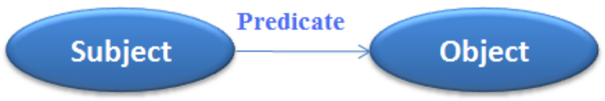
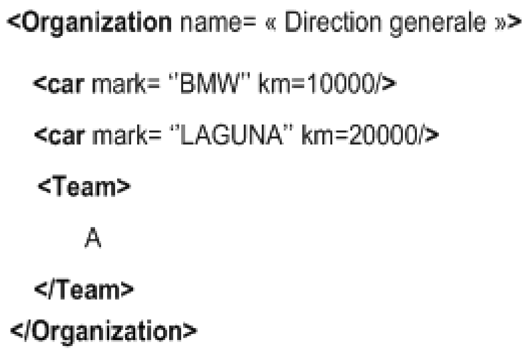


Figure 2.2: An RDF Triple as a Directed Graph.

2.1.3.2 The Extensible Markup Language (XML)

The Extensible Markup Language (XML) [Bray 1997] is a set of rules for encoding documents in machine-readable form. The goal is to enable documents to be served, received, and processed in the Web. XML data are labeled ordered trees (with labels on nodes), where internal nodes define the structure, and leaves node define the data (schema and data are mixed). Figure 2.3 shows an XML document example. The schema of XML document can be described by a schema language like DTD or XSD.
Compared to graph data, XML has an ordered tree-like structure, which is a restricted type of graph. Nevertheless, XML additionally provides a referencing mechanism among elements that allows simulating arbitrary graphs. In this sense XML can simulate semi-structured data.
In XML, the information about the hierarchical structure of the data is part of the data (in other words XML is self-describing). In contrast, in graph models this information is described by the schema graph in a more flexible manner using relations between entities. From this point of view, graph models use connections to explicitly represent generalization, compositions, hierarchy, classification, and any other type of relations. In the literature, many propositions have extended XML in order to improve the graphs description.

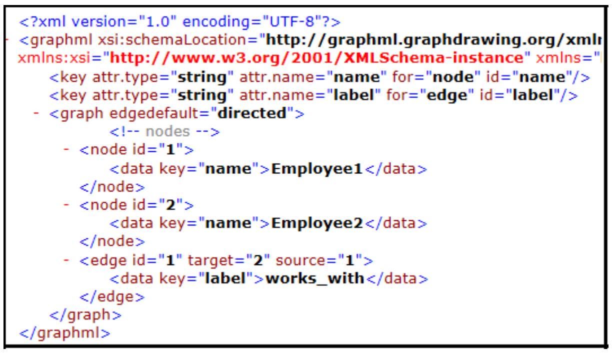


```
<Organization name= « Direction generale »>
  <car mark= "BMW" km=10000/>
  <car mark= "LAGUNA" km=20000/>
  <Team>
      A
  </Team>
</Organization>
```

Figure 2.3: XML Document.

A. GraphML (Graph Markup Language)
GraphML (Graph Markup Language) [Brandes 2001] is an XML-based format for the description of graph structures, designed to improve tool interoperability and reduce communication overhead. Thanks to its XML syntax, GraphML-aware applications can take advantage of a growing number of XML-related technologies and tools, such as parsers and validators.

GraphML is a comprehensive and easy-to-use file format for graphs. It consists of a core language used to describe the structural properties of a graph and a flexible extension mechanism to add application specific data. The basic graph model of GraphML is a labeled mixed multigraph with optional node-ports, hyperedges, and nesting. Graph drawing information is intended to be separated into topological and geometric information, with a graphics layer on top. Like any other associated data, it will be encapsulated in a special tag. The main features of GraphMl include support of: directed, undirected, and mixed graphs, hypergraphs, hierarchical graphs and etc. An example of a GraphML file is presented in figure 2.4



```
<?xml version="1.0" encoding="UTF-8"?>
- <graphml xsi:schemaLocation="http://graphml.graphdrawing.org/xmlı
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xmlns="
    <key attr.type="string" attr.name="name" for="node" id="name"/>
    <key attr.type="string" attr.name="label" for="edge" id="label"/>
  - <graph edgedefault="directed">
            <!-- nodes -->
      - <node id="1">
            <data key="name">Employee1</data>
        </node>
      - <node id="2">
            <data key="name">Employee2</data>
        </node>
      - <edge id="1" target="2" source="1">
            <data key="label">works_with</data>
        </edge>
    </graph>
</graphml>
```

Figure 2.4: GraphML Document.

In this GraphML, a graph is presented by a <graph> element. The nodes of a graph are represented by a list of node elements. Each node must have an id attribute. The edge set is represented by a list of edge elements.

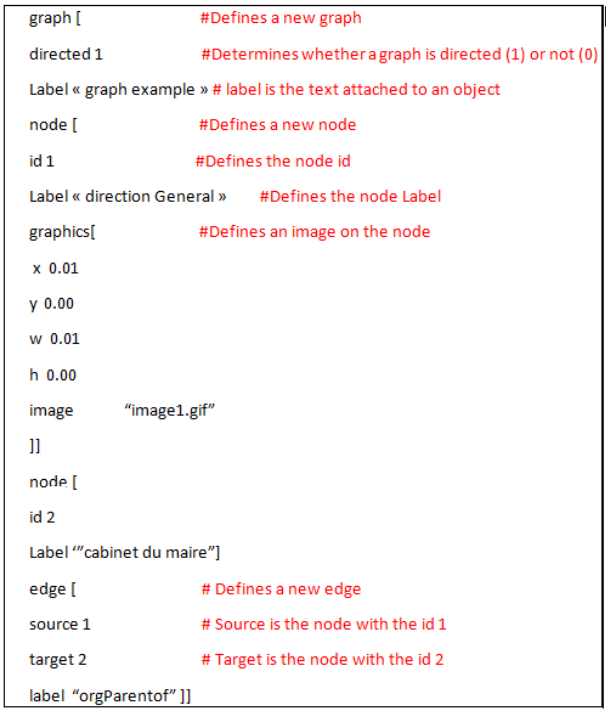
B. Graph Modelling Language (GML)

Graph Modelling Language (GML) [Himsolt 1996] is a hierarchical ASCII-based file format for describing graphs. It has also been named Graph Meta Language. GML is a text-based file format and it is the ancestor of GraphML. GML supports the entire range of possible graph structure constellations. This means that beyond "flat" graphs, hierarchically organized graphs can be saved, preserving all relevant hierarchical information. This information includes, e.g., any inter-edges from the hierarchy, the structure of any inner graphs, and whether a node containing an inner graph is a group node or a folder node. GML has some key elements:

- A GML file is made up of key-value pairs. Examples for keys are graph, node and edge.
- The key idea behind GML is that there are some standard keys like graph, node and edge, and anybody is free to add its own keys to add specific information.
- Values can be integers, floating point numbers, strings and lists, the latter one having to be enclosed in square brackets.

Figure 2.5 shows a GML file containing a graph composed by two labeled nodes.

C. XGML



```
graph [                     #Defines a new graph
directed 1                  #Determines whether a graph is directed (1) or not (0)
Label « graph example » # label is the text attached to an object
node [                      #Defines a new node
id 1                        #Defines the node id
Label « direction General »       #Defines the node Label
graphics[                   #Defines an image on the node
 x 0.01
y 0.00
w 0.01
h 0.00
image       "image1.gif"
]]
node [
id 2
Label '"cabinet du maire"]
edge [                      # Defines a new edge
source 1                    # Source is the node with the id 1
target 2                    # Target is the node with the id 2
label "orgParentof" ]]
```

Figure 2.5: GML Document.

XGML[1] is an XML-like variant of the GML file format where sections and attributes as listed in the various tables of section GML File Format are wrapped in $< section >$ and $< attribute >$ tags, respectively.

D. GXL

GXL (Graph eXchange Language) [Holt 2000] is designed to be a standard exchange format for graphs. GXL is an XML sublanguage and the syntax is given by a XML DTD (Document Type Definition). This exchange format offers an adaptable and flexible means to support interoperability between graph-based tools.
E.XGMML
XGMML (the eXtensible Graph Markup and Modeling Language) [Punin 2001] is an XML application based on GML which is used for graph description. Technically, while GML is not related to XML nor SGML, XGMML is an XML application that is designed in a way where there is a 1 : 1 relation towards GML for trivial conversion between the two formats.

2.2 Graph Manipulation

sec:GMnp
In order to extract, search or analyze information or transform data into graph

[1]http://www.yworks.com/products/yfiles/doc/developers-guide/xgml.html

structure, methods for manipulating, querying and transforming graphs are needed. In this section, we will describe some important techniques to manipulate graphs: Graph matching, Graph querying and Graph transformation.

2.2.1 Graph Matching

In many applications comparing two objects or an object and a model is an important task. The process of evaluating such similarity in the context of graph models is commonly referred to as graph matching. Graph matching has been used in many research fields: Bioinformatics [Borgwardt 2005], image analysis [Harchaoui 2007], document processing, web content mining [Schenker 2005], data mining [Cook 2006]. In this thesis, the interest is primarily on the use of Graph matching in the field of graph querying (which will be presented in the next section).

Graph matching is the process of finding a correspondence between the nodes and the edges of two graphs that satisfies some constraints, ensuring that similar substructures in one graph are mapped to similar substructures in another graph [Conte 2004]. Based on this matching, a dissimilarity or similarity score can eventually be computed indicating the proximity of two graphs.

The graph matching methods are divided into two categories: The first contains exact matching methods that require a strict correspondence among the two objects being matched or at least among their subparts. The second category defines inexact matching methods, where a matching can occur even if the two graphs being compared are structurally different to some extent. In the following, two types of matching are described.

2.2.1.1 Exact Graph Matching

The exact matching [Riesen 2010b] aims to determine whether two graphs, or at least part of them, are identical in terms of structure and labels. More formally, given two graphs $g_1 = (V_{g_1}, E_{g_1})$ and $g_2 = (V_{g_2}, E_{g_2})$, with $|V_{g_1}| = |V_{g_2}|$, the exact matching is to find a one-to-one mapping $f : V_{g_2} \rightarrow V_{g_1}$ such that $(u, v) \in E_{g_2}$ if $(f(u), f(v)) \in E_{g_1}$. When such a mapping f exists, this is called an isomorphism, and g_2 is said to be isomorphic to g_1 [Bengoetxea 2002]. This isomorphism can have different strengths:

- **Graph isomorphism:** The most stringent form of exact matching. A one-to-one correspondence must be found between each node of the first graph and each node of the second graph such that the edge structure is preserved and node and edge labels are consistent.

- **Subgraph isomorphism** is a weaker form of matching. It requires that an isomorphism stays between one of the two graphs and a node-induced subgraph between the other. Intuitively, subgraph isomorphism is the problem to detect if a smaller graph is identically present in a larger graph. Some works

[Fomin 2005] [Wong 1990]use the term subgraph isomorphism in a slightly weaker sense. They use what they call a monomorphism. The monomorphism requires that each node of the first graph is mapped to a distinct node of the second one, and each edge of the first graph has a corresponding edge in the second one. The second graph, however, may have both extra nodes and extra edges.

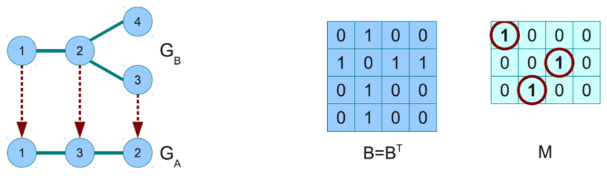
- **Homomorphism:** Each edge/node in the first graph is mapped to a edge/node of the second graph. The correspondence can be many-to-one.

In order to perform the different exact graph matching types, various algorithms have been proposed. Standard algorithms of exact pattern matching are based on tree search techniques with backtracking. The basic idea is that a partial match (initially empty) is iteratively expanded by adding new node-to-node correspondences. This process is repeated until a constraint imposed by the matching type is violated (edge structure constraint violated or node or edge label inconsistent). Then, the backtracking is performed. The algorithm undoes the last node added until a partial matching is found for which an alternative extension is possible. If all the possible mappings that satisfy the constraints have already been tried, the algorithm terminates. Several algorithms use this technique, the most popular and the more used algorithm is the Ullmann algorithm [Ullmann 1976]. In [Larrosa 2002], the tree search is used with another heuristic which improves the algorithm speed. Other new algorithm of pattern matching based on tree search is proposed in [He 2008]. However, there are other algorithms of exact pattern matching which are not based on tree search. For example, Nauty [McKay 1981], which is based on group theory, or an algorithm based on decision tree [Irniger 2007] [Lazarescu 2000].
Figure 2.6 presents an example of Ullmann exact matching algorithm [Ullmann 1976]. We take as input the two graphs $GA(VA, EA)$ and $GB(VB, EB)$and their adjacency matrices A and B. The first step of the Ulmamann approach is to define a matrix M to be $|V_A|$ X $|V_B|$ those elements are 1 and 0 such that each row contains more than one 1. Secondly, B is permuted by M to compare adjacency. Indeed, a third matrix $C = M(MB)^T$ is computed, where T denotes transposition. If $\forall i, j \in |V_A|$ $(a_{i,j} == 1) \Rightarrow (c_{i,j} == 1)$ then M specifies an isomorphisme between A and a subgraph of B. In this case, if $m_{i,j} = 1$, then the j^{th} point in B corresponds to the i^{th} point in A in this isomorphism.

2.2.1.2 Inexact Graph Matching

The constraints imposed by the exact graph matching are too rigid in some applications. Two graphs must be completely identical or at least in a large part. Therefore, the matching process must be more tolerant.

A large number of inexact graph matching methods have been proposed, dealing with a more general graph matching problem than the one formulated in terms of (sub)graph isomorphism. These inexact matching algorithms measure the discrepancy of two graphs in a broader sense, that better reflects the intuitive understanding



$$M(MB)^T = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 \end{bmatrix} \times \left(\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 \end{bmatrix} \times \begin{bmatrix} 0 & 1 & 0 & 0 \\ 1 & 0 & 1 & 1 \\ 0 & 1 & 0 & 0 \\ 0 & 1 & 0 & 0 \end{bmatrix} \right)^T$$

$$= \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 \end{bmatrix} \times \begin{bmatrix} 0 & 0 & 1 \\ 1 & 1 & 0 \\ 0 & 0 & 1 \\ 0 & 0 & 1 \end{bmatrix}$$

Figure 2.6: Ullmann Exact Matching Algorithm.

of graph similarity or dissimilarity.

In an inexact graph matching problem, since $|V_{g1}| < |V_{g2}|$, the goal is to find a mapping $f' : V_{g2} \to V_{g1}$ such that $(u, v) \in E_{g2}$ iff $(f'(u), f'(v)) \in E_{g1}$. This corresponds to the search for a small graph within a big one. An important sub-type of these problems are sub-graph matching problems, in which we have two graphs $G = (V, E)$ and $G' = (V', E')$, where $V' \subseteq V$ and $E' \subseteq E$, and in this case the aim if to find a mapping $f' : V' \to V$ such that $(u, v) \in E'$ iff $(f'(u), f'(v)) \in E$. When such a mapping exists, this is called a subgraph matching or subgraph isomorphism [Bengoetxea 2002].

Various approaches to inexact graph matching have been proposed in the literature [Riesen 2010a]:

Edit distance: Originally, edit distance was developed for string matching and a considerable amount of extensions have been proposed for graphs. The key idea is to model structural variation by edit operations reflecting modifications in structure and labeling [Neuhaus 2007]. Computing the edit distance of two graphs g1 and g2 is equivalent to finding a sequence of edit operations transforming graph g1 into graph g2 (insertions, deletions and substitutions). Such a sequence of edit operations is also termed an edit path from g1 to g2. The problem of measuring the similarity of two graphs hence turns into to the problem of finding the best model of the structural differences of two graphs. The edit operations used to modify the graphs are evaluated via an edit cost functions. Edit cost functions assigns a cost value to each edit operation reflecting the strength of the modification applied to the graph. To obtain a cost function on edit paths, individual edit operation costs of the edit path are accumulated. An edit path from g1 to g2 with minimal costs can then be defined as the best model for the structural differences of g1 and g2. This minimum cost edit path assigned to two graphs is called edit distance. In order to compute graph edit distance and related measures, A* based search techniques using some heuristics are often employed [Riesen 2007]. A* is a best-first search algorithm [Hart 1968],which always finds a solution if there is one and it never overestimates the cost of reaching a goal.

Applying the edit distance on the graph g1 and g2 described in Figure 2.7 means to transform the graph g2 into the graph g1. All edit operations are performed on the data graph. One of the edit operation sequences includes node insertion and edge insertion (node f and its relative edge), node deletion and edge deletion (node H and its relative edges), node substitution (node a) and edge substitution (the edge relative to node e and node c). A cost function is defined for each operation and the cost for this edit operation sequence is the sum of costs for all operations in the sequence.

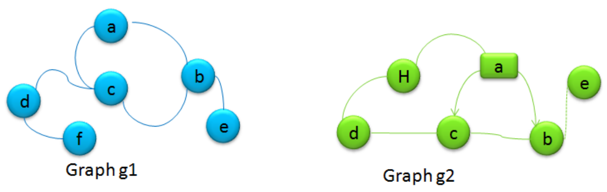


Figure 2.7: Graph Edit Distance.

Relaxation Labeling : The basic idea of this particular approach is to formulate the graph matching problem as a labeling problem [Fischler 1973]. Each node of one graph is to be assigned to one label out of a discrete set of possible labels, specifying a matching node of the other graph. During the matching process, Gaussian probability distributions are used to model compatibility coefficients measuring how suitable each candidate label is. The initial labeling, which is based on the node attributes, node connectivity, or other information available, is then refined in an iterative procedure until a sufficiently accurate labeling, i.e. a matching of two graphs, is found. This initial idea is extended by using [Wilson 1997] Bayesian Measure, by taking in account the edge label [Huet 1999].

Spectral methods: Many inexact matching methods are based on the spectral approach [Umeyama 1988] [Xu 2001] [Wilson 2005]. The basic idea of this approach is based on the observation of the proper values and if the proper vectors of the adjacency matrix of a graph are invariant with respect to node permutations. Hence, if two graphs are isomorphic, their adjacency matrices will have the same proper values and proper vectors. The reverse is not true in general. The major problem in this approach is that it is purely structural, in the sense that it is only applicable for unlabeled graphs, or that it does not exploit node or edge attributes.

Graph Kernel: Numerous inexact graph matching methods have integrated a graph kernel. The graph Kernel can be divided into three classes: (1) the convolution kernel [Watkins 2000] which provide general framework for dealing with complex objects that consist of simpler parts, (2)Kernels based on the analysis of random walks in graphs (3)Diffusion kernels which are defined with respect to a base similarity measure which is used to construct a valid kernel matrix.

Miscelanous Methods : Several approaches have been proposed in the literature; approaches based on Artificial Neural Networks [Sperduti 1997], decomposition methods [Messmer 1998], neural networks [Shonkry 1996], genetic algorithms [Suganthan 2002], methods based on bipartite matching [Baeza-Yates 2000] and methods based on local properties [Depiero 1996].

2.2.1.3 Summary

In this section we have given an overview of both exact and inexact graph matching. The emphasis has been on the fundamental concepts.

In the case of exact graph matching, the conversion of the underlying data into graphs always proceeds without errors. Otherwise, if distortions are present, graph and subgraph isomorphism detection are rather unsuitable. It restricts the applicability of exact graph matching algorithms. Inexact methods, sometimes also referred to as error-tolerant methods, are characterized by their ability to cope with errors, or non-corresponding parts, in terms of structure and labels of graphs. Hence, in order for two graphs to be positively matched, they need not be identical at all, but only similar. The notion of graph similarity depends on the error-tolerant matching method that is to be applied.

In addition to the classical approaches, many new approaches have integrated techniques like ontologies to improve the matching results. Semantic matching approaches attempt to match graphs based on their meaning by taking into account vertex and edge types and attributes as well as graph structure [Aleman-Meza 2005] [Coffman 2004]. However, since graphs can serve as conceptual representations, it can be useful to match graphs based not strictly on their similarity as graphs, but on the similarity of their interpretations in some domain of interest.

2.2.2 Graph Querying

Efficient and effective graph querying systems are critical to query and mine the ever growing graph datasets in various applications.

A query language is a collection of operators or inference rules which can be applied to any valid instance of the model data structure types, with the objective of manipulating and querying data in those structures in any desired combination [Codd 1980]. A number of query languages have been proposed for graphs. In General, a graph query takes a graph pattern as input, retrieves graphs from the database which contain (or are similar to) the query pattern, and returns the retrieved graphs or new graphs composed from the retrieved graphs. The existing query languages can be classified into visual, semantic, SQL-like and formal query languages. In the following, the graph in Figure 2.8 will be used to perform some query examples.

2.2.2.1 Visual Query Languages

Visual query languages aim to provide the functionality of textual query languages to users who are not technical database experts, and also to improve the productivity of expert database users (Harel, 1988). In general, these languages allow

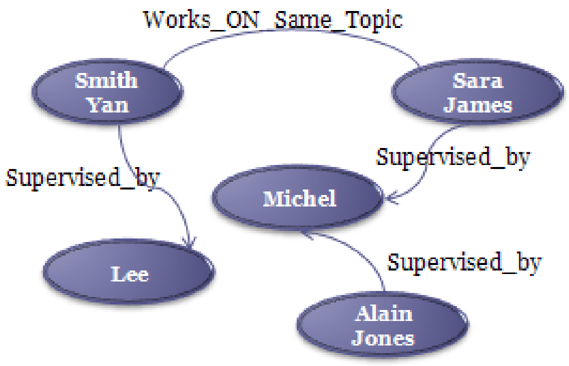


Figure 2.8: Graph Example.

users to draw a query as a graph pattern with the help of a graphical interface. The result is the collection of all subgraphs of the database matching the desired pattern [Blau 2002], [Cruz 1987], [Cruz 1988].

A. G, G+ and GraphLog

G [Cruz 1987] is a visual query language based on regular expressions that allow simple formulation of recursive queries. *G* enables users to pose queries, including transitive closure, which relational query languages cannot express. A graphical query *Q* (example Figure 2.9) is a set of labeled directed multi-graphs, in which the node labels of *Q* may be either variables or constants, and the edge labels are regular expressions defined over n-tuples of variables and constants. A path is expressed on a *G* query initially by the means of two types of edges: Dashed edges correspond to paths of arbitrary length in the graph and solid edges correspond to paths of fixed length. In *G*, simple paths are traversed using certain non-Horn clause constructs available in Prolog. However, it does not support cycles, find the shortest path or calculate node distance. Moreover, *G* does not support aggregation functions.

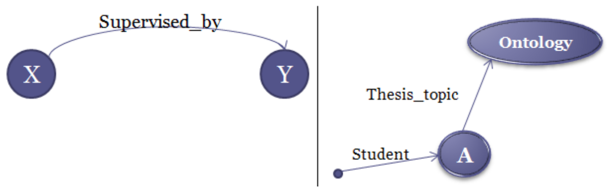


Figure 2.9: G query to find students and supervisors and query GraphLog query to find all students working on Ontology.

G evolved into a more powerful language called G+ [Cruz 1988], in which a query graph remains as the basic building block. A simple query in *G*+ has two elements; a query graph that specifies the class of patterns to search, and a summary graph, which represents how to restructure the answer obtained by the query graph. G+ provides primitive operators like depth-first search, shortest path, transitive closure and connected components. It can easily find a regular simple path. The language also contains aggregate operators that allow the path length and node degree to be found. The graph-based query language G+ provided the starting point for GraphLog [Consens 1989]. GraphLog differs from *G*+ with a more general data model, the use of negation, and the computational traceability. GraphLog queries are graph patterns which ask for patterns that must be present or absent in the database graph. Edges in queries represent edges or paths in the database. Each pattern defines a set of new edges (i.e., a new relation) that are added to the graph whenever the pattern is found. An edge used in a query graph either represents a base relation or is itself defined in another query graph. GraphLog supports computing aggregate functions and summarizing along paths. Figure 2.9 shows an example of a GraphLog query.

B. Hyperlog

Hyperlog [Levene 1991] is a declarative query and update language for the Hypernode Model (Figure 2.10). It visualizes schema information, data, and query output as sets of nested graphs, which can be stored, browsed and queried in a uniform way.

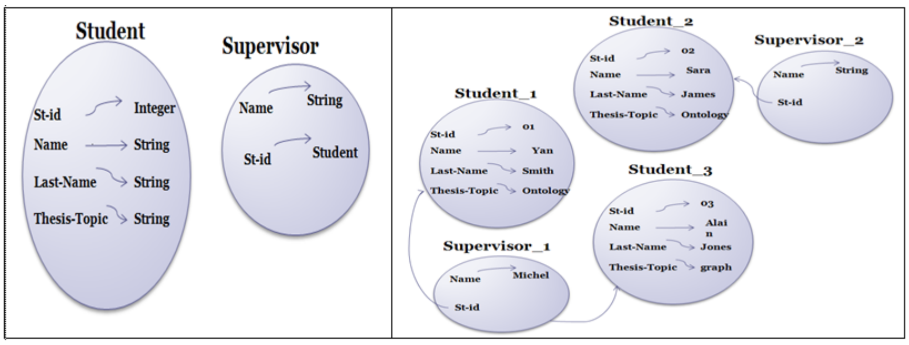


Figure 2.10: Hypernode Model Schema and Instance

A hyperlog query consists of a number of graphs (templates) which are matched against the hypernodes and which generate graphical output.

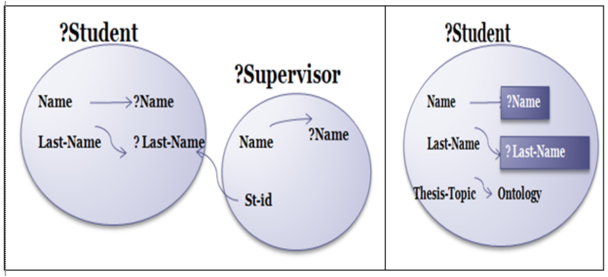


Figure 2.11: Template and Query with Hyperlog.

The user chooses which variables in the query, that should have their instantiations output in the query result. Hyperlog programs contain sets of rules. The body of a rule is composed of a number of queries, which may contain variables. The head of a rule is also a query and indicates the updates (if any) to be undertaken for each match of the graphs in the body. In order to illustrate the template and the query in the Hyperlog query language, an example is given in Figure 2.11. The template can find the students and their supervisors. The query can find the students working on Ontology. Hyperlog does not offer a special notation or expression to express paths. The existing rules can find only simple ones. The absence of aggregation functions explains the absence of answers to a query about node degree or path lengths.

C. QGRAPH

QGRAPH [Blau 2002] query is a labeled connected graph in which the vertices correspond to objects and the edges to links with a unique label. The query specifies the desired structure of vertices and edges. It may also place Boolean conditions on the attribute values of matching objects and links, as well as global constraints. A query consists of match vertices and edges and optional update vertices and edges. The former determines which subgraphs in the graph database constitute a match for the query. The latter determine modifications made to the matching subgraphs. A query with both match and update vertices and edges can be used for attribute calculation and for structural modification of the database. The query processor first finds the matching subgraphs using the query's match elements, and then makes changes to those subgraphs as indicated by the query's update elements. QGRAPH offers a good support to express paths by means of

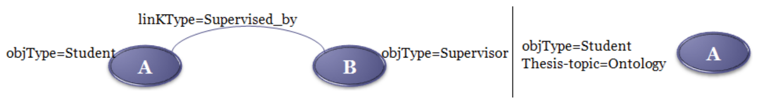


Figure 2.12: Queries with QGRAPH.

sub-queries, conditions and annotations on edges and nodes. However, it does not offer an operator for aggregation. Figure 2.11 contains two queries. The right query finds all subgraphs with a supervised link between a Student and a Supervisor. The left one finds just the students that have Ontology as the Thesis-topic.

D. GOOD and Languages based on GOOD

The GOOD [Gyssens 1990b] data transformation language is a database language with graphical syntax and semantics. This query language is used for the GOOD graph-based data model (Figure 2.13). GOOD query language is based on graph-pattern matching and allows the user to specify node insertions and deletions in graphics.

GOOD contains five operators. Four of these correspond to elementary manipulation of graphs: Addition of nodes and edges, deletion of nodes and edges. The fifth operation called "abstraction" is used to group nodes on the basis of common

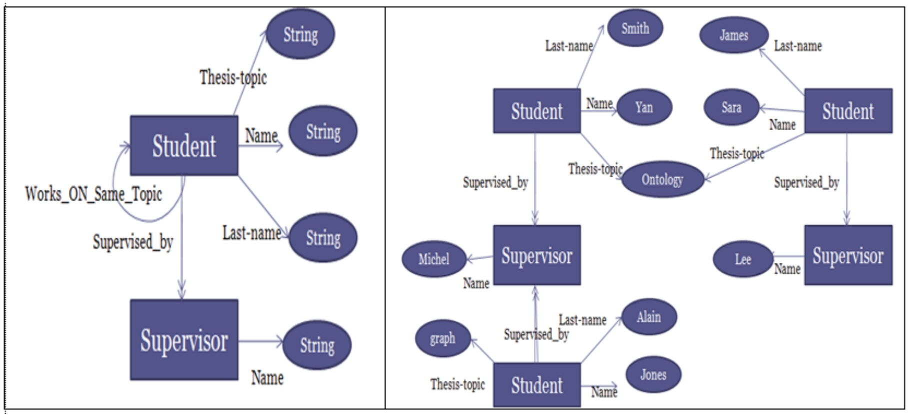


Figure 2.13: GOOD Data Model Schema and Instance.

functional or non-functional properties. The specification of all these operations relies on the notion of a pattern to describe subgraphs in an object-based instance. GOOD presents other features like macros (for a more succinct expression of frequent operations), computational-completeness of the query language, and simulation of object-oriented characteristics like encapsulation and inheritance.

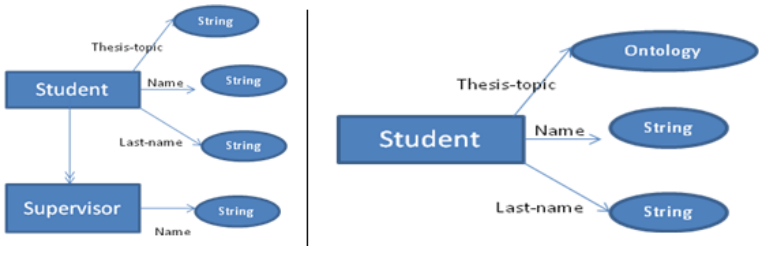


Figure 2.14: GOOD Queries.

A simple path can be discovered by using a pattern. Moreover, GOOD is incapable of finding a path with no fixed length. Figure 2.14 illustrates two examples of GOOD query: The first to find students and their supervisors; The second to find students working on the ontology topic. GOOD was followed by the proposals GMOD [Andries 1992], PaMaL [Gemis 1993] and GOAL [Hidders 2002]. These languages use GOOD's principal features and add several new functionalities.

2.2.2.2 SQL-Like Languages

SQL-like languages are declarative rule query languages that extend traditional SQL and, propose new SQL-like operators for querying graphs and objects [He 2008] [Abiteboul 1997] [Flesca 2000].

A.Lorel

Lorel [Abiteboul 1997] is implemented as the query language of the Lore prototype database management system at Stanford[2]. It is used for the OEM (Object Exchange Model) data model (Figure 2.15). A database conforming to OEM can be thought of as a graph where Object-IDs represent node-labels and OEM-labels represent edge-labels. Atomic objects are leaf nodes where the OEM-value is the node value. Lorel allows flexible path expressions, which allow querying without precise knowledge of the structure. Path expressions are built from labels and wildcards (place-holders) using regular expressions, allowing the user to specify rich patterns that are matched to actual paths in the graph database. Lorel also includes a declarative update language.

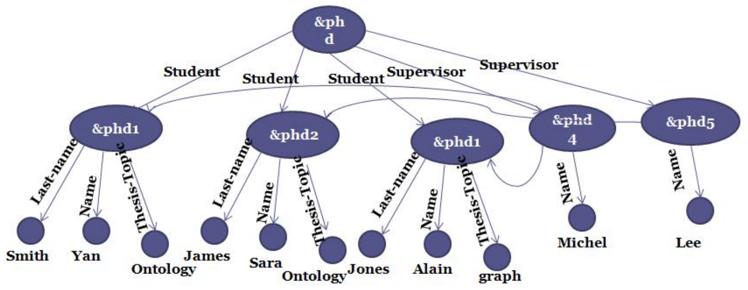


Figure 2.15: Object Exchange Model (OEM). Schema and Instance are Mixed.

B. GraphDB

Güting [Güting 1994] proposes an explicit model named GraphDB, which allows simple modeling of graphs in an object-oriented environment. A database in GraphDB is a collection of object classes where objects are composed of identity and tuple structure; attributes may be data- or object-valued. There are three different kinds of object classes called simple classes, link classes, and path classes. Simple objects are just objects, but also play the role of nodes in the database graph. Link objects are objects with additional distinguished references to source and target simple objects. Path objects are objects with an additional list of references to simple and link objects that form a path over the database graph. GraphDB uses graph algorithms in order to implement graph operations. Both the shortest path and cycle were implemented using the A* algorithm. Moreover, nodes, paths and subgraphs are indexed using path classes and index structures like B-Tree and LSD-Tree. GraphDB allows aggregation by using aggregate functions.

C. GOQL

GOQL [Sheng 1999] is an extension of OQL enriched with constructs to create, manipulate and query objects of type graph, path and edge. GOQL is applied to graph databases that use an object-oriented data model. In this data model, they define, similar to GraphDB, a special type: Node type, edge type, path type and graph type. GOQL is capable for querying sequences and paths. In addition to the OQL sequence operators, GOQL uses the temporal operators "Next", "Until"

[2]http://www.db.stanford.edu/lore

and ”Connected" for queries involving the relative ordering of sequence elements. For processing, GOQL queries are translated into an operator-based language, O-Algebra, extended with new operators. O-Algebra is an object algebra designed for processing object-oriented database (OODB) queries. To deal with GOQL's extension for path and sequence expressions, O-Algebra is extended with three temporal operators, corresponding to the temporal operators: ”Next", ”Until" and ”Connected".

D. SOQL

SoQL (Social networks Query Language), [Ronen 2009] is an SQL-like language for querying and creating data in social networks. SoQL enables the user to retrieve paths to other participants in the network, and use a retrieved path in order to attempt to create a connection with the participant at the end of the path. The main element of a SoQL query is either a path or a group, with subpaths, subgoups and paths within a group defined in the query. Creation of new data is also based on the path and group structures. Indeed, SoQL presents four new operators:
-SELECT FROM PATH query which retrieves paths between network participants, starting at a specific node and satisfying conditions in the path predicates.
- SELECT FROM GROUP query which retrieves groups of participants that satisfy conditions as a set of nodes.
-The CONNECT USING PATH and CONNECT GROUP commands are presented. These commands automate the process of creating connections between participants.
The language uses operators which specify conditions on a path or a group. It also proposes aggregation functionalities, as well as existential and universal quantifiers on nodes and edges in a path or a group, and on paths within a defined group.

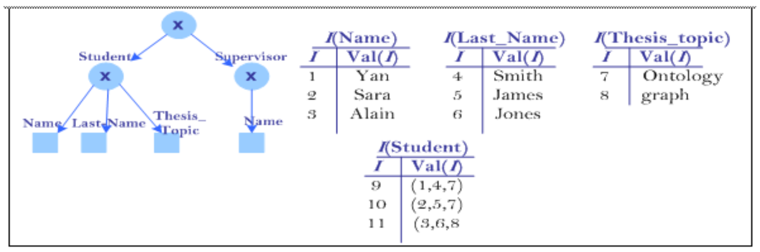
B.GraphQL

GraphQL [He 2008] is a graph query language for graphs with arbitrary attributes and sizes. In GraphQL, graphs are the basic unit of information. Each operator takes one or more collections of graphs as input and generates a collection of graphs as output. It is based on graph algebra and the FLWR (For, Let, Where, and Return) expressions used in Xquery. In the graph algebra, the selection operator is generalized to graph pattern matching and a composition operator is introduced for rewriting matched graphs using the idea of neighborhood subgraphs and profiles, refinement of the overall search space, and optimization of the search order.

2.2.2.3 Formal languages

A. LDM

The Logical Database Model [Kuper 1993] presents a logic very much in the spirit of relational tuple calculus, which uses fixed types of variables and atomic formulas to represent queries over a schema using the power of full first order languages. Figure 2.16 presents the LDM schema and instances.



I(Name)	
I	Val(I)
1	Yan
2	Sara
3	Alain

I(Last_Name)	
I	Val(I)
4	Smith
5	James
6	Jones

I(Thesis_topic)	
I	Val(I)
7	Ontology
8	graph

I(Student)	
I	Val(I)
9	(1,4,7)
10	(2,5,7)
11	(3,6,8

Figure 2.16: Logical Data Model, the schema (on the left) and part of instances (on the right).

The result of a query is another LDM schema called "query schema", which consists of those objects over a valid instance, that satisfy the query formula. In addition, the model presents an alternative algebraic query language proven to be equivalent to the logical one.

B. Gram

Gram [Amann 1992] is an algebraic language based on regular expression and supporting a restricted form of recursion. Figure 2.17 shows the data model used by

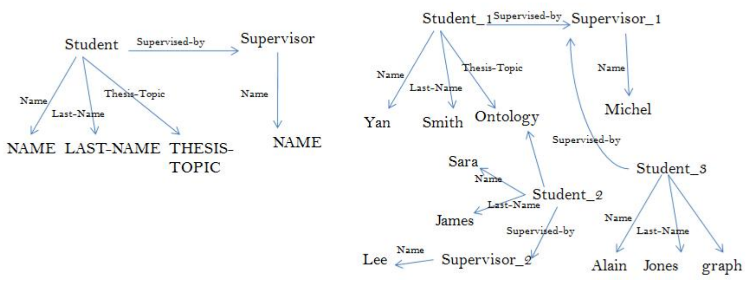


Figure 2.17: Gram Data Model, the schema (on the left) and the instances (on the right)

Gram. Regular expressions over data types are used to select walks (paths) in a graph. It uses a data model where walks are the basic objects. A walk expression is a regular expression without union, whose language contains only alternating sequences of node and edge types, starting and ending with a node type. The query language is based on hyperwalk algebra with operations closed under the set of hyperwalks. This hyperwalk facilitates the query of paths and the finding of adjacent node and edge. A Gram query example is presented in Figure 2.18.

C. G-Log

G-Log [Paredaens 1995] is a declarative, non-deterministic complete language for complex objects with identity. The data model of G-Log is (right down to minor details) the same as that of GOOD (Fig 2.19). The main difference between G-Log and GOOD is that the former is a declarative language, and that the latter is imperative. In G-Log, the basic entity of a program is a rule. Rules in G-Log are graph-based and are built up from colored patterns. A G-Log program is defined

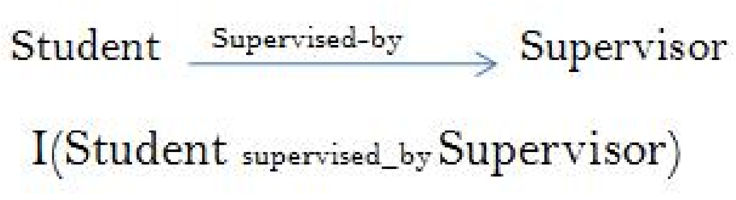


Figure 2.18: Gram query to find students and their supervisors.

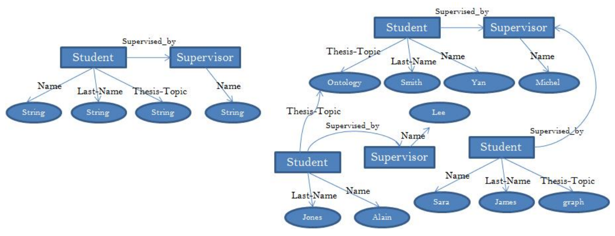


Figure 2.19: G-Log Data Model: The schema (on the left) and the instances (on the right)

as a sequence of sets of G-Log rules.

D. HNQL

HyperNode Query Language (HNQL) is a query and update language for the hypernode model [Levene 1995]. HNQL consists of a basic set of operators for declarative querying and updating of hypernodes. In addition to the standard deterministic operators, HNQL provides several non-deterministic operators, which arbitrarily choose a member from a set. HNQL is further extended in a procedural style by adding to the said set of operators an assignment construct, a sequential composition construct, a conditional construct for making inferences and, finally, loop and while constructs for providing iteration (or equivalently, recursion) facilities.

2.2.2.4 Semantic Languages

A semantic query language is a query language which is defined for querying a semantic data model.

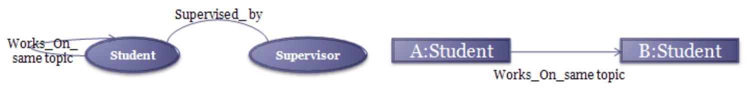


Figure 2.20: Ontology describing the graph (left) and the pattern to extract students working on same topic (right).

The semantic query language presented in [Kaplan 2006] provides a foundation for extracting information from the semantic graph where the possible structure of the graph is described by ontology (Figure 2.20) that defines the vertex types, the edge types and how edges may interconnect vertices to form a directed graph. The language uses a query with a specific format containing a function which specifies patterns and conditions for matching graphs in the database. Figure 2.20 shows an example of a pattern used by Kaplan query language.

2.2.2.5 Query Languages for Graph-Like Data

A. Query Languages for RDF

Several languages for querying RDF data have been suggested, some in the tradition of database query languages (i.e. SQL, OQL): RQL [Karvounarakis 2002], SeRQL [Broekstra 2003], RDQL [Seaborne 2004] and SPARQL [Pérez 2009]. Others more closely inspired by rule languages: Triple [Sintek 2002], Versa, N3 and RxPath. The currently available query languages for RDFs support a wide variety of operations. However, several important features are not well supported, or not supported at all. RDF query languages support only querying for patterns of paths which are limited in length and form. Nevertheless, RDF allows the representation of irregular and incomplete information (e.g the use of blank node). Of the original approach, only Versa and SeRQL provide a built-in means of dealing with incomplete information. For example, the SeRQL language provides so-called optional path expressions (denoted by square brackets) to match paths whose presence are irregular. Usually, such optional path expressions can be simulated, if a language provides set union and negation. Others work on RDF query languages to try to extend the original languages to improve path expressiveness. For example, Alkhateeb et al., [Alkhateeb 2009] allow an RDF knowledge base to be queried using graph patterns whose predicates are regular expressions. In RDF Path, N3 and Graph Path, they try to use specifications similar to those in XPATH to query paths in RDF. Moreover, RDF query languages are not well adapted to query paths of unknown length or which include multiple propriety on RDF graph. Neighborhood retrieval cannot be performed well for languages that do not have a union operator. Many of the existing proposals support very little functionality for grouping and aggregation. Moreover, aggregated functions like COUNT, MIN, MAX applied to paths could be used to answer queries in order to analyze data (like the degree of a node, the distance between nodes, and the diameter of a graph). Exceptions can be found in Versa, RQL and N3 which support count functionality aggregation in path and nodes, are not explicitly treated by any languages which need to be considered as a requirement.

B. Query Languages for XML

The Extensible Markup Language (XML) is a subset of SGML. XML data are labeled ordered trees (with labels on nodes), where internal nodes define the structure, and leaves the data (scheme and data are mixed.). XML additionally provides a referencing mechanism among elements that allows the simulation of arbitrary graphs. In this sense, XML can simulate semi-structured data. Also, many new extensions of XML are designed to represent graphs like GML, GraphML, XGML and etc. Current query languages [Bonifati 2000] for XML do not support the majority features for graph-structured XML document. The principal feature supported is path. For example, XPath [Consortium 2010a] uses path expressions to select nodes or node-sets in an XML document. Also, the set of axes defined in XPath is clearly designed to allow the set of graph traversal operations that are seen to be atomic in XML document trees. An XPath axis is fundamentally a mapping from nodes to nodesets and defines a way of traversing the underlying graph. Each axis encapsulates two things: a type of edge to follow (eg. child vs. attribute) and whether it is followed transitively (e.g. child vs. descendant). Also, XQuery [Consortium 2010b] uses XPath to express complex paths and supports flexible query semantics. In XML-QL [Deutsch 1998], path expressions are admitted within the tag specification and they permit the alternation, concatenation and Kleene-star operators, similar to those used in regular expressions. In XML-GL [Ceri 1999], the only path expressions supported are arbitrary containment, by means of a wildcard* as the edge label; this allows the traversal of the XML-GL graph, reaching an element at any level of depth. However, current query languages for XML are designed for tree-structured XML data and do not support the matching of schema in form of general graph. Even though XPath can express a node with multiple parents by multiple constraints with axis "parent", it cannot express a graph with cycles. While XML will not allow multiple parents, there is nothing in XQuery (or XPath in particular) which precludes a traversal from parent to child to a different parent. This insufficiency does not allow the presentation and the query of all kinds of graphs.

2.2.2.6 Summary

In order to effectively handle graph management applications, we need query languages which allow expressivity for management and manipulation of structural data. We have detailed in this section the different query languages for graph or graph like data model. These languages can be compared via the following functionalities:

- **Data Selection:** all the previous languages offer operator, function or pattern that can select data from any graph model. Some query languages are specific to a particular graph data model like Lorel, HNQL which cannot be generalized to a general graph. Thus, the XML or RDF query languages cannot be applied to general graph. Some query languages allow the selection of data from multiple documents (like Lorel and some XML query languages).

- **Aggregation and grouping**: few graph query languages propose aggregation operator. For instance, Lorel uses the SQL operators count and Group by and GraphDB offers the count operator. For the RDF query languages Versa, RQL and N3 support the count functionality. The work in [Chong 2005] is based on an aggregate approach which does not work for aggregations that require graph awareness. The same approach is also employed in [?], in order to extend the RDQL query language with grouping and aggregation functionalities. In the LAGAR [Chen 2005] algebra for querying RDF data, grouping and aggregation are defined over graph sets which essentially correspond to n-ary bindings. For SPARQL, some applications try to extend it by adding an aggregate function. For example, ARQ implements COUNT() and SUM() with syntax like $(COUNT(*)AS?c)$. The various proposals for grouping and aggregate functions for XML (e.g. [Borkar 2004] [S. Paparizos 2002]) rely on a nested representation resulting from XML's tree data model where only nodes are labeled and edges have implicit containment semantics. Aggregation in path and nodes are not explicitly treated by any language which needs to be considered as a requirement.

- **Graph Features**: a Path expression is a central feature to support graph matching which is typically used to traverse a graph. A path expression can be decomposed into several joins and is often implemented by joins. Almost all the previous query languages support simple path queries. Moreover, in a path query we can distinguish different types: simple path (path with no repeated vertices), path queries without variables [Cruz 1988] [Consens 1989], path queries involving uncorrelated paths, parametric regular path queries [Liu 2004], and universal regular path queries. Many approaches in graph database languages have proposed solutions to query different kinds of paths. For example, G+ proposes regular expressions for finding regular paths in a graph. As we have mentioned previously path queries are treated partially. Indeed, paths with unknown length or including multiple properties are not treated by the existing RDF query languages. In XML query languages, all languages are able to find paths. However, we have also the same problem to query optional path expressions. XPATH can only handle paths containing nodes that are descendants of the current node, and this cannot result in non-terminating processing loops. Also, conditions apply only to the last node on the path extended from the context node. Other graph features can be treated by the graph query languages. Adjacent node and adjacent edge are expressed as a union of two queries: one for outgoing edges and another for ingoing edges. Thus, only languages which can express union can support these features. The length of the shortest path and the degree of node need aggregation function which is not supported by all languages Graph query languages are the most adapted to support graph features. For example, GraphLog and G+ languages support all the graph features from adjacent node and edges to the diameter.

2.2.3 Graph Transformation

In this section, another field of the graph manipulation is presented which is the graph transformation.

2.2.3.1 Graph Transformation Presentation

Graph transformation concerns the technique of creating a new graph out of an original one using some automatic techniques. It has been used for many fields such as: Computational model for term rewriting systems and functional programming languages [Barendsen 1999], for specifying visual languages and generating associated editors [Bardohl 1999] [Minas 2002]. Besides specific applications of graph transformation, several programming languages have been developed that are based on graph transformation rules (Agg [Ermel 1999], Gamma [Banâtre 2001], Grrr [Rodgers 1998] and Dactl [Glauert 1997].)

In fact, graph transformation has at least three different roots [Ehrig 2006a]

- from Chomsky grammars on strings to graph grammars
- from term rewriting to graph rewriting
- from textual description to visual modeling.

The different approaches of graph transformation can be resumed like in the following [Rozenberg 1997]:

- **The node label replacement approach:** Allows replacing a single node as left hand side L by an arbitrary graph R. The connection of R with the context is determined by embedding rules depending on node labels.
- **The hyperedge replacement approach:** This has as left hand side L a labeled hyperedge, which is replaced by an arbitrary hypergraph R with designated attachment nodes corresponding to the nodes of L. The gluing of R with the context at corresponding attachment nodes leads to the target graph.
- **The algebraic approaches** are based on pushout and pullback constructions in the category of graphs, where pushouts are used to model the gluing of graphs.
- **The logical approach** allows expressing graph transformation and graph properties in modanic second order logic.
- **The theory of 2-structures** was initiated as a framework for decomposition and transformation of graphs.
- **The programmed graph replacement approach** used programs in order to control the nondeterministic choice of rule applications.

2.2.3.2 Graph Transformation Formalism

Graph Transformation deals with the manipulation of (labeled, directed) arbitrary graph shape by rules, combining the strengths of graphs and rules into a single computational model. It provides a pattern and rule-based manipulation of graph models. Each rule application transforms a graph by replacing a part of it by another graph [Varró 2005].
The idea of graph transformation is to consider a rule or production $p = (L, R)$ where L is the left hand side and R is the right hand side. If the match m found an occurrence of L in a given graph G, then $G \overset{p,m}{\Longrightarrow} H$ denotes the direct derivation where p is applied to G leading to a derived graph H. H is obtained by replacing the occurrence of L in G by R. A graph transformation [Baresi 2002] from a pre-state G to a post-state H, denoted by $G \overset{p(o)}{\Longrightarrow} H$, is given by a graph homomorphism $o : L \cup R \to G \cup H$, called occurrence, such that:
-$o(L) \subseteq G$ and $o(R) \subseteq H$, i.e., the left-hand side of the rule is embedded into the pre-state and the right-hand side into the post-state, and
-$o(L \setminus R) = G \setminus H$ and $o(R \setminus L) = H \setminus G$, i.e., precisely that part of G is deleted which is matched by elements of L not belonging to R and, symmetrically, that part of H is added which is matched by elements new in R.

A simple example of a direct derivation is presented in Figure 2.21.

In order to apply p1 to the graph G1, we should find a match m1 between L1 and G1. The match m1=n1,n2, n3. Applying production p1 to G1 at match m1 is to delete every object from G1 which matches an element of L1 that has no corresponding element in R1. In this example, we delete n2. Symmetrically we add to G1 each element of R1 that has no corresponding element in L1. Thus, e5 is added. All the other elements of G1 are maintained like the edge e4.

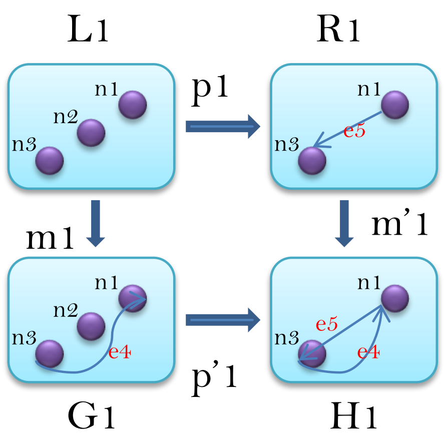


Figure 2.21: Graph Transformation: Direct Derivation.

2.2.3.3 Typed Attributed Graph Transformation

Graph transformation has been applied not only for labeled graph but also for typed attributed graphs. In [Heckel 2002], an attributed graph is modeled as a graph with node attributed (as defined in definition 5). However in [Ehrig 2006b], they model attributed graph with node and edge attribution. They called this new concept E-graph. An E-graph is defined as follows in [Ehrig 2004]:

Definition 11 (E-graph) *An E-graph* $G = (V1, V2, E1, E2, E3, (source_i, target_i)_{i=1,2,3})$ *consists of sets*

- *V1 and V2 called graph resp. data nodes,*
- *E1, E2, E3 called graph, node attribute and edge attribute edges respectively,*

and source and target functions

- $source_1 : E_1 \longrightarrow V_1$*,* $source_2 : E_2 \rightarrow V_1$*,* $source_3 : E_3 \rightarrow E_1$*,*
- $target_1 : E_1 \rightarrow V_1$*,* $target_2 : E_2 \rightarrow V_2$*,* $target_3 : E_3 \rightarrow V_2$*.*

An E-graph morphism $f : G_1 \rightarrow G_2$ *is a tuple* $(f_{V_1}, f_{V_2}, f_{E_1}, f_{E_2}, f_{E_3})$ *with* $f_{V_i} : G_{1,V_i} \rightarrow G_{2,V_i}$ *and* $f_{E_j} : G_{1,E_j} \longrightarrow G_{2,E_j}$ *for i = 1, 2, j = 1, 2, 3 such that f commutes with all source and target functions. E-graphs combined with E-graph morphisms form the category EGraphs.*

Thus, an attributed graph AG consists of an E-graph G and a data type D, where parts of the data of D are also vertices in G.

For this complex graphs the graph transformation is realized via a typed attributed graph transformation system which is defined in [Ehrig 2004] as follows:

Definition 12 (Typed Attributed Graph Transformation System) *A typed attributed graph transformation system* $GTS = (DSIG, ATG, S, P)$ *based on* $(AGraphs_{ATG}, M)$ *consists of a data type signature DSIG, an attributed type graph ATG, a typed attributed graph S, called start graph, and a set P of productions, where:*

1. $AGraphs_{ATG}$ *is a category formed by typed attributed graphs over an attributed type graph ATG and tyed graph morphisms.*
2. *a production* $p = (L \longleftarrow K \longrightarrow R)$ *consists of typed attributed graphs L, K and R attributed over the term algebra* $T_{DSIG}(X)$ *with variables X, called left hand side L, gluing object K and right hand side R respectively, and morphisms* $l, r \in M$ *i.e. l and r are injective and isomorphisms on the data type* $T_{DSIG}(X)$*,*
3. *a direct transformation* $G \overset{p,m}{\rightarrow} H$ *via a production p and a morphism* $m : L$*, called match,*

4. *a typed attributed graph transformation, short transformation, is a sequence* $G0 \Longrightarrow G1 \Longrightarrow ...Gn)$ *of direct transformations, written* $G0 \overset{*}{\Longrightarrow} Gn$,

5. *the language* $L(GTS)$ *is defined by* $L(GTS) = \left\{G|S \overset{*}{\Longrightarrow} G\right\}$

An example of the attribute typed graph transformation is depicted in Figure 2.22. In this example, we modify the input graph G by adding a new relation "Affected-to-Project" and deleting an old one "Supervised".

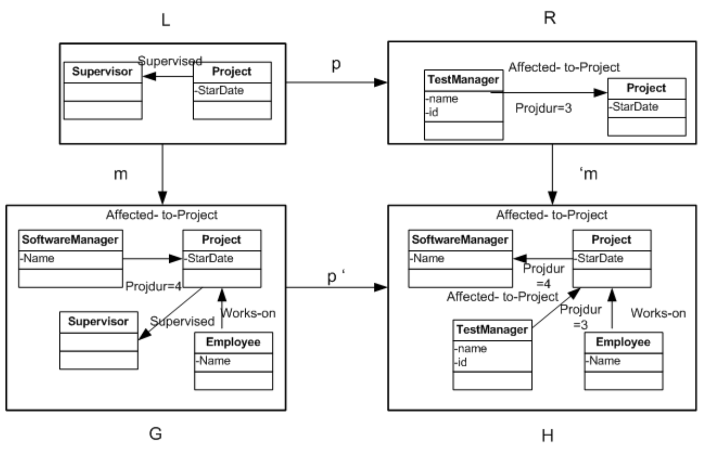


Figure 2.22: Attributed Typed Graph Transformation with Pushout.

2.3 Relational Database and Graph Models

2.3.1 Transforming Relational Database to a Graph Model

2.3.1.1 Transforming RDB to Entity Relationschip Model

The Entity-Relationship (ER) model, originally proposed by Chen in 1976 [Chen 1976], has gained wide acceptance in the area of database design and related fields. ER is an abstract and conceptual representation of data which has been used to produce a type of conceptual schema or semantic data model of a system. An ER model is composed of entities, relationships and attributes which can be modeled by a graph where entities and attributes are represented by nodes and the relationships are represented by labeled edges. Numerous approaches have been proposed in the past for accomplishing this task. [Fahrner 1995] presents a survey about these approaches and classifys them in three categories:

- The first category is based on the evaluation of the inclusion dependencies [Casanova 1984] (IND) of a given relational schema. Each IND is interpreted using the attributes role: ”Key", ”part of a key", ”foreign key" or ”non-key". These approaches can require the use of all the relations INDs and keys, key-based INDs [Casanova 1983] [Ji 1991] or general INDs [Mannila 1992].

- The second category is based on the use of keys and attribute names [Fonkam 1992] [Navathe 1988]. The transformation in these approaches is done using the classification of keys construction (e.g. whether the primary key of a relation is the concatenation of the primary keys of other relations) and the keys name (since relationships between keys are identified through their names which impose a proper name assignment).

- Some approaches use the techniques adopted in the two previous categories like the approaches of Chiang *et al.* [Chiang 1994] [Chiang 1995]

The RDB transformation of to an ER model can be a first step for another model transformation. The conceptual description offered by the ER model can help to extract the RDB semantics and transform it to other models.

2.3.1.2 Transforming RDB to RDF

Mapping Relational Databases to an RDF graph is an active field of research. The majority of data in the current Web is stored in RDBs. In order to make these data available to semantic web applications, it is necessary to transform them to RDF. Many surveys have been presented to classify the existing work on this field [Sahoo 2009] [Hert 2011] [Beckett 2003]. The existing approaches can be compared using a technique to create a mapping between the relational database and the RDF graph. The mapping can be done with three approaches: automatic direct mapping, automatic mapping using an existing ontology and Manual/semi-automatic mapping.
Automatic direct Mapping: These approaches allow to directly transform a RDB to an RDF graph. [Berners-Lee 1998] presents a direct mapping where relational tables are transformed to classes in an RDF vocabulary, and the attributes of the tables are transformed to properties in the vocabulary. The URIs of the instances as well as those of the vocabulary classes are generated automatically based on the RDB schema and data. This direct mapping has been adopted by many approaches. For example, the Virtuoso RDF View [Blakeley 2007] uses the unique identifier of a record (primary key) as the RDF object, the column of a table as RDF predicate and the column value as the RDF subject. In D2RQ [Bizer 2007] (D2RQ also allows users to define customized mappings) and SquirrelRDF [Seaborne 2007].
Automatic mapping using an existing ontology: These approaches automatically generates the RDB to RDF mappings where an existing ontology is used to enhance the quality of the mappings [Hu 2007]. For example, Dragut and Lawrence [Dragut 2004] transform relational schemas and the RDF ontologies into directed labeled graphs respectively, and reuse the schema matching tool COMA [hai Do 2002] to exploit simple mappings. The approach Ronto [Petros Papapanagiotou 2006], introduces six different strategies to discover mappings by distinguishing the types of

entities in relational schemas.
Manualsemi-automatic mapping: These approaches use also domain ontology to generate the mapping from RDB to RDF but the process of matching is done manually or semi-automatically. The domain ontology may be preexisting and sourced from public or may be bootstrapped from local ontologies created by automatic mapping tools. Green *et al.* [Green 2008] presented an approach of mapping spatial data to RDF using a hydrology ontology [Hart 2007] as the reference knowledge model. Sahoo *et al.* [Sahoo 2008] proposed an approach to generate mappings using the Entrez Knowledge Model (EKoM). This approach is also called Domain ontology mapping and the process is the same as an ontology population technique where the transformed data are instances of the concepts defined in the ontology schema. Many mapping tools such as D2RQ [Bizer 2007] allow the users to create customized mapping rules in addition to the automatically generated rules.

2.3.1.3 Transforming RDB to XML

XML has been extended to allow graph data modeling. Many approaches in the literature have ben proposed to transform a relational database to a XML document. None of them explicitly transform the RDB to a GML, GraphXML, GXL or GraphML. However, these approaches can be extended to support the graph in XML. The proposed approaches can be classified in two principal categories:

Direct translation: These approaches use a set of rules to direct transform a RDB to a XML document. In XPERANTO [Carey 2000] and Agora [Manolescu 2000], the XML schema is obtained by performing a set of SQL queries against the XML view. The mapping from the relational schema to the XML schema is done manually by experts. In XML Extender [Cheng 2000], The users have as input the relational schema and the XML target. Users of this tool need to manually supply the mapping between the relational and the XML schema. The tool has the feature to convert operations from XML to relational and obtain the results as XML.

Indirect Translation: These approaches use an intermediate structure to extract the semantic of the relational database, and to then translate it to the XML model. The intermediate models allow acquiring more information about the existing objects and their relationships. Such conversions are generally specified by rules which describe how to transform RDB elements (e.g., relations, attributes, data dependencies, keys) into a conceptual model such as the ER model, etc. In this category, other researchers started by transforming the relational model to a ER model then the ER model was transformed into a XML document. The Wang research can be cited here. In [Wang 2005], as a first step the ER model were extracted from the relational model using the reverse engineering technique in [Alhajj 2003], which resulted in a RID graph. Then, the RID graph is mapped into an XML Schema. After the schema is translated, a XML document is generated from the RDB data. In [Fong 2005] an extended ER model was extracted from the RDB schema and then it was mapped to a XSD graph which captured the relationships and constraints. The XSD graph is mapped to a XML schema. This mapping transforms the foreign keys into a hierarchy of element/sub-elements, which in some cases produces redundancy when an element has a relationship with more than one element. In [Fong 2006], an extended ER model was used to translate a RDB to an XML document according to a DTD schema.

In [Du 2001], they developed a method that employed an ORA-SS model to support the translation of a RDB schema into a XML Schema RDB-to-ORA-SS -to-XML Schema. They proposed translation rules for mapping a semantically enriched RDB schema into an ORA-SS model [Dobbie 2001]. In the literature, other algorithms can be found which transform an ER model to XML [Alhajj 2003], relational model to DTD [Laforest 2003], UML-to-XML [Conrad 2000].

2.3.2 Searching Information from Relational Database using Graph Techniques: Keyword Search in Relational Database

The keyword search approaches allow naive users to acquire information from the relational database without any knowledge about the schema or query languages. They allow a user to submit a query using a finite set of keywords to find the data that satisfies his/her information needs [Park 2011]. In this section, we will describe how the keyword search system transforms and uses the relational database as a graph to find the solution. With SQL, in order to search the relation between two objects, for example between an Employee and a Project: we need to specify the particular attributes, and generate complex queries with many joins and unions (if the two objects are not directly linked by a foreign key). In keyword search, stating two keywords, i.e., Employee and Project, is enough for the user. The keyword search system will compute the most meaningful answers on behalf of the user. Most of the recent studies in this domain have viewed a relational database as a graph, which represents a relational model easily. Actually, Keyword search systems with graph-based approach first build a materialized graph over the database. The values in tables or tuples can be modeled as nodes in the graph; or instead, the schema itself, such as the tables and their relationships, can be modeled in the graph. More precisely, graphs have been modeled in two ways: data graph models and schema graph models. In a data graph, nodes represent tuples and edges represent the foreign key relationships between pairs of tuples. [He 2007] [Kacholia 2005] [Din 2007] [Li 2008]. Once the data graph is constructed, the search system does not need to access the underlying database because the data graph keeps the values in redundant schema elements. Therefore, internal queries for a data-graph are used to get the seed nodes for graph traversing. Searching with a data-graph performs better than searching with a schema-graph because schema-graph searching requires the additional task of querying the underlying database after the graph traversing. It may be suitable to cite some approaches based on data-graph.
BANKS [Kacholia 2005] models tuples as nodes in a graph, connected by links induced by foreign key and other relationships. It introduces a backward edge to

enable the search system to traverse backward, so that a few nodes or edges with very large weights do not skew the answers. Answers to a query are modeled as rooted trees connecting tuples that match individual keywords in the query. BANKS employs the backward expanding search algorithm, which is based on Dijkstra's single source shortest path algorithm. It expands node clusters by visiting nodes backward until the rooted tree is completed.

DataSpot [Dar 1998] is based on a novel representation of data in the form of a schema-less semi-structured graph called a hyperbase. The DataSpot Publisher takes one or more possibly heterogeneous databases, predefined knowledge banks such as a thesaurus, and user-defined associations, and creates the hyperbase. To find query answers, the system performs a weighted best-first search using all of the nodes that contain the keywords (i.e., query source) until it finds nodes connected to all the query sources. The query answer is a connected sub-hyperbase that contains the keywords in the query. The answers are ordered according to the score, which can be computed by counting edges.
BLINKS [He 2007] system proposes a bi-level indexing and query processing scheme for top-k keyword search on a data graph. It introduces a cost-balanced expansion technique that optimizes the backward search by balancing the number of accessed nodes for expanding each cluster. In a schema-graph, nodes represent tables and edges represent the foreign key relationships between pairs of tables. [Agrawal 2002] and [Hristidis 2002] use a schema graph as their data representation method. A schema-graph exploits only the schema of the underlying database itself, and it has a smaller size than a data graph. A system based on schema-graph produces internal queries, such as SQL joins queries to retrieve values from the database.
DBxplorer [Agrawal 2002] and Discover [Hristidis 2002] are two examples of keyword search approaches based on a schema-graph. Given a set of query keywords, DBXplorer returns all rows (either from single tables, or by joining tables connected by foreign-key joins) such that each row contains all keywords. As a first step, DBXplorer performs a preprocessing step called "Publish", that enables databases for keyword search by building the symbol table, which locates the query keywords in the relational entities, and associated structures. The symbol table is the key data structure used to look up the respective locations of query keywords in the database. Many location granularities have been taken into consideration, where for every keyword the symbol table maintains the list of this granularity that contains it: Column level, cell level and hybrid. In order to remove the duplicate values from the database, compression algorithms have been proposed FK-Comp and CP-Comp. The duplicate values are for foreign key entry and common keywords. Secondly, all potential subsets of tables (Join tree) in the database that might contain tuples having all keywords are identified and enumerated. A subset of tables can be joined only if they are connected in the schema. Finally, each Join tree is then mapped to a SQL statement that joins the tables as specified in the tree. The enumerated Join trees are ranked by the number of connections.

DISCOVER is like DBXplorer in the sense that it also finding all Join trees which it calls candidate networks, by constructing Join expressions. For each candidate

Join tree, an SQL statement is generated. The generated SQL statement may contain many common Join structures that are due the fact that the trees may have many common components. DISCOVER uses an algorithm that maximizes the reusability and minimizes the size of the intermediate result. A new extension of Discover was proposed in [Hristidis 2003] which adopts the IR-style document-relevance ranking strategies.

2.4 Summary and Discussion

In this chapter, we study the different approaches related to our work. Firstly, we give an overview of the graph models and their manipulation techniques. Then, we present some relational database transformation to graph like model approaches. Now, we will present the important observations from each part.

Graph data models

Graphs provide a powerful primitive for modeling data in a variety of applications. A variety of real-world objects and relationships can easily be represented by nodes and edges in the graph for example social networks, technological networks, etc. In this work, the graph model will be used to model data extracted from multiple resources and representing heterogeneous objects. The presented models can be compared according to their abilities to model complex graphs with heterogeneous nodes, multiple attributes and multiple relations. Therefore, we have compared the basic foundation of the graph model (graph, hypernode or hypergraph), the attributes and edges type, the existence of the schema and instance level, the different support (inheritance, grouping, nested relation and complex object) by the models and finally examined whether it is adapted for a clear visualization of big graph. Using the result of the table 2.1, we can notice that models based on hypernode or hypergraph present a good model for complex graphs. Indeed, by using the grouping, each object will have its attributes encapsulated in the object node. Also, the encapsulation will facilitate the visualization of the graph in the case of multiple attributes.

Relational databases and graphs

In a business context, important expertise information is often stored in relational databases. Relational databases pervade almost all businesses. Many kinds of data, from e-mails and contact information to financial data and sales records, are stored in databases. Also, databases used in business contain information about all people, objects and processes related to the enterprise. In order to transform this relational model into a graph model, and also obtain from it graphs describing the interaction between enterprise objects (like social networks), we have studied the approaches allowing transforming relational database to graph-like data or to extract graphs. In other terms, our purpose is not only to transform the relational model into a graph model, but also to obtain graphs of the enterprise objects.

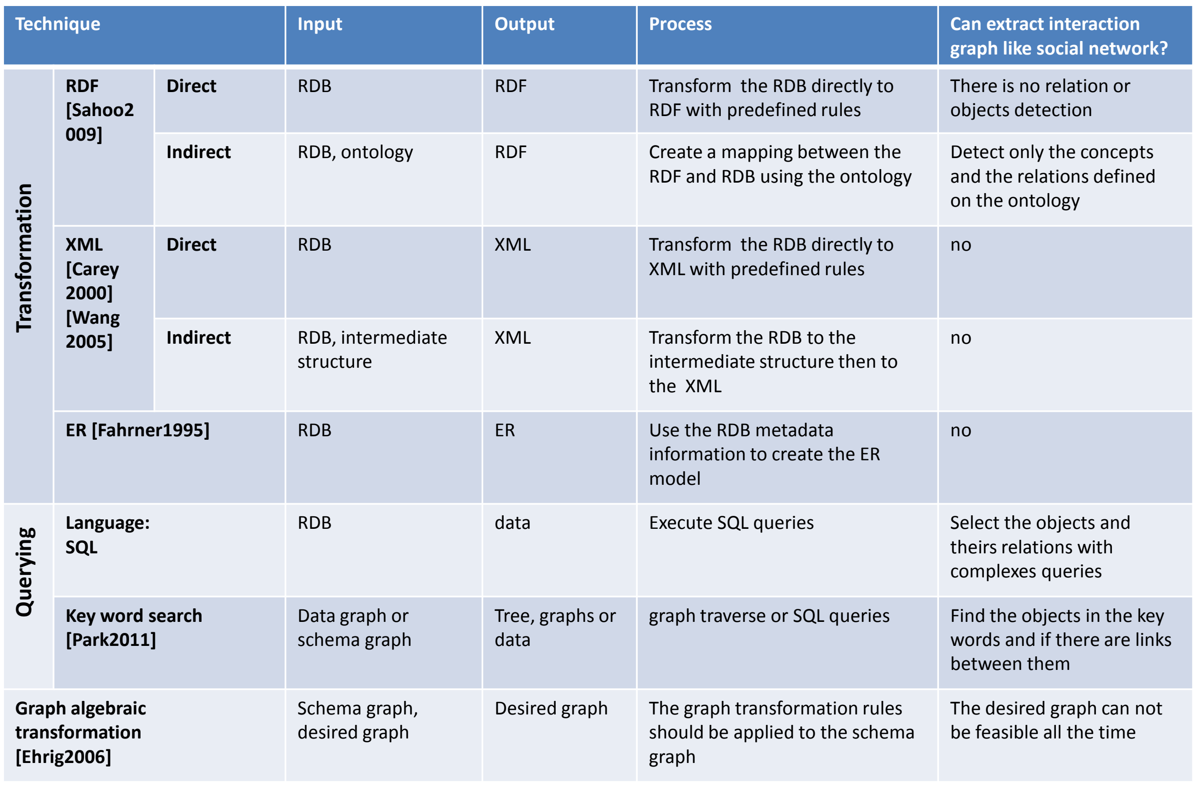
In the context of transformation approaches, we have presented the transformation to the entity relation model, RDF and XML. We have started by studying the transformation to ER model techniques. This transformation facilitates the

Table 2.1: Graph Models Comparison

		Graph Database							XML		RDF
		GMOD	GOOD	PaMaL	LDM	Hypernode	GGL	Grouvy	Graphml	GML	
Basic	Graph	+	+	+	+	-	-	-	+	+	+
	Hypernode	-	-	-	-	+	-	-	-	-	-
-	Hypergraph	-	-	-	-	-	+	+	+	-	
Node	Attributed	+		+	+	+	+	+	+	+	+
	Labeled	+	+	+	+	+	+	+	+	+	+
Edge	Attributed	-	-	-	-	-	-	-	+	+	+
	Labeled	+	+	+	-	-	+	+	+	+	+
Graph	Schema	+	+	+	+	+	-	+	-	+/-	+/-
	Instance	+	+	+		+	+	+	+	+	+
Support	Inheritance	-	-	+	-	-	-	+	-	-	-
	Grouping	-	-	-	-	+	+	+	+	-	-
	Nested Relation	-	-	+	-	+	+	+	+	-	-
	Complex Object	-	-	-	+	+	+	+	+	+	+
Visualization		+/-	-	-	+/-	+/-	+/-	+/-	+	+	+

transformation of the relational model to another model. Indeed, it detectes all the existing entities and relations in the database. These techniques are based only on the relational database information (keys, attributes name, etc). Using the ER model, we can then obtain the important information to get a graph model.However, the ER model does not model data as a real object graph (the nodes are objects and links relations between these objects). For instance, we can found entities that do not represent an object but a relation between two other entities.
The transformation to RDF or XML data model is performed directly or indirectly. The direct approaches allow to directly transform a relational database to the target model. These approaches use an intermediate structure to extract the semantic of the relational database, then to translate it to the target model.
The query techniques can be query languages based on predefined and formal queries and key word queries. The first one is by using directly the query languages such as SQL [ISO9075:1999 1999]. However, in order to query such data, the user must master a complex query language and understand the underlying data schema. In relational databases, information about an object is often scattered in multiple tables due to normalization considerations. In the case of a complex schema the queries will be very complexe with many joins and unions. The keyword search techniques presented in the section can extract the relation between specific objects. In these approaches a relational database can be viewed as a graph where tuples are modeled as vertices connected via foreign-key relationships. Using that, a query containing the name of two objects can extract if there is a relation between two objects.
Other techniques can be used to extract specific graphs from relational database. As we can consider the relational database as graph where table are nodes and related by the foreign keys which are the link, we can use the graph transformation presented in section 2.2.3. This transformation can be processed only if the user has an idea about the resulting graph that he would like to extract. However, in many cases this graph is not feasible to extract (e.g objects not connected or not existing). All these different approaches are resumed in the table presented in the Figure 2.23.

However, the approaches present some inconveniences:
-The transformation approaches can only transform the relational model to a graph like model. It does not allow extracting specific graphs which describe the relations between specific objects.
-The search approaches can only detect the links between two objects.
-The algebraic transformation needs a well defined right graph (desired graph).
-All these approaches cannot automatically discover hidden relations between objects or detect all the nodes that present the some object.
We can say that there is no approach which extracts interaction graphs from relational databases.



Technique			Input	Output	Process	Can extract interaction graph like social network?
Transformation	RDF [Sahoo2009]	Direct	RDB	RDF	Transform the RDB directly to RDF with predefined rules	There is no relation or objects detection
		Indirect	RDB, ontology	RDF	Create a mapping between the RDF and RDB using the ontology	Detect only the concepts and the relations defined on the ontology
	XML [Carey 2000] [Wang 2005]	Direct	RDB	XML	Transform the RDB directly to XML with predefined rules	no
		Indirect	RDB, intermediate structure	XML	Transform the RDB to the intermediate structure then to the XML	no
	ER [Fahrner1995]		RDB	ER	Use the RDB metadata information to create the ER model	no
Querying	Language: SQL		RDB	data	Execute SQL queries	Select the objects and theirs relations with complexes queries
	Key word search [Park2011]		Data graph or schema graph	Tree, graphs or data	graph traverse or SQL queries	Find the objects in the key words and if there are links between them
Graph algebraic transformation [Ehrig2006]			Schema graph, desired graph	Desired graph	The graph transformation rules should be applied to the schema graph	The desired graph can not be feasible all the time

Figure 2.23: Different Approaches to Extract Graphs from RDB.

2.5 Issues and Solution description

From the studied states-of-the art, we can enumerate the following problems from each part:

- **Absence of an adapted graph model for the graphs extracted from relational databases:** From the presented study we can notice that the graph model based on simple graph model (like (GOOD [Gyssens 1990a], GMOD [Andries 1992], etc.) is unsuited for heterogeneous graph with complex objects. For instance, in these models the attributes are presented as separated nodes which complicates the visualization and the treatment of these graphs. The graph model based on complex nodes can resolve these problems. However, these models do not offer explicit labeled edges. They use hyperedges or node encapsulation. In the relational databases, objects are related by the mean of foreign keys. Thus, the graph model used for modeling the extracted graph should maintain this information.

-**Absence of an interaction graph extraction approach from relational database Transformation:** The proposed approaches to transform a relational database to a graph like model RDF or XML or even ER only translate the relational model to a target model. Relational databases querying approaches facilitate the information search on relational database and permit to detect if there is a link between specific objects. However, all the previous techniques do not allow to identify similar objects or to discover hidden or new relations between these objects. For instance, with these approaches we cannot extract a social network.

-**Absence of a "simple" graph querying method containing advanced graph features and various graph models:** Many approaches are proposed in the literature allowing to manage graphs as described in section **??**. The graph query languages are the first tool that users use to query and extract information from graphs. The majority of the languages are based on specific syntax which needs advanced knowledge in computer science. In this context, visual query languages can be a good alternative for inexperienced users. However, the existing one treats specific data model (XML, RDF or Hypernode) or does not support advanced functionalities like path search or aggregation. Thus, business users need an efficient visual query language which is easy to manipulate, treat different graph types and support advanced features (aggregation and graph analysis) in order to better analyze their data.

In order to resolve these issues, we have proposed new approaches, which are presented in Figure 2.24:
- **An enterprise ontology learning approach :** which extracts a personalized enterprise ontology using a generic enterprise ontology and lexico-syntactic patterns from the enterprise unstructured data. The resulting ontology will bridge the gap between the structured and unstructured data. It will play the role of the enterprise concepts reference. This approach will be detailed in chapter 3.
- **An object interaction graph extraction approach :** which allows users to extract specific objects and their relations from relational database (or not) using the enterprise ontology. As a first step, the relational database is transformed to a graph model called the SPIDER-Graph, a new graph model adapted to this kind of graph. Then, the user can select the enterprise concepts that he would like to see their interactions. In this case, object identification and ontology relation enrichment processes are performed before the relation extraction step. The user can also select directly the desired objects from the input graph. In this case only a relation extraction process is performed. All this details will be detailed in the chapter 4.
- **A visual query language :** the proposed visual query language can query both extracted graph from relational databases and existing graph in the enterprise stored in RDF , XML or GraphML files. This language covers different query types from simple selection query to social network analysis queries. It is based on a pattern matching process to extract sub-graphs from input graphs. This language is detailed in the chapter 5.

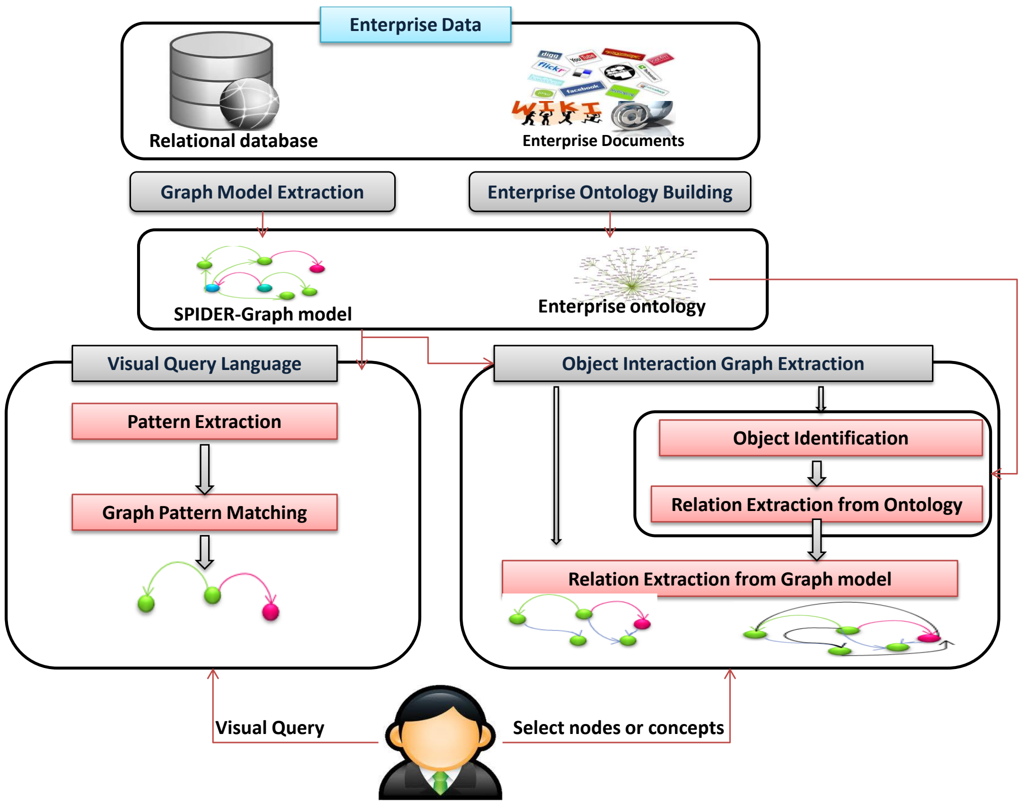


Figure 2.24: The Proposed Approach.

Chapter 3

The Enterprise Ontology Learning Approach

Contents

Enterprises daily manipulate unstructured data (documents, emails, web pages, contacts, blog posts) as well as structured data. These different data types are complementary. The description of an object, person or process can be disseminated in several sources with several structures. For example, the description of a project (start-date, budget, people and etc.) can be stored in a database and the analysis of its results and other characteristics can be detailed in a document.
Using a unified graph model can facilitate querying and integrating heterogeneous data sources. In order to extract graphs from relational databases and unstructured data, we need a reference of all the enterprise objects and process. Enterprise ontology has been used for describing enterprise objects and processes in order to represent a common semantic layer of enterprise resources. It can be a good reference of all the existing objects and processes of a particular enterprise. Furthermore, it can describe the characteristics of each enterprise element, their relations and their various names and connotations.
The predefined concepts and relations in the enterprise ontology can help to extract from unstructured data a graph model. In this context, we have defined an approach for enterprise ontology learning coping with both generic and specific aspects of enterprise information. Our approach is based on two main steps. First, general enterprise ontology is semi-automatically built in order to represent general aspects (core ontology); Then, an ontology learning method from enterprise unstructured

data is applied to enrich and populate this latter with specific aspects.
The resulting enterprise ontology, which represents a graph model of the enterprise unstructured data, will be used to extract graph from relation databases and to enrich them.
We start by presenting the basic concepts related to the ontology domain and the related work on ontology learning. After this, we detail the different aspect of our approach.

3.1 Theoretical Basis

3.1.1 Ontology Definition

The notion of ontology is old and stems from philosophy. Ontologies have been widely used in various domains like artificial intelligence, knowledge engineering and management, information retrieval etc. The importance of ontologies has reemerged with the proposal of semantic web [Berners-lee 2001] by Tim Berners Lee.
Ontology has been defined using different definitions as it was presented in [Guarino 1998]. The most used definition comes from Gruber [Gruber 1993] where an ontology is described as an "explicit specification of a conceptualization". In this definition, *Conceptualization* refers to an abstract model of an object in the world by having identified the relevant concepts referring to it. *Explicit* means that the type of concepts used, and their constraints of use, are explicitly defined. The definition by Gruber was later developed further by Studer *et al.* [Studer 1998] who stated that "an ontology is a formal, explicit specification of a shared conceptualization", which means that an ontology should also be shared within some group of people or agents within a domain.

In addition, there is a variety of other ontological definitions, which share the following set of components:

- *Concepts* of a domain are abstract or concrete entities derived from specific instances or occurrences.
- *Attributes* are characteristics of the concepts which may or may not be concepts by themselves.
- *Taxonomy* provides hierarchical relations between the concepts.
- *Non-taxonomic Relations* specify non-hierarchical semantic relationships between the concepts.
- *Instances* are used to represent elements or individuals in ontology.

More formally, an ontology is defined as follows:

Definition 13 (Ontology) *An ontology is defined by the set* $O := \{C, H^c, OP, DP, A\}$ *where*

- *C is the set of concepts where $C := c\,|c =< c_l, I >$ where c_l is the c label and I is its set of instances.*
- *H^c is the set of directed relations $H^c \subset C$ which is called concept hierarchy or taxonomy.$H^c(C_1, C_2)$ means that C_1 is a subconcept of C_2.*
- *OP is the set of object properties $op \in OP$ with $op := \langle n, c_1, c_2 \rangle$ where n is the name of the object property, c_1 , $c_2 \in C$ with c_1 is the range of op and c_2 is the domain of op.*
- *DP is the set of datatype properties $dp \in DP$ then $dp := \langle n, c_1, t \rangle$ where n is the name of the datatype property, $c_1 \in C$ with c_1 is the range of op and t is the type of the relation.*
- *A is the set of axioms.*

Several categorizations of ontologies have been defined [Guarino 1998]according to their level of dependence of a particular task or a point of view [Gomez-Perez 2004] (see Figure 3.1):

- Top-level ontology or upper-level ontology: is a generic ontology and domain independent. It describes very general concepts
- Domain ontology: describes the vocabulary related to a generic domain by specializing the concepts introduced in the top-level ontology. e.g. medicine, agriculture, politics, etc.
- Task ontology: describes the vocabulary related to a generic task or activity by specializing the top-level ontologies.
- Application ontology: concepts in application ontologies often correspond to roles played by domain entities.
- Domain-task ontology : defines domain-level ontologies of domain-specific tasks and activities.
- Method ontology: gives definitions of the relevant concepts and relations applied to specify a reasoning process, so as to achieve a particular task.

The following section focus on ontologies building methods and tools.

3.1.2 Ontology Building

Ontology building is a process that aims to produce ontology. It is a complex process covering the complete life-cycle of ontologies, from requirements engineering to usage and maintenance of the ontologies. The ontology life-cycle can be summarized by these principal steps [Suarez-Figueroa 2008]:

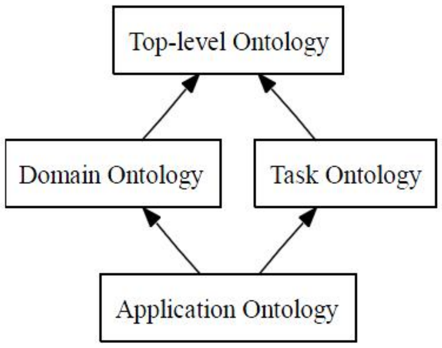


Figure 3.1: Guarino's types of ontologies.

- **Specification:** identify the purpose and scope of the ontology. The purpose answers the question: "Why is the ontology being built?" and the scope answers the question: "What are its intended uses and end users?"

- **Conceptualization:** describe, in a conceptual model, the ontology to be built. The conceptual model of an ontology consists of concepts in the domain and relationships among those concepts.

- **Formalization:** transform the conceptual description into a formal model, that is, the description of the domain found in the previous step is written in a more formal form, although not yet its final form. Concepts are usually defined through axioms that restrict the possible interpretations for the meaning of those concepts. Concepts are usually hierarchically organized through a structuring relation, such as *is-a* (class-superclass, instance-class) or *part-of*.

- **Implementation:** implement the formalized ontology in a knowledge representation language. For that, one commits to representation ontology, chooses a representation language and writes the formal model in the representation language using the representation ontology.

- **Maintenance:** update and correct the implemented ontology.

There are also other activities that should be performed during the entire life-cycle:

- **Knowledge acquisition:** acquire knowledge about the subject either by using elicitation techniques on domain experts or by referring to relevant bibliography. Several techniques can be used to acquire knowledge, such as brainstorming, interviews, questionnaires, text analysis, and inductive techniques.

- **Evaluation:** judge the quality of the ontology technically.

- **Documentation:** report what was done, how it was done and why it was done.

This life-cycle plays the role of guideline for building ontology. The existing tools and methodology can modify the ontology life-cycle or ignore some steps. The existing ontology building approaches can be divided in two categories: Manual approaches and semi-automatic or automatic, called learning approaches. These different approaches are described below.

3.1.2.1 Manual Building

In the literature, many methodologies have been proposed in order to guide the manual ontology building or the building from scratch. These methodologies are guidelines describe each step of the building process. A comparative and detailed study of these methods and methodologies can be found in Jones *et al.* [Jones 1998], Fernandz Lopez [Lopez 1999], Ohgren [Ohgren 2005] and Suarz-Figueroa *et al.* [Suarez-Figueroa 2007]. Some examples of these methods are described in order to show how an ontology building methodology works.
[Lenat 1989] presented a methodology based on three phases. The first phase consists in the manual codification of articles and pieces of knowledge in which common sense knowledge, that is implicit in different sources, is extracted by hand. The second and third phases consist in acquiring new common sense knowledge using natural language or machine learning tools. The difference between them is that in the second phase this common sense knowledge acquisition is aided by tools, but mainly performed by humans, while in the third phase the acquisition is mainly performed by tools.
METHONTOLOGY [Gomez-Perez 1996] is a more general methodology used to build ontologies. They are either built from scratch, reusing other ontologies as they are, or by a process of reengineering. The METHONTOLOGY framework enables the construction of ontologies at knowledge level. The first step is to specify the purpose of the ontology, the level of formality and the scope. Next, all knowledge needs to be collected. Then, a conceptualization phase is performed. In this step, Fernandz *et al.* [Lopez 1999] first propose to build a glossary of terms with all possibly useful knowledge in the given domain. Terms are grouped according to concepts and verbs, and these are gathered together to form tables of formulas and rules. The next task to perform is to check whether there are any already existing ontologies that can be used. After that, the ontology is implemented using a formal language, that can be evaluated according to some references. The final part is the documentation.
A more recent methodology is proposed by Noy and McGuinness [Noy 2001]. Their methodology is iterative, starting with a rough concept and then revising and filling in the details. The first step consists of determining the domain and the scope of the ontology. It is important to think of whether to use already existing ontologies, and if so, how to use them. A list of all the terms that could be needed or used is then produced. The class hierarchy should represent an "is-a" relation, cycles should be avoided, siblings should have the same level of generality, multiple inheritance could lead to some problems; guidelines regarding when to introduce

new classes or instances are given. When the classes are defined, i.e. the terms and the relations, then the properties of the classes need to be specified (attributes). Here it is important to check whether some relations are inverse or not, and whether a default value for an attribute could be useful. After this the value type of both classes and class properties are defined, including cardinality, domain and range. Finally, the individual instances are created.
In the context of the Enterprise project, Uschold and King [Uschold 1995] proposed a methodology to build a top level enterprise ontology. Their methodology is based on four phases: (1) Identify the purpose of the ontology, (2) build it, (3) evaluate it, and (4) document it. During the building step, the authors propose capturing knowledge, coding it and integrating other ontologies inside the current one. They also propose three strategies for identifying the main concepts in the ontology: A top-down approach, in which the most abstract concepts are identified first, and then, specialized into more specific concepts; a bottom-up approach, in which the most specific concepts are identified first and then generalized into more abstract concepts; and finally they propose a middle-out approach, in which the most important concepts are identified first, then generalized and specialized into other concepts.

The manual approach can be useful to model relevant ontologies. However, the manual approaches are time-consuming and not well adapted to construct various ontologies for various domains/applications.

3.1.2.2 Ontology Learning

The term ontology learning was originally coined by Alexander Madche and Steffen Staab [Maedche 2001] and can be described as the acquisition of a domain model from data. The term ontology learning refers to the automatic or semi-automatic support for the construction of ontology [Buitelaar 2005]. Ontology learning uses different data sources to learn the concepts relevant for a given domain, their definitions as well as the relations holding between them. The used data source can be:

- Structured data such as database schema, XML-DTDs and UML diagrams. In this case, ontology learning mainly consists in mapping definitions from the schema to corresponding ontological definitions.
- Semi-structured data such as dictionaries like WordNet [Miller 1995], XML or HTML documents or tabular structures.
- Unstructured data: natural language text documents, like the majority of the HTML based web pages; for extracting knowledge from these sources, statistical and linguistic approaches are often used.

In this work, the focus is on ontology learning from unstructured data. According to Maedche [Maedche 2002] and Cimiano [Cimiano 2006] the field of ontology

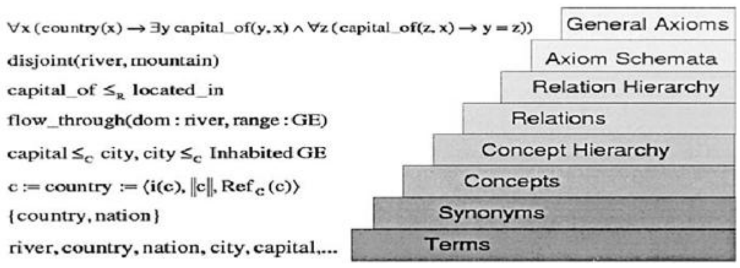

$\forall x\,(\text{country}(x) \rightarrow \exists y\ \text{capital\_of}(y,x) \wedge \forall z\,(\text{capital\_of}(z,x) \rightarrow y = z))$
General Axioms
disjoint(river, mountain)
Axiom Schemata
$\text{capital\_of} \leq_R \text{located\_in}$
Relation Hierarchy
flow_through(dom : river, range : GE)
Relations
$\text{capital} \leq_C \text{city}, \text{city} \leq_C \text{Inhabited GE}$
Concept Hierarchy
$c := \text{country} := \langle i(c), \|c\|, \text{Ref}_C(c) \rangle$
Concepts
{country, nation}
Synonyms
river, country, nation, city, capital,...
Terms

Figure 3.2: Ontology "Layer Cake".

learning from text can be divided into a "layer cake" of methods and algorithms. The layer shows the different subtasks of learning ontology: (see figure 3.2)

- Acquisition of the relevant terminology,
- Identification of synonym terms/linguistic variants (possibly across languages),
- Formation of concepts,
- Hierarchical organization of the concepts (concept hierarchy),
- Learning relations, properties or attributes, together with the appropriate domain and range,
- Hierarchical organization of the relations (relation hierarchy),
- Instantiation of axiom schemata,
- Definition of arbitrary axioms.

The ontology learning process needs techniques from other domains to realize the tasks of the "layer cake". Techniques from Natural Language Processing, computational linguistics, and text mining are used for extracting different elements from the text. The ontology learning from text techniques can be classified as presented in [Gomez-Perez 2003]:

- **Pattern-based extraction** [Morin 1999] [Hearst 1992]. These approaches are based on the use of templates or patterns to extract various ontology elements. Patterns are in the form of regular expressions. [Hearst 1992] is a primary work on pattern based extraction. In her work, she uses six lexico-syntactic patterns in order to extract hyponymy/ hyperonymy relations from English texts.

- **Association rules** were initially defined by [Agrawal 1994]. The association rules method for ontology learning have been originally described and evaluated in [Maedche 2000]. They have also been used to discover non-taxonomic relations between concepts, using a concept hierarchy as background knowledge [Maedche 2001].

- **Conceptual clustering** [Faure 2000]. Concepts are grouped according to the semantic distance between each other to make up hierarchies. Formal Concept Analysis (FCA) can be seen as a conceptual clustering technique as it also provides intentional descriptions for the abstract concepts or data units it produces. The FCA has been used to learn new concepts [Cimiano 2005b] [Coulet 2008].

- **Ontology pruning** [Kietz 2000]. The objective of ontology pruning is to build a domain ontology based on different heterogeneous sources. It is based on the following steps: Firstly, generic core ontology is used as a top level structure for the domain-specific ontology. Secondly, a dictionary which contains important domain terms described in natural language is used to acquire domain concepts. These concepts are classified based on the generic core ontology. Thirdly, domain-specific and general corpora of texts are used to remove concepts that were not domain specific. Concept removal follows the heuristic that domain-specific concepts should be more frequent in a domain-specific corpus than in generic texts.

- Concept learning [Hahn 2000]. A given taxonomy is incrementally updated when new concepts are acquired from texts.

These approaches have been used separately or combined in many ontology learning frameworks. In the literature, several tools have been proposed to learn ontology using the previous approaches. An overview is proposed in [Shamsfard 2003]. Some examples are cited below:

- The Mo'K workbench [Bisson 2000], basically relies on unsupervised machine learning methods to induce concept hierarchies from text collections. In particular, the framework focuses on agglomerative clustering techniques and allows ontology engineers to easily experiment with different parameters.

- The ASium [Faure 2000] framework learns verb frames and taxonomic knowledge, based on statistical analysis of syntactic parsing of French texts. It uses a conceptual clustering algorithm to generate the structure of the ontology.

- OntoLT [Buitelaar 2004] is an ontology learning plug-in for the Proté gé ontology editor. It is targeted more at end users and heavily relies on linguistic analysis. It basically makes use of the internal structure of noun phrases to derive ontological knowledge from texts.

- The framework by Velardi *et al.*, OntoLearn [Navigli 2004], mainly focuses on the problem of word sense disambiguation, i.e. of finding the correct sense of a word with respect to a general ontology or lexical database. In particular, they present a novel algorithm called SSI relying on the structure of the general ontology for this purpose. Furthermore, they include an explanation component for users consisting of a gloss generation component which generates definitions for concepts, which were found relevant in a certain domain.

- TEXT-TO-ONTO [Maedche 2001] is an ontology learning environment, based on a general architecture for discovering conceptual structures and engineering ontologies from text. It also supports the acquisition of conceptual structures as mapping linguistic resources to the acquired structures. It makes an environment to discover conceptual relations to build ontologies. The new version Text2Onto [Cimiano 2005a] which supports learning ontologies from web documents, allows the import of semi-structured and structured data as input as well as texts. It also has a library of learning methods which use each one on demand. Their learning method is a multi-strategy method, combining different methods, for various inputs and tasks.

The task of ontology learning is in general followed by an ontology population task which allows the automatic or semi-automatic instantiation of a given ontology. This task is presented in the next section.

3.1.3 Ontology Population

Ontology population can be seen as an ontology enrichment process that inserts concepts instances and relations instances into an existing ontology. The process of ontology population does not change the structure of ontology (as the concept hierarchy and non-taxonomic relations are not modified). As mentioned in [Cimiano 2006], ontology population is related to the named entity recognition (NER) and information extraction tasks. The existing approaches of ontology population can be divided in two categories:

- **Approaches based on natural languages processing (NLP) techniques:** there are a number of approaches that use NLP-based techniques for ontology population. The first category are pattern-based approaches relying on Hearst patterns [Hearst 1992] [Schlobach 2004] [Zouaq 2009] [Etzioni 2004] or on the structure of words [Velardi 2005]. These approaches try to find explicitly stated "is-a" relationships.
  Other linguistic approaches are based on the definition or the acquisition of rules. For example, Amardeilh [Amardeilh 2005] proposes a rules acquisition approach that uses linguistic tags on text. These tags are mapped to concepts, attributes and relationships from the ontology and enable to find their instances.

- **Statistical and machine learning techniques:** these approaches can be divided into supervised and weakly supervised approaches [Tanev 2006]. Among the weakly supervised approaches, Cimiano proposed an approach in [Cimiano 2005a] which uses a vector-feature similarity between each concept c and a term to be categorized. Cimiano and Volker evaluated different context features (word windows, dependencies) and proved that syntactic features

work best. Their algorithm assigned a concept to a given instance by computing the similarity of this instance feature vector and the concept feature vector. In [Tanev 2006], they used syntactic features extracted from dependency parse trees. This algorithm requires only a list of terms for each class under consideration as training data.
Supervised approaches for ontology population reach higher accuracy. However, they require the manual construction of a training set, which is not scalable [Tanev 2006]. An example of a supervised approach is the work of [Fleischman 2001] [Fleischman 2004] who designed a machine learning algorithm for fine-grained Named Entity categorization. Web→KB [Craven 2000] relies also on a set of training data, which consists of annotated regions of hypertext that represent instances of classes and relations, in order to extract named entities. Based on the ontology and the training data, the system learns to classify arbitrary Web pages and hyperlink paths.

3.1.4 Enterprise Ontology

Enterprise ontology is an ontology used to describe the domain, or parts of the domain of an enterprise [Blomqvist 2007]. It was defined in [Uschold 1995] as a top level ontology containing basic concepts, that concern enterprises and their activities, such as Project, Strategy and People. Even though enterprise ontology has been used in many areas of research [Maedche 2003] [Billig 2008] the number of existing enterprise ontology is too restricted. As a first example, we can mention the TOVE ontologies which were developed as a part of the TOVE Enterprise Modeling project [Fox 1995] and built from scratch. The goal of the TOVE project was to develop a set of integrated ontologies for the modeling of both commercial and public enterprises. The following ontologies have been developed to model Enterprises:

- Foundational Ontologies:
  - Activity
  - Resource
- Business Ontologies
  - Organization
  - Product and Requirements
  - ISO9000 Quality
  - Activity-Based Costing

The organization ontology describes the set of constraints on the activities performed by agents. In this ontology, the concepts are grouped into thematic sections. The ontology contains generic concepts, like time, causality, activity and constraint. For each concept, properties and relations are also defined. The concepts are structured into taxonomies and are represented by constants and variables, while the attributes and relations are represented with predicates, giving rise to micro-theories.

Another example of enterprise ontology is The Enterprise Ontology (EO) [Uschold 1995] which was developed manually to support and enable communication between different people, people and computational systems, and among different computational systems. The proposed enterprise ontology is presented in [Uschold 1998]. This ontology is based on the following main content:

- Meta-Ontology: Terms used to define the terms of the Ontology. e.g. Entity, Relationship, Role.
- Activity, Plan, Capability and Resource: Terms related to the process and planning .e.g. Activity, Planning, Authority, Resource Allocation.
- Organization: Terms related to how Organizations are structured e.g. Person, Legal Entity, Organizational Unit, Manage, Ownership.
- Strategy: Terms related to high level planning for an enterprise e.g. Purpose, Mission, Decision, Critical Success Factor.
- Marketing: Terms related to marketing and selling goods and services Sale, Customer, Price, Brand, Promotion.
- Time: Terms related to Time e.g. Duration, Date.

In the SEMCO Project [Blomqvist 2006], a semi-automatically method was proposed for building enterprise ontology using ontology design patterns with the aim of the creation of enterprise ontologies in small-scale application contexts. The resulting ontology was an enterprise ontology which was build for a specific enterprise and a specific application.

The main purpose of enterprise ontology is to promote the common understanding between people across enterprises, as well as to serve as a communication medium between people and applications, and between different applications. However, in this thesis, the main object of creating the enterprise ontology is to collect the characteristics of objects and processes of a specific organization. The role of this ontology is to allow other applications to identify of a specific enterprise object using its characteristics and different names used in this enterprise. The enterprise ontology used here should not just be a top level ontology (like Tove ontology) or merely describe a specific task like in [Blomqvist 2006], it should contain each existing detail of all the enterprise concepts to facilitate its detection. In the next section, we describe the proposed enterprise ontology building approach.

3.2 Building the Enterprise Ontology

The general purpose of building the enterprise ontology is to model the unstructured data as graphs. This ontology should model all the enterprise concepts, which define the different objects and process, and their shared relations. Also, for a specific object, the ontology should define all its characteristics, synonyms and its different names used at the enterprise. The built ontology represents the reference of the objects and their characteristics in an enterprise.
In the enterprise context, there are two types of concepts: General concepts of the business domain and specific concepts and terminology related to a particular enterprise. We propose an enterprise ontology learning approach which takes general ontology as input and adapts it to specific enterprise.

The proposed approach is based on two main phases (see Figure 3.3):

1. Building the generic part of the ontology: Generic ontology is minimal enterprise ontology built manually using existing enterprise models and patterns;

2. Building the specific part of the ontology: Specific enterprise ontology is learned and populated using the enterprise documents (web sites, wiki, emails and etc.) and the generic ontology.

In what follows, we describe the different steps of the our approach. First, we present the manual building of the top level part of the ontology. Then, the learning process, which enriches the generic ontology by learning new concepts and relations using the enterprise documents, is presented. In order to obtain the final enterprise ontology, a population method is additionally performed using the enterprise documents.

3.2.1 Building the Generic Part of the Ontology

The generic part of the enterprise ontology aims to outline the common processes and elements that enterprises share. For example, all the organizations have employees, products and customers and a production process as activity. In the literature, several existing resources can be useful to build the generic ontology. Before describing the approach of building, the used resources are shown.

3.2.1.1 The Used Resources

In order to collect the common concepts of enterprises, we have used several existing resources: Enterprise ontologies and enterprise data models. Below, the resourced are described in detail:
Existing Enterprise Ontologies. as it mentioned in the previous section some enterprise ontologies have been designed in the literature: The Enterprise Ontology EO [Uschold 1995] and the TOVE ontology [Gruninger 1995]. Uschold presents a unified methodology by combining the "best" parts of EO and TOVE into a unified

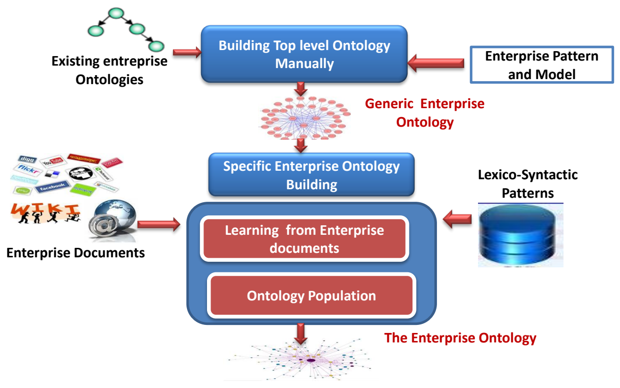

Existing entreprise Ontologies
Building Top level Ontology Manually
Enterprise Pattern and Model
Generic Enterprise Ontology
Specific Enterprise Ontology Building
Lexico-Syntactic Patterns
Enterprise Documents
Learning from Enterprise documents
Ontology Population
The Enterprise Ontology

Figure 3.3: Enterprise Ontology Building Approach.

method. The resulting enterprise ontology is presented in [Uschold 1998]. An expert of this ontology is presented in the Figure 3.4.

We have used this latter ontology as one of the input resources.

Enterprise Data Model: An Enterprise Data Model is an integrated view of the data produced and consumed across an enterprise. It can describe in a formal and a generic way the enterprise function and active objects. In general, it is designed to benefit many different industries and enterprises, representing common data constructs that appear in most organizations. Indeed, this data model can be a good starting point to extract the generic concepts to enter into the enterprise ontology.
The data model pattern proposed by [Silverston 2001] have also been used. These patterns are designed for the database domain. The database schemas share many properties with ontologies. Schemas can even be considered to be ontologies, from which to extend data instances. Furthermore, it facilitates the use of this pattern to build ontology. The Silverston data model models different part of an enterprise and describes in details each element in this part, i.e. the attributes and the relations of the element are detailed. This data model is based on the following main contents:

- People and organizations: Elements related to how Organizations are structured and how people are modeled, how they communicate within an organization e.g. Person, Party roles, Party relationships, Party contact, etc.
- Products: Elements related to common product information e.g. Product

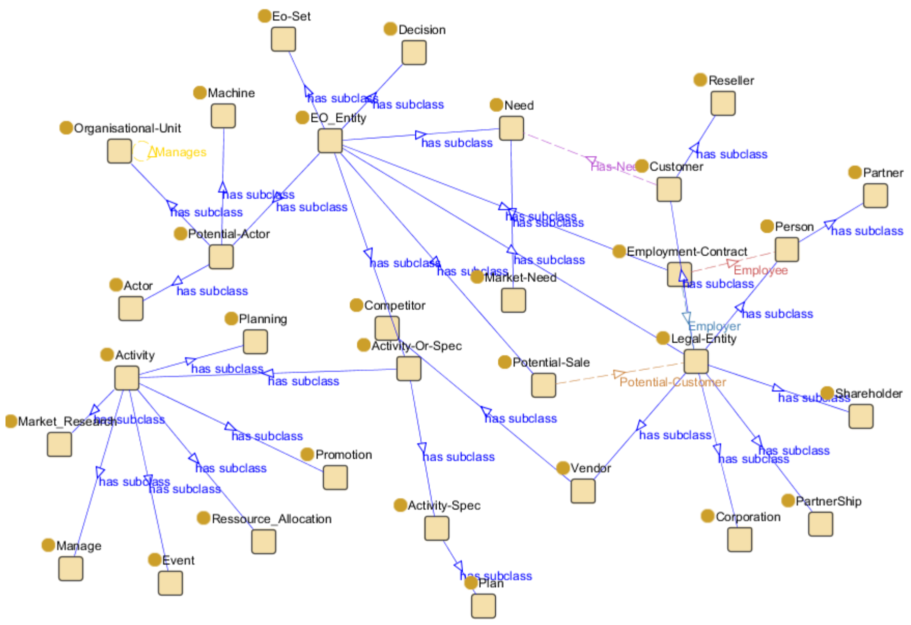


Figure 3.4: The Uschold Ontology.

definition, Product category, Suppliers and manufacturers, etc.

- Ordering Products: Elements related how an enterprise obtains a products e.g. Requirements, Requests, Quotes, Agreements, etc.
- Shipments: Elements deal with the shipment of items that are scheduled to be or have been delivered e.g. Shipment document, Shipment method, etc.
- Work effort: Elements deal with the efforts made within organizations to accomplish tasks such as completing a project or producing e.g. work requirements, fixed asset assignment and etc.
- Invoicing: Elements related to invoices flow e.g. Invoices roles, billing for order items and billing for shipment items, etc.
- Accounting and Budgeting: Related to financial information e.g. Budget, Accounts and etc.
- Human resources: Contains elements related to the employment process within an organization e.g. Employment, Position, Employee and etc.

In this data model an organization (see Figure 3.5) is modeled via an entity called *Organization* that stores information about a group of people with common purpose such as corporation, department, division, government agency or nonprofit organization. An organization is designed by its name and its federal tax. Other information, which can change on time, is putted in other entities; for instance, the address is putted in the entity Postal Address Information. An organization may

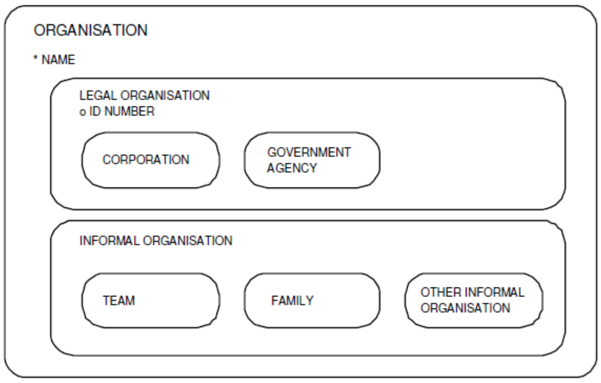


Figure 3.5: Data Model Describing Organization [Silverston 2001].

be Legal organization such as a Corporation or Government Agency, or an informal organization, such as Family, Team or other informal organization.

This data model, which is based on quite similar foundations as the entity relationship (ER) model, is translated into the ontology formalism. By transforming entity to a concept, attribute to attribute on concept, relationship to relations and so on.

These different resources represent the input for the manual ontology building process which is described in the next section.

3.2.1.2 The Building Approach

In order to build our minimal enterprise ontology from scratch, we have used a methodology similar to the one proposed by [Noy 2001] (see section 3.1.2.1). To begin with, the common concepts from the different resources have been regrouped. Secondly, the relations between the concepts were defined. In this last step, the hierarchical relation was defined: Defining the synonyms, the super class, and sub class. Moreover, resolving contradictory relations, the other existing relations were sorted to avoid cycles and multiple inheritance.

The third step consisted of specifying the concept properties by merging existing ones and adding new ones. For this generic part of the ontology, individual instances were not added.

We take the example of the concept "Person". In the enterprise data model, this entity is a sub-concept of the concept "Party" and has the relation "Acting_as" with the concept "Person_Role". In the Ushold ontology, "Person" is a sub-concept of the concept "Legal_Entity". First, we regroup the two concepts having the same label "Person". Secondly, the "Person" concept has two super class "Legal_Entity" and "Party". In order to resolve this multi inheritance problem, we analyze the other concepts. We found that "Legal_Entity" is a synonym of "Legal_Organization" which is sub-concept of "Organization". Then, we put "Party" as super-class of

"Person" and we add "Legal_Entity" as synonym to "Legal_Organization". The other relations are added automatically and we verify in each time if there is a conflict of synonyms or multiple inheritance. Finally, the "Person" attributes found in the enterprise data model and the Ushold enterprise ontology are merged.

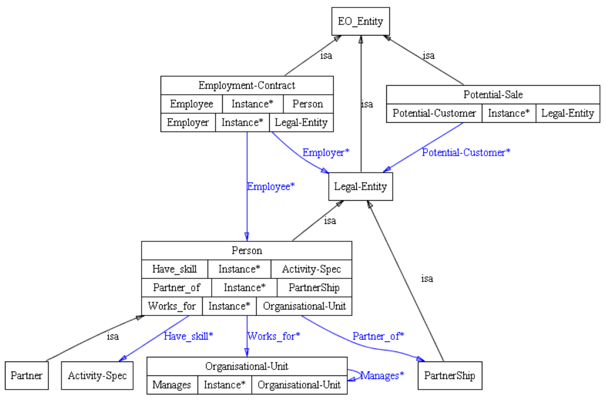

EO_Entity
isa
isa
isa
Employment-Contract
Employee | Instance* | Person
Employer | Instance* | Legal-Entity
Potential-Sale
Potential-Customer | Instance* | Legal-Entity
Employer*
Potential-Customer*
Employee*
Legal-Entity
isa
isa
Person
Have_skill | Instance* | Activity-Spec
Partner_of | Instance* | PartnerShip
Works_for | Instance* | Organisational-Unit
isa
Have_skill*
Works_for*
Partner_of*
Partner
Activity-Spec
Organisational-Unit
Manages | Instance* | Organisational-Unit
Manages*
PartnerShip

Figure 3.6: The "Legal_Entity" Concept in the Ushold Ontology.

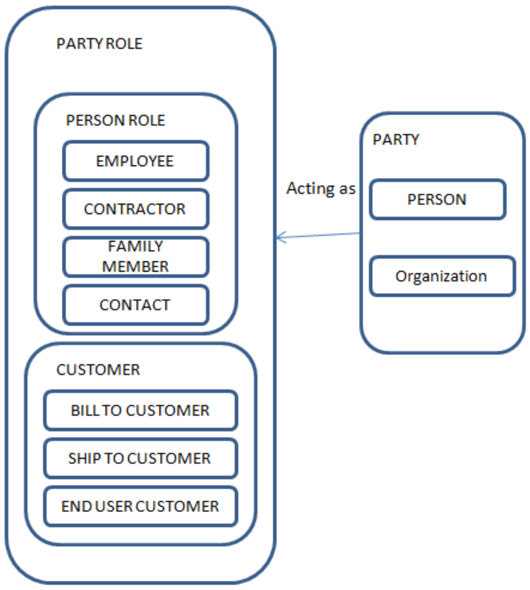


Figure 3.7: The Entity "Person".

The resulting part containing the Person concept is presented in the Figure 3.8.

The resulting ontology contains 90 concepts about the enterprise actors (Person, Team and etc), their role (Employee, Customer, Agent and etc), the production (Product, Requirement, Order , Price, etc), the communication and the work effort.

3.2.2 Learning and Populating the Specific Enterprise Ontology Using the Enterprise Data

In the previous section, we have built a generic enterprise ontology containing the common concepts of the business domain. Apart from the common concepts, an enterprise can use local terminology to describe e.g the employees or products. In order to collect this local information and make the generic ontology adapted to the specific enterprise, we have used a learning process to add this specific part using the enterprise documents such as web sites, wikis and e-mails. The learning process is followed by a population process using the same document in order to add new instances to the ontology. In what follow, we detail these steps to create the specific enterprise ontology.

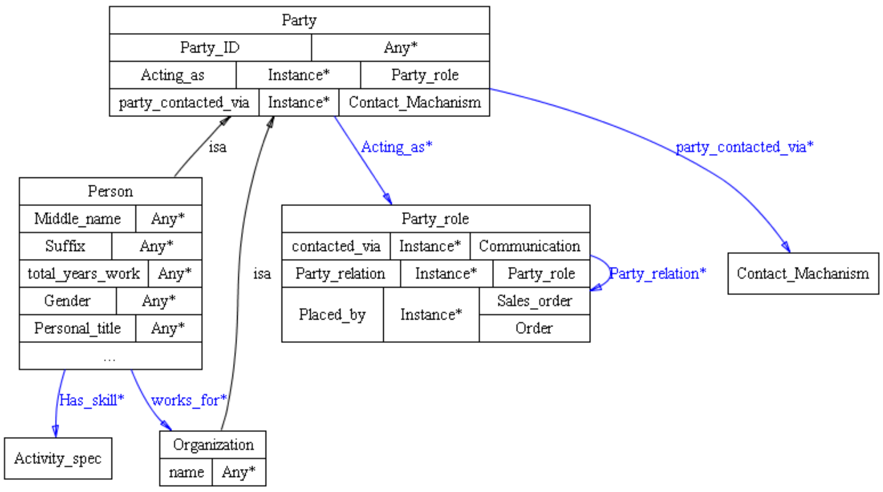


Figure 3.8: The Person concept in the generic enterprise ontology.

3.2.2.1 Phase1: Documents Treatment

In order to facilitate the learning and populating process, the documents were first processed using linguistic method. Linguistic preprocessing starts by the tokenization process, which divides the text into tokens, and a sentence splitting process, which divides the text into sentences.
The resulting annotation set serves as an input for a Part-of-speech (POS) tagger which assigns appropriate syntactic categories to all tokens. Then, lemmatizing or stemming process is done by a morphological analyzer or a stemmer, respectively. After that, the sentences are divided into noun phrase chunks using a noun phrase NP chunker.
After the linguistic preprocessing of the documents, a semantic analysis is performed which consists in identifying the named entity and the concepts of the generic ontology. Named entity recognition process allows to locate and to classify atomic elements in text into predefined categories such as the names of persons, organizations, locations, expressions of times, quantities, monetary values, percentages, etc. In the input documents, we are interested to identify named entities related to cities, organizations, person names, enterprise names and software products names. These named entities will be used to identify new instances in the population process. In this context, we have used different gazetteers, which are sets of lists containing names of entities, in order to detect named entities related to cities, organizations, person names, enterprise names and software products names.
After that, we annotate the documents using the input ontology. The existing concepts of the treated documents are searched. This different treatment performed on the input documents produce the annotation which are the inputs on the learning and the population algorithms.

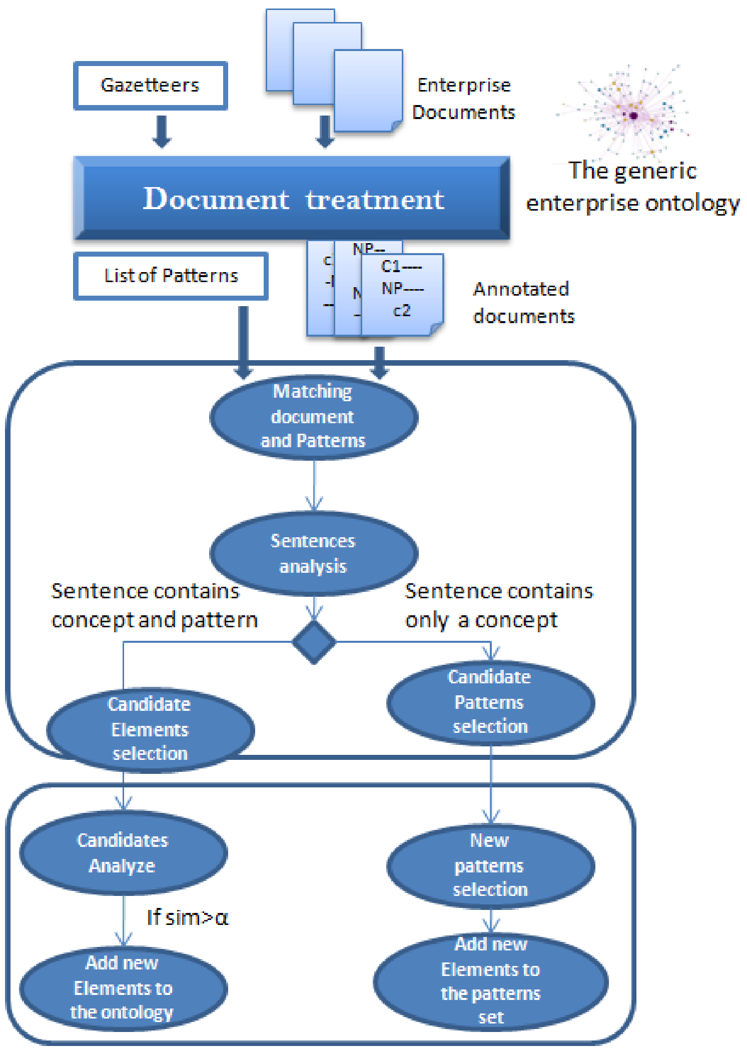

Gazetteers
Enterprise Documents
The generic enterprise ontology
Document treatment
List of Patterns
C1----
NP----
c2
Annotated documents
Matching document and Patterns
Sentences analysis
Sentence contains concept and pattern
Sentence contains only a concept
Candidate Elements selection
Candidate Patterns selection
Candidates Analyze
If sim>α
Add new Elements to the ontology
New patterns selection
Add new Elements to the patterns set

Figure 3.9: The Learning Process.

3.2.2.2 Phase 2: Learning Process

This learning process aims to add new elements (concepts, relations and properties) to the generic enterprise ontology. It is based on two steps: (1) Detection of candidate elements by using lexico-syntaxic patterns and (2) filtering the candidate elements by using similarities measures.
The learning process is described in figure 3.9. The steps are detailed in the following.
A. Candidate concepts, attributes and relations detection. The document processing step has allowed to annotate the input documents with named entities and the generic ontology concepts. Using the annotated documents, a candidate detection algorithm is performed. First, the algorithm matches each document with the set of predefined lexico-syntaxic patterns. Then, each document is analyzed sen-

tence by sentence. For each sentence containing a known concept or named entity, a new element can be annotated as a candidate. The type of new element, which can be concept, relation or attribute, depends on the detected pattern. In this context, different sets of patterns were identified, which are classified by the type of element that it can detect. In the following, *NPC* designs the noun phrase containing a concept from the initial ontology and *NP* is another noun phrase which contains a candidate element.

Pattern for Taxonomic relations. These patterns are used to extract new concepts which share a hyponymy or a meronymy relation with an existing concept. They include the Hearst patterns and the Lexico-Syntactic Patterns taken from the Ontology Design Patterns. Hearst defined six patterns which can be reduced to the set in Table 3.1.

Table 3.1: Hearst Patterns

Pattern	**Detected Relation**	**Example**
NPC *such as* {NP,}* {*and*|*or*} NP	hyponym(NPC, NP)	Online business, such as web design and programming⇒ hyponym(Business,web design), hyponym(business, programming).
Such NPC *as* {NP,}* {*and*|*or*} NP	hyponym(NPC, NP)	such good as fourniture or equipment ⇒ hyponym(good, fourniture), hyponym(good, equipment).
NP {NP,}* {,} {*and*|*or*} other NPC	hyponym(NPC, NP)	Engineers, technicians and other IT Employee ⇒ hyponym(Employee, Engineers)
NPC {,} (*including*|*especially*) {NP,}* {*and*|*or*} NP	hyponym(NPC, NP)	Employee including CEO or Manager ⇒ hypnym(Employee,CEO), hyponym(Employee, Manager).

The second type of patterns investigated was the lexico-syntactic patterns corresponding to Ontology Design Patterns. For example the: *NP be (the same as\|equivalent to\|equal to\|like) NPC*. If we detect one of these patterns containing one of the generic ontology concepts (here designed by *NPC*), the *NP* will be added to the candidate concept set.

For instance in the sentence "*Other forms of company, such as the cooperative*", we detect the Hearst pattern "$NPC\ suchas\ \{NP,\} * \{and|or\}\ NP$", and the concept "*Company*". In this case the name "*Cooperative*" is added to the candidate concepts set.

Pattern for non hierarchical relation. The verbal sentences can reveal semantic relations between two terms. In the sentence "Organization provides services", the verb "provides" indicates the type of the relation between an organization and a service. Based on this kind of observation, Cimiano [Cimiano 2006] defined the verbal patterns. The two verbal patterns are resumed in Table 3.2, where P represents a proposition. In the previous sentence, if "*Organization*" is a concept in the generic enterprise ontology and using the verbal pattern we can detect the relation: *Provide (Organization, service)*. Then "*Service*" is a candidate concept to be added in the ontology with the relation *"Provide"*.

Table 3.2: Verbal Patterns

Pattern	**Detected Relation**	**Example**
NPC [P] V NP	[P] V (NPC,NP)	**Organization** provides **services.** ⇒ Provide(Organization, Services)
NP [P] V NPC	[P] V (NP,NPC)	*Products* are manufactured by an **enterprise**. ⇒ Manufactured by(Products, Enterprise)

Pattern for attributes. Attributes are defined as relations with a datatype as range. Typical attributes are, for example, name or color with a string as range, date with a date as range or size with an integer or real as range. For example, in the sentence "*Company has a capital*" by using the pattern "*NPC have NP*", we add capital as an attribute to the concept "*Company*". The attribute patterns are resumed in Table 3.3

Table 3.3: Attribute Patterns.

Pattern	**Detected Relation**
NPC have NP (ODP pattern)	Datatype :name=NP; Range=NPC
Adjective NPC	Datatype :name=Adjective;Range=NPC

B. Candidate elements processing. The previous patterns allow the identification of candidate elements to add to the enterprise ontology. Each candidate element is in the form of $CR := \prec t, ce, c, r \succ$ where:

- t: The type of relation which can be an object property or a data property depending on type of the pattern.
- *ce*: The candidate element, a new data property where t is a data property or a new concept where t is a new object property (obtained from the *NP* element).
- c: The existing concept of the ontology (obtained from the *NPC*).
- r: The name of the relation between the concept c and the candidate element *ce*. In the case of the detection with taxonomic pattern r="Is-A", verbal pattern r= verb and for the attribute r=" ".

For each candidate relation, the similarity $sim(ce, c)$ is computed between the concept c and the candidate element *ce*. For the similarity, we have used the WebOverlap measure [Bollegala 2007]which is based on the web co-occurrence. This measure exploits page counts returned by a Web search engine to measure the semantic similarity between words. WebOverlap is defined depending on the value of r. In the case of r="Is-A" or r="'"':

$$WebOverlap(c, ce) := \frac{hits(C\ and\ ce)}{Min(hits(c), htis(ce))}$$

where the notation *hits(P)* denotes the page counts for the query P in a search engine. If $sim(ce, c) \geq \alpha$ (where α is a threshold value) then *ce* is added to the ontology as synonym to c if r="IS-A" and as datatype of c if r="'"'.

In the case of r=verb,

$$WebOverlap(c, ce) := \frac{hits(C\ verb\ ce)}{Min(hits(c), htis(ce))}$$

If $sim(ce, c) \geq \alpha$ then a new concept *ce* is added to the ontology and a new object property having the name of the verb between the concepts c and *ce* is also added to the ontology.

We present an example of the candidate elements evaluation using the WebOverlap measure in the table 3.4 and using the Google results. By analyzing the obtained results, the similarity between "company" and "cooperative" is too high (more than 1) which reveal the importance of this relation. Then the relation *Hyponym(company, ccoperative)* is added to the ontology. When a new element appears in more than patterns, the pattern having the highest score is added to the ontology. In the example of "portfolio" the relation *Has(company, portfolio)* is the relation selected to add in the enterprise ontology.

Table 3.4: Example of the patterns evaluation.

Example	**Detected Pattern**	**Candidate relation**	**Candidate Evauation**

Other forms of company, such as cooperative	$NPCsuchas\{NP,\}*$ $\{and\|or\}$ NP	c=company
		
		ce=cooperative
			Hits(company)=9.070.000.000
			
			Hits(cooperative)=42.200.000
			Hits(company AND cooperative)=74.300.000
			Score=1,769

Company may create a portfolio	NPC [P] V NP	c=company
		
		ce=portfolio
			Hits(company)=9.070.000.000
			
			Hits(potfolio)=693,000,000
			Hits (company create portfolio)= 200.000.000
			score=0,288

Company has portfolio	NPC [P] V NP	c=company
		
		ce=portfolio
			Hits(company)=9.070.000.000
			
			Hits(potfolio)=693,000,000
			Hits (company has portfolio)= 473,000,000
			Score=0,682

C. New pattern identification Using the enterprise documents, the set of predefined lexico-syntactic patterns are sought to be enriched with new ones. These new patterns will be used in the population process. On the annotated document, the ontology concepts can be detected in many sentences which do not verify a pattern. Then, we use these sentences to discover new patterns in order to enhance the set of patterns. For each sentence containing one of the generic ontology concept and does not verifying an existing pattern:

1. The regular expression is extracted from the sentence.
2. The pattern is pruned and selected manually.
3. A type of relation is added to the identified pattern.

For example, from the text:
"the most common forms of company are:

- A company limited by guarantee
- A company limited by shares."'

We detect the new pattern: $NPbe : .NP*$

3.2.2.3 Phase3: Population Process

In the previous step, we have described the learning process which enriches the generic ontology with new concepts and relations. In order to build more specific enterprise ontology dedicated to a particular enterprise and useful in its local application, the learning concepts are enriched, from the enterprise document instances using an ontology population process. The population process contains three main steps: Document treatment, Instance identification and Instance Treatment.
Step 1. Document treatment: In phase 1, the enterprise documents have been annotated linguistically. Also some named entities and the concepts of the generic ontology have been detected. However, in the learning steps, new concepts have been identified, for this the process of searching the ontology concepts is performed in order to detect the new added elements.
Step 2. Instance identification: In order to identify the concept instances, we use the set of lexico-syntaxic patterns:
-Instances extracted from taxonomic relations: These instances are extracted using the pattern presented in the section pattern for hierarchical relations. If the identified pattern contains an existing concept, (designed in the pattern by NPC) and the hypernyms are proper nouns, these hypernyms will be added as candidates to be a concept instances.
Example: in the sentence "Product such as Phone-7", a Hearst's pattern is identified and the concept "Product", the proper noun "Phone-7" is added as a candidate to be an instance of the concept "Product".
-Instances extracted from non taxonomic relations: These instances are extracted using the pattern presented in the section pattern for non hierarchical relations. The patterns have the form $NP[P]VNPC$ or *NPC [P] V NP*. To search instance using these patterns, the following conditions are verified:
If *NPC* is a concept and *[P] V* is the name of one of its relation *R(NPC, C2)*, then *NP* is a candidate to be an instance of *C2*.
For example, in the sentence "Products bought by ECP", "ECP" is an instance added to the concept "Client" because we have the relation "Product buy by client" in the ontology.
Step3. Candidate instance treatment :

The previous patterns allow the identification of candidate instance to add to the enterprise ontology. However, some instances can be applied to many concepts. In order to resolve this problem, a similarity measure has been used which calculates the similarity between the candidate instance and the other concepts. Each candidate instance *Ci* is applied to a concept *C* using this notation $CI :=< Ci, C >$. If *Ci* is applied to more than one *CI*, the similarity is searched between this instance and the set of the concepts. For each *CI*, the similarity *sim(Ci,C)* is computed between the concept *C* and the candidate element.

3.3 Summary

In this chapter, we have presented an enterprise ontology learning approach which extracts enterprise ontology from the enterprise unstructured data. Before the description of our approach, we have presented in the section 3.1 the basic concepts related to the ontology domain and the related works on ontology building, which can be manual or semi-automatic (called in this case ontology learning), and ontology population which can be seen as an ontology enrichment process. In the section 3.1.4, we have presented the definition and the approaches to build an Enterprise ontology. The existing enterprise ontology building approaches are manual, which produces generic ontology [Uschold 1998], or semi-automatic [Blomqvist 2006] which produces very specific ontology for a particular enterprise. However, the generic approaches do not integrate the specific concepts related to a particular enterprise. For example, a generic ontology does not specify the different products in an enterprise. In the other hand, a specific ontology contains only a part of the local enterprise concepts which exist on the input documents used for the learning process. However, many interesting concepts can be found in the enterprise but does not figure in these documents. In our work, the enterprise ontology will be used to detect enterprise objects from the relational database and to enrich the extracted graphs from these databases by adding other objects relations. Thus, this ontology should integrate the generic concepts of the enterprise domain and the concepts related to a particular enterprise. Therefore, we propose in section 3.2 an enterprise ontology learning approach which is based on a generic ontology built from the existing enterprise ontology and the enterprise models. The learning process enriches the generic part with new concepts, instances and relations from the enterprise unstructured data. The evaluation of the building ontology is performed manually.
In the next chapter, the integration of the resulting ontology in the objects interactions graph extraction process, will be presented. The ontology concepts will play the role of enterprise objects reference.

Chapter 4

Object Interaction Graph Extraction from Relational Database

Contents

In the enterprise, business objects like employees, projects, products and so on can share different relations or interactions. We can name these graphs objects interactions graphs. In this chapter, we propose a new approach to extract these graphs. The proposed approach is based mainly on two steps: (1) The Conversion of the relational database model to a Graph model; and (2) the extraction of the interactions graphs, with the chosen objects, from the graph model.
In order to facilitate the treatment of these processes and to improve the visualization and the manipulation of the extracted graphs, we propose a new graph model called SPIDER-Graph (Structure Providing Information for Data within Edge or Relations). This graph data model, which is inspired from an existing model, allows to model complex graphs with heterogeneous objects and multiple relations.
We then give an overview of the proposed approach in section 4.1. The used graph model is detailed in section 4.2. Section 4.3 presents the relational database transformation to a graph model. Finally, the interaction graph extraction process is detailed in section 4.4.1.

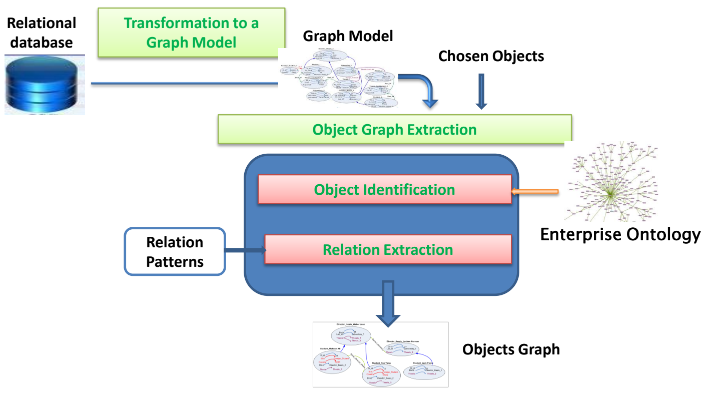


Figure 4.1: The Approach Overview.

4.1 The Approach Overview

In the enterprise context, the most important data are stored in relational databases. A heterogeneous object graph, which describes the interactions between different objects like social network graph, products and custumers graph, project and employee, will facilitate the analysis of the interactions and will help make better decisions. The object graph extraction approach is based essentially on two main steps (see figure 4.1):
(1)Relational database transforming into a graph model: This transformation allows the extraction of all the objects in the relational database in the form of nodes and outlines the relations between them, which in further step facilitate the selection of the desired one. Also, nodes in SPIDER-Graph are more complex than a simple graph. A SPIDER-Graph node can encapsulate all the attributes of an object.
(2) Objects interaction graph extraction. This contains two sub steps: objects identification and relations extraction. After transforming the relational database into a graph model, the object identification process is performed. In order to make this identification automatic a priori knowledge should be used. Here, we use the enterprise ontology built with the enterprise documents (described in the previous chapter). Then, having the set of desired object a relation detection algorithm based on a set of patterns and the ontology relations is applied.

4.2 The SPIDER-Graph Model

In section 2.1.2, we have presented a variety of models for graph databases. All these models have their formal foundation as a variation of the basic mathematical definition of a graph. The structure, used for modeling entities and relations, influences the way that data are queried and visualized. These models can be divided in two categories:

1. Models based on simple nodes data being represented by a (directed or undi-

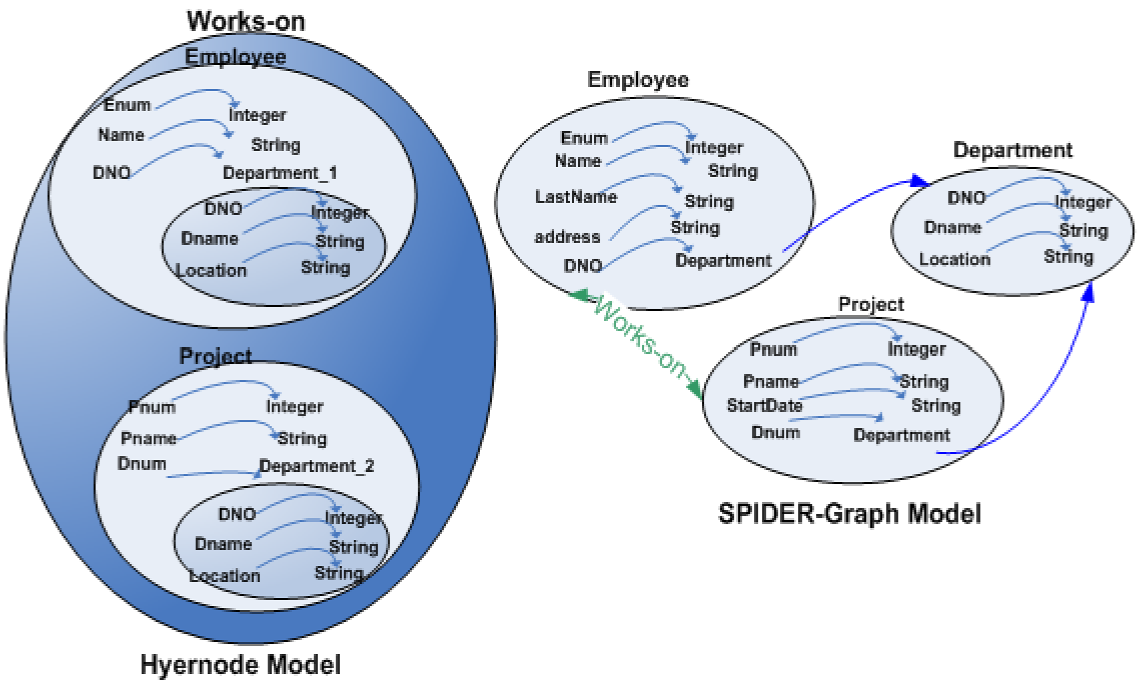


Figure 4.2: SPIDER-Graph Model VS Hypernode Model.

rected) graph with simple nodes and edges (like GOOD [Gyssens 1990c], GMOD [Andries 1992], etc.).

2. Models based on hypernode: in these models, the basic structure of a graph (node and edge) and the presentation of entities and relations are based on hypernodes and hypergraphs (like Hypernode Model [Levene 1995],GGL [Graves 1995],etc.).

From this comparison, models based on a simple graph are unsuitable for complex networks where entities have many attributes and multiple relations. However, models based on hypernodes can be very appropriate to represent complex and dynamic objects. In particular, the hypernode model with its nested graphs can provide an efficient support to represent every real-world object as a separate entity. Hypernode also encapsulates all the attributes related to each object in a same node, thus facilitating the visualization of objects with different and multiples attributes. However, the hypernode model does not offer an explicit representation of labeled edges and with the multiple encapsulations the relation representation can be lost.

In order to have a better suited model to represent complex heterogeneous graphs, we propose a graph model based on complex-nodes which have a structure similar to a hypernode. The data model we propose here contains some characteristics of the hypernode model. Indeed, both models encapsulate the object attributes, inside the complex-nodes or the hypernode. Attributes can also be multi-valuated. The differences between a hypernode and our model are the following (see Figure 4.2):

1. Our model proposes an explicit representation of labeled edges between nodes to describe objects relations instead of encapsulating two hypernodes within a further hypernode in the case of the hypernode.
2. Attributes type can be a reference to another node instead of using the encapsulation in the hypernode model in order not to have a multiple encapsulation leading to hidden interactions between objects.
3. SPIDER-Graph separatee between the schema level and the instance level: this separation can help in the graph treatment. Indeed, the schema level represents the meta-model or the global view of how the instance level should follow.

The differences between the SPIDER-Graph model and the hypernode model are summarized in table 4.1.

Table 4.1: The difference between the SPIDER-Graph model and the hypernode model

	Hypernode model	**SPIDER-Graph model**
Node labeled	supported	supported
Edge labeled	By encapsulation	Explicit labeled edge
Nested relation	supported	supported
Complex object	supported	supported
Schema graph	Not supported	supported
Instance graph	supported	supported
Attribute Node	Encapsulated in the node	Encapsulated in the node
Simple Attribute type	supported	supported
Complex Attribute type	Encapsulate the object	Reference the object
Encapsulation depth	With no limit	One level

SPIDER-Graph, our model is built on the traditional graph-based model. This model is designed to explicitly represent the interaction between objects having multiple attributes. The underlying data structure of the SPIDER-Graph model is the complex-node, which is used to represent objects. A schema graph in the SPIDER-Graph model is schematized by an attributed labeled graph: objects are represented by means of typed complex-nodes which may carry attributes. Attributes can be atomic, multi-valued, or a reference to another object (another complex-node). Relations between objects are modeled by labeled edges without attributes. Our model is defined as follows:

Definition 14 (Complex-node) *A Complex-node CN is defined by* $CN := (cn, A_{cn})$ *where:*

- *cn denotes the name of CN*
- *A_{cn}denotes a set of attributes $A_{cn} := \{\ a_{cn}|\, a_{cn} := \langle n, type, kf, kp\rangle\ \}$where*
  - *n is the attribute name.*
  - *type is the attribute type. it can be a basic type (such as: Integer, String,...) or a reference to another CN.*
  - *kf mentions if the attribute is extracted from a foreign key.*
  - *kp mentions if the attribute is extracted from a primary key.*

Now we can formally represent the graph schema (Example in Figure 4.3) by:

Definition 15 (SPIDER-Graph schema) *The schema of SPIDER-Graph model is defined by $S = (N_{cn}, R)$ where:*

- *N_{cn} is the set of complex nodes,*
- *R is the set of relations between N_{cn} defined by $R := \{r_{cn}\,|r_{cn} = \langle r, CN_s, CN_d\rangle$,CN_s,$CN_d \in N_{CN}\}$ with :*
  - *r denotes the name of r_{cn}*
  - *CN_s denotes the node source name*
  - *CN_ddenotes the node destination name*

In the SPIDER-Graph model, an instance graph is schematized by an attributed graph: concrete objects represented by an instance complex-node where values are added to the attributes and relations represented as edges labeled with the corresponding relation name according to the schema. The instance of a complex node is defined by:

Definition 16 (Complex-node instance) *A Complex-node instance CN_I of CN is defined by $CN_I := (cni, cn, A_{cni})$ where:*

cni denotes the name of the instance,

cn denotes the name of CN ,

A_{cni} denotes a set of valuated attributes A_{cni}:=$\{\ a_{cni}|\, a_{cni} := \langle n, V\rangle\ \}$with:

- *n is the attribute name (according to the schema)*
- *V is the set of values which can be atomic values or a CN_I (having the same type mentioned in the schema).*

Definition 17 (SPIDER-Graph instance) *The SPIDER-Graph instance is defined by $IS := (N_{Icn}, R_I)$ where:*

- *N_{Icn} is a the set of instance complex-nodes,*

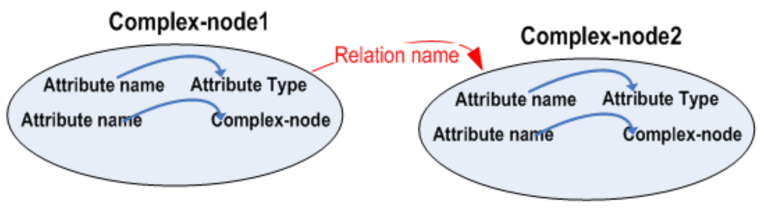


Figure 4.3: Complex-node Schema.

- R_I *is the set of relations between* N_{Icn} *defined by* $R_I := \{\langle r, CN_{Is}, CN_{Id}\rangle$ $r \in R, and CN_{Is}, CN_{Id} \in N_{cn}\ \}$

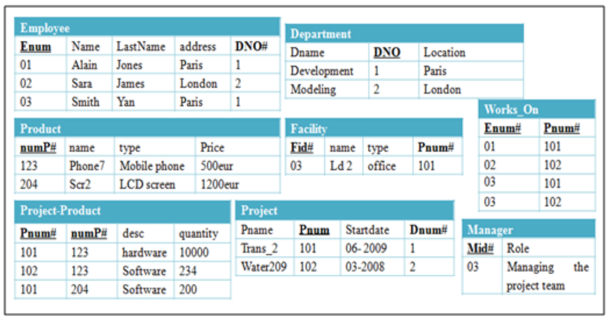
the SPIDER-Graph model is used in our graph extraction approach to model the input data and the final extracted graph.

4.3 Transforming the Relational Database to a SPIDER-Graph

Having a SPIDER-Graph model instead of a relational model can provide a clear view of the existing objects in the initial database, show explicitly the implicit relations (expressed by keys in the relational schema) and discover hidden ones.
Using this graph can facilitate the selection of the desired objects and their interactions at a further step.
In this section, we detail the process of transforming the relational data model into a SPIDER-Graph data model. The transformation of a relational model into a target model includes schema translation and data conversion [Maatuk 2008]. The schema translation can turn the source schema into the target schema by applying a set of mapping rules. In our work, we propose a translation process which directly transforms the relational schema into a SPIDER-Graph schema.
Once the schema level is extracted, data contained in the relational schema are used in order to get the instance level.

4.3.1 Process1: Schema Translation

The schema translation process is based on two main steps:(1) extracting the relational database schema, and (2) transforming it into the SPIDER-Graph schema.
The first step is to extract the schema of the relational database by using the schema metadata of the relational database management system (it contains information about tables and columns). The idea is to identify the primary key, composite key(s) and foreign key(s) of each relation. This information is then used to design the new schema (complex-nodes and relations within and between them). This process is performed along the following steps.



Employee

Enum	Name	LastName	address	DNO#
01	Alain	Jones	Paris	1
02	Sara	James	London	2
03	Smith	Yan	Paris	1

Department

Dname	**DNO**	Location
Development	1	Paris
Modeling	2	London

Works_On

Enum#	**Pnum#**
01	101
02	102
03	101
03	102

Product

numP#	name	type	Price
123	Phone7	Mobile phone	500eur
204	Scr2	LCD screen	1200eur

Facility

Fid#	name	type	Pnum#
03	Ld 2	office	101

Project-Product

Pnum#	**numP#**	desc	quantity
101	123	hardware	10000
102	123	Software	234
101	204	Software	200

Project

Pname	**Pnum**	Startdate	Dnum#
Trans_2	101	06-2009	1
Water209	102	03-2008	2

Manager

Mid#	Role
03	Managing the project team

Figure 4.4: The Relational Database.

4.3.1.1 Step1 : Relational Schema Extraction

The Relational schema extraction process extracts the schema of the relational database using the schema metadata of the relational database management system (it contains information about tables and columns). It identifies the primary keys, composite key(s) and foreign key(s) of each table. This information is extracted using SQL queries. The extracted relational schema is represented as follow:

Definition 18 (Relational database schema) *Relational database schema is defined by* $RS : \{T\,|T := \langle t_n, A, K_{P,F}\rangle\}$*, where:*

- t_n *denotes the name of* T*.*
- A *denotes a set of attributes of* T*;* $A := \{a = \langle < a_n, t, kf, kp > \rangle\}$*, with:*
  - a_n *is an attribute name,*
  - t *is its type*
  - kf *mentions if* a *is a foreign key or not,*
  - kp *mentions if* a *is a primary key or not*
- $K_{P,F}$ *denotes a set of keys of* T*;* $K_{P,F} := \{\beta\,|\beta := \langle kr, kf, kp, re, f_a\rangle\}$*, with:*
  - β *represents a key (* a_n *attribute which can a part of a composed key),*
  - kr *is the name of the attribute containing the key,*
  - kf *mentions if* β *is a foreign key or not*
  - kp *mentions if* β *is a primary key or not,*
  - re *the table that contains the exported primary key,*
  - f_a *is the attribute name of the foreign key.*

This schema provides an image of metadata obtained from an existing relational database. It provides more information than traditional schema. Indeed, it gives information about primary and foreign keys to facilitate relations extraction in further steps.

Consider the database shown in Figure 4.4. For example for the Table "Employee", the relation $Employee(\underline{\textbf{Enum}}, Name, LastName, address, \textbf{DNO\#})$ was extracted, where primary key is stressed and written in bold (here is *Enum*) and the foreign key is marked with # and written with bold (here is *DNO*). In table 4.2, we expose an excerpt of the relational database schema extracted from the relational database in Figure 4.4.

Table 4.2: An Excerpt of the Extracted Relational Database schema

	t_n	**A**				$\mathbf{K}_{p,f}$				
		a_n	t	kf	kp	kr	kf	kp	re	f_a
T1	Employee	Enum	Integer	false	true	Enum	false	true	-	-
		Name	String	false	false	-	-	-	-	-
		LastName	String	false	false	-	-	-	-	-
		Adress	String	false	false	-	-	-	-	-
		DNO	Integer	true	false	DNO	true	false	Department	DNO
T2	Project	Pnum	String	false	true	Pnum	false	true	-	-
		Pname	String	false	false	-	-	-	-	-
		Startdate	Date	false	false	-	-	-	-	-
		Dnum	Integer	true	false	-	-	-	-	-
T3	Works_On	Enum	Integer	true	true	Enum	true	true	Employee	Enum
		Pnum	Integer	true	true	Pnum	true	true	Project	Pnum

4.3.1.2 Step 2: Mapping the Relational Schema to the SPIDER-Graph Schema

In order to transform a relational schema into a SPIDER-Graph schema, two steps are performed: First, the extraction of the complex-nodes then the identification of their relations.

A. Complex-nodes Extraction

Using the relational schema, algorithm 1 is used to extract the complex-nodes. The proposed algorithm creates from each table T a complex-node CN. CN takes the attributes of T. However, the algorithm changes the attributes types if the attribute is a foreign key. In this case, the attribute type will be a reference to the table that contains the exported key.

From the relational database tables (Figure 4.4), the proposed algorithm extracts six complex-nodes (objects) (Figure 4.6): *Employee, Manager, Department, Product, Project, Project-Product.*

The complex-node Employee = ("Employee",{$\langle Enum, Integer\rangle$,$\langle Name, String\rangle$,$\langle$ L*astName, String*>, $\langle$ *address, String*>,$\langle$ *DNO, Department* >); has four attributes (Enum, Name, LastName, address) with predefined types and one attribute *DNO* having *Department* as type because it represents a foreign key

Algorithm 1: Complex-nodes Extraction

```
Data: $RS$ the relational database schema.
Result: the set of the complex-node $N_{cn}$.
begin
    foreach $T \in RS$ do
        CreatComplexNode($T$)
        $CN.cn = T.t_n$
        create the $CN.A_{cn}$
        foreach $a \in T$ do
            create $a_{cn} \leftarrow a$
            if $a.ce = true$ then
                $a_{cn}.type$ =Table that contains the exported key
            else
                $a_{cn}.type = a.type$
```

exported from the table *Department*.

B. Relation Identification
The second step of the SPIDER-Graph schema transformation is the relation identification which is performed using algorithm 2.
The proposed algorithm uses the set of identified complex-nodes and the keys extracted from the relational database. Indeed, for each complex-node *CN* in the SPIDER-Graph schema, the algorithm analyzes its set of attributes A_{CN}. The detected relations can be classified in four categories:

- **Association relations** (line 3 →6): as a first step, the algorithm checks if all the existing attributes on the complex-node are at the same time foreign keys and primary keys. In this case, *CN* is deleted and transformed into relations between the complex-nodes referenced by the attributes. In each complex-node mentioned on the foreign keys reference, new attributes containing the others foreign keys are added.
  For example, *Works-on* is composed only by two attributes which are referencing other complex-nodes. Then, it is transformed into a relation between *Employee* and *Project* having the name "Works-on". Also, two new attributes $< Enum, Employee >$ and $< Pnum, Project >$ are added to the complex-nodes *Project* and *Employee* ,respectively.

- **Dependence relations** (line 8 → 6): if A_{CN} contains an attribute a_{cn} which is a foreign key ($a_{cn}.kf = true$) and not a primary key ($a_{cn}.kp = false$), then the current complex-node *CN* is in relation with the complex-node which is referenced by a_{cn} (which is mentioned by $a_{cn}.type$). We cannot add a name for this relation because we only know the existence of a dependency between

the two tables.
For example, the relation $r :=< "", Employee, Department >$ is created between the complex-nodes *Employee* and *Department* because *Employee* contains the attribute *DNO* which references the complex-node *Department.*

- **"IS-A" relations** (line 13 $\rightarrow$ 17): if A_{CN} contains only one attribute a_{cn} which is a foreign key ($a_{cn}.kf = true$) and also a primary key ($a_{cn}.kp = true$), then the current complex-node CN has the relation "IS-A" with the complex-node the name of which is mentioned in a_{cn}*.type.* In the relational database to design an inheritance relation between a table A and B, we put the A primary key as a foreign primary key in B.
For example, the relation $r :=< "IS - A", Manager, Employee >$ is created between the complex-nodes *Employee* and *Manager* because *Manager* contains the attribute *Mid* which a primary key and also a foreign key referencing *Employee.*

- **"Part-of" relations** (line 18 $\rightarrow$ 21): if the complex-node CN has more than one attribute which is a foreign key and also primary keys and has other attributes which do not verify this condition, then the algorithm builds a relation "Part-of" with the current complex-node CN and with the complex-node referenced by a_{cn}.
For example, the complex-node $Product - Project$ has the attributes *Pnum* and *NumP* which are primary keys and also foreign keys referencing *Project* and *Product*, respectively. Then, the algorithm creates the relations $r1 :=< "Part - of", Product, Product - Project >$ and $r2 :=< "Part - of", Project, Product - Project >$.

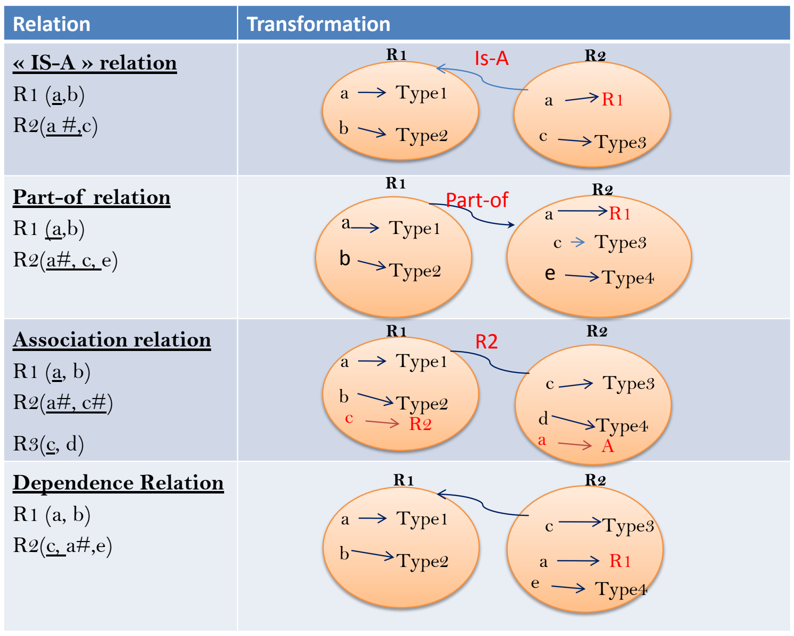
A summary of the previous relations is presented in Figure 4.5.

Applying the complex-nodes extraction and the relation extraction processes to the relational database presented in Figure 4.4, we obtain the SPIDER-graph schema depicted in Figure 4.6.

In the next section, the process to populate the extracted schema is described.

4.3.2 Process2: Data Conversion

In order to populate the pattern already identified, we propose an approach of data conversion that uses the relational database tuples to create the SPIDER-Graph instance graph. This process is based on three steps.
First, the relational database relations tuples are extracted using SQL queries. Second, for each complex-node in the SPIDER-Graph schema, a complex-node instance set (see definition 16) is extracted from the corresponding tuples.
For example, from the relational database in figure 4.4, the table *Employee* contains three tuples. Then, the process creates three instances for the complex-node *Employee*: *Employee_ 1*, *Employee_2* and *Employee_3.* The value of the instances follows the SPIDER-Graph schema. Indeed, *Employee_3* is defined by



Relation	Transformation
« IS-A » relation R1 (a,b) R2(a #,c)	R1 a → Type1 b → Type2 Is-A R2 a → R1 c → Type3
Part-of relation R1 (a,b) R2(a#, c, e)	R1 a → Type1 b → Type2 Part-of R2 a → R1 c → Type3 e → Type4

Association relation
R1 (a, b)
R2(a#, c#)
R3(c, d)
	R1
	a → Type1
	b → Type2
	c → R2
	R2
	R2
	c → Type3
	d → Type4
	a → A

Dependence Relation R1 (a, b) R2(c, a#,e)	R1 a → Type1 b → Type2 R2 c → Type3 a → R1 e → Type4

Figure 4.5: The Set of the Identified Relations.

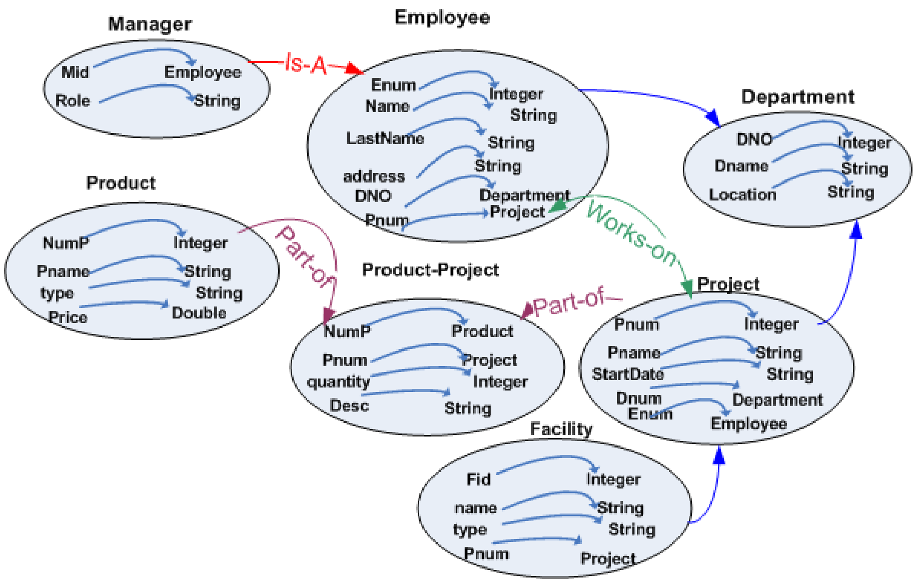


Figure 4.6: The SPIDER-Graph Schema.

Employee_3:=("Employee_3","Employee",$A_{Employee\_3}$)$where$:
$A_{Employee\_3}$:={$\langle Enum, 03\rangle$,$\langle Name, Smith\rangle$,$\langle LastName, Yan\rangle$, $\langle address, Paris\rangle$, $\langle DNO, Department\_1\rangle$ }
Finally, for each relation in the SPIDER-Graph schema, a set of instance relations is extracted using the value of keys on the relational tables. Transformed data are

Algorithm 2: Relation Extraction

```
Data: $RS$ the relational database schema,the set of the complex-node $N_{cn}$
Result: the set of relations $R$.
1  begin
2      foreach $CN \in N_{cn}$ do
3          if $number(kp) = number(kf)$ then
4              foreach $(ai_{cn}, aj_{cn}) \in A_{cn} \cap i \neq j$ do
5                  create $r := \langle < cn, ai_{cn}.type, aj_{cn}.type > \rangle$
6              Delete $CN$ from $R$
7          else
8              foreach $a_{cn} \in A_{cn}$ do
9                  if $(a_{cn}.kf = true)\, and\, (a_{cn}.kp = false)$ then
10                 $CN_{kf} = a_{cn}.type$
11                 create $r := \langle <, "", CN, CN_{kf} > \rangle$
12                 add $r$ to $R$
13                 if $(a_{cn}.kf = true)\, and\, (a_{cn}.kp = true)$ then
14                 if $(number(kp) == 1)\, and\, (number(kf) == 1)$ then
15                     $CN_{kf} = a_{cn}.type$
16                     create $r := \langle < "IS - A", CN, CN_{kf} > \rangle$
17                     add $r$ to $R$
18                 if
                   $(number(kp \cap kf) \rangle\, 0 and\, (number(kf \cap kp) < number(kp))$
                   then
19                     $CN_{kf} = a_{cn}.type$
20                     create $r := \langle < "Part - Of", CN_{kf}, CN > \rangle$
21                     add $r$ to $R$
```

loaded into the SPIDER-Graph schema. An excerpt of the SPIDER-Graph instance extracted from the relational database in figure 4.4 and following the SPIDER-Graph Schema in figure 4.6 is shown in 4.7. The extracted SPIDER-Graph contains the same information found in the input relational database. It represents a graph view of the database which models explicitly the relations between the existing objects.

4.3.3 Process Performance

The relational database transformation into a SPIDER-Graph process is based on the set of algorithms executed sequentially:
-The relational schema mapping to a SPIDER-Graph (see section4.3.1.2): complex-node extraction (algorithm 1) and relation extraction (algorithm 2).

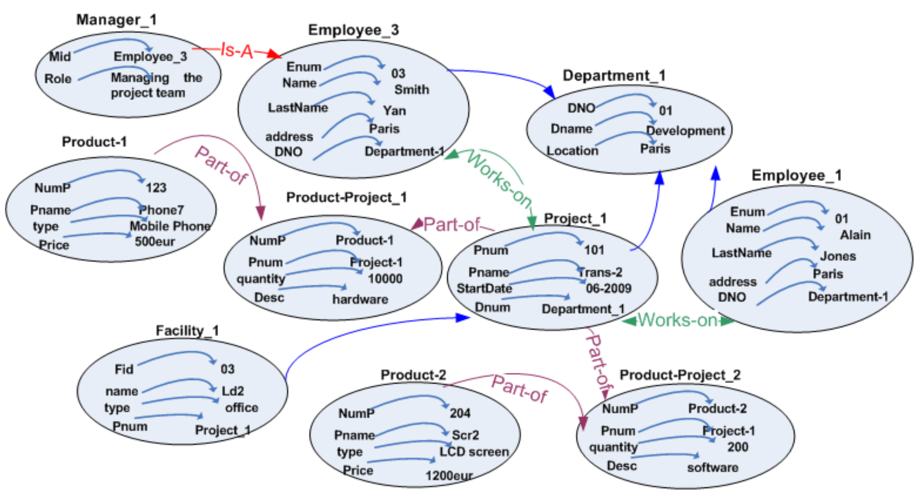


Figure 4.7: An Excerpt of the SPIDER-Graph Instance.

- The data migration (see section 4.3.2): Instance complex-nodes extraction and relations building.

Then, the time complexity of the whole process is the sum of these algorithms time complexity, that we detail in the following of this section. The schema translation process time complexity depends on the database size (number of tables and of the attributes of each table) and the complexity of the used SQL queries. The used queries are simple select queries which are created to extract the relational database meta-data. For example to extract the table names from the PostgreSQL database, the following SQL query is used:

"Select tablename from pg_tables where schemaname="public";"

Another example, to extract the attribute information related to each table, the following SQL query is used:

"SELECT column_name,data_type,is_nullable FROM INFORMATION_SCHEMA.COLUMNS where table_name=name";

The time complexity of these queries can be considered as an elementary instruction. Then, the schema translation complexity is in the order of $O(n \times m)$ where n the number of tables and m the average of attributes in the relational database tables.

The relation extraction algorithm has the same time complexity which is $O(n \times m)$ where n is the number of complex-nodes and m the average of attributes related to each complex-nodes. The relation extraction is accelerated by using a HashMap to store the complex-nodes that have been extracted.

The data Migration process has the complexity of $O(n_2)$. Indeed, for each complex-node, the algorithm searches the values of its attributes from the relational database. In the case of attributes referencing other complex-node (for example in the attribute <Lab_id, Laboratory, Laboratory_1>), the algorithm should search for the corresponding reference in the set of the instance complex-nodes.

This complexity depends on the relational database management system used. For instance, the maximum column per table[1] is 1000 for Oracle, 4096 for MySQL and 250-1600 for PostgreSQL. Thus, the $O(n_2)$ is generally not reached.

4.4 Graph Transformation According to the User

From the graph depicted in the previous section, we apply extraction rules according to the user's interest (the set of objects interactions of which the user would like to see). Extracting the graph leads to cope with two main problems: objects of interest (named objects in the following) identification and relations extraction and transformation.

4.4.1 Object Interaction Graph Extraction Process

The object interaction graph is a graph describing the different possible relations between heterogeneous objects. From the graph depicted in the previous section, we apply extraction rules according to the interest ofthe user (the set of objects interactions which the user would like to see) in order to extract the object interaction graph. This graph is defined as following.

Definition 19 (The objects graph) *The object interaction graph is the graph defined by $GO := (O_I, R_O)$ where:*

- *O_I is a finite set of objects such $O_I := \{o_I \,|o_I \in N_{cn}\}$.*
- *R_O is a finite set of relations between objects such $R_O :=\{r\,|r := \langle l, o_{1I}, o_{2I}\rangle$ $o_{1I}, o_{2I} \in O_I$ $\}$where l is the relation name.*

The extracted graph is also a SPIDER-Graph.

4.4.2 Object Identification

Object identification is the process used to identify complex-nodes that contain the objects of interest for the user (for example Employee or Project). These objects constitute the nodes of the extracted graph.
Many problems may occur during this step. First, an object can be described by the means of different tables in the initial relational database, so many complex-nodes on the SPIDER-Graph can represent the same object (for example *Manager* and *Employee* represent *Person*). Second, the names of the complex-nodes may not be significant (for example a complex-node describing an employee can have the name "TB-Emp"). Each object has a number of characteristics which help to identify it. Then, in order to identify the desired objects on the SPIDER-Graph, we have proposed an approach that analyzes not only the name of each complex-node but also the attributes related to each one. For this, we have used the enterprise ontology in

[1]http://en.wikipedia.org/wiki/Comparison_of_relational_database_management_systems

the identification process as an external reference containing the basic characteristics of each enterprise elements.
The process of object identification takes place as follows. Firstly, the user selects the concepts, representing the objects which he wants to see in the graph, from the enterprise ontology. Then, the SPIDER-Graph schema is analyzed, considering the chosen concepts. If the complex-node contains some characteristics related to the desired object (represented by the selected concept) or bears the same name, it will be selected to be one of the objects in the extracted graph. After identifying the complex-nodes that contain the chosen objects, we add their complex-nodes instances to the set of identified objects O_I.
This process is carried out via the algorithm 3. The proposed algorithm takes as input the set of chosen concepts and the SPIDER-Graph schema and it produces as an output the set of corresponding objects to the chosen concepts. Algorithm 3 is based on the following steps:

Step 1.Names treatment. In this step, the name of each complex-node *CN*in the input SPIDER-Graph is treated by the following process ($2 \longrightarrow 5$). First, the algorithm cuts the labels into a set of tokens e.g. if a complex-node *CN* has the name "*Projects-Products*", the set of extracted tokens is {*Projects, and, Products*}. The tokenizer recognizes punctuation, blank characters, digits, etc.
Then, the algorithm extracts the corresponding lemmas to each token in order to find its basic form (e.g., Projects$\mapsto$Project). The tokens such as articles, prepositions, conjunctions, and so on, are marked to be discarded. At the end of this process, each name of *CN* is designed by its set of tokens T_{CN}:=$\{t_i \mid t_i \subseteq cn\}$ e.g. for the name "Projects-Products", the corresponding set of tokens is {Project, Product}. We name the set of T_{CN} TCN.
The same process is applied to the labels of each selected concept ($6 \longrightarrow 9$). Then, each label of a concept $c \in C$ is composed by a set of tokens $T_c := \{t_{ci} \mid t_{ci} \subseteq c\}$. The set of T_c is named TC.

Step 2. String matching between the name of *CN* and the concept label. ($10 \longrightarrow 22$) In order to compare the *CN* name *cn* and the label of a concept $c \in C$, the algorithm uses the two sets T_{CN} and T_c. This comparison is summarized by the following cases:

1. If T_{CN} is exactly equal to T_c, then *CN* is added to the set of objects O_I.
2. If the T_{CN} elements are sub-concepts or synonyms of *c* then, *CN* is added to O_I. Example: if *c*=*Enterprise* and *cn* = *Organization* then *CN* is an object to add in the final graph because *cn* is synonym of *c*.
3. If one of the T_{CN} elements has one of the T_c elements as a suffix or prefix then *CN* is candidate to be an object to add in the final graph. Example *c*=*Enterprise*$\longrightarrow$ Tc={Enterprise} , cn=*TBEnterprise* $\longrightarrow$ T_{CN} ={*TBEnterprise*}. In this case, "*TBEnterprise*" contains "*Enterprise*" then "*TBEnterprise*" is candidate to represent the concept "*Enterprise*".

Algorithm 3: Object Identification Algorithm

```
Data: $C_{user}$, SPIDER-Graph schema $S := (N_{cn}, R)$, SPIDER-Graph
      instance $IS := (N_{Icn}, R_I)$
Result: The chosen objects $O_I$
1  begin
2      foreach $CN_i := (cn, A_{cn}) \in N_{cn}$ do
3          Extract tokens and extract lemmas from each token
4          $T_{CN_i} := \{ti \,|ti \subset cn\}$
5          add $T_{CN_i}$ to $TCN$;
6      foreach $C_i := (c_l, I) \in C_{user}$ do
7          Extract tokens and extract lemmas from each token
8          $T_{C_i} := \{ti \,|ti \subset c_l\}$
9          add $T_{C_i}$ to to $TC$;
10     foreach $T_{CN_i} \in TCN$ do
11         foreach $T_{C_i} \in TC$ do
12             if $T_{CN_i} == T_{C_i}$ then
13                 Add $CN_i$ in $O_I$;
14             else if $T_{CN_i}.containsSynonyms(T_C) \,||T_{CN_i}.containssubC(T_C)$
               then
15                 Add $CN_i$ in $O_I$;
16
17             else if $T_{CN_i}.containsSuffix(T_C)||T_{CN_i}.containsPreffix(T_C)$
               then
18                 Add $CN_i$ in $CAND$;
19
20             else
21                 if $Sim(T_{CN_i}, T_{C_i}) > \alpha$ then
22                     Add $CN_i$ in $CAND$;
23     foreach $CN_i \in CAND$ do
24         foreach $C_i \in C$ do
25             if $Simatt(CN_i, C_i) > \beta$ then
26                 Add $CN_i$ in $O_I$;
27     foreach $CN_i \in O_I$ do
28         Remove $CN_i$ from $O_I$ ;
29         Add $CN_i$ instances in $O_I$ ;
30     return $O_I$;
```

if c=*Employee* then T$_c$={Employee} and *cn*=*TB_Emp* then T$_{CN}$ ={TB,Emp}, *Emp* is a suffix of *Employee* then this complex-node is a candidate to be an object in the final graph and it is added in the set of candidates $CAND$.

4. In the other cases, we calculate the similarity between T$_c$ and T$_{CN}$ using the name similarity measure proposed in [Madhavan 2001]:

$$Sim(T_c, T_{CN}) = \frac{\sum_{t_1 \in T_c} \left[\max_{t_2 \in T_{CN}} sim(t_1, t_2) \right] + \sum_{t_2 \in T_{CN}} \left[\max_{t_1 \in T_c} sim(t_1, t_2) \right]}{|T_c| \times |T_{CN}|}$$

where sim(t$_1$,t$_2$) is calculated using the Jiang and Conrath measure [Jiang 1997] and WordNet as reference.

The CN is added as a candidate object if Sim(T$_c$,T$_{CN}$) exceeds a given threshold value α.

Step 3. Candidate object treatment. (10 $\longrightarrow$ 22) In this step, the algorithm analyzes the set of candidate complex-nodes by using their attributes. For each chosen concept C, it compares its datatype properties (DP) with each candidate CN attributes by using a similarity measure. The similarity between a concept C and a complex-node CN, based on their attributes similarities, is calculated as follows:

$$Simatt(C, CN) = \frac{\sum_{dp \in DP} \sum_{a_{cn} \in A_{cn}} sim(dp.n, a_{cn}.n)}{|DP| + |A_{cn}|}$$

where dp is a datatype property of C having the name $dp.n$ and a_{cn} is an attribute in A_{cn} having the name $a_{cn}.n$. If *simatt (C,CN)* $> \beta$ (where β is a threshold value) then CN is added to the objects set.

In order to calculate the similarity $sim(dp.n, a_{cn}.n)$ between the name of an attribute and the name of datatype property, we use the following process.
Firstly, the name of attribute and the name of datatype property is preprocessed by tokenizing and producing T_{acn} and T_{dp}. Then, the attributes similarities are calculated using a string matching measure. We use the Jaro-Winkler measure [Winkler 1999] which is based on the number and order of the common characters between two strings and which gives good result with short string [Cohen 2003]. We combine this measure with the Monge and Elkan measure one, a hybrid measure based on the use of tokens [Cohen 2003]:

$$sim(dp.n, a_{cn}.n) = \frac{1}{k} \sum_{k}^{i=1} max_l^{j=1} simJaro - Winkler(a_i, b_j)$$

where $a_i \in T_{acn}$, $b_j \in T_{dp}$, i the number of elements in T_{acn} and l the number of elements in T_{dp} .

For instance, we perform the object identification process to identify from the SPIDER-Graph the related complex-nodes to the concept *"Project"*.

Step 1. From each complex-node a set of tokens is extracted:
$T_{CN:Project}$={Project},$T_{CN:Manager}$={Manager}, $T_{CN:Employee}$={Employee},

$T_{CN:Product-Project}$={Product, Project}, $T_{CN:Product}$={Product}, $T_{CN:Facility}$={Facility},
$T_{C:Project}$={Project}
Step2. We compare each complex-node T_{CN} with the tokens set of the concept $T_{C:Project}$:

- $T_{CN:Project}$ is exactly equal to $T_{C:Project}$ then the complex-node *Project* is added to the set of object O_I
- $T_{CN:Product-Project}$ contains the $T_{C:Project}$ elements then the complex-node $Product - Project$ is a candidate to represent a Project.
- For the other complex-nodes, we calculate the $Sim(Tc, TCN)$: Sim(Project,Manager)=0,0694, Sim(Project,Employee)=0,0760, Sim(Project,Product)=0,117, Sim(Project,Facility)=0,1551. The calculated similarities do not exceed the value of α (α=0,4) (see in the evaluation presented in chapter 6). Then, we do not add them to the candidates set.

Thus, in the candidates set we put the complex-node $Product - Project$.
Step3. We calculate the similarity *Simatt* between CN=$Product - Project$ and c=*Project*, we found that simatt(CN,C)=0,557 which is not enough to add $Product - Project$ in the objects set O_I (less than β=0,7).

4.4.3 Relation Construction

After the objects identification step, we define the set of relations between the identified objects. These relations are obtained from two different sources: the ontology relations and the SPIDER-Graph extracted from the relational database. For each data source, a specific process is performed.

4.4.3.1 Relations Construction from the Relational Database

The identified objects can already share relations in the SPIDER-Graph. The objective here is to put these relations in the object interaction graph and find hidden ones. In our process to transform the relational database into a graph database model, we have defined four types of relations: IS-A, Part-of, dependency with known name (using the initial relational tables containing only foreign keys), dependency with unknown name. Then, in order to detect all the relations, we have proposed an approach based on these existing relations. From the different existing relations and by using the SPIDER-Graph schema, the proposed approach creates a set of pattern relations. After that, it seeks these patterns in the SPIDER-Graph instance and creates the set of relations between the identified objects. A pattern relation P_r is defined formally as follows:

Definition 20 (Pattern Relation) *A pattern relation* P_r *is defined by* $P_r = \langle n, o_{I1}, o_{I2}, o_m \rangle$ *such as:*

- *n is the name of the relation,*
- *o_{I1} and o_{I2} are the two types of the two objects that share a relation. o_{I1} and $o_{I2} \in O_I$*
- *o_m is a complex-node mediator (a connector) for the relation. In other word, this complex-node is used to build this relation.*

The relation pattern is able not only to find the existing relations between the objects that should be added to the final graph but also to add new attributes on the objects and find new objects that cannot be identified using the previous process. In order to create these patterns, we use the SPIDER-Graph schema which mentions the type of complex-nodes and all the relations between them. Then, these patterns are searched for the SPIDER-Graph instance. In the following part, we describe the extraction process of each pattern and how it is used to find the relations between the objects.
Patterns to identify new objects. The relation "*IS-A*" allows to find hidden objects which may not be identified with the objects identification process (Table 4.3). In the relation construction process, we start by analyzing this kind of relations to find in the next steps the relations related to the hidden objects.

Table 4.3: Patterns to identify new objects.

	Initial Relation	**Process and description**
Definition	R_1 :=$\langle$"$IS-A''$, CN_s, $CN_d\rangle$ where CN_s or CN_d instances$\in\text{O}_I$	CN_s or CN_d instances are added to O_I
Example	R :=<"IS-A", Employee, Manager>	Manager instances are added to O_I

Patterns based on relation between chosen objects. The objects on O_I can be directly linked by an existing relation in the SPIDER-Graph. If two objects CN_s and CN_d exist in O_I and share a relation R_2 then two patterns are created (see table 4.4):
-Pr_1 finds all the instances of CN_s and CN_d which share the relation R_2.
- Pr_2 creates a new relation between all the instances of CN_s that have the relation R_2 with the CN_d instance.

In some cases, the chosen objects are not directly linked. Then, we search whether there is a path between them or not. The relations between the complex-nodes are directed or bidirectional relations and we assume that each edge has a weight of 1. Then, we can apply the Dijkstra's Algorithm [Dijkstra 1959]. By applying this algorithm, we distinguish three cases:

- There is no path between the two objects,

Table 4.4: Patterns based on relations between chosen objects.

Initial Relation	**Pattern**	**Process and description**
R_2 :=$\langle r, CN_s, CN_d\rangle$ where CN_s and CN_d instances$\in O_I$	Pr_1 :=< r, CN_{si} ,$CN_{di}, null$ > where CN_{si} and CN_{di} are CN_S and CN_d instances.	Find all the existing relations between CN_s and CN_dinstances then add them to R_O
	Pr_2 :=$\langle Same\_(CN_d.name)$,CN_{SIi}, CN_{SIj}, CN_d > where CN_{SIi} and CN_{SIj} are CN_s instances and i!=j	Find all the CN$_s$ instances which have relations with a same CN$_d$ instances and link them with a new relation "Same_(CN$_d$.name)"
CN_s and CN_d instances$\in O_I$there is no relation between them	Pr_{path}:=<"",CN_{si}, CN_{di}, P> *where* P is a semi-path or a path between CNs and CN_d	See description below.

- There is a direct path and it is denoted by the path :< $CN_s, CN_j,\ldots, CN_d$> where $\{CN_j\}$ is the set of complex-nodes between the two chosen objects CN$_s$ and CN$_d$

- There is a semi-directed path. The semi directed path is denoted by the semi-path :< CN_s, $CN_j \ldots CN_k, CN_d$> where $\{CN_j\}$is the set of complex-nodes between the first chosen objects CN_s and CN_k and $\exists R_i \in R$ such as $R_i := \langle ``Part-of'', CN_d, CN_k\rangle$. This kind of relation is represented by a dotted edge in the graph depicted in Figure 4.10.

An example of the previous patterns,where the chosen objects are Employee, Project and Product, is shown in table 4.7.

Patterns based on relations between a chosen object and another complex-node (Table 4.6). Naturally, chosen objects may also have relations with other complex-nodes (which are not chosen to be added on the object graph). These relations can reveal hidden relations between objects or enrich the attributes of a complex-node. If an object CN_s has the relation $R_3 := \langle r, CN_s, CN_d\rangle$ with a complex-node CN_d which is not in O_I, then two patterns can be extracted:
- The pattern Pr_3 finds the object instances which are in relation with the same CN_d instance.
- The pattern Pr$_4$ is more complex and corresponds to the following case: if an object has a *"part-of"* relation with another complex-node CN_d and other complex-nodes have a "*part-of*" relation with CN_d, then the chosen object has a relation with this complex-node. Pr$_4$ detects these relations.
In the case where the shared relation is $R_4 := \langle r, CN_d, CN_d\rangle$, only the pattern Pr_5 can be created. Pr_5 creates a new relation between all the instances of CN_d that

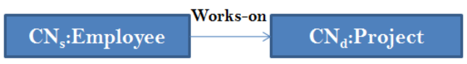
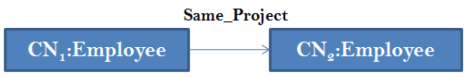
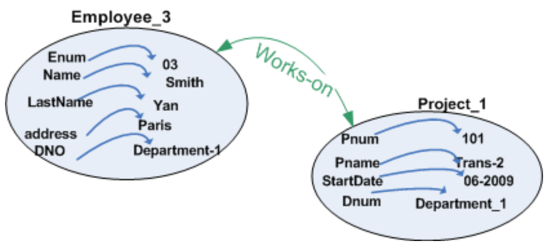
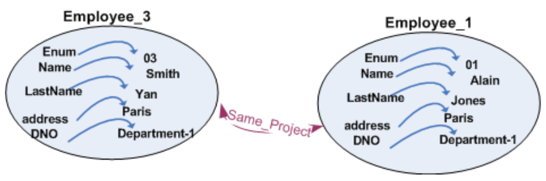
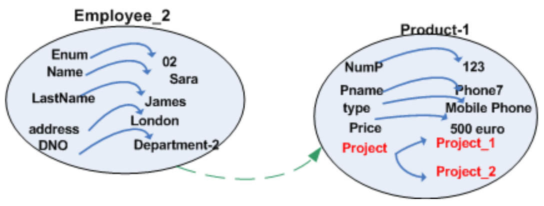
Relation and identified pattern	Example of extracted relations
Relation: R_1:=< "*Works_on*", *Employee*, *Project* > **Patterns:** Pr_1 :=< "*Works_on*", *Employee*, *Project*, *null* >  Pr_2 :=< *Same_Project*, *Employee_i*, *Employee_j*, *Project* > 	 Employee_3 Enum Name LastName address DNO 03 Smith Yan Paris Department-1 Works-on Project_1 Pnum Pname StartDate Dnum 101 Trans-2 06-2009 Department_1  Employee_3 Enum Name LastName address DNO 03 Smith Yan Paris Department-1 Same_Project Employee_1 Enum Name LastName address DNO 01 Alain Jones Paris Department-1
Relation: Path between Employee and Product **Patterns:** semi-path:⟨*Employee*, *Project*, *Product-Project*, *Product*⟩	 Employee_2 Enum Name LastName address DNO 02 Sara James London Department-2 Product-1 NumP Pname type Price Project 123 Phone7 Mobile Phone 500 euro Project_1 Project_2

Table 4.5: Example of Patterns based on relations between chosen objects.

have the relation R_4 with the same CN_s instance.

These different patterns are created and applied using algorithm 4. Firstly, the algorithm extracts from the SPIDER-Graph schema all the relations in which the complex-node source or destination are on O_I (line 2 $\rightarrow$ 3). Then, each pattern is searched for and applied on the SPIDER-Graph instance. The function $Treat-UnlinkedObject()$ is called to search if there is a path between unlinked objects.

Table 4.6: Patterns based on relations between a chosen object and another complex-node

Initial Relation	**Pattern**	**Process and description**
R$_3$ $:=\langle r, CN_s, CN_d\rangle$ where CN$_s$ $\in$O$_I$ and CN$_d$ $\notin$O$_I$	Pr$_3$:=<*Same_ (CN$_d$.name)*, CN_{SIi}, CN_{SIj}, CN_d > where CN_{SIi}, CN_{SIj} are CN_S instances, i!=j and r != "Part-of"	Find all the CN_s instances which have relations with a same CN_d instances and link them with a new relation
	Pr$_4$:=<*Same_ CN$_k$.name*, CN_{SIi}, CN_{SIj}, CN_k > where CN_{SIi}, CN_{SIj} are CN_S instances, i!=j, r="*Part-of*" and CN_k $\in\{CN \setminus CN$ has the relation *R*:=<"*Part-of*", *CN*, *CN$_d$*>}	1.Find all the complex-nodes CN$_k$ having a "Part-of" relation with CN$_d$ such as R:= <"Part-of", CN$_k$, CN$_d$ > 2. add a new attribute to CN$_s$ containing the name of CN$_k$ 3.Then the pattern Pr$_4$ is applied : find all the CN_s instances which have relations with a same CN_k instances and link them with a new relation
R_4 $:= \langle r, CN_s, CN_d\rangle$ where CN$_s$ $\notin O_I$ and CN$_d$ $\in O_I$	Pr$_5$:=<*Same_ CN$_s$.name*, CN_{dI}, CN_{jI}, CN_s > where CN_{dI} is CN_d instance and CN_{jI} is instance of $CN_j$where $CN_j \in O_I \bigcap\{CN \setminus CN$ has the relation R:=<"", *CN*, *CN$_s$*>}.	1. add a new attributes in CN_dcontaining CN_s reference. 2. Then the pattern Pr$_5$ is applied : if CN_shas relations with other objects,link each object instance with a CN_d instance if they are in relation with the same CN_sinstance.

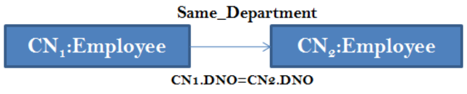
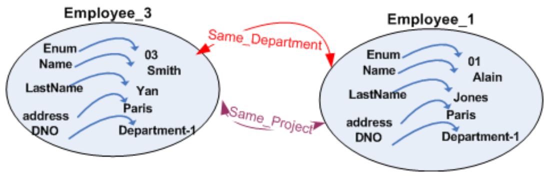
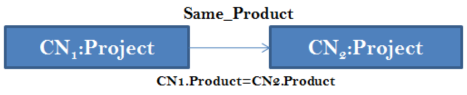
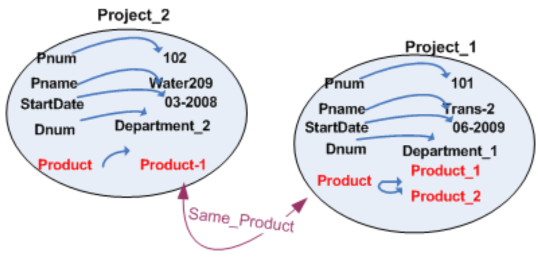
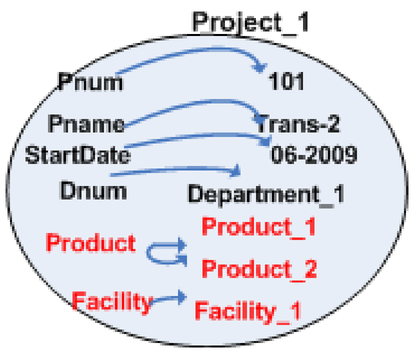
Relation and identified pattern	Example of extracted relations
Relation: R_3:=< “”, $Employee, Department$ > **Patterns:** Pr_3 :=< “$Same\_Department$”, $Employee_i$, $Employee_j$, $Department$ >  Same_Department CN1:Employee CN2:Employee CN1.DNO=CN2.DNO	 Employee_3 Enum Name LastName address DNO 03 Smith Yan Paris Department-1 Same_Department Same_Project Employee_1 Enum Name LastName address DNO 01 Alain Jones Paris Department-1
Relation: R'_3:=< “$Part-of$”, $Project, Product-Project$ > **Patterns:** Pr_2 :=< $Same\_Product$, $Project\_i$, $Project\_j$, $Product$ > -Add the attribute n:=< $Product, Product_i$ > to each instance of $Project$.  Same_Product CN1:Project CN2:Project CN1.Product=CN2.Product	 Project_2 Pnum Pname StartDate Dnum Product 102 Water209 03-2008 Department_2 Product-1 Same_Product Project_1 Pnum Pname StartDate Dnum Product 101 Trans-2 06-2009 Department_1 Product_1 Product_2
Relation: R_4:=< “”, $Facility, Project$ > **Patterns:** -*Facility* has no relation with other object then no pattern detected. -Add the attribute n:=< $Facility, Facility_i$ > to each instance of $Project$.	 Project_1 Pnum Pname StartDate Dnum Product Facility 101 Trans-2 06-2009 Department_1 Product_1 Product_2 Facility_1

Table 4.7: Example of Patterns based on relations between chosen objects and other complex-nodes.

Algorithm 4: Relation Construction Algorithm.

```
Data: $S := (N_{cn}, R)$, $IS := (N_{Icn}, R_I)$
Result: Relation set for the object graph $R_o$
1  begin
2      Create the set $\Phi$;
3      $\Phi = \{r \,|r := \langle r_n, CN_s, CN_d\rangle, r \in R \cap ((CN_s \cup CN_d \in O_I) \cap (CN_s \cup CN_d \notin O_I))\};$
4      foreach $r \in \Phi$ and $r_n ==$ "$IS - A$" do
5          Add $CN \notin O_I$ in $O_I$;
6      foreach $r \in R$ do
7          if $CN_s \in O_I \cap CN_d \in O_I$ then
8              Mark(r);
9              foreach $ri \in R_I$ do
10                 if $ri.isinstance(r)$ then
11                     add $ri$ to $R_o$;
12             SearchSameRelation(r);
13         if $r \in \Phi$ then
14             if $CN_s \in O_I$ then
15                 if $rn ==$ "$Part - of''$ then
16                     create $\Gamma$ and $\Psi$;
17                     $\Gamma = \{CN \,|CN\ has\ the\ relation\ R :=< \text{“}Part - of'', CN, CNd >\}$
18                     $\Psi = \{r|\, r :=< Same\_(CN_k.name), CN_{is}, CN_{js}, CN_k >, i \neq j, CN_k \in \Gamma$
19                     add $\Psi$ to $R_o$;
20                 else
21                     AddAttribute(CN$_s$, $CN_k$); $SearchSameRelation(r, CN_s, CN_d)$;
22             if $CN_d \in O_I$ then
23                 AddAttribute(CN$_d$, $CN_s$); $SearchSameRelation(r, CN_d, CN_s)$;
24     Treat-UnlinkedObject();
25     SearchSameRelation(r,CN$_s$, $CN_d$) begin
26         foreach $ri := \langle ri_n, CN_{is}, CN_{id}\rangle$ instance of $r$ do
27             Mark(ri);
28             foreach $rj$ instance of $r$ and $rj.unmarked$ do
29                 if $CN_{id} == CN_{jd}$ then
30                     create $rsame := \langle Same\_(CN_d.name), CN_{is}, CN_{js}\rangle$;
31                     add $rsame$ to $R_o$;
```

4.4.3.2 Relations Construction from the Enterprise Ontology

The enterprise ontology includes relations between the enterprise objects which are extracted from enterprise data. These relations between the concepts can reflect real relations between the enterprise objects which are not described on the relational database.
For example, in Figure 4.8, the relation "Leader-of" between the concepts *Employee* and *Project* is not specified on the *GO* (SPIDER-Graph on the right of the figure). Also, the Employee "Alain" which is an instance of the concept *Employee* and the project "Trans-2" which is an instance of the concept *Project* have inherited this relation.
However, these two instances exist on the *GO* (Project_1 and Employee_1). Indeed, the relation $R := \langle ''Leader-of'', Employee\_1, Project\_1 \rangle$ can be added to the graph *GO* and can enrich it.
In this section we describe the process to add relations from the enterprise ontology *O* to the object graph *GO*.

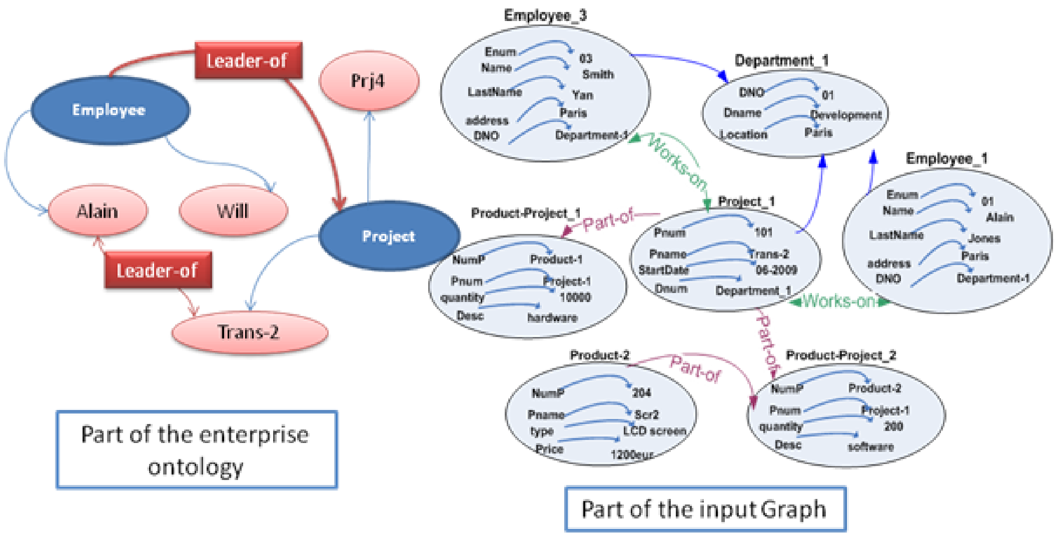


Figure 4.8: Example of Relations Added From the Ontology.

Algorithm 5: Ontology relations extraction

```
Data: $C_{user}$, the set of the relation in the ontology $OP$, the set of
      identified object $O_I$
Result: relations to add in $R_o$
1  begin
2      foreach $c_i \in C_{user}$ do
3          if $\exists r_o \in OP \cap i \neq j \,|r_o(c_i, c_j)$ then
4              search_instance_relation $(r_o)$
5              if $\exists r_o \in OP \,|r_o(c_{i_i}, c_{j_i})$ then
6                  $CN_i = search - instance - CN(c_{i_i}, M_{OS})$
7                  $CN_j = search - instance - CN(c_{j_i}, M_{OS})$
8                  if $CN_i \neq null \cap CN_j \neq null$ then
9                      Create $r = r_o$
10                     Add $r \in R_o$
```

The proposed algorithm 5 can be described by these steps.

Step 1. *Search for the existing relations between the chosen concepts.* As a first step, the user has chosen a set of concepts to search the input graph from the enterprise ontology O. This process can be described as a matching between the ontology and the SPIDER-Graph This matching between the ontology and the SPIDER-Graph is denoted by M_{OS} where $M_{OS} := \{m \,|m := (c, CN), c \in C, CN \in S\}$. With c is a chosen concept and CN is a complex-node

If a concept c is matched against at least one complex-node then it is added to the set of identified concepts C_{user}. The set of identified concepts C_{user} can already share relations among them on the ontology. Then these relations are searched (line 3). If there is a relation $r \in OP$ between two concepts of C_{user}, we go to the next step.

For instance, if the C_{user}=\{ *Employee*, *Project* \}, then the matching between the ontology and the SPIDER-Graph is $M_{OS} := \{|m1 := (Employee, Employee), m2 := (Project, Project)\}$ In Figure 4.7, the two concepts *Employee* and *Project* share the relation "Leader-of". In this case, the algorithm goes to the step2.

Step 2. Search the existence of instance relations. (line 4) Each concept $c \in C_{user}$ can have a set of instances I on the ontology. If two concepts share a relation on the ontology, we search for the instances of these two concepts which share the same relation.

For the concept *Employee*, we detect two instances $I1 := Alain$ and $I2 := Will$ and only one instance $I1 := Trans - 2$ for the concept *Project*.

Step 3. Search the instance relations on the GO (Line 5 $\longrightarrow$ 7). In order to find the corresponding complex-nodes instances to a concept instance we use the set M_{OS}.

If $R_i =: \langle n, c_{ii}, c_{jj} \rangle$ is an instance relation of the relation $R := \langle n, c_i, c_j \rangle$ where

$ci, cj \in C_{user}$, we search for the corresponding complex-node instances to c_{ii} and c_{jj}.
Indeed, for each $m_k := (c_k, CN_k) \in M_{OS}$ where $c_k = c_i$ or $c_k = c_j$, we calculate the similarity between c_{ii} or c_{jj} and the CN_k instances.
The similarity between a complex-node instance and a concept instance is calculated via this formula:

$$sim(CN_I, c_i) = \frac{\sum_{a_c \in DPc} \sum_{a_{cn} \in A_{cni}} sim(a_c, a_{cn})}{|DPc| \times |A_{cni}|}$$

where:

- DPc is the set of datatype properties related to the instance c_{1i},
- A_{cni} is the set of attributes related to the complex-node instance CN_{Ij},
- $sim(a_c, a_{cn}) =$
  $\frac{1}{2}(sim(a_c.name, a_{cn}.name) + sim(a_c.value, a_{cn}.value)) =$
  $\frac{1}{2}(SimJaroWinkler(a_c.name, a_{cn}.name) + SimJaroWinkler(a_c.value, a_{cn}.value))$,
  The used string similarity measure is the Jaro-Winklere measure [Cohen 2003].

If $sim(CN_I, c_i)$ is greater than a threshold value δ then CN_I is the corresponding complex-node instance to c_i.
The used similarity takes in consideration both the value and the name of an attribute. Thus, in $sim(CN_I, c_i)$, we use Jaro-Winklere. It calculates the similarity between two attributes by calculating the average of similarities between their names and their value.
Step 4. Add new relations to the object graph GO (link $8 \longrightarrow 10$). If we find the corresponding complex- nodes instance to the two instance concepts , we link the two complex-node by this new relation.

4.4.4 The Process Performance

The graph extraction process depends on the sequential execution of three algorithms: the object identification (the algorithm 3), the relation extraction from the relation database (the algorithm 4) and the relations extraction from the ontology (the algorithm 5).
The object identification process depends on the SPIDER-Graph size and the number of chosen concepts. The first step (which is the name treatment) is executed $(N + M)$ times , where N the number of complex-nodes and M the number of concepts.
The string matching process (candidate extraction), which is the second step, is executed in the worst case $N * M * Complexity(sim(T_{CN}, T_c))$ (where T_{CN} and T_c are the set of tokens of a complex-node CN and a concept c, respectively) . However, the complexity time of $sim(Tcn, Tc)$ is based on the extracted tokens

numbers which in general cases cannot exceed 10 tokens. Then, the complexity of this step can be $o(N * M)$.
Finally, in order to analyze the candidate complex-nodes (the $CAND$ set), the algorithm traverses all the complex-nodes N times and for each concept (M times), calculates the *Simatt*. *Simatt* measure depends on the complex-nodes attributes size and the concept attributes size. The complex-node attributes number have a maximum size depending on the relational database management systems: PostgreSQL 250-1600, Oracle 1000, Acess 250. Also the concept attribute does not exceed 30 attributes. Then, the algorithm complexity is in the range of $o(N * M)$.
The object identification can have the complexity time $o(N)$ because in general a user prefers to search for the interaction between two or three objects.
The relation extraction using the patterns depends on the SPIDER-Graph relations number on the schema graph n and the instance graph m. In the worst case, this algorithm can be executed on $n * m_2$ time. However, the process of Search-SameRelation can be accelerated if we store the input SPIDER-Graph relation on a HashMAP having a double key (MultiKeyMap).

The relations extraction from the ontology process depends on the chosen concepts instances n and the identified object instances m. Then, the complexity of this process can be considered $n*m*C(sim(CN_i, cn_i))$. The time complexity of the similarity measure $sim(CN_i, cn_i)$ depends on the attribute numbers of the complex-nodes and the concept (the same case as in the object identification process which allows to consider that the process complexity is $n * m$.
Finally, we can consider the graph extraction process is in the worst case $n*m_2$ and $n * m$ by using the MultiKeyMap.

4.4.4.1 Examples of Extracted Graph

In this section, we present two examples of extracted graph from the SPIDER-Graph presented in Figure 4.6: the first interaction graph is between *Project* and *Employee* and the second between *Employee* and *Product*. Each extracted graph represents a new user view of the input SPIDER-Graph.
A. The interaction graph between *Project* and *Employee*
First, the object identification process is performed. In this case, the identification is simple because we can detect the chosen objects directly from their names. Then all the instances of the complex-nodes *Employee* and *Project* are added to the set O_I (O_I = *Employee_1, Employee_2, Employee_3, Project_1, Project_2*).
The relation construction process is then performed using the O_I set. We apply the patterns on the set of existing complex-nodes relations using the following steps:
Step 1. Identify new objects using the "IS-A" relation. In our example, the process identifies "Manager" as a new object to add in the set O_I due the relation $R =<$ "$IS - A$", $Manager, Employee >$.
Step 2. Add the existing relations between identified objects by applying Pr_1. In the database schema, *Employee* and *Project* share the relation $R_1 :=<$

$``", Employee, Project >$. Then, R_1 is added to the set R_IO.
Step 3. Create new relations and new attributes using the predefined patterns. From the existing relations between *Project*, *Employee* and other complex-nodes, we have detected many new relations which are summarized in table 4.8.
Step 4. Enrich the graph with the enterprise ontology. In this step, we use the ontology described in Figure 4.8. Using the same steps in the section 4.4.3.2 , the relation $R =< Leader - of, Employee_1, Project_1 >$ is added to the graph. The result graph is presented in Figure 4.9.

Table 4.8: Relations between Projects and Employee

Initial Relation	**Pattern**	**Created relation**
R=⟨“works-on", Employee, Project>	Pr$_2$:=⟨ Same_Project, Employee_i , Employee_j> and $i \neq j$	R$_O$:=⟨ Same_Project, Employee_3 , Employee_1 > R$_O$:=⟨ Same_Project, Employee_3, Employee_2 >
R=⟨“ ", Employee, Department>	Pr$_3$:=⟨ Same_Department, Employee_i,Employee_j> and $i \neq j$	R$_O$:=⟨ Same_Department, Employee_3, Employee_1>
R=⟨“Part-of", Project, Project-Product >	Pr$_4$:=⟨Same_Product, Project_i, Project_j> and $i \neq j$ -add the attribute ⟨ Product, Product_i> to *Project*	R$_O$:=⟨Same_Product,Project_2, Project_1 > -new attribute ⟨ Product,Product_1> in *Project*_2. -new attribute <Product,$\{Product\_1, Product\_2\}$> in *Project*_1
R=⟨“ ",Facility, Project>	*Facility* is not related with other object then we have no pattern. - the attribute $\langle Facility, Facility\_i\rangle$ is added to *Project*	-new attribute $\langle Facility, Facility\_1\rangle$ in *Project*_1

B. The interaction graph between *Product* and *Employee*
The previous steps are repeated and the extracted relations are summarized in table 4.9. However, in this example, the two objects are not directly connected in the initial graph. The dotted edge means that the two objects are indirectly related. Indeed, *Employee* and *Product* are related by the semi-path:⟨*Employee*, *Project*, *Product-Project*, *Product*>, then the Pr_{path} can be applied. The result graph is presented in Figure 4.10.

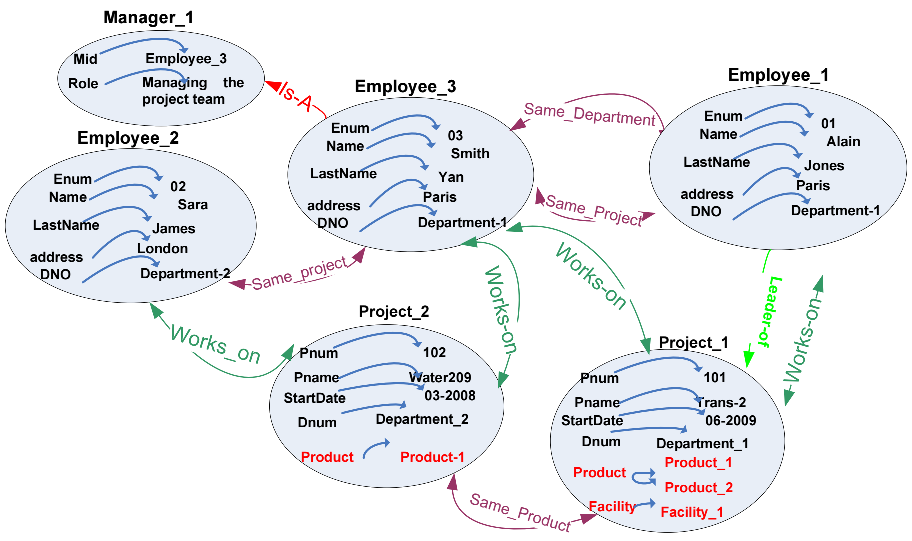


Figure 4.9: The Employee-Project Graph.

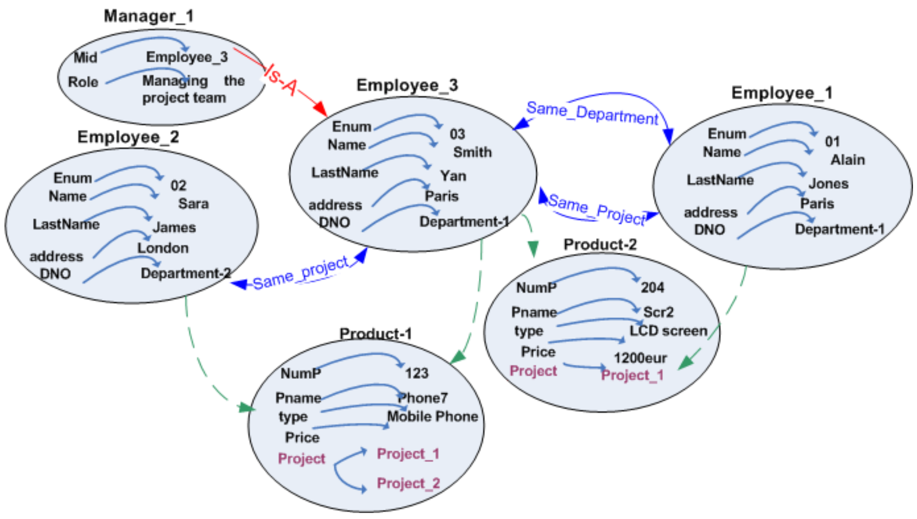


Figure 4.10: The Employee-Product graph.

4.5 Summary

In this section, we discuss our interaction object graph extraction against the graph extraction approach described in the chapter 2:

-Attributed Typed Graph transformation Approach : the graph transformation approaches can transform an input graph to a new graph by following a set of predefined rules. For example, in Figure 4.11, we can add the relation **same_product** by applying a Pushout technique (detailed in section 2.2.3.2).

Table 4.9: Relations between Product and Employee.

Initial Relation	Pattern	Created relation
R= ⟨“works-on", Employee, Project>	Pr$_2$:=⟨Same_Project, Employee_i, Employee_j> and $i \neq j$	R$_O$:=⟨Same_Project, Employee_3 , Employee_1> R$_O$:=⟨Same_Project, Employee_3, Employee_2>
R= ⟨“ ",Employee, Department>	Pr$_3$:=⟨Same_Department, Employee_i,Employee_j> and $i \neq j$	R$_O$:=⟨Same_Department, Employee_3, Employee_1 >
R= ⟨“Part-of", Product, Project-Product>	Pr$_4$:=⟨Same_Project, Project_i, Project_j> and $i \neq j$ -the attribute < $Project, Project\_i$ > is added to Product	-no *Products* are from the same *Project*. -a new attribute $\langle Project, Project\_1 \rangle$ is added to *Product_2* -a new attribute ⟨Project, $\{Project\_1, Project\_2\}$> is added to *Product_1*

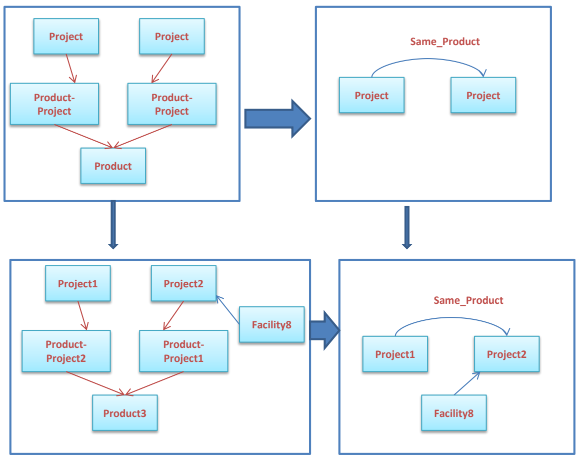


Figure 4.11: Attributed Typed Graph Transformation : rule to add a new relation.

However, this technique is based on predefined rules that control the graph transformation. Indeed, these rules specify all the elements to change or to add. The task of rule definition is manually and the user may not detect all the important relations. For instance, to obtain the interaction graph between Product and Employee, we should write manually a rule for each new relations or attributes. Nevertheless, the path relation cannot be obtained with the graph transformation because it is based

on a shortest path algorithm. Another important point is that these approaches cannot transform directly the relational schema to a graph schema. It can only process homogeneous schema.
-Keyword search : In order to search a relation between objects, these approaches can transform the relational database to a simple graph. For example, the relational database presented in Figure 4.4 can be transformed to the simple graph shown in Figure 4.12.

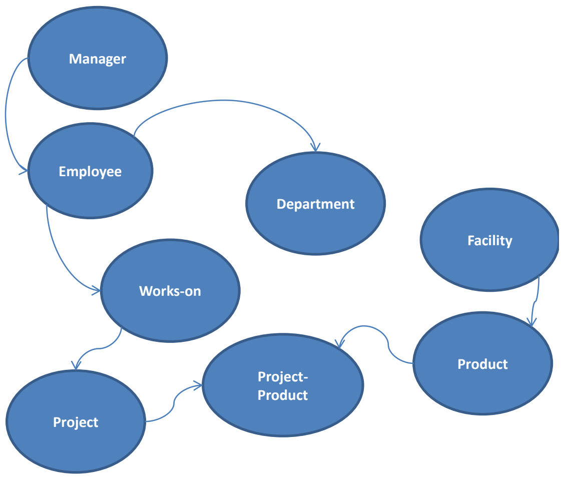


Figure 4.12: Graph in Keyword Search Approach.

However, these approaches can only detect if there is a relation between objects and cannot create new relations between them. For instance, we cannot find explicitly the relation "Same_Product"' but we can find that there is a link between two projects.

4.6 Conclusion

In this chapter, we have presented a new approach to extract object interaction graphs from relational databases. Object interaction graph can represent the enterprise objects with a dynamic way and highlight the shared relations between and within objects. In order to model these extracted graphs, we have proposed a new graph model SPIDER-Graph which is presented in section 4.2. The SPIDER-Graph is based on these:

- Separation between the instance graph and the schema graph: A SPIDER-Graph is composed of two graphs. The schema SPIDER-Graph, which can be considered like a meta-model or a DTD for the graph, describes the types of the nodes, their relations, the attributes in each node and their types. The instance SPIDER-Graph contains the concrete elements (nodes labeled by object identifier, attributes with values and their relations) according to the schema.

- Heterogeneous elements: SPIDER-Graph allows to have complex-nodes with heterogeneous types and attributes.
- Complex-nodes and objects referencing: the SPIDER-Graph represents the objects via the complex-nodes. A complex-node encapsulates all the object attributes in the same node. An attribute in a complex-node can reference another complex-node.

Thereby, the SPIDER-Graph can preserve the relational database specificities: the heterogeneous relational database tables can be modeled as separated complex-nodes and the foreign keys can be mentioned by the use of complex attributes (attributes that reference other complex-node).
In the section 4.3, we have detailed the approach to transform the relational database into a SPIDER-Graph. This transformation has presented the relational data as a set of enterprise objects related by a variety of relations. In this level, all the objects are extracted and the detected relations can be classified in four categories (Association relations, Dependence relation, "IS-A", "Part-of" relation). In the section 4.4, the object interaction graph extraction approach is described. This approach used as input the extracted SPIDER-Graph from the relational database and the enterprise ontology described in the previous chapter. Firstly, based on the enterprise concepts that the user selects from the enterprise ontology, an object identification process is performed in order to detect all the complex-nodes that correspond to this choice. Secondly, we have described the relation between the selected objects extracted process in 4 . Indeed, we have used a set of relation patterns to extract the hidden and existing relations on the input SPIDER-Graph. This step allows to discover additional information between the selected objects; e.g. the employees who work in the same project via the relation "Same-project". Then, we have added additional relations from the enterprise ontology using a schema mapping process.

The extracted graph is a SPIDER-Graph containing information about the interactions between user's selected object from the structured and unstructured enterprise data. We reckon that the extracted graph is rich enough to help business user in their analysis. For example, the graph of Employee and Product in the Figure 4.10 can help to find the team to put in a new project. Then, we need an efficient way to analyze and query this kind of graph in order to have the good result. Thus, in the next chapter, we present a new visual query language adapted to query and analyze the SPIDER-Graph and other enterprise graphs.

Chapter 5

GraphVQL: Graph Visual Query Language

Contents

Querying graphs is an important task which allows accessing and analyzing graph data. Visual query languages are the most intuitive and easy way to query graph for a non-expert user. In this chapter, our graph visual query language is presented (GraphVQL), dedicated to heterogeneous graphs that supports different types of queries. GraphVQL is a query language for a SPIDER-Graph model and also for heterogeneous graphs modeled with RDF or Graphml. In the first section, we present a global view of GraphVQL. Then, we describe the different processes performed by the language: query treatment (section 5.2), SPARQL conversion (section 5.3) and SPIDER-Graph query treatment (section 5.4).

5.1 GraphVQL Description

In the literature, different visual query languages have been proposed (see section 2.2.2.1). However, these languages are specific to a particular graph data model (graph database model, RDF or simple graph). Also, several queries types are not treated or treated partially, for example: The aggregation queries, social network

analysis queries and path queries. In this context, we propose a visual query language which permits to query the SPIDER-Graph model, RDF or GraphML graphs, called GraphVQL (**Graph Visual Query Language**). The user can query not only the extracted graph from the relational database but also the other existing graphs modeled with various graph data model. This language covers different query types from simple selection query to social network analysis queries.
GraphVQL is based on a pattern matching process to extract subgraphs from input graphs. Indeed, the user draws a query using a set of predefined symbols. This drawn graph is transformed into a graph-pattern query used to search the result in the input graph. The visual query language result is a set of sub-graphs extracted from the input graph. The graph search process changes with the input graph type. In the case of a SPIDER-Graph, GraphVQL uses a pattern matching algorithm to find the sub-graphs corresponding to the select query drawn by the user, and uses a shortest path algorithm to find a response to a path query. In the case of RDF or GraphML graph, GraphVQL uses a set of mapping rules to transform a graph pattern query to a SPARQL query which is then executed to find the result. Figure 5.1 depicts an overview of GraphVQL main processes.

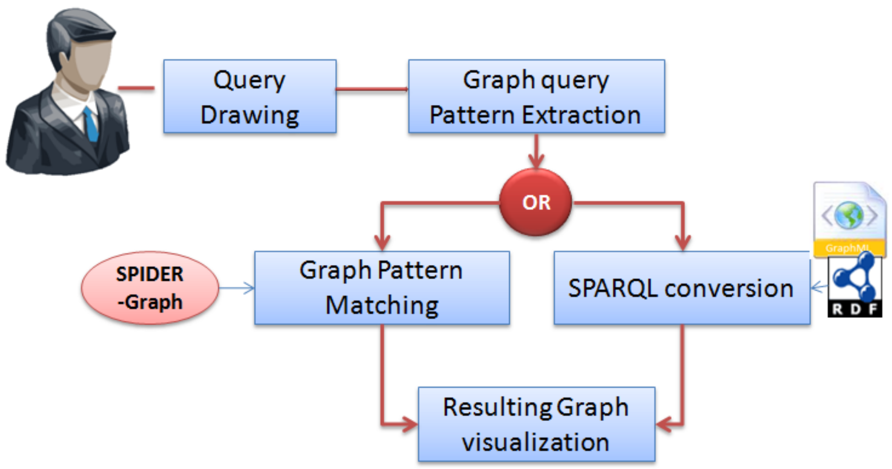


Figure 5.1: GraphVQL Architecture.

The details in each process is described below.

5.2 GraphVQL Visual Notation and Grammar

GraphVQL allows the user to draw different kind of queries using predefined symbols. These queries can be:

- Selection query: This type of query can contain condition on nodes, on attributes and/or on edges. It allows to select all the data that correspond to the drawn pattern.

- Aggregation query: Select the data, then apply an aggregate function (MAX,MIN,AVG,...) on a specific object or attribute.
- Path search query: This type of query can find a path between two specified objects.
- Analyze query: This type of query analyzes the position of a specific object on a graph using the social network metrics.

In this section, we describe how the user can perform queries using GraphVQL. First, the basic visual notation of the language is introduced, then the syntax of GraphVQL is detailed, as well as the process of graph pattern extraction.

5.2.1 Basic Visual Notation

The basic elements of GraphVQL can be divided into four categories (see table depicted in Figure 5.2) :

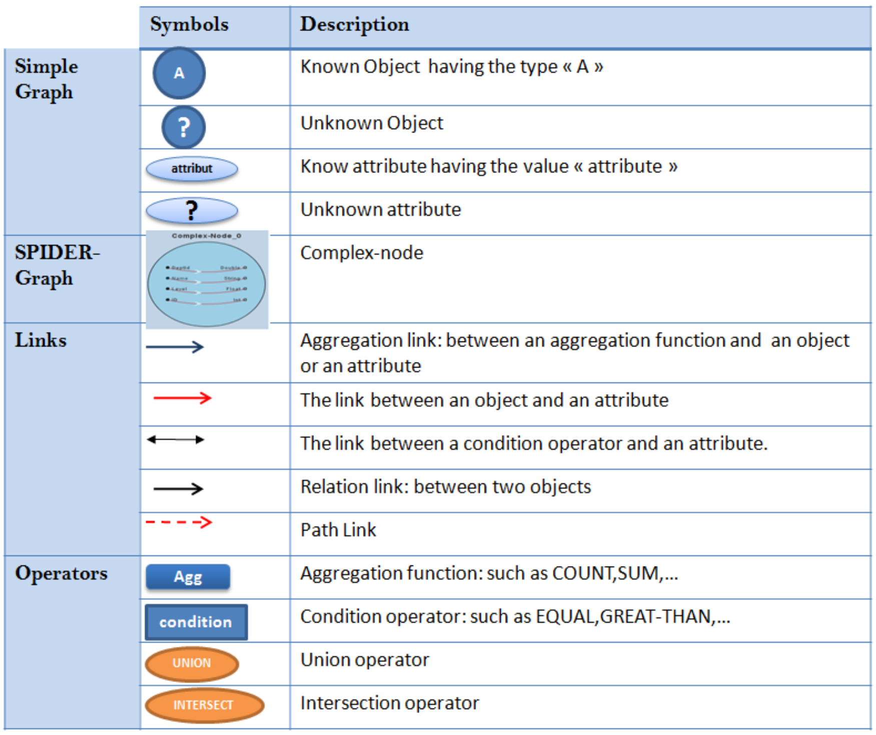
- Simple graph elements: These symbols represent nodes of simple graphs (hereafter, RDF and GraphML graphs will be called simple graph). Attributes of each node are represented using the diamond symbol.
- SPIDER-Graph elements: contain the complex-nodes that compose a SPIDER-Graph.
- The Links category regroups all the different links that a graph can contain: link between two objects (complex-node or simple node), link between objects and their attributes (for simple graph), aggregation operator link which can link an operator to an object or an operator to an attribute, condition link which links an attribute or an object to a condition. Finally, a path link which links the objects which the user would like to search the path.
- The operator category contains the operators symbols: aggregation, condition, union and intersection to explicit more complex queries.

The GraphVQL queries are composed by the previous symbols following predefined rules.

5.2.2 GraphVQL Syntax

In the following, we present the grammar rules designed with the help of EBNF [Scowen 1998]. The used meta symbols on our grammar are the following:
(. . .): Grouping
|: alternative
{. . .}: repetition zero or more time

The grammar rules are the following:



	Symbols	Description
Simple Graph	A	Known Object having the type « A »
	?	Unknown Object
	attribut	Know attribute having the value « attribute »
	?	Unknown attribute
SPIDER-Graph	Complex-Node_0	Complex-node
Links	→	Aggregation link: between an aggregation function and an object or an attribute
	→	The link between an object and an attribute
	↔	The link between a condition operator and an attribute.
	→	Relation link: between two objects
	⇢	Path Link
Operators	Agg	Aggregation function: such as COUNT,SUM,...
	condition	Condition operator: such as EQUAL,GREAT-THAN,...
	UNION	Union operator
	INTERSECT	Intersection operator

Figure 5.2: GraphVQL Symbols.

1. Query ::=$(SPIDERQuery|GraphQuery)$
2. SPIDERQuery ::=$(SelectSPIDERQuery\,|PathSPIDERQuery\,|GraphAggregation)$
3. SelectSPIDERQuery ::= Complex-node $\{Relation\ Complex-node\}$
4.PathSPIDERQuery::= Complex-node PathLink Complex-node
5. GraphQuery ::= (SelectQuery|AggregationQuery|AnalyseQuery)
6. SelectQuery::= Object $\{Relation\ Object\}$ $\{OPERATOR\ SelectQuery\}$
7. AggregationQuery::= Aggregation SelectQuery
8. AnalyseQuery::=(PathQuery |SNAQuery)
9. PathQuery::= Object PathLink Object
10.SNAQuery::= Metric Object
11. Complex-node::= Type $\{AttributeName\ AttributeValue2\}$
12. Object::=Type $\{AttributeName\ AttributeValue\,\{Condition\}\}$
13. Condition::= ConditionOperator ConditionValue Conditionlink
14. ConditionOperator::=(DIFFERENT|GREAT_THAN|GREAT_EQUAL|LESS_THAN |LESS_EQUAL | EQUAL|CONTAINS |GROUPBY|LIMIT|OFFSET)
15. ConditionValue::= String
16. Conditionlink::="condition"
17. Aggregation::= AggregationOperator Aggregationlink
18. AggregationOperator::=(MIN|MAX|AVG|SUM|COUNT)
19. Aggregationlink::="aggregation"
20. AttributeName::=String

21. AttributeValue::=String
22. AttributeValue2::= Type|String
23. Type::=String
24. OPERATOR::= (OPERATORlink) (OPERATORFUNCTION) (OPERATORlink)
25. OPERATORFUNCTION::=(UNION| INTERSECTION)
26. OPERATORlink::="operator"
27. Metric::=(*Degree* |*Betweenness* |*Closeness*)
28. Relation::=RelationName
29.RelationName::=String
30.GraphAggregation::=Complex-node {Relation} {AttributeName}

Now, we detail each rule of the grammar. By its first rule, GraphVQL queries can only be a query for SPIDER-Graph or for simple graphs.

$$Query ::= (SPIDERQuery|GraphQuery)$$

5.2.2.1 Simple Graphs Queries Syntax

In the case of a simple query, GraphVQL proposes various query types.

$$GraphQuery ::= (SelectQuery\,|AggregationQuery\,|AnalyseQuery)$$

-The *selection query* for simple graph is specified by the sixth rule :

$$SelectQuery ::= Object\,\{Relation\ Object\}\,\{OPERATOR\ SelectQuery\}$$

The selection query can contain conditions on the objects and the attributes to search; such as the query in Figure 5.3 where the user add a condition to the project name.

The query on simple graphs can be composed by two sub-queries linked by an

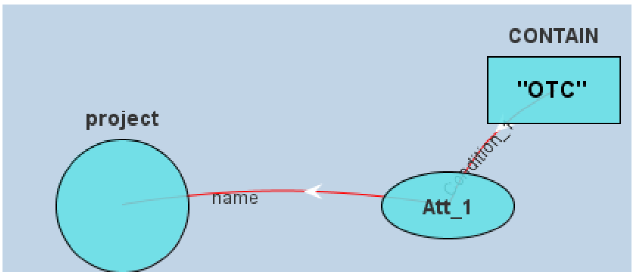


Figure 5.3: Selection query using condition on attribute.

operator UNION or INTERSECTION ; such as the query in Figure 5.4.

The *Aggregation query* (Figure 5.5) on GraphVQL can apply an aggregation operator to an attribute or to an object.

AggregationQuery::= Aggregation SelectQuery

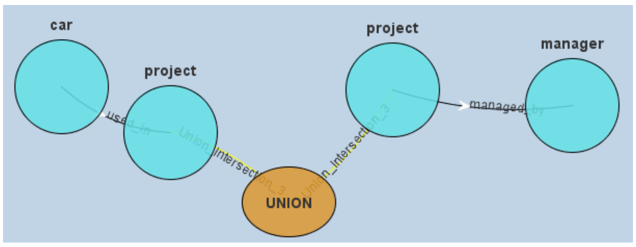


Figure 5.4: Composed selection query

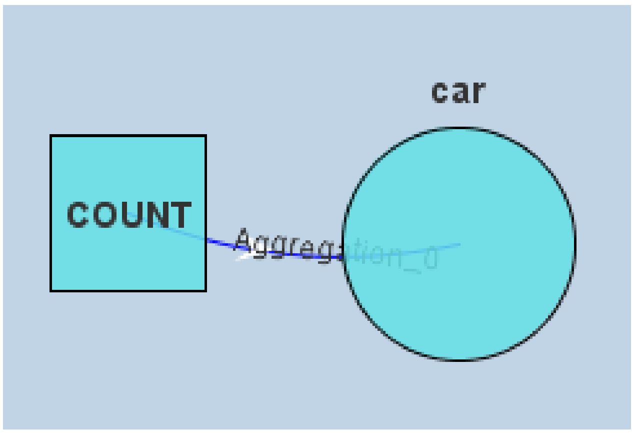


Figure 5.5: Aggregation query

The *analysis query* can be a path query or a social analysis query. The Path query can find a path between two defined objects.

PathQuery::= Object PathLink Object

The analysis query applies a social network measure to a defined object.

SNAQuery::= Metric Object

5.2.2.2 SPIDER-Graph Queries Syntax

GraphVQL queries for SPIDER-Graphs can be select, path or aggregation queries :

$$SPIDERQuery ::= \\ (SelectSPIDERQuery\,|PathSPIDERQuery\,|GraphAggregation)$$

In the selection queries the query should contain at least one complex-node with a defined type:

$$SelectSPIDERQuery ::= Complex - node\,\{Relation\ Complex - node\} \\ Complex - node ::= Type\,\{AttributeName\ AttributeValue2\}$$

The same rule mentions that the query can contain relations between a set of complex-nodes. For example, in Figure 5.6 the user search for all employees working in the same project. This query contains three complex-nodes (two Employees and a Project) linked by two relations:

SelectSPIDERQuery ::= Employee1 (works-on Project1) (same-Project Employee2)
Employee1::=Employee
Employee2::= Employee
Project1::= Project

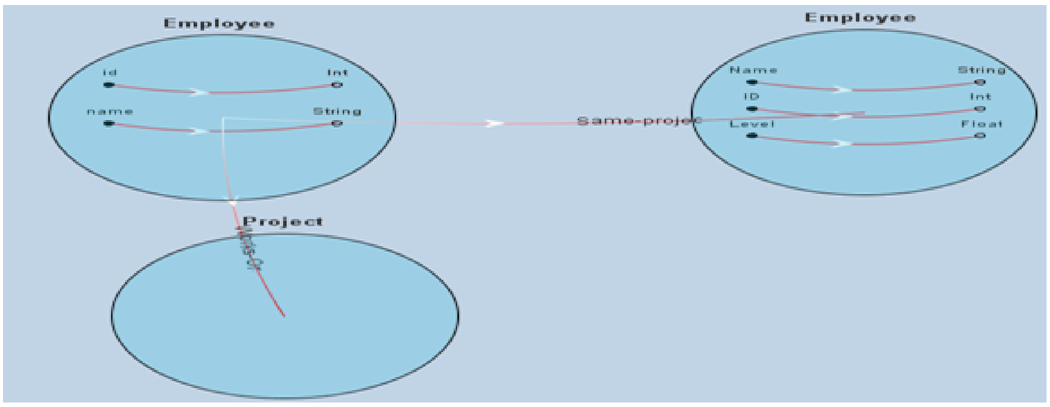


Figure 5.6: Selection queries on SPIDER-Graph.

By the fourth rule, GraphVQL allows to find paths between complex-nodes on the SPIDER-Graph. The path can be a simple path, which is a path between a start complex-node and an end complex-node, or a path indicating the set of complex-nodes that should be found in the final path:

$$\text{PathSPIDERQuery::= Complex-node } \{PathLinkComplex - node\}$$

For example in Figure 5.7, the query is a path query between two specific objects: Employee and Product.

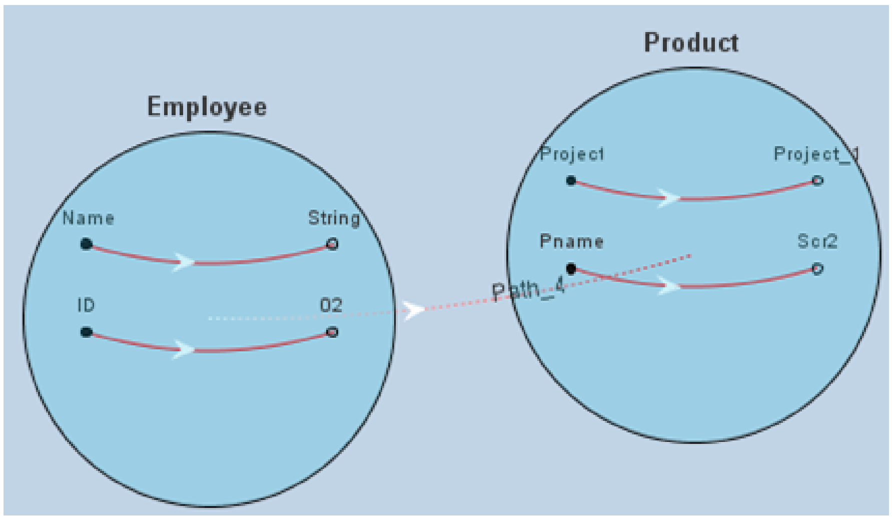


Figure 5.7: Path query on SPIDER-Graph.

GraphVQL allows to aggregate SPIDER-Graph by using an aggregation algorithm K-SNAP [Tian 2008]. The aggregation takes in input a complex-node type and the relations and attributes with which the summarization algorithm will decompose the graph in groups.

GraphAggregation::=Complex-node {Relation} {AttributeName}

5.3 Simple Graph Queries Treatment

The simple graph uploaded by a user can be a RDF or a GraphML graph. These graphs are queried with queries composed by simple graph elements. In this section, we detail the process of simple graph queries treatment. The process starts by extracting the pattern graph query from the drawn graph (section 5.3.1). Then, the extracted graph pattern query is transformed to a SPARQL query (section 5.3.2). The query pattern translation process to SPARQL query differs from query type to another.

5.3.1 Graph Pattern Query Extraction

A GraphVQL query drawn by the user is a graph composed by nodes and edges having different types and roles. This graph query can be defined formally by the definition:

Definition 21 (Graph Query) *A graph query GQ, in GraphVQL, is defined by* $GQ := (V, E)$ *where:*

- V *is the nodes set such as* $V := \{v\,|v := (t, n, A)\}$ *, where:*

  - *t is the node type which can be: Object, Attribute, Aggregation, Condition, Operator;*
  - *n is the value of the node;*
  - *A is the the attributes set attached to the node. The attributes set A is taken in consideration only if t is an Object.* $A := \{a\,|a = (n_a, v_a)\}$ *where* n_a *is the attribute name,* v_a *is the attribute value.*

- *E is the edges set such as* $E := \{e\,|e := (t, e_n, v_{start}, v_{End})\}$ *where:*

  - *t is the link type which can be: object relation (relation between two objects), Attribute link (between object and attribute) , Aggregation link, Condition link, Operator link;*
  - e_n *the name of the link;*
  - v_{start} *and* v_{end} *are respectively the start node and the end node which share the link.*

For example, the graph query $GQ1$ extracted from the drawn query in Figure 5.3 is composed by the following elements:

- V={v$_1$ $:= (Object, ”Project”, A_{Project}), v_2 := (Attribute, null), v_3 := (Condition, ”OTC”)\}A_{Project} = \{a := (”name”, null)\}$

- E={e$_1$ $:= (Attributelink, ”name”, v_2, v_1), e_2 := (Conditionlink, Contains, v_3, v_2)\}$

From this graph query, our approach extracts the corresponding graph-pattern query from the drawn query following the defined grammar. The graph-pattern query specifies the query type and elements (the different objects, the applied conditions and aggregations and the used operators). It facilitates the work of query processing. A graph query pattern is defined recursively by:

Definition 22 (Graph Pattern Query) *(1) A graph query pattern P can be a:*
-Basic graph query pattern defined by $BP := (G, C, AG)$
- Group graph query pattern defined by $GP := \{P\ op\ P \mid P = BP \cup GP$ *where* $op \in \{AND, UNION, PATH\}$
(2) If P1 and P2 are graph query patterns, then the expressions (P1 AND P2), (P1 PATH P2), and (P1 UNION P2) are graph queries patterns.

The graph pattern query can be presented in the following grammar:

$$P ::= T\ BP\,|”(”GP”)”$$
$$GP ::= P”AND”P|P”UNION”P|P”PATH”P$$

T is the query type which can be Select, Aggregation, Path or Analyze. Now, we define the basic pattern query by the following definition.

Definition 23 (Basic Pattern Query) *A basic pattern query BP is defined by* $BP := (G, C, AG)$ *where:*

- *G is the pattern graph* $G := (O, R)$ *where:*
  - *O is the objects set which the user would like to see in the result* $O \subseteq V$*;* $O := \{o \,|o := (u, n, A)\}$ *with u is a Boolean attribute which mentions if o is a known object or an unknown one, n is the object name, A is the attributes set attached to the object.*
  - *R is the set of object relations where* $R := \{r \,|r := (r_{name}, o_{start}, o_{end})\}$ *with* o_{start} *and* $o_{end} \in O$
- *C is the conditions set;* $C := \{c \,|c := (a, cop, s)\}$ *where a is the attribute of the condition, cop is the used operator (which can be DIFFERENT, GREAT_ THAN, GREAT_EQUAL, LESS_ THAN, LESS_EQUAL, EQUAL, CONTAIN) and s is the value of the condition.*
- *AG is the set of aggregations where:* $AG := \{ag \,|ag := (t_{ag}, o, a)\}$ *where* t_{ag} *is the aggregation name (which can be SUM, AVG, MIN, MAX, COUNT, GROUPBY), o the object on which the aggregation is applied, a the attribute to aggregate*

We take the previous example of Figure 5.3. The graph pattern query process can extract from its graph query $GQ1$ the following graph pattern query: P1:=Select BP1. Where BP1:=(G1,C1,AG1)=

- $G1 = \{O := \{o1 = (true, project, A_{Project})\}\}$
- C1={c=(a,CONTAIN,"OTC")}
- AG1=null

Indeed, the input query contains one objet o1 ,which is "Project" having the attribute "name" designed by a, and the condition c1.

In order to extract these patterns, we have proposed a graph query pattern extraction process which is performed by algorithm 6.

Algorithm 6: ExtractionPattern

```
Data: The user query $GQ := (V, E)$
Result: The pattern query $P$
1  begin
2      if $GQ.containsPathObject()$ then
3          $P.Tp = $ “$Path$”
4      else if $GQ.containsAggregationObject()$ then
5          $P.Tp = $ “$Aggregation$”
6      else if $This.SNA = True$ then
7          $P.Tp = $ “$SNA$”
8
9      else
10         $P.Tp = $ “$Select$”
11     if GQ. contains(UNION) or GQ. contains(AND) or GQ.
       containsPathObject() then
12         $OP = ExtractOperator(GQ)$
13         $left\_graph = Extractleftgraph(GQ)$
14         $right\_graph = Extractrightgraph(GQ)$
15         $left\_pattern = ExtractionPattern(left\_graph)$
16         $right\_pattern = ExtractionPattern(right\_graph)$
17         $P = left\_pattern + OP + right\_pattern$
18     else
19         $P = ExtractionBasicPattern(GQ)$
20     return P
```

The proposed algorithm 6 takes the graph query *GQ* as input and returns a query pattern *P*.
In the beginning, the query type is detected by analyzing the used objects in *GQ* (line 2→10). If the *GQ* contains an edge having the type path (i.e. $\exists e \in E \,|e.t = Path)$ then the query has the type "Path" or if it contains an aggregation link the query has the type "Aggregation". If the user wants to analyze an object the query type will be "SNA". Indeed, the "SNA" query is a special query which is performed not by putting a special symbol on the *GQ* but by forming a selection query. The result will be analyzed by the mean of a social network analysis metrics. Finally, in all the other cases, the query is a select query and its type is "SELECT".
For example, if the query in Figure 5.3 does not contain an aggregation or a path link then it has the type "SELECT".
After the query type detection, the extraction process algorithm extracts the correspondent pattern to the input query (line 10→18). Second, the extraction process is performed (line 10→18). In the case of composed query (query containing an UNION or AND operator) or a path query, the initial query *GQ* is divided, recur-

sively, in multiple basic graph queries which do not contain operators. The methods *Extractleftgraph()* and *Exractrightgraph()* allow to extract the left part and the right part of a graph query, each containing an operator. Each basic graph query is treated independently in order to extract its correspondent basic graph query pattern. For example, the query in Figure 5.4 is an "UNION" query. The proposed algorithm divides this graph query to two sub-graphs: The left part which is composed by (Car, used-in, Project) and the right part which is composed by (Project, managed_by, Manager) .
The pattern extraction process analyzes each simple graph query pattern using the algorithm 7.
The nodes set in the GQ is analyzed in order to build the objects set O of the pattern P (line $2 \rightarrow 9$). In this step, each node v with the type object (known or unknown) is transformed to an object o having the same attributes and type as v. The algorithm then analyzes the different edges existing in GQ in order to detect the other elements in the query (line $10 \rightarrow 22$). An edge $e \in E$ in the query can have different types which are treated differently:

- If e has the type relation (line 12), a new relation r is added to the set $R(R \in G)$. r has all the characteristics of e. Indeed, if $e := (t, e_n, v_{start}, v_{End})$, then $r := \langle e_n, v_{start}, v_{end} \rangle$.

- If e has the type aggregation (line 15), a new aggregation function ag is added to the set AG. If $e := \langle t, e_n, v_{start}, v_{end} \rangle$. then $ag := (t_{ag}, o, a)$ where $t_{ag} := v_{start}$,$o =: v_{end}$ and a is the attribute related to v_{end} (if it exists).

- If e has the type condition (line 19), a new condition c is added to the condition set C. If $e := \langle t, e_n, v_{start}, v_{end} \rangle$ then $c := (a, cop, s)$ where $a = v_{end}$, $cop = v_{start}.t$ and $s = v_{start}.n$.

Algorithm 7: ExtractionBasicPattern()

```
Data: Graph query $GQ := (V, E)$
Result: Basic pattern query $BP = (G, C, AG)$
1  begin
2      foreach $v \in V$ do
3          if $v.t == object$ then
4              o=CreateObject(v)
5              $G.o = V.v$
6              if $v.isDefined$ then
7                  $o.u = True$
8              else
9                  $o.u = False$
10     foreach $e \in E$ do
11         switch $e.t$ do
12             case relation
13                 r=RelationExtraction()
14                 R.add(r)
15             case aggregation
16                 a=AggregationExtraction()
17                 AG.add(a)
18
19             case condition
20                 c=ConditionExtraction()
21                 C.add(r)
22
23     return BP
```

5.3.2 Graph Pattern Query Translation to SPARQL

In this section, we will start by describing SPARQL query syntax. Moreover, we will detail the graph pattern query translation to SPARQL process corresponding to each query type.

5.3.2.1 SPARQL query Syntax

SPARQL is a graph matching query language for RDF. A SPARQL query is of the form $H \longleftarrow B$, where:
-B is the body of the query: an RDF graph pattern expression that may include RDF triples with variables, conjunctions, disjunctions, optional parts, and constraints over the values of the variables. The WHERE clause contains the body part B which should be matched against the triples/graphs of RDF.
-H is the head of the query: an expression that indicates how to construct the

answer to the query. It is called the result form of the query which can be:

- SELECT: "Returns all, or a subset of, the variables bound in a query pattern match".
- CONSTRUCT: "Returns an RDF graph constructed by substituting variables in a set of triple templates".
- ASK : "Returns a boolean indicating whether a query pattern matches or not".
- DESCRIBE: "Returns a RDF graph that describes the resources found".

In this work, the used SPARQL queries are "SELECT" queries and the used RDF graph pattern on the body should cover the characteristics of a GraphVQL query. Now, the principal concepts and definitions related to SPARQL query will be described.

Definition 24 (RDF Triple) *A triple (s, p, o)* $\in (I \cup B) \times I \times (I \cup B \cup L)$*is called an RDF triple where s is the subject, p the predicate, o the object and I , B, and L (IRIs [Duerst 2005], Blank nodes, and Literals, respectively) are disjoint infinite sets.*

Definition 25 (SPARQL graph pattern) *A SPARQL graph pattern expression is defined recursively as follows:*
(1) A triple pattern is a graph pattern.
(2) If P1 and P2 are graph patterns, then expressions (P1 AND P2), (P1 OPT P2), and (P1 UNION P2) are graph patterns.
(3) If P is a graph pattern and R is a SPARQL built-in condition, then the expression (P FILTER R) is a graph pattern. A built-in condition is constructed using elements of the set Variable $\cup I \cup L$ *and constants, logical connectives* $(\neg, \wedge, \vee)$*, inequality symbols* $(\prec, \leq, \succ, \geq)$*, the equality symbol (=), unary predicates like bound, isBlank, and isIRI, plus other features.*

A Filter is a restriction on solutions over the whole group in which the filter appears.

Definition 26 (Filter Condition) *Let ?x, ?y* $\in$ *variables and c, d* $\in L \cup U$*. We define filter conditions inductively as follows:*

- *The expressions ?x = c, ?x =?y, and c = d are filter conditions.*
- *The expression bound(?x) (abbreviated as bnd(?x)) is a filter condition.*
- *If R1 and R2 are filter conditions, then* $\neg R1$*,* $R1 \wedge R2$*, and* $R1 \vee R2$ *are filter conditions.*

The used SPARQL query has the following definition:

Definition 27 (SPARQL Query) *A SPARQL query SpQ is defined by SpQ:=(PRF, VR, GPspr, FI, AG) where:*

- *PRF is the prefixes set. A prefix prf ∈ PRF declares the schemas used in the query. It is designed by the PREFIX keyword which associates a prefix label with an IRI.*

- *VR is the variables set which mentions the object to see in the result. Variables are distinguished by a leading question mark symbol, e.g. ?name denotes variable names.*

- *GPspr is the graph pattern of the SPARQL query. In this work, GPspr can be, a basic graph pattern or an union graph pattern. A basic graph pattern BP is a set of triples where $BP := \{(s, p, o)\}$. Union graph pattern is a set of basic graph patterns combined by the operator "UNION".*

- *FI is the set of filters on the query.*

- *AG is the set of aggregation operators. An $ag \in AG$ is applied on variable $v \in Vr$ where $ag := COUNT, SUM, MIN, MAX, AVG, GROUPBY$*

Example of a SPARQL query that allows to select the cars number by project filtered by the car used time:

$PREFIXcar :< http : //example.org/car/ >$
$PREFIXproject :< http : //example.org/project/ >$
$PREFIXrdf :< http : //www.w3.org/1999/02/22 - rdf - syntax - ns >$
$SELECTcount(?v0car)as?count, ?v3projectWHERE$
$?v0carcar : used_in?v3project.$
$?v0carcar : used\_times?v0used\_times.$
$?v0carcar : id?car1.$
$?v3projectproject : name?project2.$
$FILTER(?v0used\_times > 2).$

It is composed by:

- The set of prefixes: PRF= http://example.org/car/ , http://example.org/project/, http://www.w3.org/1999/02/22-rdf-syntax-ns

- The variables set VR=$?v0car, ?v3project, ?v0used\_times, ?project2$

- The patterns set: GPspr= $?v0car\ car : used\_in\ ?v3project;$
$?v0car\ car : used\_times\ ?v0used\_times.$
$?v0car\ car : id\ ?car1.\ ?v3project\ project : name\ ?project2.$

- The filters set: FI=?v0used_times $>$ 2

- the aggregations set: AG= count(?v0car)

5.3.2.2 The Translation Process

Translating the graph query pattern to a SPARQL query varies with the query type. Three translation processes are described in this section: Selection and aggregation query, Path query and SNA query.

A. Selection and aggregation queries

The selection and aggregation graph queries pattern can be a basic graph pattern query or a group graph pattern query. If the graph pattern query contains an operator "AND" or "UNION", the pattern is divided into two parts treated separately as basic graph pattern. The basic graph pattern query translation to a SPARQL query algorithm 8 is based on the following steps:

Step 1. The algorithm starts by extracting the prefixes existing on the RDF graph (line 3).

Step 2. From each existing relation $r := (r_n, o_{start}, o_{end})$ in the input graph query pattern, the algorithm extracts the corresponding triple (s, p, o_t) (line 4$\rightarrow$ 15). The triple is extracted with the method $CreateTriplefromrelation()$ which takes as input two variables var1 and var2 and the relation name r_n. These variables are extracted from the objects o_{start} and o_{end} as follows: $var = ? + (o_{start}.n.tostring())$. Then, the corresponding triple (s, p, o_t) to this relation is built in this way: s takes the value of the variable var1 extracted from the o_{start} and o_t is the variable var2 extracted from o_{end} . In order to build p, the prefix label corresponding to the relation is extracted from the prefix list PRF. The value attributed to p is: $(prefix\ label : r_n)$.

Step 3. For each object $o := (u, n, A)$ in the input graph query pattern with the attributes $(A \neq null)$, the algorithm extracts the corresponding triple (s, p, o_t) using the method $CreateTriplefromattribute$(line 16$\rightarrow$ 25). In this way, s takes the value of the variable var1 extracted from the object name o, the o_t takes the value of variable extracted from the attribute name var2 and the predicate p which is composed of the prefix label of the prefix corresponding to the object URI and the attribute name $(p = prefix\ label : attribute\ name)$. Then, (s, p, o_t)=(variable of o, $(prefix\ label : attribute\ name)$, variable of attribute).
If the attribute of the object o has a value, an additional triple is created where triple=(variable of o, $(prefix\ label : attribute\ name)$, attribute value)

Step 4. The conditions set are transformed to a set of filters (line 26$\rightarrow$ 28). A filter f , in this work , has two formats:
(1)FILTER regex(variable, value) used to search pattern on string. For example, to search all the projects having in their names the string "Water", the filter FILTER regex(?name, "Water") is used.
(2)FILTER (condition) used to directly apply a condition to an attribute. For example, to search all the car having a price more than 10.000, the filter $FILTER(?price > 10000)$ is applied.

For each condition $c := (a, cop, s)$, a filter$f$ is created. If the attribute a has the type string and the operator is "CONTAIN" then the first type of filter are used. In other cases, the second filter type is adopted.

Step 5. In the case of aggregation query (line 29$\rightarrow$ 31), the graph pattern query contains a set of aggregations applied to objects or objects attributes. Then, for each aggregation $ag := (t_{ag}, o, a)$, a new aggregation ags is added to the SPARQL query. $ags = (f, var, varn)$ is composed of f, the aggregate function (SUM,MAX,MIN, or COUNT) which has the value of t_{ag}, var is the variable name extracted from the attribute a or from the object o, if a is null.

Algorithm 8: Selection and aggregation graph pattern query SPARQL-Transformation

```
    Data: Basic pattern query $P = (G, C, AG)$,the RDF Graph RDFG
    Result: $SpQ := (PRF, VR, GPspr, FI, AG)$
 1  begin
 2      SparqlQuery SpQ = new SparqlQuery();
 3      PRF = extractPrefix(RDFG);
 4      foreach relation $r := (r_n, o_{start}, o_{end}) \in G$ do
 5          Var1=CreateVariable(o$_{start}$);
 6
 7          VR.add(Var1);
 8
 9          Var2=CreateVariable(o$_{end}$);
10
11          VR.add(Var2);
12
13          Triple t=CreateTriplefromrelation(r$_n$, $Var1, Var2$);
14
15          SpQ.GPspr.add(t);
16      foreach Object $o := (u, n, A) \in G$ do
17          if $A \neq null$ then
18              Var1=CreateVariable(o);
19              foreach  $a \in A$ do
20                  Var2=CreateVariable(a);
21                  VR.add(Var2);
22                  Triple t= CreateTriplefromattribute(Var1,Var2);
23                  SpQ.GPspr.add(t);
24                  if $a.value \neq null$ then
25                      Triple t= CreateTriplefromattributeValue(Var1,Var2);
26                      SpQ.GPspr.add(t);
27      foreach condition $c \in C$ do
28          Filtre f=CreateFilter(c);
29          SpQ.FI.add(f)
30      foreach  aggregation $a \in AG$ do
31          Aggregation ag=CreateAggregation(a);
32          SpQ.AG.add(ag);
33      return SpQ;
```

The final SPARQL query is built by regrouping the different sets using the following grammar:

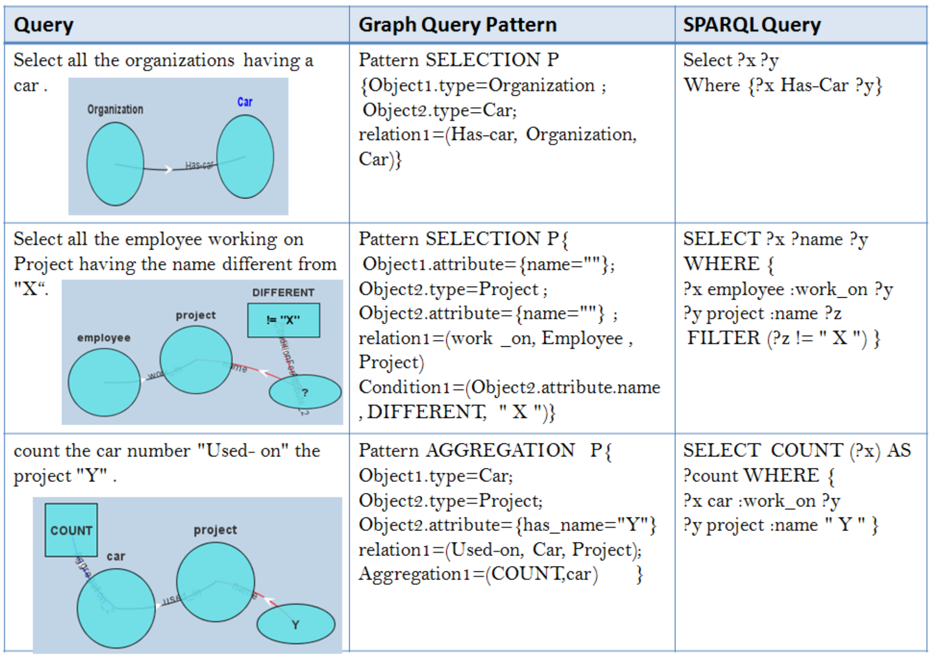
SPARQLQUERY:="SELECT" (VAR|AG) (,VAR) * (,AG)*

"WHERE{" TRIPLE (TRIPLE)*
(FILTER)* ""

If the graph pattern query is a group pattern query containing an UNION operator, an analyze step is performed to detect the similar variables and the different detected filters. The resulting SPARQL follows this grammar:

SPARQLQUERY:="SELECT" (VAR|AG) (,VAR) * (,AG)*
"WHERE{{ " TRIPLE (TRIPLE)* ""
"UNION {" TRIPLE (TRIPLE)* ""
(FILTER)* ""

Examples of selection and aggregation queries transformation are shown in Table 5.8. From each drawn query in the first column, a graph query pattern is extracted then transformed to a SPARQL query. For instance, in the first query, the graph query pattern contains two objects and a relation. The two objects (object1 and object2) are transformed to two variables in the SPARQL query (?x and ?y) and the relation is putted in the Where condition.



Query	Graph Query Pattern	SPARQL Query
Select all the organizations having a car .	Pattern SELECTION P {Object1.type=Organization ; Object2.type=Car; relation1=(Has-car, Organization, Car)}	Select ?x ?y Where {?x Has-Car ?y}
Select all the employee working on Project having the name different from "X“.	Pattern SELECTION P{ Object1.attribute={name=""}; Object2.type=Project ; Object2.attribute={name=""} ; relation1=(work _on, Employee , Project) Condition1=(Object2.attribute.name , DIFFERENT, " X ")}	SELECT ?x ?name ?y WHERE { ?x employee :work_on ?y ?y project :name ?z FILTER (?z != " X ") }
count the car number "Used- on" the project "Y" .	Pattern AGGREGATION P{ Object1.type=Car; Object2.type=Project; Object2.attribute={has_name="Y"} relation1=(Used-on, Car, Project); Aggregation1=(COUNT,car) }	SELECT COUNT (?x) AS ?count WHERE { ?x car :work_on ?y ?y project :name " Y " }

Figure 5.8: Selection and Aggregation Queries on Simple Graph.

B. Path Query

Using a path query, a user specifies two objects where he would like to search the possible links between them. The path pattern query contains information about the start object and the end object of the path. The approach to treat this query

uses a path search algorithm in order to find the links between the two objects. In this context, we use the A* algorithm [Hart 1972]. A* is based on a heuristic function to increase the rapidity of the search process. The used heuristic function has the form: $F(n) = d(n) + h(n)$ where $d(n)$ is the distance from node n to root node and $h(n)$ is the depth of node n.
The algorithm 9 transforms a path graph pattern query to a SPARQL query. As the path query can be seen as two simple select queries related by the link Path, the algorithm treats each part separately in order to find the start and the end nodes for the path (line 1 $\rightarrow$ 5). Then, the path is searched using the A* algorithm (line 5 $\rightarrow$ 6).

Algorithm 9: Path pattern query transformation to SPARQL

```
   Data: pattern query P,the RDF Graph RDFG
   Result: $SpQ := (PRF, VR, GPspr, FI, AG)$
1  begin
2      SparqlQuery q_left =SPARQLTransformation (p.left_pattern);
3      SparqlQuery q_right = SPARQLTransformation (p.right_pattern);
4      RDFNode start_node = executeQuery(q_left);
5      RDFNode end_node = executeQuery(q_right);
6      GraphRDFNode graph = getGraph();
7      GraphRDFNode path = retrievePath(graph,start_node,end_node);
8      return path;
```

This algorithm is based on a retrieve function (see algorithm 10) which represents the application of the $A*$ algorithm.

An Example of the path query is shown by Figure 5.9: Find the path between Hyundai49 and Ford77.

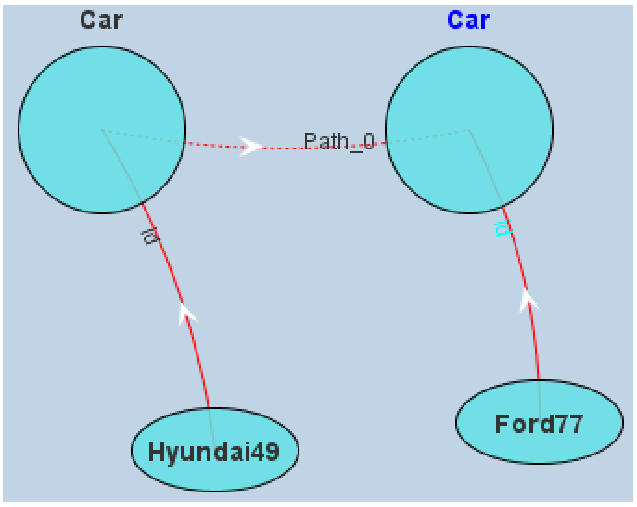


Figure 5.9: Path Query.

Algorithm 10: RetrievePath()

```
Data: the RDF Graph g , RDFNode start, RDFNode end
Result: path
begin
    Queue queue = new Queue();
    List list_visited_nodes;
    queue.push(start);
    while queue ≠ null do
        RDFNode node = queue.pop();
        list_expanded_nodes = expand(node);
        if list_nodes_expanded.constain(end) then
            break;
        queue.push(list_expanded_nodes);
    //update the weight of node by using F(n) = d(n) + h(n)
    updateWeightFunction(queue)
    list_visited_nodes(list_expanded_nodes)
    // get back the found path.
    GraphRDFNode path = traceBack(list_visited_nodes,start,end)
    return path
```

The coresponding graph pattern query is:

Pattern PATH P{
Pattern Left{
Object1.type=Car
Object1.attribute={$id = "Hyundai49"$}
Pattern Right{
Object1.type=Car
Object1.attribute={$id = "Ford77"$}

The coresponding SPARQL query for this pattern is: SELECT * WHERE {
(SELECT * WHERE {
?v0Car ?p ?o.
?v0Car car:id ?v0id.
FILTER regex ($?v0id, "Hyundai49"$) .
UNION
(SELECT * WHERE {
?v1Car ?p ?o.
?v1Car car:id ?v1id.
FILTER regex($?v1id, "Ford77"$) .

C. SNA query

SNA is a query that analyzes the position of a particular object in the input graph. This analysis is performed using the centrality measure which gives a "rough indication of the social power of a node based on how well it "connect" the Network"'. There are three measures of centrality that are widely used in social network analysis: degree, betweenness and closeness.

(1)The **Degree centrality** of a node refers to the number of edges attached to the node.

If the graph is directed, then we usually define two separate measures of degree centrality, namely indegree and outdegree. Indegree is a count of the number of edges directed to the node, and outdegree is the number of edges that the node directs to others. For positive relations such as friendship or advice, we normally interpret indegree as a form of popularity, and outdegree as gregariousness.

For a graph $G := (V, E)$ having n vertex, the degree centrality of a vertex v:

$D_c(v) := \frac{Degree(v)}{(n-1)}$

(2)**Betweenness centrality** is a centrality measure of a vertex within a graph. Vertices that occur on many short paths between other vertices have higher betweenness than those that do not. Betweenness reflects the number of nodes which a node is connecting indirectly through their direct links.

For a graph G , the betweenness for vertex is computed as follows:
1. For each pair of vertices (s,t), compute all shortest paths between them.
2. For each pair of vertices (s,t), determine the fraction of shortest paths that pass through the vertex in question (here, vertex v).
3. Sum this fraction over all pairs of vertices (s,t).

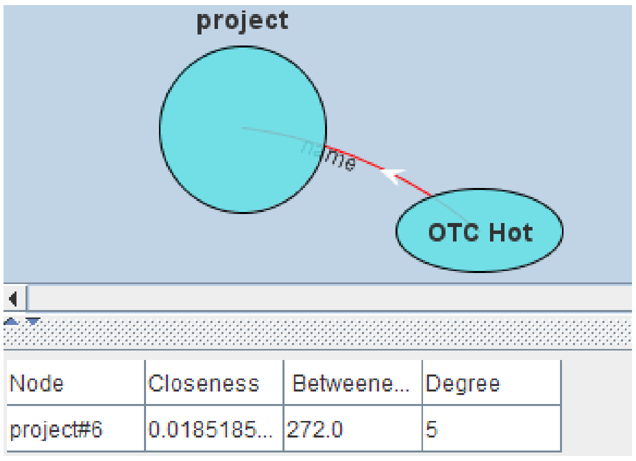
$B_c(v) := \sum_{v_i \neq v_j \neq v \in V} \frac{Path_{v_i v_j}(v)}{Path_{v_i v_j}}$, where $Path_{v_i v_j}$: the number of shortest path from v_i to v_j

$Path_{v_i v_j}(v)$:the number of shortest path from v_i to v_j pass through a vertex v.

(3)**Closeness centrality** is the number of links that a person must go through in order to reach everyone else in the network. The closeness centrality of a node is measured as the inverse of the sum of distances from this node to all the other nodes [Freeman 1977].

$C_c(v) := \frac{1}{\sum_{v_i \in G} \frac{1}{length(PATH(v,v_i))}}$ where $PATH(v, v_i)$ is the shortest path between v and v_i and $length(PATH(v, v_i))$ is the length of such a shortest path.

These different measures have been implemented with the use of SPARQL queries. An example of SNA query applied to the object "Project OTC HOT" is presented in Figure 5.10.



Node	Closeness	Betweene...	Degree
project#6	0.0185185...	272.0	5

Figure 5.10: Analyse Query.

5.4 SPIDER-Graph Queries Treatment

In order to query SPIDER-Graph, GraphVQL proposes for the users to draw SPIDER-Graph queries. These queries can be selection or path queries. For this, a graph pattern matching algorithm is used in the case of selection query and a shortest path algorithm is used in the case of a path query. In what follows, we start by defining the graph pattern query for a SPIDER-Graph query. Thereafter, the process to treat the selection query and then the path query, will be described thoroughly.

5.4.1 Graph Pattern Query for the SPIDER-Graph

The SPIDER-Graph queries are composed by complex-nodes which can be linked with relation links or path links in the case of a path query. The basic pattern query will be a particular case from the one defined in definition 23. Actually, the objects in the graph pattern are simply complex-nodes. Also, there are no aggregation operators. The graph pattern query is defined as follows:

Definition 28 (Basic SPIDER-Graph Pattern Query) *A basic pattern query for a SPIDER-Graph $S := (N_{cn}, R)$ is defined by $BP_S := (G, C)$ where:*

G is the SPIDER-Graph pattern $G := (O, R)$:

- *O is the complex-nodes set which the user would like to see in the result;$O := \{o\,|o := (u, CN), CN \in N_{cn}\}$ with*
  - *u is a Boolean attribute which mentions if o is a known or an unknown object.*
  - *CN is the complex-node defined $CN := (cn, A_{cni})$ (see the definition 16).*
- *R is the set of object relations where $R := \{r\,|r := (r_{name}, o_{start}, o_{end}), o_{start}$ and $o_{end} \in O\}$*

C is the set of conditions; $C := \{c \,|c := (a, cop, s)\}$ where a is the attribute of the condition, cop is the used operator (which can be DIFFERENT, GREAT_THAN, GREAT_EQUAL, LESS_THAN, LESS_EQUAL, EQUAL,CONTAIN) and s is the value of the condition.

The extracted pattern from the GraphVQL query can be a set of BP_S or a set of BP_S related by the link PATH. Now, we start by detailing the selection query process.

5.4.2 Selection Query: Graph Pattern Matching

The graph pattern matching algorithm takes the graph pattern extracted from the drawn query and the SPIDER-Graph as input; retrieves sub-graphs from the input graph matching the graph pattern query and returns the retrieved graphs.
We have used the pattern matching algorithm proposed in [He 2008] and we have modified it to support a SPIDER-Graph data model. This algorithm supports typed, attributed nodes and labeled edges; and its standard allows us to include other pruning techniques easily.

It takes the SPIDER-Graph Pattern Query and a SPIDER-Graph as input, and produces one or all feasible mappings as output. This process is based on two phases:
(1)The first phase retrieves the feasible matches for each node in the graph pattern and builds search space with the resulting products. The feasible matches of a node u is the set of nodes in graph G that satisfies a predicate F_u: $\phi(u) = \{v \,|v \in V(G), F_u(v) = true\}$.
The search space is the product of feasible matches for each node of graph pattern. In the query in Figure 5.11a, there are two complex nodes: $CN1$ and $CN2$, with the types "Employee" and "Project", respectively. The feasible matches for each complex-node are: $\phi(CN1) = \{v \,|v \in V(G), F_{CN1}(v) = true\}$where $F_{CN1}(v) = \{v.type = \text{“}Employee\text{”}\}$
$\phi(CN2) = \{v \,|v \in V(G), F_{CN2}(v) = true\}$where $F_{CN2}(v) = \{v.type = \text{“}Project\text{”}\}$
(2)The second phase consists in searching the matching between the graph pattern query and the SPIDER-Graph using the search space. Now, we detail the adopted algorithm 11.

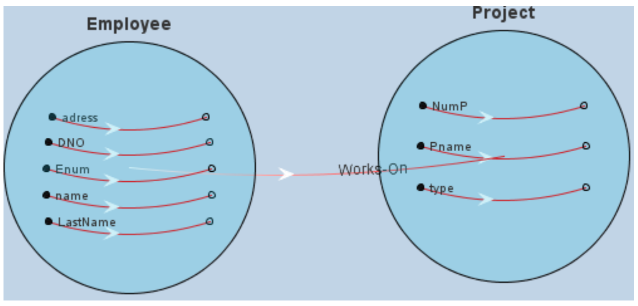
Step 1: The algorithm searches all the feasible matches of each complex-node in the pattern in the input graph (line 2-4). The used predicate on this matching is the following:
Fu (v) (nodes): Gathers all the complex-nodes having the same type and the same attributes values than those defined in the graph pattern query. If no attribute is defined in the pattern complex-nodes, all complex-nodes of the same type are accepted.
The feasible matches of the graph pattern on the SPIDER-graph depicted in Figure 5.11 are the following:
$\phi(Employee) = \{Employee\_1, Employee\_2, Employee\_3\}$

$\phi(Project) = \{Project\_1, Project\_2\}$
Step 2: The detected feasible matches are pruned (line 6 $\rightarrow$ 10). The algorithm compares the neighborhood sub-graphs profiles of each complex-node on the graph query pattern and its feasible complex-node neighborhoods profiles in order to reduce the size of $\phi(u)$. By definition in [He 2008], the neighborhood sub-graph of node v consists of all nodes within distance 1 from v and all edges between the nodes. The profile is a sequence of the node labels in lexicographic order. Then, node v is a feasible match of node u_i if they have the same profile. In the graph in Figure 5.11, for the complex-node $Employee\_1$, the neighborhood sub-graphs of radius 1 are {Employee_1, Project_2, Employee_3} and the corresponding profile is “Employee Employee Project". This profile contains the profile of the complex-node "Employee" which is "Employee Project". Then $Employee\_1$ is maintained in the search space.
Step 3: The search process searches the matching between the input graph pattern query and the SPIDER-Graph (line 12 $\rightarrow$ 26). The procedure $Search(i)$ iterates on the i^{th} node to find feasible mappings for that node. The procedure $Check(u_i, v)$ examines if u_i can be mapped to v by considering their edges.
$F_e(e')(edges)$: Compares the labels of the SPIDER-Graph edge and the pattern edge. If no label is specified in the pattern, all corresponding edges of the SPIDER-Graph are accepted.
$F_\phi(G)(match)$: Considering the analyzed criteria, there was no other relevant criteria to be added to this function, which means that it will always return true. However, it was considered anyway in the structure in case of a definition of a better parameter.

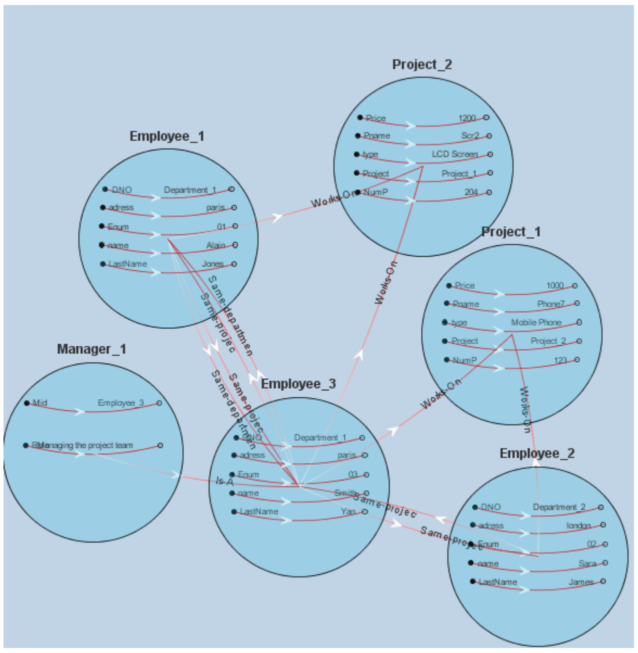
Algorithm 11: Graph Pattern Matching

```
Data: Basic pattern query $BP_S$,SPIDER-Graph $S$
Result: one or set of feasible graphs
begin
    //find all the feasible matches
    foreach complex-node $u \in BP_S$ do
        $\phi(u) = \{v \,|v \in S, F_u(v)(nodes)\}$

    //Local pruning of the feasible mates
    foreach complex-node $u \in BP_S$ do
        foreach complex-node $v \in \phi(u)$ do
            if $\neg Feasable(Profile(u), Profile(v))$ then
                Remove v from $\phi(u)$;
    Search(1);
    void Search(i)
    begin
        foreach complex-node $v \in \phi(u_i)$, $v$ is free do
            if $\neg Check(ui, v)$ then continue;
            $\phi(u_i) = v$;
            if $i < |V(BP_S|$ then Search(i+1);
            else if $F_\phi(S)$ then
                Report $\phi$;
                if $\neg exhaustive$ then Stop;

    boolean Check(u$_i, v$) begin
        foreach edge $e(u_i, u_j) \in BP_S, j < i$ do
            if edge $e'(v, \phi(u_j)) \notin Sor\neg F_e(e')$ then
                return false ;
        return true;
```

Now, we present an application example. Figure 5.11 shows the sample graph pattern and the SPIDER-Graph.



(a) Pattern



(b) SPIDER-Graph

Figure 5.11: A sample graph pattern and SPIDER-Graph

In the previous example, the matched sub-graphs are: (Employee_1, Works-on, Project_2) , (Employee_2, Works-on, Project_1) ,(Employee_3, Works-on, Project_1) ,(Employee_3, Works-on, Project_2)

5.4.3 Path Query

The purpose of the path query is to discover if a set of complex-nodes are connected or not. This query is composed by two or more complex-nodes related by path links. In order to process a path query, we have proposed an algorithm based on three main steps:
Step 1: The algorithm analyzes the path pattern query in order to extract the set of paths to find. Each path is defined by its start node and end node. For example, if the path query is: CN1 PathLink CN2 PathLink CN3, then there are two paths to search $path1$ between ($CN1$, $CN2$) and $path2$ between ($CN2$, $CN3$).

In the query depicted in Figure 5.7, we have only one path to search $path1$ between $CN_{start} = Manager$ and $CN_{end} = Project$

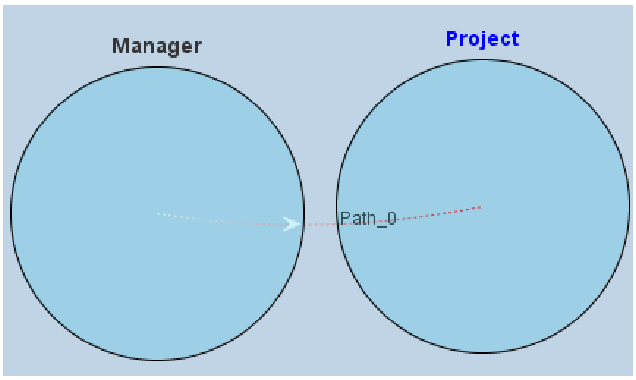


Figure 5.12: Path Query.

Step 2: For each path between two complex-nodes, we have chosen to find the shortest path between them. In order to find the shortest paths between the complex-nodes, we have used the algorithm proposed in [Newman 2001] which is a modified form of the traditional graph traversal algorithm breadth-first search. The algorithm finds the path between a node n_i and a node n_j by stepping from n_i to its predecessor and then to the predecessor of each successive node until n_j is reached. The algorithm can find one or multiple path between two well defined nodes if it exists. We have integrated two adaptations to this algorithm:
(1) Support path queries between nodes in which attributes are partially specified;
(2) Take into account that an attribute in a complex-node can reference another complex-node, which means, that it considers the "implicit edges" provided by the SPIDER-Graph model.
The algorithm 12 is made to find the node candidates for the two nodes and, for each pair of them, calculate the shortest path. Firstly, for the two complex-nodes CN_{start} and CN_{end}, the algorithm found their correspondent complex-nodes in the

input SPIDER-Graph (line 3⟶11).

Algorithm 12: Path query processing

```
Data: Complex-node $CN_i$, $CN_j$,SPIDER-Graph $S$
Result: set of paths between $CN_i$ and $CN_j$
1  begin
2      //find all the complex-nodes which can match $CN_i$ and $CN_j$
3      if $CN_i.A \neq null$ then
4          $cand(CN_i) = \{v \,| v \in S, v.type = CN_i.type \; and \; v.A = CN_i.A\}$;
5      else
6          $cand(CN_i) = \{v \,| v \in S, v.type = CN_i.type\}$ ;
7      if $CN_j.A \neq null$ then
8          $cand(CN_j) = \{v \,| v \in S, v.type = CN_j.type \; and \; v.A = CN_j.A\}$;
9      else
10         $cand(CN_j) = \{v \,| v \in S, v.type = CN_j.type\}$;
11     //Research of paths
12     $List < SPIDER - Graph >$ result;
13     foreach $complex\text{-}node \; u \in cand(CN_i)$ do
14         foreach $complex\text{-}node \; v \in cand(CN_j)$ do
15             if $u \neq v$ then
16                 SPIDER-Graph path=FindShortestPath(S,u,v);
17             if $path \neq null$ then
18                 result.add(path);
19     return result ;
```

Then, for each couple of matched, complex-nodes the shortest path is searched (line 14⟶20) using an extension of the algorithm presented in [Newman 2001]. One important aspect of SPIDER-Graph is that an attribute in a complex-node can refer to another complex-node. While [Newman 2001] expands only the edges of a node while setting the predecessors in the breadth-first search, the search is expanded additionally to the attributes of the node. In the next interaction, the edges of the node that was an attribute will be expanded as well, revealing the implicit relation that BFS cannot find. The shortest path is calculated with procedure 13:

Algorithm 13: FindShortestPath()

```
    Data: Complex-node $CN_i$, $CN_j$,SPIDER-Graph $S$
    Result: set of paths between $N_i$ and $N_j$
1   begin
2       Assign $CN_j$ distance zero;
3       $d \longleftarrow 0$;
4       foreach complex-node $CN_k$ whose assigned distance is $d$ do
5           follow each attached edge to $CN_l$ at its other end;
6           follow each attribute of $CN_k$ that points to another node l;
7           if $l$ has not already assigned distance then
8               $CN_l$.setDistance(d+1);
9               declare $CN_l$ as predecessor of $CN_k$;
10          d=d+1;
```

Step 3: The set of found paths are regrouped and returned as the final result.

The result is displayed to the user as a set of sub-graphs.

For example, if the previous path query is searched in the SPIDER-Graph described in Figure 5.11. Firstly, the algorithm searches for the complex-nodes that match with $CN_{start} = Manager$ and $CN_{end} = Project$: cand(Manager)={ $Manager_1$} and $cand(Project) = \{Project_1, Project_2\}$. Then, all the possible shortest paths between $(Manager_1, Project_1)$ and $(Manager_1, Project_2)$ are searched via the shortest path algorithm. The displayed result is the all the paths between these complex-node instances (see Figure 5.13).

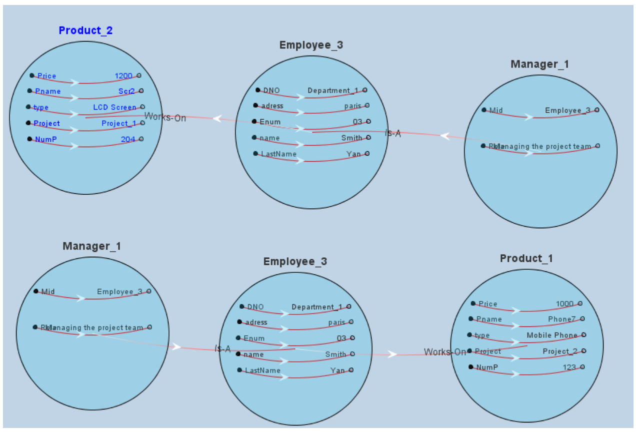


Figure 5.13: Path Query Result.

5.4.4 Aggregation Query

When the input graphs are large, effective graph aggregation methods are helpful for the user to understand the underlying information and structure. Graph aggregations produce small and understandable summaries and can highlights groups or even communities in the graph, which greatly facilitates the interpretation. Thus, adding an aggregation operator in GraphVQL will help the user to divide the input graphs in a significant groups. The existing algorithms like partitional algorithms [MacQueen 1967], Hierarchical clustering algorithms [Rodrigues Jr. 2006] or spectral algorithms [Luxburg 2007], use only links between nodes of the graph of the network, and do not take into account the internal values contained in each node, while classical clustering algorithms applied on tables of values, work only on these values ignoring completely the possible link between individual.

An algorithm which can take into account both kind of information would be very valuable. Designed for graphical graph aggregation the k-SNAP algorithm [Tian 2008], in its divisive version, begins with a grouping based on attributes of the nodes, and then tries to divide the existing groups thanks to their neighbours groups, trying minimizing a loss information measure.

The K-SNAP algorithm produces a summary graph through a homogeneous grouping of the input graph's nodes, based on user-selected node attributes and relationships. The K-SNAP controls the number of resulting groups. Indeed, a user can give as input the K-group to see as result. All the details of this algorithm is specified in the the paper [Rania Soussi 2012].

In GraphVQL, the aggregation is performed as follows. The user draws the object to see and specifies the attributes and the relations with which the algorithm will obtains the groups. For example, if the user would like to dived the Employee in four groups. First, the algorithm starts by extracting the employees graph (with attribute gender and address) and the relation between them (Same_Department and Same_Project). Then, the algorithm produces a summary graph shown in Figure :ksnap. This summary contains four groups of employees and the relationships between these groups. Employees in each group have the same gender and are in the same address (enterprise agency address), and they relate to employees belonging to the same set of groups with Same_Department and Same_Project relationships. For instance, each employee in group G_1 has at least another employee that work with him in the Same_Department and the Same_Project in the in $groupG_2$.

5.5 Summary and Conclusion

In this chapter, we have presented GraphVQL which is a visual query language for a SPIDER-Graph model and graphs modeled with RDF or GraphML. GraphVQL allows to query:

- Simple graphs (RDF or GraphML) via selection, aggregation or analysis

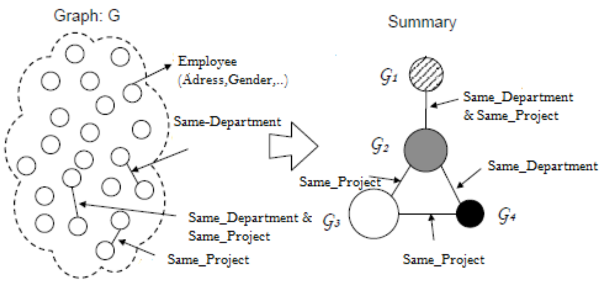


Figure 5.14: Aggregation of the Employee Graph.

queries. Here, the queries are modeled as graphs from which graph pattern queries are extracted and transformed to a SPARQL query.

- SPIDER-Graph via selection, path or aggregation queries. In this case, the queries are model as SPIDER-Graphs model from which pattern queries are extracted. Then, for the selection queries, we use a pattern matching algorithm, which finds the corresponding part of the input graph to the pattern. In the path query case, we use a path query algorithm to find the response. Finally, to aggregate the input query a graph aggregation algorithm is used.

In the section 5.2, we have described the visual notation and the grammar rules of GraphVQL. GraphVQL visual symbols allow inexperienced users to draw their queries as graph patterns. These patterns are treated with two different methods depending on the input graph. Then, we have detailed in the section 5.3 the query treatment in the case of RDF or GraphML graphs. The graph pattern query is transformed to a SPARQL query taking in consideration the different types of symbols drawn. For instance, if the initial query contains the aggregation operator count, then the SPARQL query will contain in the SELECT part the operator count. In the case of the SNA query, GraphVQL can for the moment calculate the degree, betweenness and closeness for a specific object in the input graph. Then, in the future work, adding other metrics like the cohesion or the density can help users to improve their decision making.

In the section 5.4, we have explained the different SPIDER-Graph queries treatment which can be summarized in three different process:

- The selection queries are treated via an exact pattern matching (section 5.4.2). The used algorithm treats the input graph pattern in three main steps: (1)search all the feasible matches of each complex-node in the input pattern, (2) the feasible matches are pruned in order to minimize the search space and (3) a search process is performed on the built search space. The result will be all the sub- graphs that match the input pattern.

- The path queries are treated via a two steps algorithm. First, the algorithm will search all the existing path in the input graph pattern query. Then, each path is searched with our extension of the Newmann algorithm that takes in consideration the attributes in the complex-nodes source and destination.

- The aggregation queries can only be performed on a part of the input SPIDER-Graph that contains homogeneous complex-nodes. Then, the aggregation algorithm will take as input this homogeneous sub-graph, the user selected attributes and edges related to the complex-node in the graph. The result will be different groups of complex-nodes grouped by their attributes and relation similarities.

We have not made a social analyze query on the SPIDER-Graph for instance but we are working on this. For this, we need to create an adapted algorithms that take in consideration the structure of the complex-nodes. In the section 6.4 of the next chapter, we will present the implementation and the evaluation of GraphVQL.

Chapter 6

Implementation and Evaluation

Contents

Up to this point we have presented the contributions of our work. In order to test their viability in a real world environment, a prototype has been developed. It includes the three main approaches: The enterprise ontology building, the object graph extraction and the visual query language. The system has been designed to contain three sub-systems which work independently. The performance of each sub-system is evaluated separately. In the first section we present the implemented architecture. The data sets used in the evaluation are presented in section 6.2. In section 6.3, we have applied the objects interactions graph extraction from the relational database to extract social network. After this, we have evaluated the performance of this approach. Finally, the GraphVQL performance is evaluated as well.

6.1 System Implementation

The different approaches presented in the previous chapters have been implemented and evaluated using JAVA. The prototype architecture is depicted in Figure 6.1.

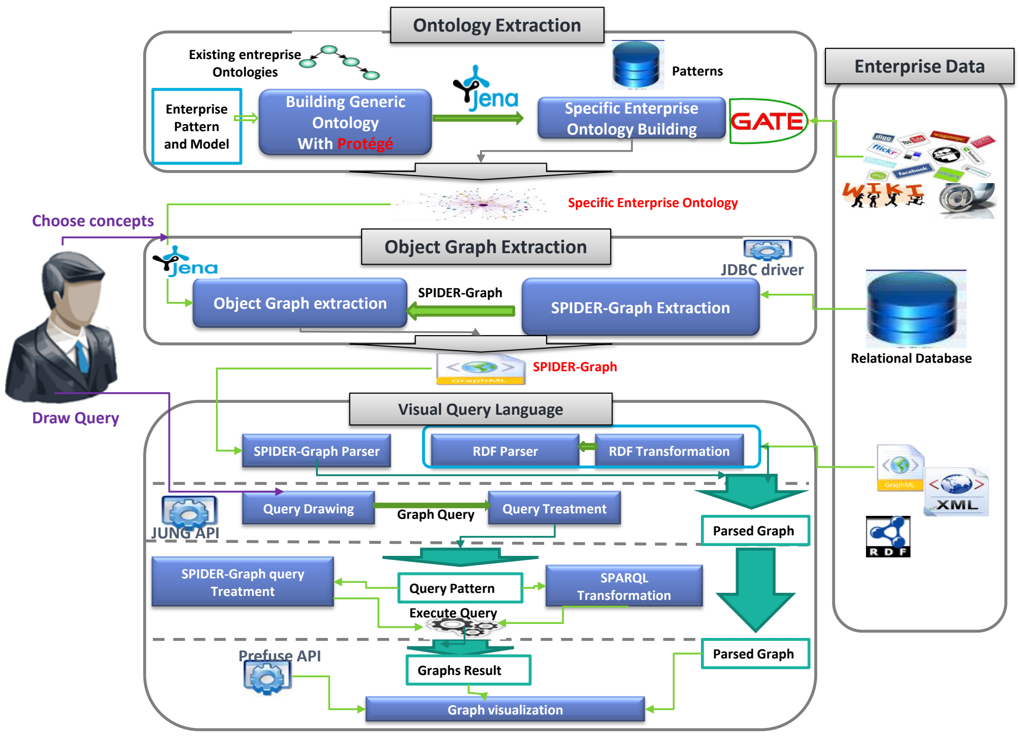


Figure 6.1: The System Implementation Architecture.

The prototype contains three main functionalities presented as three independent sub-systems: ontology extraction, object graph extraction and the visual query language.
The ontology extraction system provides the specific enterprise ontology to the objects graph extraction module. This latter, which is modulated as a SPIDER-Graph, can be queried and visualized by the visual query language module. In what follows, the functionalities and the used API of each module is shown.

6.1.1 The Ontology Extraction Module

The ontology extraction module is a sub-system, which can be described as the implementation of the enterprise ontology learning process. It is charged to build the enterprise ontology by taking as input a generic enterprise ontology, which is built manually using Protégé 3.7.4, and the enterprise unstructured and semi-structured data (websites and wiki). It uses the Protégé JAVA API [Knublauch 2004] to load, edit and save ontologies.
The architecture of this sub system is depicted in Figure 6.2. As a first step the enterprise documents are treated with GATE components. GATE [Cunningham 2002] is an open source architecture and infrastructure for the building and deployment of Human Language Technology applications. GATE can be used to process doc-

uments in different formats including plain text, HTML, XML, RTF, and SGML. The following components were used to process documents:

- The Tokeniser divides the text into tokens (such as numbers, punctuation, symbols, and words of different types),
- The Sentence Splitter divides the text into sentences. This module is required for the tagger,
- Pos-Tagger adds part-of-speech information to tokens,
- Lemmatizer reduces each token to its lemma, i.e. base form,
- NP chunker divides the text into noun phrase chunks,
- OntoRootGazetteer looks up items from the ontology and matches them with the text, based on root forms,
- JAPE transducers annotates text and adds new items to the ontology.

Based on the previous GATE components, the documents are treated and analyzed to extract the candidate concepts, relations, and attributes to enrich the generic enterprise ontology. Then, the new detected concepts and attributes (Datatype) are used in the learning process to enrich the generic ontology and the new instances in the population process. The produced ontology is modeled and stored with OWL1.0.

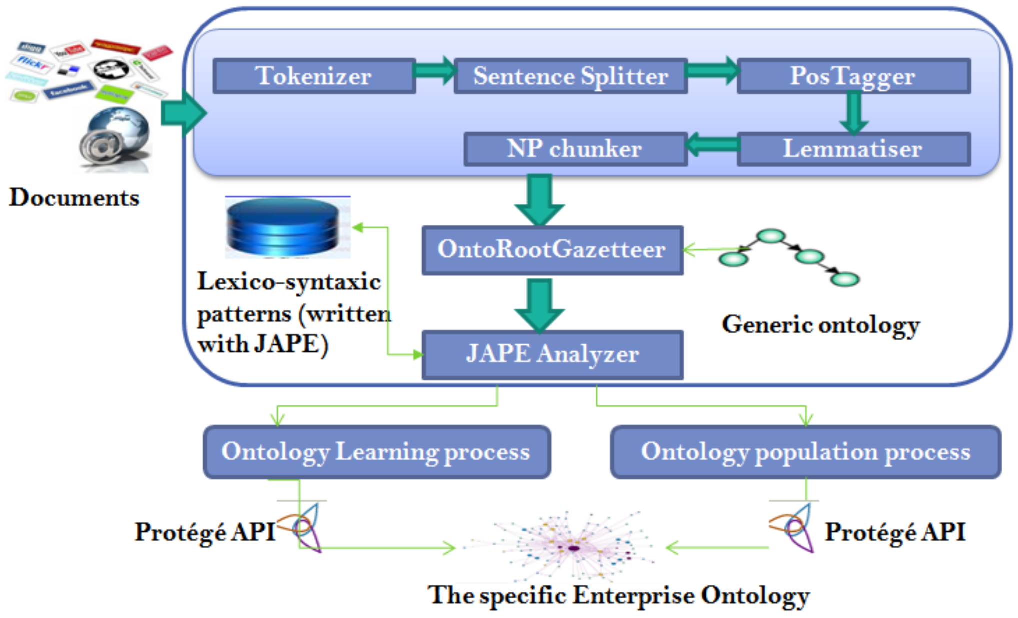
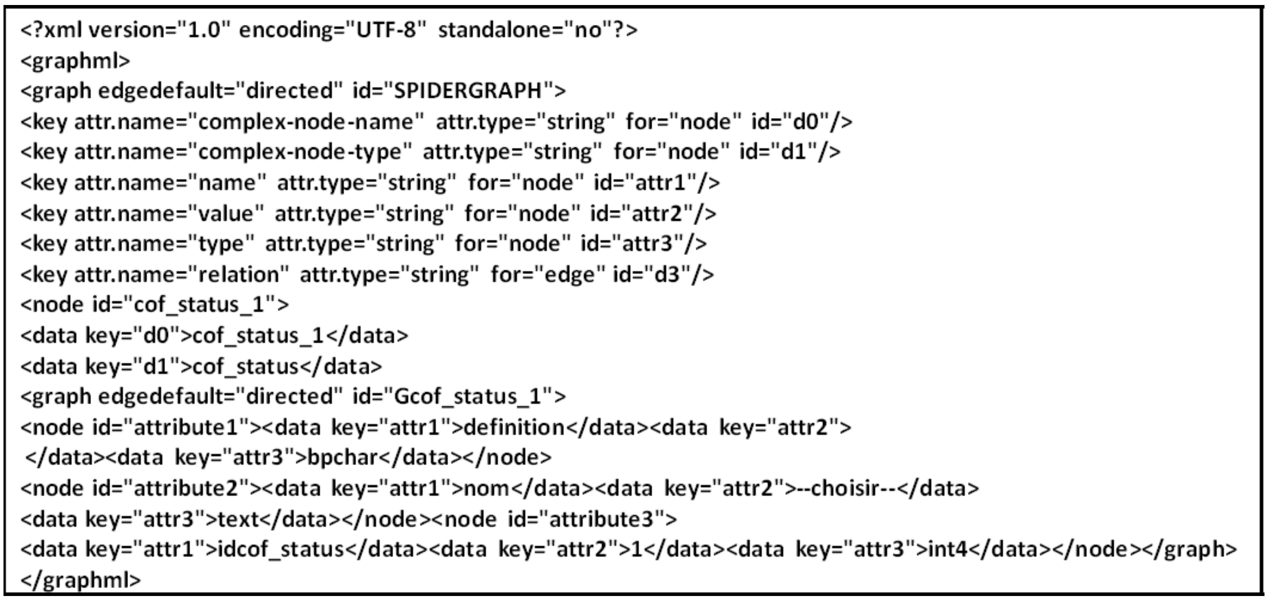


Figure 6.2: The ontology Extraction Module.

6.1.2 Object Graph Extraction

The object graph extraction is a sub-system based on two modules(see Figure 6.4):)
-The SPIDER-Graph extraction module: this module takes the relational database as input, and then processes it to extract a SPIDER-Graph. The connection between the system and the relational database is allowed by the use of a JDBC API. As a



```
<?xml version="1.0" encoding="UTF-8" standalone="no"?>
<graphml>
<graph edgedefault="directed" id="SPIDERGRAPH">
<key attr.name="complex-node-name" attr.type="string" for="node" id="d0"/>
<key attr.name="complex-node-type" attr.type="string" for="node" id="d1"/>
<key attr.name="name" attr.type="string" for="node" id="attr1"/>
<key attr.name="value" attr.type="string" for="node" id="attr2"/>
<key attr.name="type" attr.type="string" for="node" id="attr3"/>
<key attr.name="relation" attr.type="string" for="edge" id="d3"/>
<node id="cof_status_1">
<data key="d0">cof_status_1</data>
<data key="d1">cof_status</data>
<graph edgedefault="directed" id="Gcof_status_1">
<node id="attribute1"><data key="attr1">definition</data><data key="attr2">
 </data><data key="attr3">bpchar</data></node>
<node id="attribute2"><data key="attr1">nom</data><data key="attr2">--choisir--</data>
<data key="attr3">text</data></node><node id="attribute3">
<data key="attr1">idcof_status</data><data key="attr2">1</data><data key="attr3">int4</data></node></graph>
</graphml>
```

Figure 6.3: Graphml for the SPIDER-Graph Instance.

-Object Graph extraction: From the extracted graph and by using the user chosen concepts, this module extracts the objects interactions graph. The resulting graph is saved as a specific GraphML file which is modified in order to store a SPIDER-Graph model. The GraphML for the SPIDER-Graph model, which is presented in Figure 6.3, contains the following elements:

- A complex-node is represented by a node $< node >$ having two elements "complex-node-name" and "complex-node-type" putted on the tag $< data >$.
- Each complex-node contains one nested $< graph >$ including its set of attributes.
- A complex-node attribute is represented by a node which has three elements: "name", "value" and "type" putted in three $< data >$ tags.

We choose to store the resulting graphs in GraphML due the fact that GraphML is an extension of XML and it is supported by many graph visualization API like

Prefuse[1] and Gephi[2].

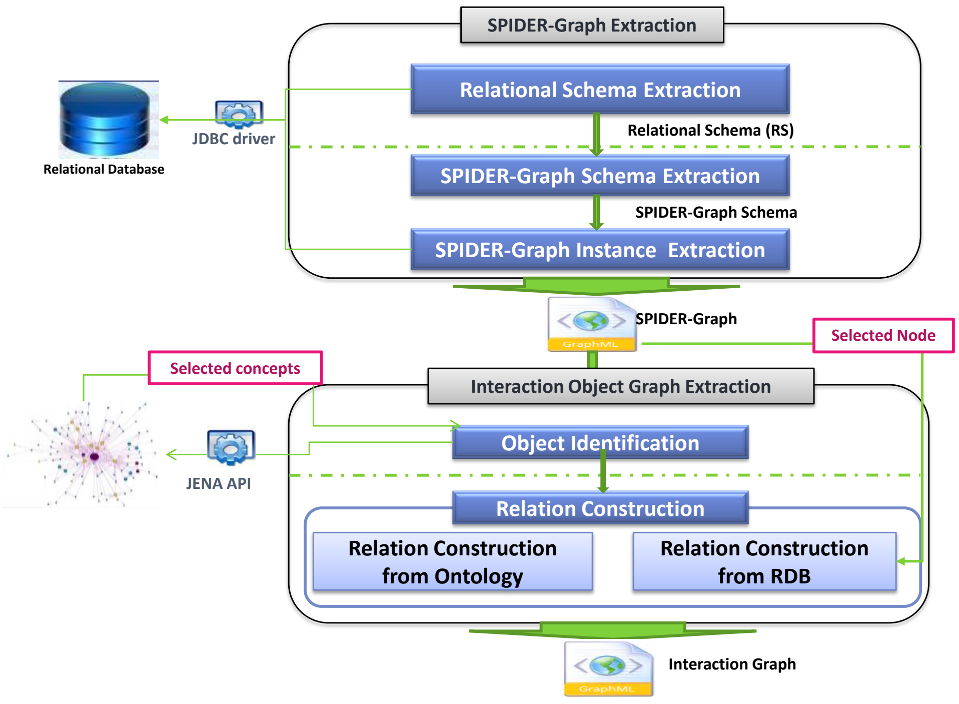


Figure 6.4: Object Graph Extraction Module.

6.1.3 The Visual Query Language

The visual query language is a sub-system, which is known to be the implementation of GraphVQL. It allows users to visualize different kind of graphs and to query them. This sub-system is divided in four modules (see Figure 6.5):

File treatment: This module contains all the classes that treat the input files. For the GraphML files containing SPIDER-Graph model a specific parser is implemented to extract information about the graph. For other file-formats (XML, GraphML), we transform them to RDF in order to facilitate the extraction of information using the SPARQL queries.

The RDF parser is based on the Jena API which can be used to create and manipulate RDF graphs.

Drawing query treatment: Using the Jung API[3], this module provides the draw functionalities and treats the input drawn graph query to extract a pattern graph query.

[1]http://prefuse.org/
[2]http://gephi.org/
[3]http://jung.sourceforge.net/

Query mapping: This module transforms the graph pattern query to the corresponding query and finds the result. It uses the ARQ component of the Jena API for the SPARQL translation. ARQ is an implementation of the SPARQL query language for Jena. It allows executing the SPARQL query with the expansions. For the SPIDER-Graph query, the graph pattern are treated via the pattern matching algorithm or the shortest path algorithm.
Graph drawing: Based on the Perfuse API, this module visualizes the resulting and the input graphs.

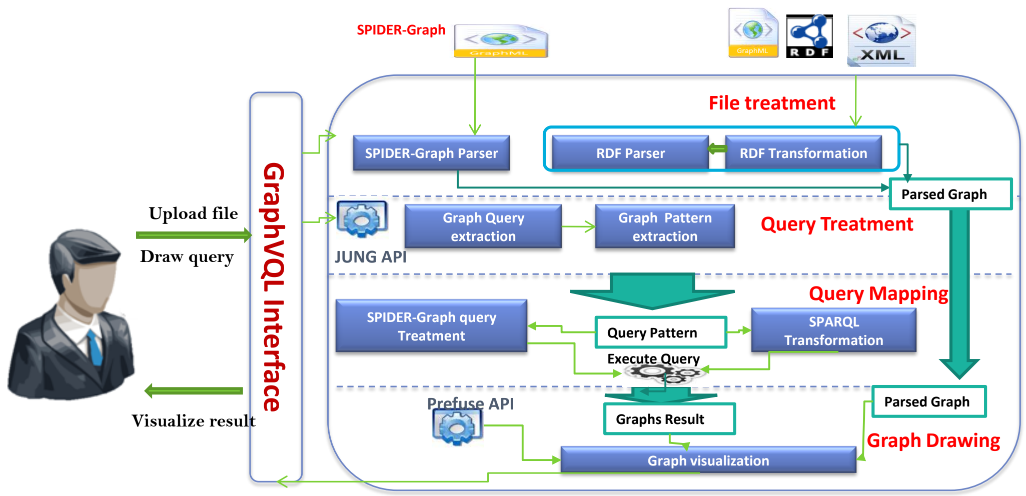
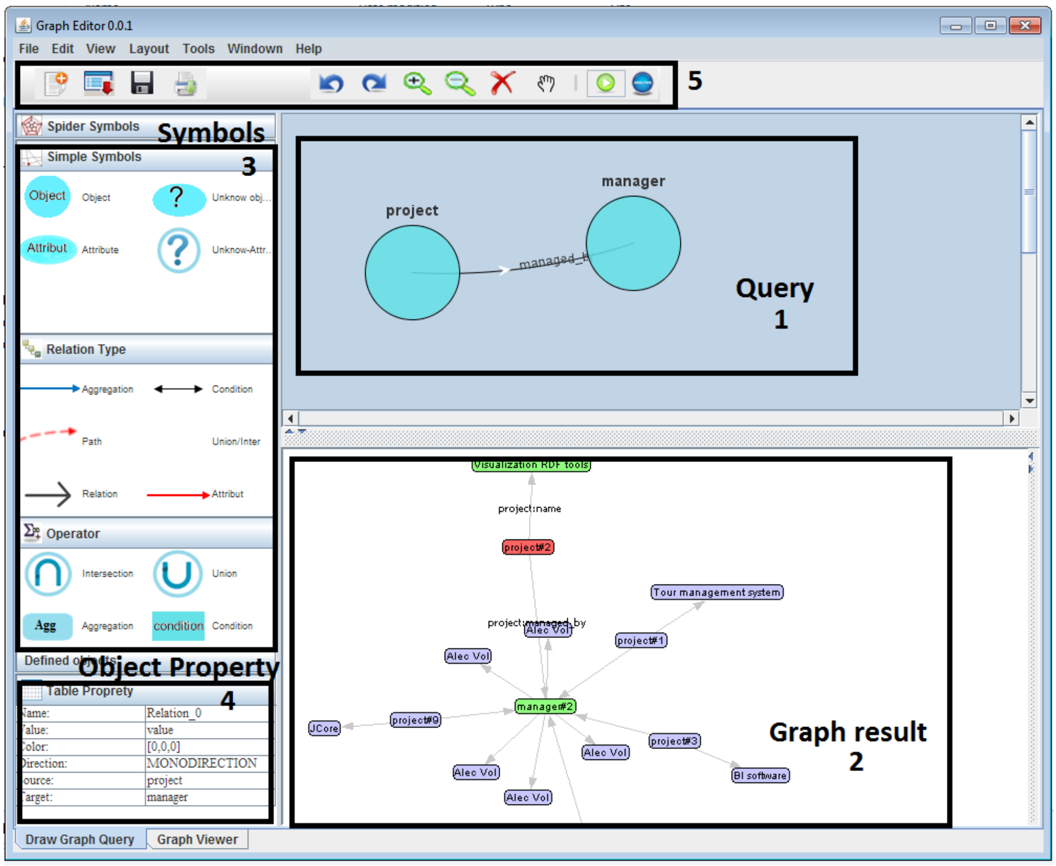


Figure 6.5: The Visual Query Language.

The visual interface of this sub-system is presented in the following screenshots.
The screenshot in Figure 6.6 shows the interface where the user can draw queries.
Figure 6.7 and Figure 6.8 show a visualization of an RDF graph and a SPIDER-Graph, respectively.

6.2 Experiment Data Sets and Evaluation Context

In the following experiments, we have used two sources of data:
(1) The ADEME database:
The used relational database is a real enterprise database which describes the thesis funded by the ADEME. The database contains 30 tables about students, their thesis, their directors and their laboratories, as well as the engineers and the co-financers. The database contains the theses of 1788 students. Each table contains in average 20 attributes. This database is used to evaluate the objects interaction graph extraction approach applied to a social network case. In order to simplify the process, we detail the process using a simplified part of the database which is exposed in Figure 6.9.
(2) Data provided from the ARSA project:
The ARSA project (Analyses des Réseaux Sociaux pour Administrations, i.e. Social



ame:	Relation_0
alue:	value
olor:	[0,0,0]
irection:	MONODIRECTION
ource:	project
arget:	manager

Figure 6.6: A snapshot of the query language interface.

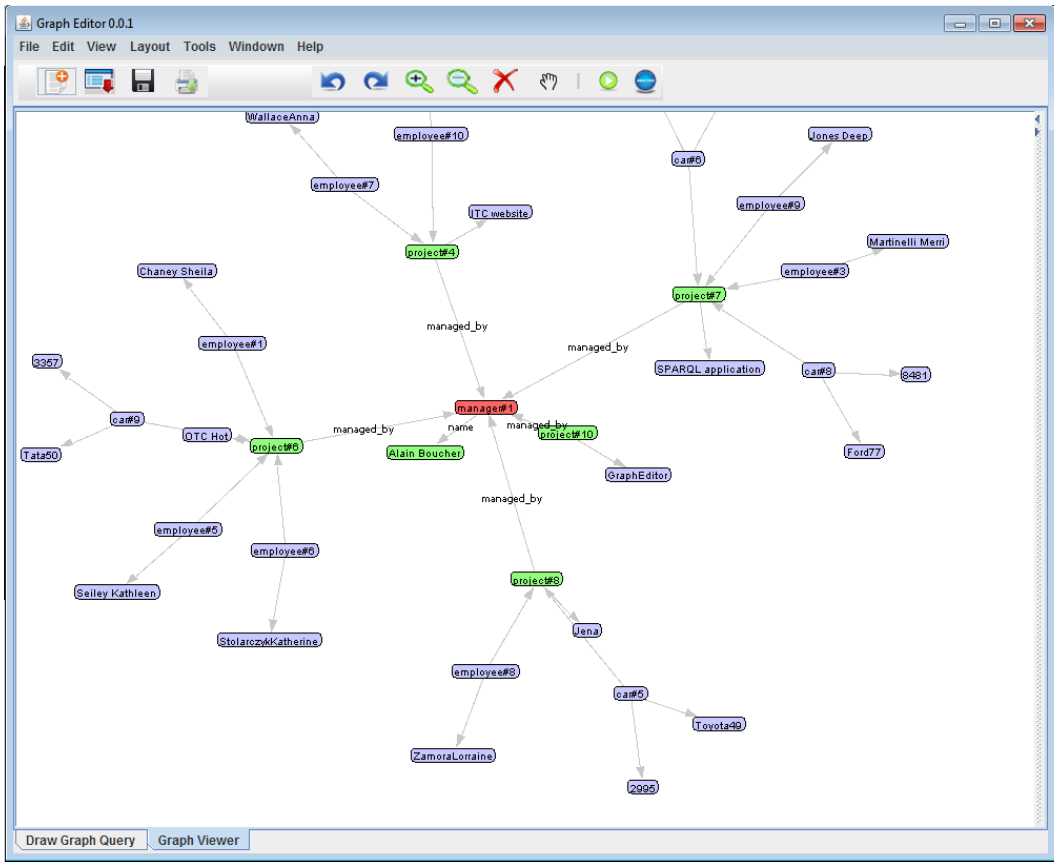


Figure 6.7: A Snapshot of a RDF Graph Visualization.

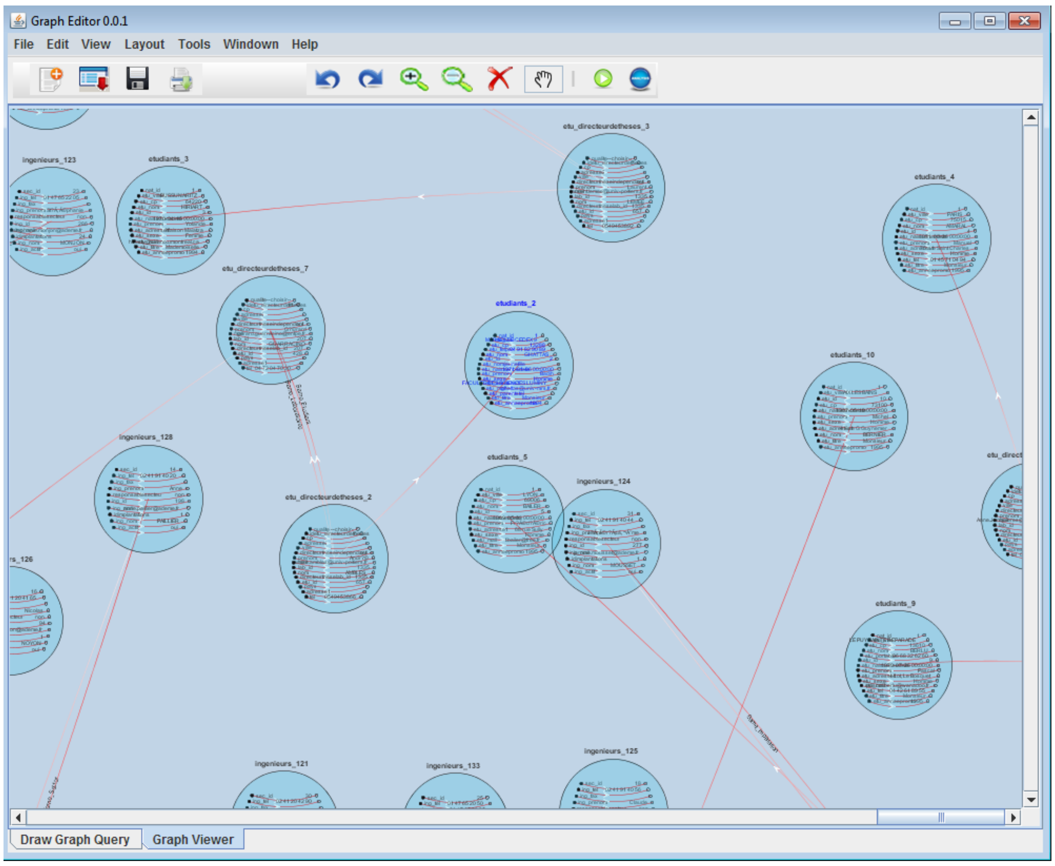
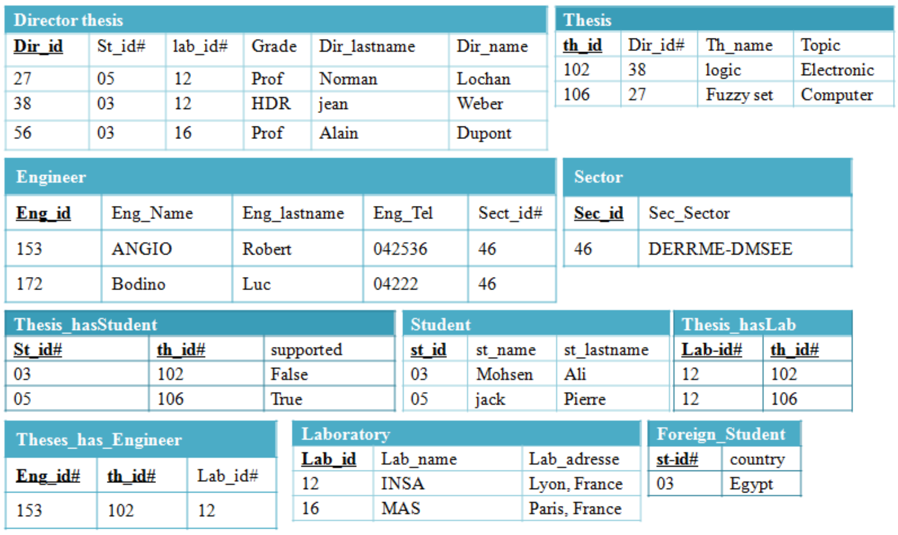


Figure 6.8: A Snapshot of a SPIDER-Graph Visualization.



Director thesis

Dir_id	St_id#	lab_id#	Grade	Dir_lastname	Dir_name
27	05	12	Prof	Norman	Lochan
38	03	12	HDR	jean	Weber
56	03	16	Prof	Alain	Dupont

Thesis

th_id	Dir_id#	Th_name	Topic
102	38	logic	Electronic
106	27	Fuzzy set	Computer

Engineer

Eng_id	Eng_Name	Eng_lastname	Eng_Tel	Sect_id#
153	ANGIO	Robert	042536	46
172	Bodino	Luc	04222	46

Sector

Sec_id	Sec_Sector
46	DERRME-DMSEE

Thesis_hasStudent

St_id#	**th_id#**	supported
03	102	False
05	106	True

Student

st_id	st_name	st_lastname
03	Mohsen	Ali
05	jack	Pierre

Thesis_hasLab

Lab-id#	**th_id#**
12	102
12	106

Theses_has_Engineer

Eng_id#	**th_id#**	Lab_id#
153	102	12

Laboratory

Lab_id	Lab_name	Lab_adresse
12	INSA	Lyon, France
16	MAS	Paris, France

Foreign_Student

st-id#	country
03	Egypt

Figure 6.9: The PhD Students Database.

Networks Analysis for Administrations) is dedicated to the promotion of the transparency in administrations using social networks. In the context of this project, we have been concerned to carry out a visual query language for novice user. Then, in order to evaluate the performance of the realized query language GraphVQL, two files from the ARSA project were used. One file was a RDF one, the other GraphML one, containing information about the Antibe town hall: agents, projects and materials. The files contain 2500 objects.

6.3 Evaluation of the Relational Database Transformation to an Object Graph: The Social Network Use Case

In this section, we apply the different processes of the object graph extraction approach from the relational database on the ADEME database in order to extract a social network. This social network describes all the interactions between the persons existing in the database. The object graph $GO := (O_I, R_O)$ extracted from the relational database (defined by the definition 19) will be a social network where O_I is the set of complex-nodes instances representing persons and R_O the relations between these persons. The extraction approach is based on two main steps:

- The relational database model transformation to a SPIDER-Graph model.
- Graph extraction according to the user chosen objects: in the case of social network the chosen objects should have the type *"Person"*.

The obtained results are presented along with the input dataset. Then, the process results and performances are evaluated.

6.3.1 Mapping the Relational Model to the SPIDER-Graph Model

6.3.1.1 The Obtained Result from the Dataset

The process of mapping the relational model to a SPIDER-Graph is based on two main processes (see section 4.3):

- The schema translation which is start by extracting the relational database schema; then, transforming it into a SPIDER-Graph schema.

- The data conversion which uses the relational database tuples and the SPIDER-Graph schema to create the SPIDER-Graph instance.

As a first step of the transformation, the SPIDER-Graph schema is extracted from the relational database schema. The extracted SPIDER-Graph schema is the following: $S = (N_{cn} \cup R)$ where:
− The set of complex-nodes N_{cn} = { *Thesis, Laboratory, Thesis_hasStudent, Student, Director_thesis, Foreign_Student, Engineer, Sector,Theses_has_Engineer* }
$- R := \{\langle \text{“}IS-A\text{”}, Foreign\_Student, Student\rangle$,
$\langle \text{“}Part-of\text{”}, Student, Thesis\_hasStudent\rangle$,
$\langle \text{“”}, Director\_thesis, Student\rangle$,
$\langle \text{“”}, Director\_thesis, Laboratory\rangle$,
$\langle \text{“”}, Thesis, Director\_thesis\rangle$,
$\langle \text{“”}, These\_has\_Engineer, Laboratory\rangle$,
$\langle \text{“}Thesis\_hasLab\text{”}, These, Laboratory\rangle$,
$\langle \text{“}Part-of\text{”}, Thesis, Thesis\_hasStudent\rangle$,
$\langle \text{“}Part-of\text{”}, Thesis, These\_has\_Engineer\rangle$,
$\langle \text{“}Part-of\text{”}, Engineer, These\_has\_Engineer\rangle$,
$\langle \text{“”}, Engineer, Sector\rangle\}$
In the extracted schema, the relation $\langle \text{“}Thesis\_hasLab\text{”}, These, Laboratory\rangle$ is provided from the table "Thesis_hasLab" which is constructed only with a primary key composed with two foreign keys Lab_id and th_id. Then, the complex-node "Thesis_hasLab" is deleted and the attributes $< Th\_id, Thesis >$ and $< Lab\_id, Laboratory >$ are added to the complex-nodes $Laboratory$ and $Thesis$, respectively.
The resulting SPIDER-Graph schema is depicted in Figure 6.10.

The second step is the data migration which consists in populating the extracted schema by using the table tuples.
By using the set of relational tuples, we instantiate each complex-node in the SPIDER-Graph schema.

For example, from the complex-node Thesis:

$$\begin{aligned} Thesis = (\text{``}Thesis\text{''} \quad ,\{ \quad & < Th\_id, Integer >,\\ & < Th\_name, String >,\\ & < Topic, String >,\\ & < Dir\_id, Director\_thesis >,\\ & < Lab\_id, Laboratory >\}) \end{aligned}$$

By using the set of tuples related to the table *Thesis* and the value of the foreign keys, this complex-node has two instances: $Thesis\_1$ and $Thesis\_2$.

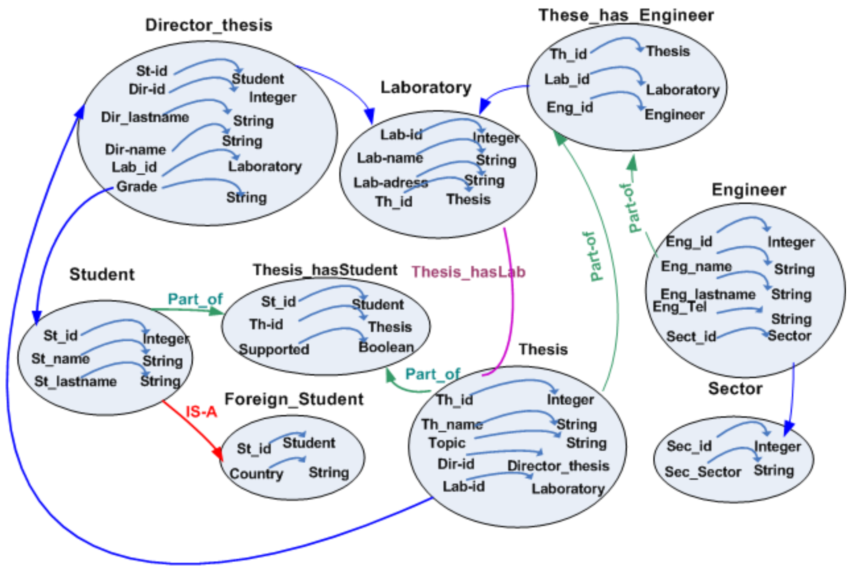


Figure 6.10: SPIDER-Graph Schema.

For example, $Thesis\_1$ is an instance of *Thesis* and is defined by:

$$\begin{aligned} Thesis\_1 = \langle Thesis, Thesis\_1 \quad ,\{ \quad & < Th\_id, Integer, 102 >,\\ & < Th\_name, String, \text{``}Logic'' >,\\ & < Topic, String, \text{``}Electronic'' >,\\ & < Dir-id, Director\_thesis, Director\_thesis\_1 >,\\ & < Lab\_id, Laboratory, Laboratory\_1 >\}\rangle \end{aligned}$$

For each relation in R in the SPIDER-Graph schema, a set of instances relations R_I is extracted using the value of keys on the relational tables.
Finally, transformed data are loaded into the SPIDER-Graph schema. An excerpt of the SPIDER-Graph instance is shown in Figure 6.11.

From the ADEME database, the following data were obtained:
-A schema SPIDER-Graph containing 30 complex-nodes and 48 relations.
-An instance SPIDER-Graph containing 12213 instance complex-nodes and 13282 instance relations.

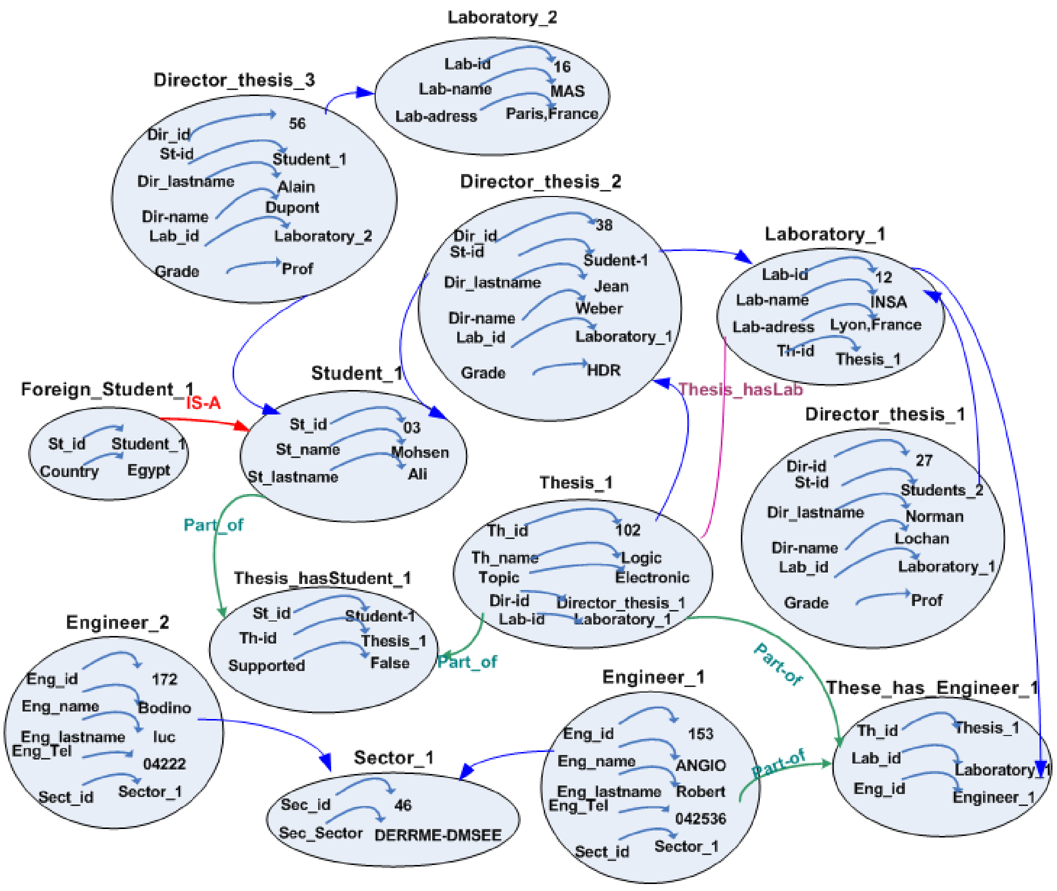


Figure 6.11: SPIDER-Graph Instance.

6.3.1.2 Evaluation

The relevance of the previous process is evaluated by using the database migration evaluation method proposed in [Maatuk 2008]. This method uses a set of queries which have been designed to observe any differences between the source database and the target database. If the same result is obtained, it means that the transformation method is correct, and no data is lost by using it.
In our case, the input database is a relational database and the resulting SPIDER-Graph is stored in a GraphML file. For this evaluation, we have stored the resulting graph in a relational database in order to perform some SQL queries in the two data models and to evaluate the approach.
For this, we have modeled the SPIDER-Graph model using a relational model. The proposed schema is composed by four tables:
-A complex-node table for the complex-nodes in the SPIDER-Graph schema.
-A complex-node_IN table for the complex-nodes instances relatives to each complex-node.
-Relation and Relation_IN tables store information about the relations between complex-nodes and complex-nodes instances, respectively.
For the purpose of securing a better modeling of the SPIDER-Graph, the database schema is incremented by two new types: Node and Node_IN to facilitate the storage of the complex-nodes attributes. This an example of the used queries:

Query1: Find the Thesis id of the student having the st_id=03;
RD: *Select th_id from Student where st_id='03';*
SGD: *Select listn[2].value from complex-node_ins where complex-node_name='Student' and listn[1].value='03';*
Query2: Find the engineer name working on the th_id='102';
RD: *Select E.Eng_name from thesis_has_Engineer T, Engineer E where T.Eng_id=E.Eng_id;*
SGD: *Select T1. listn[2] from complex-node_ins T,complex-node_ins T1 where T.complex-node_name ='theses_has_ingenieurs' and T1.complex-node_name ='ingenieurs' and T.listn[7]=T1.listn[10];*

After comparing the results between the two databases, the SPIDER-Graph schema and instance are generated without loss (the same result is found) or redundancy (the result is not duplicated) of data. It proves the correctness of this transformation.

6.3.2 Evaluation of the Object Graph Extraction from the SPIDER_Graph: The Social Network Case

6.3.2.1 The Resulting Social Network

In order to evaluate the object graph extraction process, we have chosen to transform the extracted SPIDER-Graph (instance in Figure 6.11 and schema in Figure 6.10) to a Social network. The object Graph extraction approach is based on two main steps:

1. Object identification
2. Relation construction

A. Result of the object identification process

The chosen objects in this case are the complex-nodes representing the persons. These objects are identified using the object identification process (see section 4.4.2). The SPIDER-Graph schema is used as an input to extract the type of candidate complex-nodes (those which may represent persons). Then, the instances of these complex-nodes are put in set O_I.

The algorithm proposed in section 4.4.2 is composed of three main steps:

1. Complex-nodes and concepts Names treatment,
2. String matching between the complex-nodes names and the chosen concepts labels.
3. Candidates objects treatment.

In the social network case the selected concept is "*Person*". In this particular case, we won't compare the label of the concept "*Person*" with the name of the existing complex-nodes in the SPIDER-Graph schema instead we proceed directly to the third step. Indeed, "Person" is a generic concept and it is unusual to find table with the name "Person". The complex-nodes attributes are compared with the attributes (Datatype properties) of the concept "*Person*". A person has a number of characteristics like name, surname, birthday, address, email and etc. The enterprise ontology already built (see the enterprise ontology chapter) contains the concept "*Person*" (see Figure 6.12) with its different characteristics (described by the datatype properties).

In order to compare each complex-node *CN*, attributes are compared with the datatype properties of the concept "*Person*", using the following similarity measure presented in the section 4.4.2:

$$Simatt(C_{Person}, CN) = \frac{\sum_{n_{C_{Person}} \in DP} \sum_{a_{cn} \in A_{nc}} sim(n_{C_{Person}}, n_{a_{cn}})}{|DP| + |A_{nc}|}$$

Where $n_{C_{Person}}$ is the name of a datatype property related to the concept *Person* and a_{cn} is the name of an attribute of the complex-node *CN*.

The proposed similarity is based on the name similarity $sim(n_{C_{Person}}, n_{a_{cn}})$ of each attribute of the complex-node and each datatype of the concept person. The result is the average of these similarities.

If $simatt(C_{Person}, CN) > \beta$ (where β is a threshold value that we will specify in the next section) then *CN* is added to the objects set.

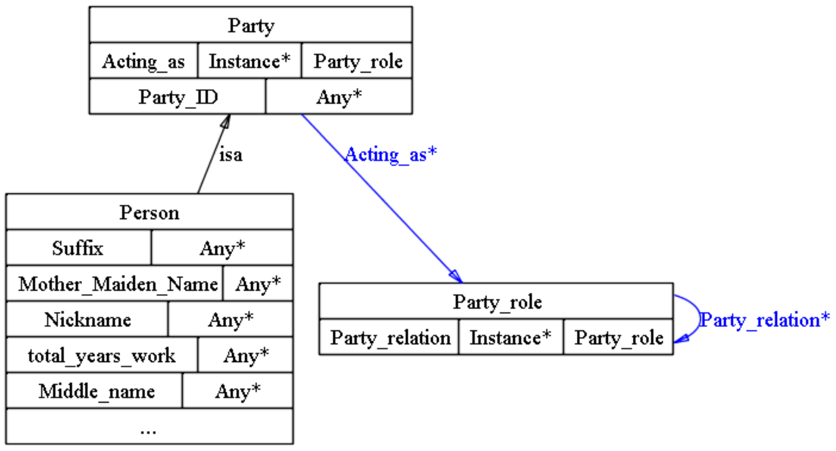


Figure 6.12: The Concept "Person".

By applying this process to the SPIDER-Graph schema presented in Figure 6.10, the complex-nodes representing a person detected are: "Student", "Director_thesis" and "Engineer". Then, all their instances are added to the object set O_I.

B. Result of the Relations Construction Process

Here, we apply the relation construction process presented in section 4.3.1.1. The relation construction process is performed using the identified objects set O_I and the predefined relation patterns. The extracted relations for each kind of pattern is described below.

B.1. Identification of New Objects

By using the "IS-A" relation pattern and the relation $r =< \text{"}IS-A\text{"}, Foreign-Student, Student >$, the process identifies "Foreign-Student" as an object. Then, the instances of the complex-node *Foreign-Student* are added to the set O_I.

B.2. Relations between Chosen Objects

By using the set of patterns already defined in section 4.4.3, we detect the hidden relations between the O_I objects. This process uses each existing relation in the input SPIDER-Graph schema.
From the relation $R_{cn1} :=< \text{""}, Director\_thesis, Student >$, two patterns are identified (Table 6.1):

- $Pr_1 :< \text{""}, Director\_thesis, Student, null >$: In the SPIDER-Graph schema, *Student* and *Director_thesis* share the relation R_{cn1}. Pr_1 indicates that the R_{cn1} instances that exists in the SPIDER-Graph instance relations should be added to R_o.

- $Pr_2 :=< \text{"}Same\_Student\text{"}, Director\_thesis\_i, Director\_thesis\_j, Student >$: Two thesis directors may have the same *Student* (same value of $St-id$). Then, the SPIDER-Graph is then searched for all instances of *Director_thesis* which have the same *Student* (mediator for this pattern) in order to add between them the relation *Same_ Student*.

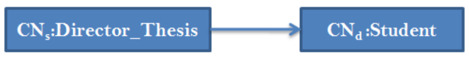
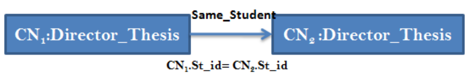
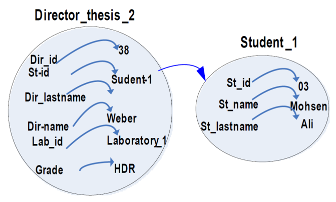
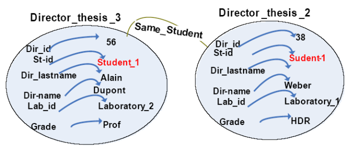
Relation and identified pattern	Example of extracted relations
Relation: R_{cn1}:=< "", $Director\_thesis, Student,$ > **Patterns:** Pr_1 :=< "", $Director\_thesis, Student, null$ >  Pr_2 :=< $Same\_Student$, $Director\_thesis\_i$, $Director\_thesis\_j$, $Student$ >  Same_Student CN$_1$:Director_Thesis CN$_2$:Director_Thesis CN$_1$.St_id= CN$_2$.St_id	 Director_thesis_2 Student_1 Dir_id St-id Dir_lastname Dir-name Lab_id Grade 38 Sudent-1 Weber Laboratory_1 HDR St_id St_name St_lastname 03 Mohsen Ali  Director_thesis_3 Same_Student Director_thesis_2 Dir_id St-id Dir_lastname Dir-name Lab_id Grade 56 Student_1 Alain Dupont Laboratory_2 Prof 38 Sudent-1 Weber Laboratory_1 HDR

Table 6.1: Relation R_{cn1} Pattern.

In the case when no relations exist between the identified objects, we try to search the path or the semi-path between these objects. Afterwards, it is investigated if there is a path between ("Student", "Engineer") and ("Engineer","Director_thesis") by applying the Dijkstra algorithm.

- For ("Student", "Engineer"): The semi-path Pr_{path} :=< "", $Engineer$,$Thesis\_has\_Engineer$, $Laboratory, Thesis, Thesis\_has\_student, Student$ > is detected.
- For ("Director_thesis","Engineer"): The semi-path Pr_{path} :=< "", $Engineer$, $Thesis\_has\_Engineer, Thesis, Director\_thesis,$ > is detected.

B.3. Relations between Chosen Objects and Other Complex-Nodes

In this step, we use the predefined patterns in Table 4.3 to discover new relations between the detected objects using their relations with other complex-nodes (complex-nodes that are not in the set O_I).

From the relation R_{cn2} :=< "", $Director\_thesis, Laboratory$ >, one pattern is identified (Table 6.2): $Laboratory$ is not an identified object and therefore its instances are not included in the final graph:

- Pr_3 :=< $Same\_Laboratory, Director\_thesis\_i$, $Director\_thesis\_j$, $Laboratory$ >: using the value of the foreign key Lab_id in each complex-node instance of the object $Director\_thesis$, we will link those having the same value of Lab_id by the relation $Same\_Laboratory$.

From the relation R_{cn3} :=< "", $Engineer, Sector$ >, one pattern is identified (Table 6.3):

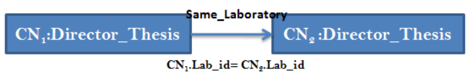
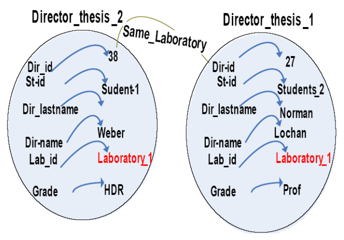
Relation and identified pattern	Example of extracted relations
Relation: $R_{cn2} :=< ``", Director\_thesis, Laboratory >$ **Pattern:** $Pr_3 :=< Same\_Laboratory, Director\_thesis\_i, Director\_thesis\_j, Laboratory >$  Same_Laboratory CN1:Director_Thesis CN2 :Director_Thesis CN1.Lab_id= CN2.Lab_id	 Director_thesis_2 Same_Laboratory Director_thesis_1 Dir_id St-id 38 Student-1 Dir_lastname Weber Dir-name Lab_id Laboratory_1 Grade HDR Dir-id St-id 27 Students 2 Dir_lastname Norman Lochan Dir-name Lab_id Laboratory_1 Grade Prof

Table 6.2: Relation R_{cn2} Pattern

- $Pr_3 :=< Same\_Sector, Engineer\_i, Engineer\_j, Sector >$: using the value of the foreign key *Sect_id* in each complex-node instance of the object *Director_thesis*, we will link those having the same value of *Sect_id* by the relation *Same_Sector*.

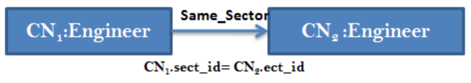
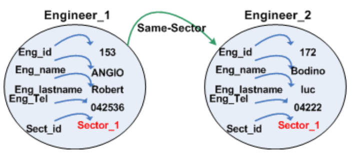
Relation and identified pattern	Example of extracted relations
Relation: $R_{cn3} :=< ``", Engineer, Sector >$ **Pattern:** $Pr_3 :=< Same\_Sector, Engineer\_i, Engineer\_j, Sector >$  CN1:Engineer Same_Sector CN2 :Engineer CN1.sect_id= CN2.ect_id	 Engineer_1 Same-Sector Engineer_2 Eng_id 153 Eng_name ANGIO Eng_lastname Robert Eng_Tel 042536 Sect_id Sector_1 Eng_id 172 Eng_name Bodino Eng_lastname luc Eng_Tel 04222 Sect_id Sector_1

Table 6.3: Relation R_{cn3} Pattern

From the relation $R_{cn4} :=< ``Part-of", Student, thesis\_hasStudent >$, we identify one pattern (Table 6.4) and we add some information:

- *Thesis_hasStudent* shares two relations "Part-of" with *Student* and *Thesis*. We add a new attribute on the complex-node $Student < Thesis, Thesis\_i >$, corresponding to his Thesis. Then, we can apply the pattern Pr_4.

- $Pr_4 :=< Same\_Thesis, Student\_i, Student\_j, Thesis\_hasStudent >$, by this pattern all the students who share the same thesis are searched. No semantically inexact relation is found.

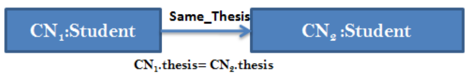
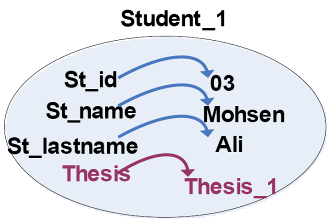
Relation and identified Pattern	Example of extracted relation
Relation: $R_{cn4} :=< \text{“}Part-of\text{”}, Student, thesis\_hasStudent >$ **Pattern:** - $Thesis\_hasStudent$ shares two relations “Part-of” with $Student$ an $Thesis$ - Add the attribute $n :< Thesis, Thesis\_i >$ to each instance of $Student$ - $Pr_4 :=< Same\_Thesis, Student\_i, Student\_j, Thesis\_hasStudent >$  CN1:Student Same_Thesis CN2 :Student CN1.thesis= CN2.thesis	 Student_1 St_id 03 St_name Mohsen St_lastname Ali Thesis Thesis_1

Table 6.4: Relation $Rcn4$ Pattern.

From the relation $R_{cn5} :=< \text{“}Part-of\text{”}, Engineer, thesis\_has\_Engineer >$, we identify one pattern (Table 6.5) and we add some information:

- $thesis\_has\_Engineer$ shares two relations “Part-of" with $Engineer$ and $Thesis$. An attribute is added to the the complex-node $Engineer < Thesis, Thesis\_i >$, corresponding to his Thesis. Then the pattern Pr_4 can be applied.

- $Pr_4 :=< Same\_Thesis, Engineeri, Engineer_j, thesis\_has\_Engineer >$, with all the engineers sharing the same thesis are searched.

From the relation $R_{cn6} :=< \text{“}'', Thesis, Director\_thesis >$, there are no identified patterns because $Thesis$ is not related to other objects (Table 6.6).

Finally, the schema of the resulting social network is depicted in Figure 6.13 and the corresponding social network in Figure 6.14.

The resulting social network contains: 1788 complex-nodes “Student" instances, 1735 complex-nodes “Directeur-thesis" instances and has 303 complex-nodes “Engineer" instances. The approach built 47844 relations between these instances.The number of relations is higher than the number of relation in the relational database. This is due the fact that from each detected relation in the database, the relation extraction process can extract one or two new relation. We can reduce the relation number by adding a filter to the relation extraction process which calculate the importance of the relation to add to the user.

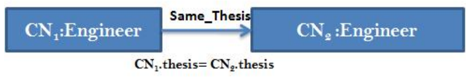
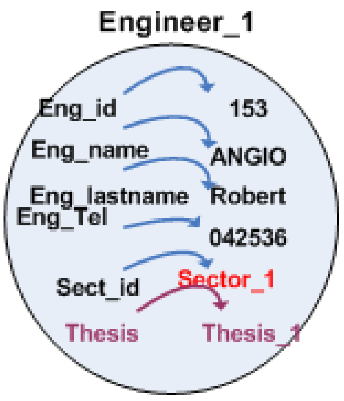
Relation and identified Pattern	Example of extracted relation
Relation: $R_{cn5} :=< \text{“}Part-of'', Engineer,$ $thesis\_has\_Engineer >$ **Pattern:** - $Tthesis\_has\_Engineer$ shares two relations “Part-of” with $Engineer$ an $Thesis$ - Add the attribute n :< $Thesis, Thesis\_i >$ to each instance of $Student$ - Pr_4 :=< $Same\_Thesis, Engineer_i, Engineerj,$ $Thesis\_hasStudent >$  CN₁:Engineer Same_Thesis CN₂ :Engineer CN₁.thesis= CN₂.thesis	 Engineer_1 Eng_id 153 Eng_name ANGIO Eng_lastname Robert Eng_Tel 042536 Sect_id Sector_1 Thesis Thesis_1

Table 6.5: Relation $Rcn5$ Pattern.

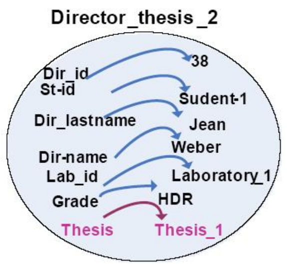
Relation and identified Pattern	Example of extracted relation
Relation: R_{cn4} :=< “”, $Thesis, Director\_thesis >$ **Pattern:** - Thesis has no relations with other objects then no pattern detected. - Add the node n :< $Thesis, Thesis\_i >$ to each instance of Student	 Director_thesis_2 Dir_id St-id Dir_lastname Dir-name Lab_id Grade Thesis 38 Sudent-1 Jean Weber Laboratory_1 HDR Thesis_1

Table 6.6: Relation R_{cn4} Pattern

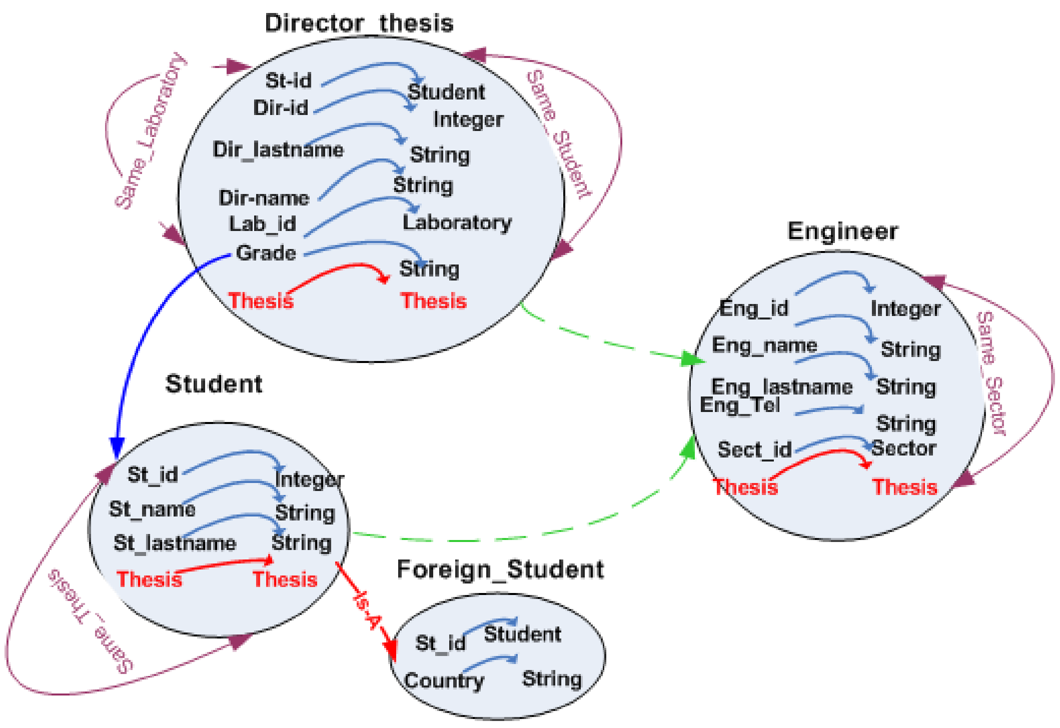


Figure 6.13: The Schema of the Social Network.

6.3.2.2 Evaluation

The pertinence of the previous process was evaluated in two steps. First, the object identification was evaluated using the recall and precision measures:

Precision= (number of identified objects correctly)/(total of identified objects)
Recall=(number of identified objects correctly)/(total of objects to identify)

In the case of the ADEME database, the application detected all the tables containing persons because the attributes tables were well designed.
However, the precision and the recall of the approach depend on the α value (threshold of the name similarity) and β value (threshold of the attributes similarity, simatt). The value variations of the precision are represented by the schema in Figure 6.15 and for the recall are presented in Figure 6.16.

From the graphic, we can notice that when α and β have a high values the result precision increase and the recall decrease.
α has more influence on the recall value. α controls the similarity measure to select the candidate complex-nodes to be filtered. Then, when α has a high value, the number of candidates increases and the recall increases. However, this can decrease the candidate number and then we can lose some objects. Thus, from the evaluation we take a high value of $\beta = 0.7$ and $\alpha = 0.4$ in order to guaranty a good recall and high precision.

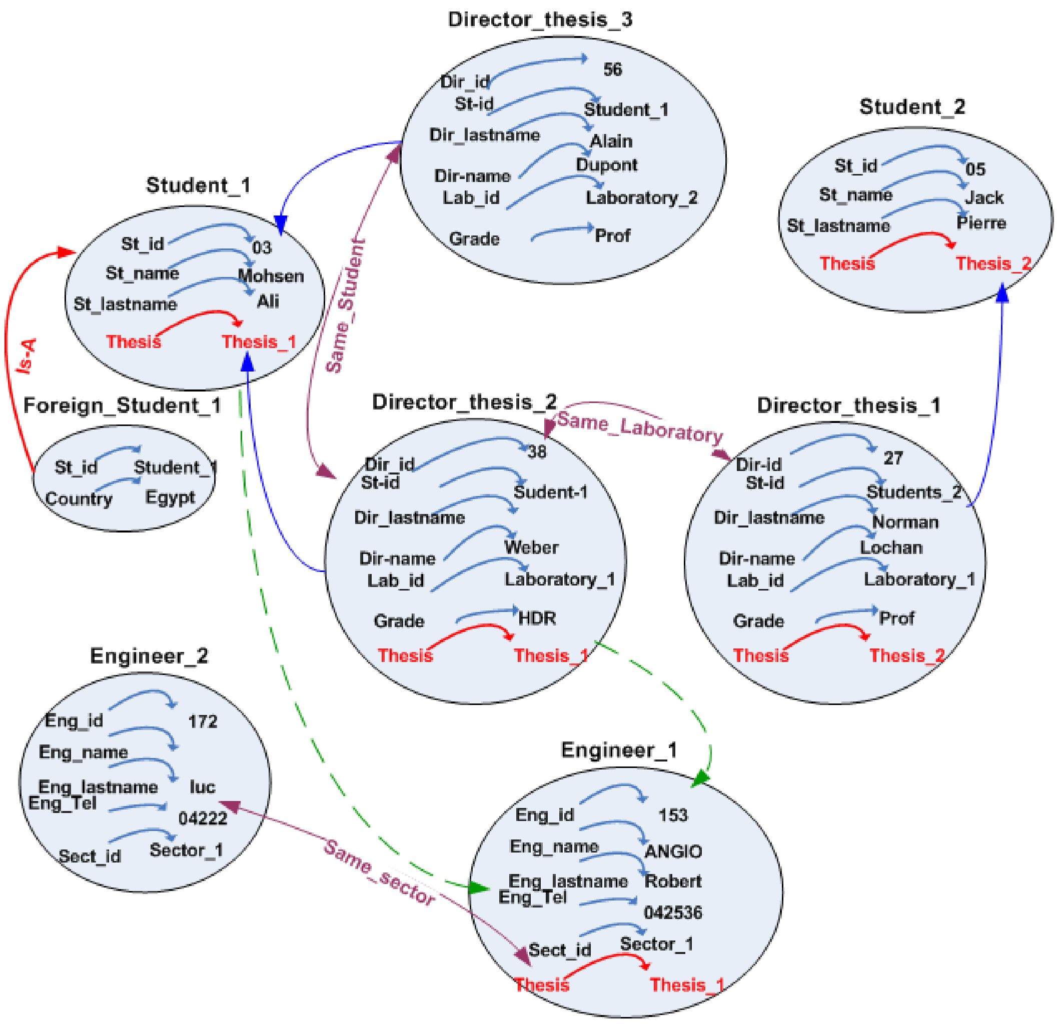


Figure 6.14: The Extracted Social Network.

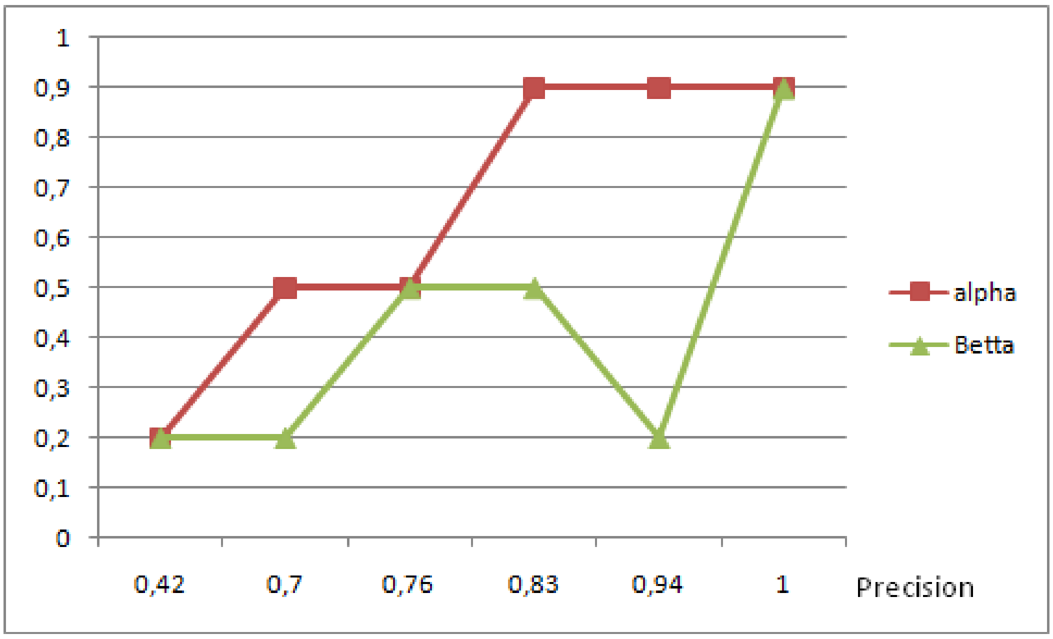


Figure 6.15: Variation of the Precision Depending on β and α Value.

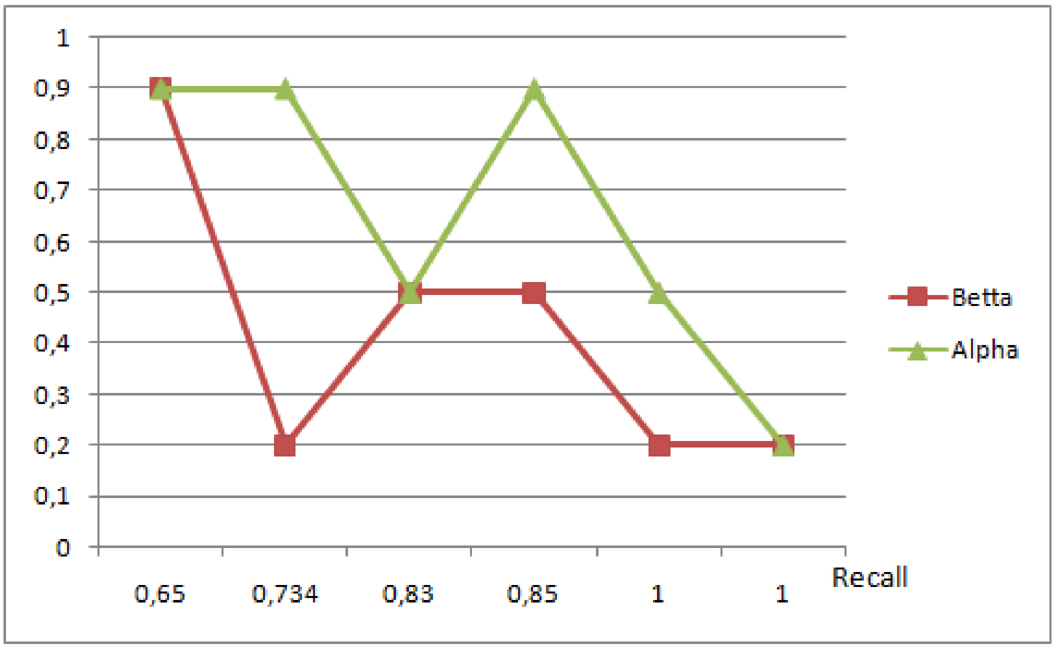
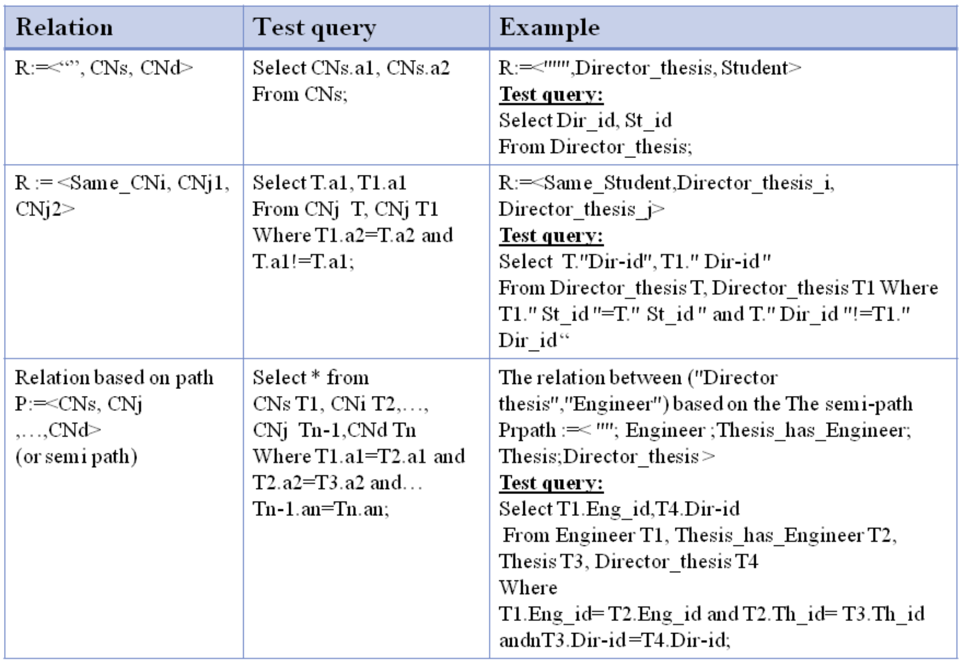


Figure 6.16: Variation of the Recall Depending on β and α Value.

In order to evaluate the relation construction process, we have defined a set of SQL queries in order to verify the correctness of the extracted data. These queries are used to search the existence of the extracted relations on the initial relational database. We summarized some examples of these queries in table 6.17.



Relation	Test query	Example
R:=<"", CNs, CNd>	Select CNs.a1, CNs.a2 From CNs;	R:=<"""",Director_thesis, Student> **Test query:** Select Dir_id, St_id From Director_thesis;
R := <Same_CNi, CNj1, CNj2>	Select T.a1, T1.a1 From CNj T, CNj T1 Where T1.a2=T.a2 and T.a1!=T.a1;	R:=<Same_Student,Director_thesis_i, Director_thesis_j> **Test query:** Select T."Dir-id", T1." Dir-id" From Director_thesis T, Director_thesis T1 Where T1." St_id "=T." St_id " and T." Dir_id "!=T1." Dir_id"
Relation based on path P:=<CNs, CNj ,…,CNd> (or semi path)	Select * from CNs T1, CNi T2,…, CNj Tn-1,CNd Tn Where T1.a1=T2.a1 and T2.a2=T3.a2 and… Tn-1.an=Tn.an;	The relation between ("Director thesis","Engineer") based on the The semi-path Prpath :=< ""; Engineer ;Thesis_has_Engineer; Thesis;Director_thesis > **Test query:** Select T1.Eng_id,T4.Dir-id From Engineer T1, Thesis_has_Engineer T2, Thesis T3, Director_thesis T4 Where T1.Eng_id= T2.Eng_id and T2.Th_id= T3.Th_id andnT3.Dir-id=T4.Dir-id;

Figure 6.17: SQL queries for the relation extraction process evaluation.

The SQL test queries, presented on the second column, are generated automatically from the new extracted relations. This is feasible because the complex-nodes names preserve the tables' names. The queries model the extracted relations and their results should identify the objects of the graph relations. For example, for the relation $R =<$ "", $Director\_thesis, Student >$, the query should find all the couple of Director_thesis and Student which share this relation and it should be verified that is the same couple of objects having this relation in the social network.
In table 6.17, In the first rows, $a1$ is the first attribute related to CNs and it corresponds to the primary key of the table from which CNs is extracted. $a2$ is the attribute of CNs which references CNd.
In the third row, in the case of $i < n$: ai is the primary key of Ti (the first attribute of CNi which is extracted from Ti) and ai is the reference of Ti in $Ti + 1$.
if $i = n$, an is the primary key of Tn and an is the reference of Tn in $Tn - 1$.
The evaluation of the queries has shown the pertinence of the process. However, the extraction of the SQL query becomes more and more complicated especially where the complex-nodes are not directly connected. For instance, to evaluate a relation

based on a path, we need multiple joins that sometimes are hard to find in the relational database.

6.4 GraphVQL Evaluation

In order to evaluate the performance of the query language GraphVQL, we have calculated the time response according to the input graph size. The evaluation will be divided into two parts: queries on the RDF files and queries on the SPIDER-Graph model.

6.4.1 Queries on Simple Graph

In the case of simple graphs, the input graphs in GraphVQL system are transformed to a RDF model. Then, in order to evaluate the time response of GraphVQL, we have used different query types to query RDF graphs with various sizes. The first graph is a synthetic graph about projects and employees, the second graph is a sub-graph from the ARSA data and the third one is the graph of the ARSA data (12213 nodes and 13282 edges). The evaluation results are presented in Figure 6.18. The presented performance values are the average values founded with a set of different queries.

For the select queries, a query that contains many constraints are executed faster than a select all query. Also, for the path query, the length of the path influences the time response. The select queries used for the evaluation cantain many constraints. In this case, the path query can be more faster than a select query. For the social

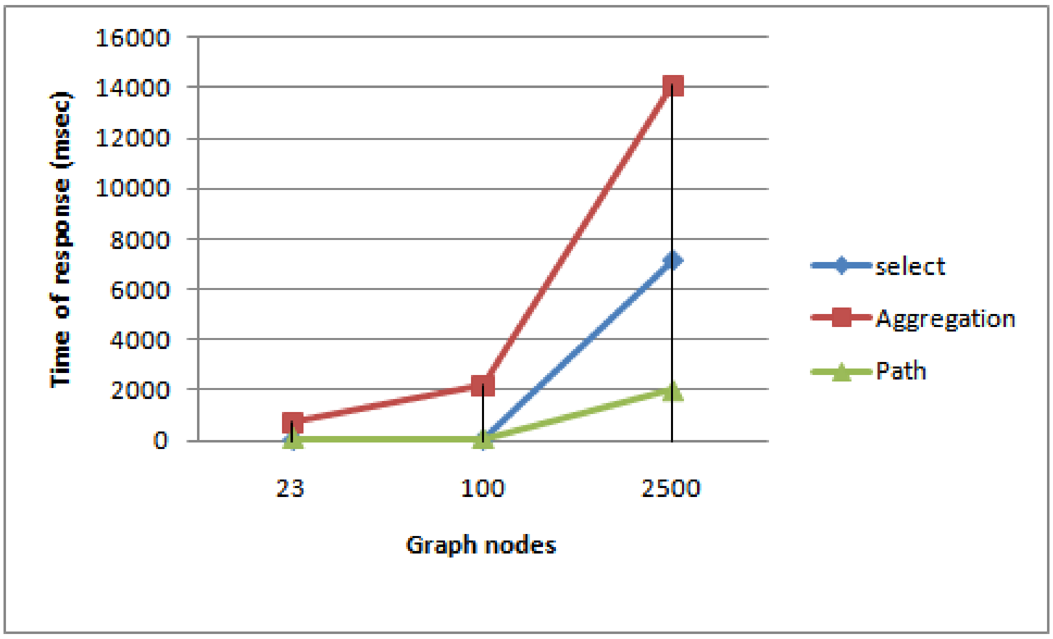
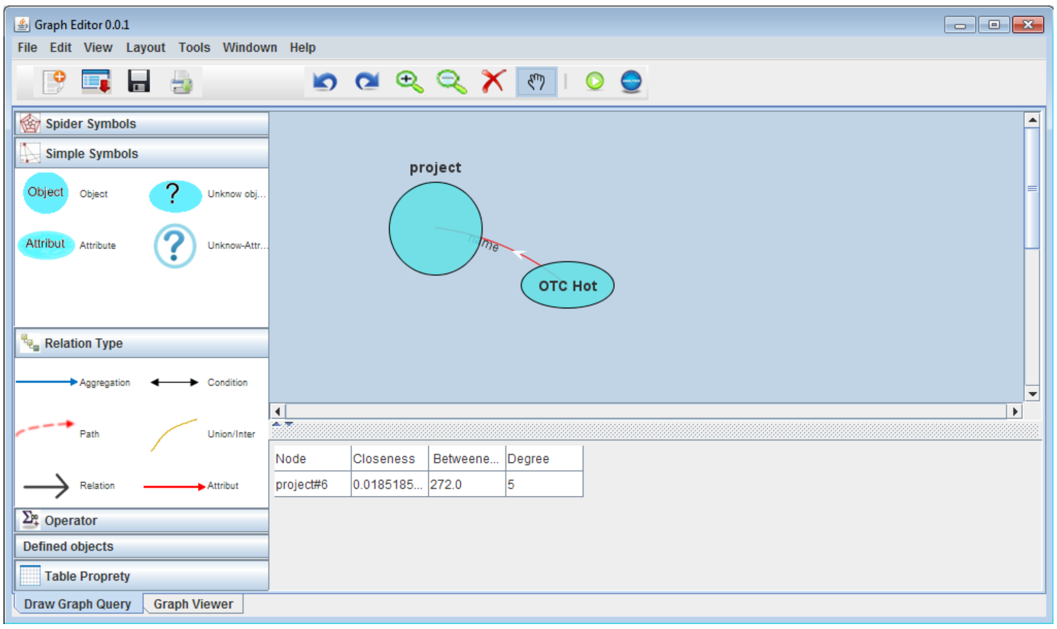


Figure 6.18: Performance of the GraphVQL Queries on a RDF Graphs.

network analysis query, the system gives the user a table containing the information about the closeness, degree and betweenness. Figure 6.19 shows an example of a SNA query which calculates the different measures for a specific project. This query is executed on the synthetic graph.
The high value of betweenness shows the greater amount of the project influence over what happens in a network. The positive degree value shows that this project is active in the network. The closeness is near to zero because the first graph is divided in two unconnected graphs, which means that this project can not be reached by other objects from the second graph.



Node	Closeness	Betweene...	Degree
project#6	0.0185185...	272.0	5

Figure 6.19: Social Network Analysis Query on a RDF Graph.

6.4.2 Queries on SPIDER-Graph Evaluation

The SPIDER-Graph queries are based on two algorithms: The graph pattern matching algorithm for the selection queries and the shortest path algorithm for the path queries. Below, the performance of each query type is given.

6.4.2.1 Selection Queries

We have evaluated these queries by varying the input queries size (number of nodes used on the query) and the graph size. The first chart 6.20 represents the time of response of the graph pattern matching algorithm according to the number of nodes used in the query, the use or not of the pruning by profiles on the algorithm presented in the section 5.4.2 and if the graph query pattern contains attributes. This evaluation is made on the SPIDER-Graph instance extracted from the ADEME database (12213 nodes and 13282 edges). The results show that for a graph pattern

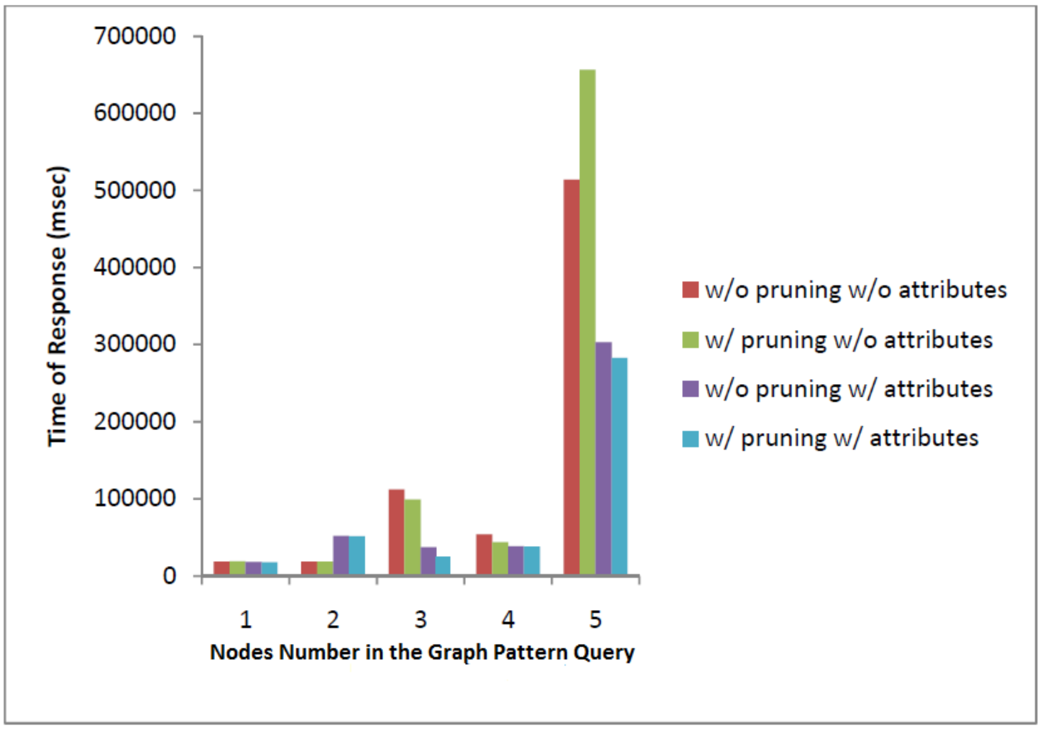


Figure 6.20: Performance of the Graph Pattern Matching: Pruning Technique and Attributes Influence.

query without attributes or with few attributes (one attribute for each node), the use of the pruning techniques accelerates the time response when the graph contains attributes. When the pattern query contains more nodes, the use of attributes decrease the candidate number and it accelerates the search process.
The second analysis concerns the size of the input graphs. We have used three graphs. The first one is a social network composed of 21 nodes, 94 edges and with no attributes in the nodes. In this case, the performance obtained is represented in Figure 6.21.

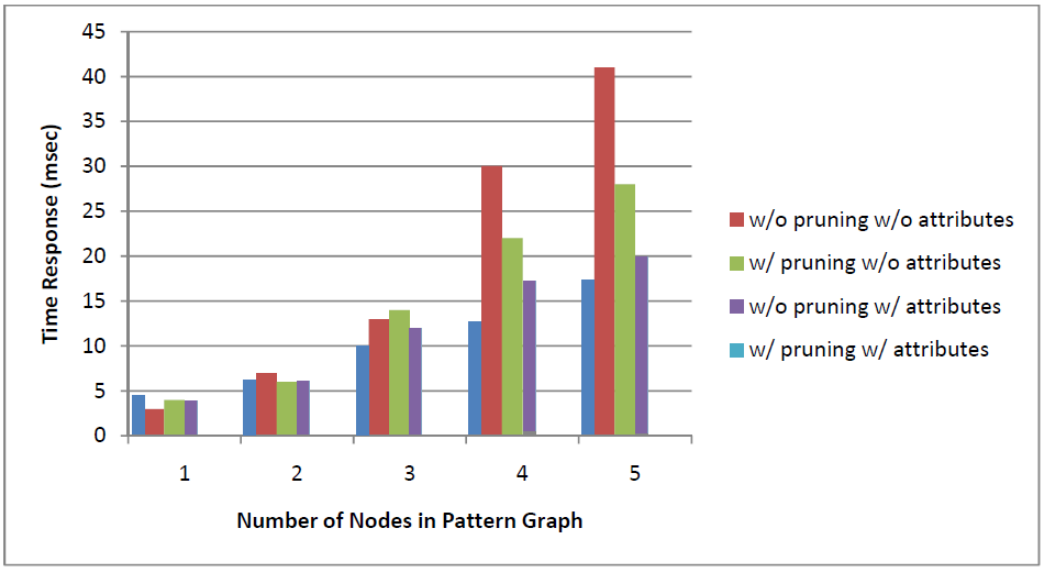


Figure 6.21: Performance of the Graph Pattern Matching with the Social Network.

The second graph is composed of 135 nodes and 642 edges, representing the relations between Computer Science researchers and their publications (see Figure 6.22). Each node has attributes attached. The third one is the SPIDER-Graph instance extracted from the ADEME database (12213 nodes and 13282 edges).

Comparing all the graphs together (see Figure 6.23), we obtained the following result:

The time response used in each case is the average of the results presented before. It is noticeable that when it comes to a very large graph, the time of response increases considerably, though, it can be considered as implement complementary techniques to reduce the time, once any small perceptual decrease in the time spent represents a sensitive difference for the final user.
Finally, considering the results presented, we can conclude that the presence of attributes in the graph makes the search much slower, but, the use of this attributes in the pattern makes the number of candidates decrease substantially.

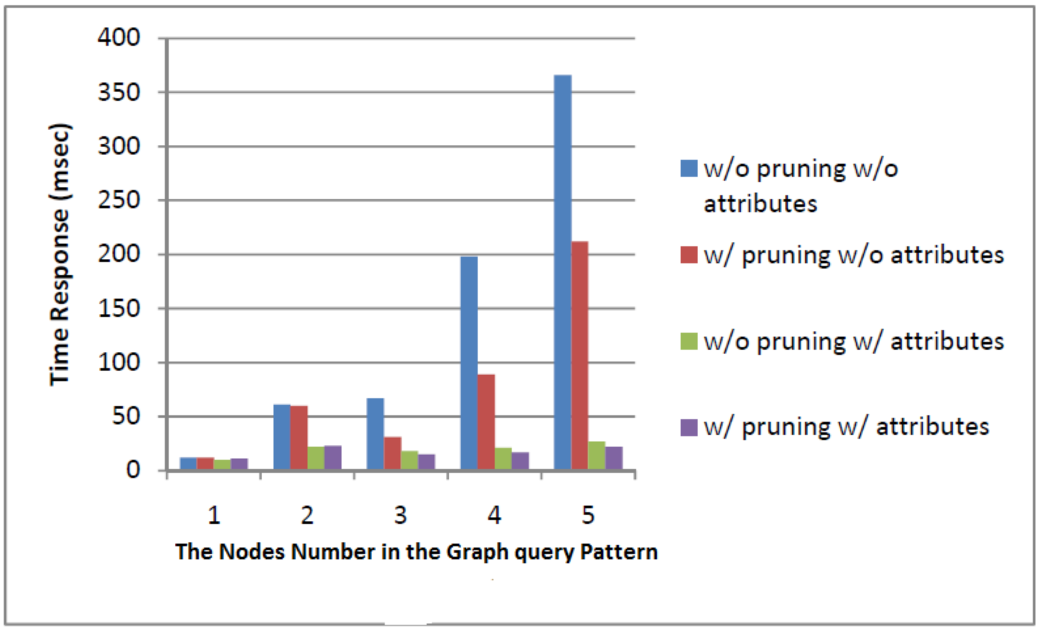


Figure 6.22: Performance of the Graph Pattern Matching with the Second Graph.

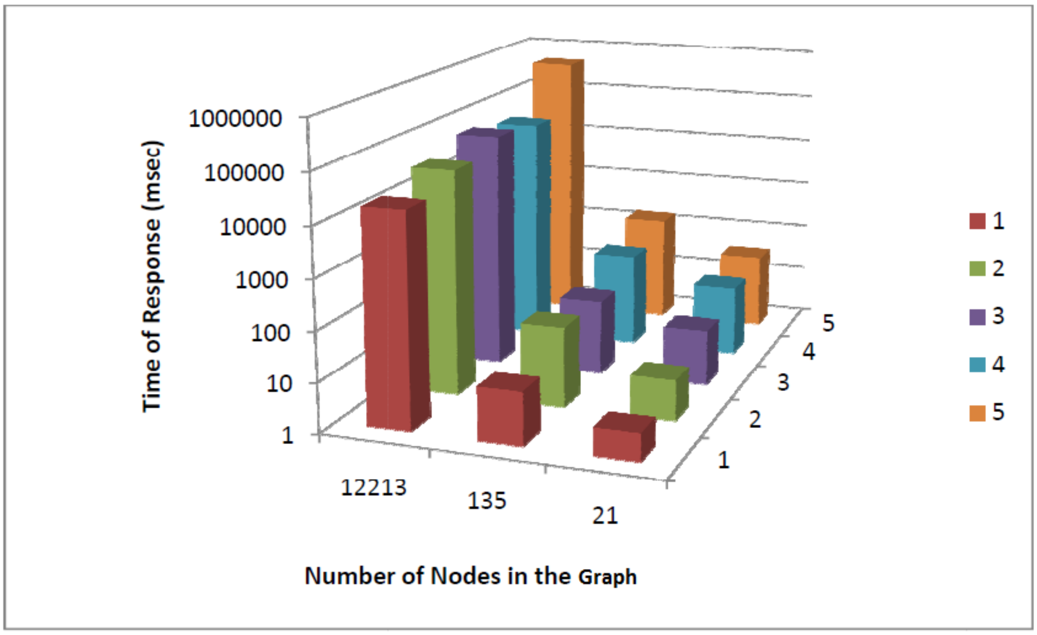


Figure 6.23: Performance of the Graph Pattern Matching: Graph Size Influence.

6.4.2.2 Path Queries

We have evaluated the performance of the path query process by varying the size and the type of the searched path using the ADEME graph. We have used different path queries size: Queries containing between two or five nodes that are with or without attributes.

In Figure 6.24, we present two examples of the used path queries.

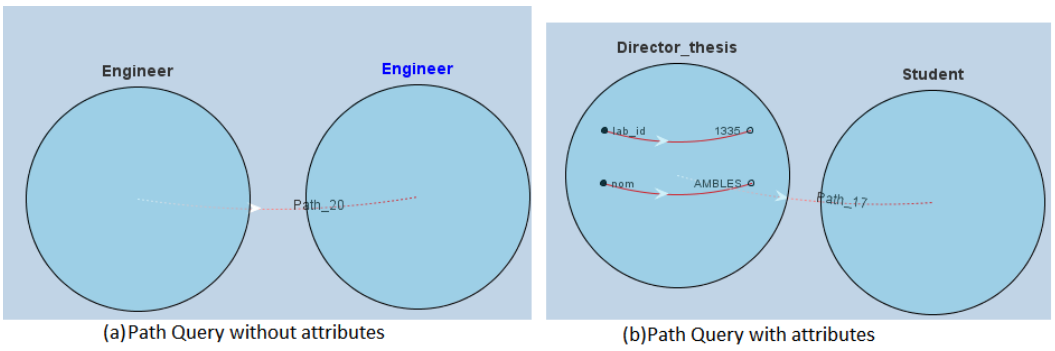


Figure 6.24: Example of the Used Path Queries.

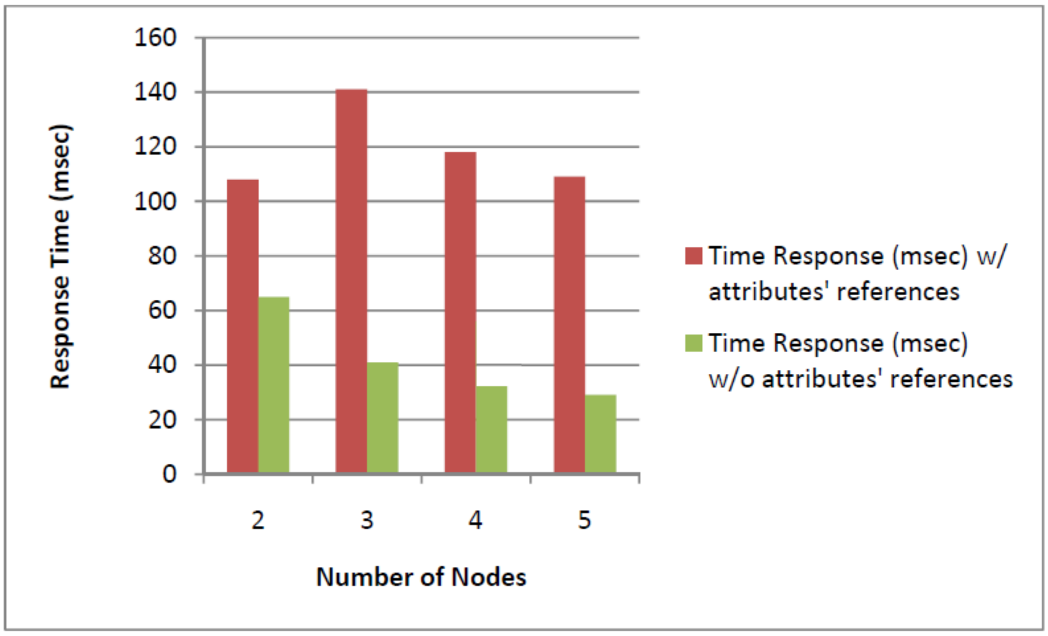


Figure 6.25: Performance of the Shortest Path Algorithm: Path Query Size Influence

The results shown in Figure 6.25 show that a query path without attributes is faster to find. In the case of the path without attributes, the algorithm does not have constraint to search then it will return all the paths that match with the query. In other cases, the algorithm should filter the result.

When queries have four or five nodes, the algorithm finds the results faster than otherwise. This is due to the fact that the numbers of nodes limits the number of possible paths to find.

6.5 Summary and Conclusion

In this chapter, we have presented the implementation of the different proposed approaches in this thesis. Then, we elaborated on various experimentations realized thanks to object interaction graph extraction approach and the GraphVQL language, previously introduced in Chapters 4 and 5.

We first detailed the three main sub system implemented to evaluate the work: the ontology learning, the object interaction graph extraction and the visual query language. Then, we applied the object graph extraction approach to build a social network from the ADEME database. As a first step, we have transformed the input relational database to a SPIDER-Graph. Despite that the resulting graph is stored in a GraphML, we have stored it in a relational database to evaluate it via a set of SQL queries. In the section 6.3.2.1, we have detailed the process to transform the obtained SPIDER-Graph to a social network. The resulting social network, which describes the relations between three objects: student, thesis directors, and engineers, is modeled also as SPIDER-Graph stored in a GraphML file. The relevance of the results has been evaluated using SQL queries.

In the section 6.4, we have presented the evaluation of GraphVQL using the ARSA project data. This evaluation is divided in two parts. Firstly, we have also evaluated the time responding of GraphVQL in the case of the simple graph (stored in RDF, GraphML or XML). This evaluation shows an accepted timing with medium graphs (2500 nodes). However, the time responding of the aggregation query can be improved for example by using a graph aggregation algorithm instead of the adaptation of the SPARQL operators. Secondly, we have evaluated GraphVQL time responding in the case of SPIDER-Graph (stored in a modified GraphML). The pattern matching algorithm used for the selection queries has been evaluated by varying the input graph size, the input query size and the technique used. The results have shown an improvement of the time response by using a pattern matching algorithm based on a pruning step and taking in consideration the complex-node attributes (see Figure 6.20). Indeed, a pruning step reduces the space of search and accelerates the matching process. For the path queries, we have evaluated the performance of the shortest path algorithm by varying the input query size.
For the moment, we use medium or small graphs (e.g. the ADEME SPIDER-Graph contains 12213 nodes). Then, we plan to process large and very large graph. We will optimize the process of pattern matching by improving the pruning technique and reducing as possible the search space. Another possible solution is by dividing the input graph and processing it using distributed solution like using the MAPRe-

duce [Lin 2010].

CHAPTER 7

Conclusion and future work

7.1 Summary of Contributions

The aim of this thesis, as indicated in the beginning, was to model enterprise data as graph model in order to facilitate the information search and analysis. The objective is also to provide a cartographi of data manipulated in an enterprise, and to bridge the gap between structured data and unstructured content. The main research questions were how to extract graphs from the different enterprise data (structured and unstructured) and facilitate the use of these graphs in the information search process. In order to answer these questions, we have proposed:

- An approach to extract graphs from relational database and a new graph data model to model these graphs.
- An approach to extract a specific enterprise ontology using the unstructured and semi-structured enterprise data.
- An approach to extract objects interaction graphs using the graphs extracted from the relational database and the knowledge learnt from the enterprise ontology.
- A visual query language allowing querying the different graphs types existing on the enterprise for non expert user.

Now, we will summarize the different contributions for each proposed approach.
1. Extracting graphs models from the different enterprise data source:
Graphs are a natural way to model data. Indeed, having all the data modeled as graphs allows the users querying and analyzing them using the same techniques and using the powerful graph manipulation and access techniques. As the enterprise data can be structured, semi-structured and unstructured, we have adopted a specific approach for each data type. However, the semi-structured data as XML and RDF can be modeled directly as a graph.
For the relational database, we have proposed a new approach based on two steps: (1) transform the relational schema to, a graph database schema and (2) Migrate the data to graph instance.
In order to preserve the relational data characteristics (dependency between tuples with foreign keys, etc) and improve the visualization and the comprehension of the resulting graph, we have defined the SPIDER-Graph data model. The SPIDER-Graph allows to model data having multiple attributes and multiple relations. The

attributes in this model can refer to other nodes which can model foreign keys.
In the case of unstructured data, the way to extract a graph model is not intuitive. For this we have choosen to extract the enterprise objects and their relations with the help of an enterprise ontology. This ontology plays the role of a guide to extract the important concepts and relations of the business domain in the enterprise data.

Then, we have proposed an enterprise ontology learning approach from the enterprise data. The input of this process is a generic enterprise ontology containing all the important concepts about the business domain. The advantage of this approach, which is based on adapted patterns for the business context, is that it provides new enterprise ontology for each company.
2. Extracting objects interaction graphs from multiples resources:
The graphs describing the interactions between heterogeneous objects, like social networks, facilitate complex data analysis. In this context, we have proposed a new approach that allows the extraction of the interaction graphs from the graph extracted from the relational database and enriched with the enterprise ontology. This approach based on two main steps:
First, the objects identification: the user selects the enterprise objects, he is interested by, represented by the ontology concepts then the process identify from the graph the corresponding objects.
Second, the relations extraction: this step is based on a set of relation patterns extracted from the relational database which detect explicit and create new relations between the chosen objects from the relational database. Then, other relations are added from the ontology relations. The use of the graph extracted from the relational database combined with the ontology extracted from the other enterprise data, makes this approach original. The proposed approach has been applied to extract social network.
3. The visual query language
We have proposed a visual query language (GraphVQL) which allows querying different graph models: SPIDER-Graph, RDF and GraphML. This language cover different query types from the simple selection query to social network analysis query and use:

- Graph pattern matching process to extract subgraphs from input graphs.
- Set of mapping rules to transform a graph pattern query to a SPARQL query.

GraphVQL can be seen as a query language related to the new graph data model SPIDER-Graph which uses a new extensions of the pattern matching algorithm proposed in [He 2008] and the shortest path algorithm proposed in [Newman 2001]. It is also a new visual query language for RDF which extends SPARQL to add new query types as the analysis or the shortest path query.

7.2 Future Work

In this section, we describe several future lines of research and present some preliminary ideas on how they can be tackled. Regarding the graphs extraction process, some issues can be addressed in order to improve the final results.

1. In the enterprise some existing social networks can be found on the social websites such as LinkedIn, Facebook or Tweeter among others. These social networks can be used to enrich the extracted interaction graphs. This enrichment can be performed by using merging graph techniques.

2. In this thesis, we have focused, on the graph extraction from one relational database. However, in the real enterprise context, we can find multiple databases created for different contexts and applications. Then, merging the different extracted graphs from the different databases can improve the quality of the resulting graphs.

3. The identified relation patterns from the relational database can be improved by using not only the schema but also the relational database tuple to learn new relations types. For example, these tuples , using the data mining techniques, allows to discover objects having the same characteristics.

4. In order to extract graph model from the unstructured and semi structured data, we have used an enterprise ontology with a predefined concept. However, in the case of building interaction graphs with known objects, we can use machine learning techniques to find these objects in the text and extract their relations. Some of these techniques have been applied in the social network extraction context.

5. The implementation of the proposed approach has revealed that we cannot do it on line. Then, in our future work we will try to adopt new techniques such as parallel computing algorithm to increase the time complexity of the used algorithms.

Regarding the visual query language some issues can be addressed in order to improve the final results.

1. In order to query the SPIDER-Graph graph model, the GraphVQL language uses a pattern matching algorithms to find select query and a shortest path query algorithm to find the links between two objects. In our future work, we

will improve the pattern matching algorithm to realize aggregation queries on the SPIDER-Graph data model. The enterprise ontology can be integrated in this algorithm to improve the matching result. Indeed, we can search for the existing objects on the graph semantically similar to the object on the query.

2. In addition to the visual query, we can add a keyword search module to find information on the different input graph: RDF, Graphml.

3. The queries drawn by the user are stored. Then, the history of theses queries can be used to recommend new queries for the user. Also, these queries can be used to detect communities and to analyze the users behavior.

Bibliography

[Abiteboul 1997] Serge Abiteboul, Dallan Quass, Jason Mchugh, Jennifer Widom and Janet Wiener. *The Lorel Query Language for Semistructured Data.* International Journal on Digital Libraries, vol. 1, pages 68–88, 1997. (Cited on page 23.)

[Agrawal 1994] Rakesh Agrawal and Ramakrishnan Srikant. *Fast Algorithms for Mining Association Rules in Large Databases.* In Jorge B. Bocca, Matthias Jarke and Carlo Zaniolo, editeurs, VLDB'94, Proceedings of 20th International Conference on Very Large Data Bases, September 12-15, 1994, Santiago de Chile, Chile, pages 487–499. Morgan Kaufmann, 1994. (Cited on page 51.)

[Agrawal 2002] Sanjay Agrawal. *DBXplorer: A System for Keyword-Based Search over Relational Databases.* In Proceedings of the 18th International Conference on Data Engineering, ICDE '02, pages 5–, Washington, DC, USA, 2002. IEEE Computer Society. (Cited on pages 2 and 38.)

[Aleman-Meza 2005] Boanerges Aleman-Meza, Christian Halaschek-Wiener, Satya Sanket Sahoo, Amit Sheth and I. Budak Arpinar. *Template based semantic similarity for security applications.* In Proceedings of the 2005 IEEE international conference on Intelligence and Security Informatics, ISI'05, pages 621–622, Berlin, Heidelberg, 2005. Springer-Verlag. (Cited on page 19.)

[Alhajj 2003] Reda Alhajj. *Extracting the extended entity-relationship model from a legacy relational database.* Inf. Syst., vol. 28, pages 597–618, September 2003. (Cited on pages 36 and 37.)

[Alkhateeb 2009] Faisal Alkhateeb, Jean-François Baget and Jérôme Euzenat. *Extending SPARQL with regular expression patterns (for querying RDF).* Web Semant., vol. 7, pages 57–73, April 2009. (Cited on page 28.)

[Amann 1992] Bernd Amann and Michel Scholl. *Gram: a graph data model and query languages.* In Proceedings of the ACM conference on Hypertext, ECHT '92, pages 201–211, New York, NY, USA, 1992. ACM. (Cited on page 26.)

[Amardeilh 2005] Florence Amardeilh, Philippe Laublet and Jean-Luc Minel. *Document annotation and ontology population from linguistic extractions.* In Proceedings of the 3rd international conference on Knowledge capture, K-CAP '05, pages 161–168, New York, NY, USA, 2005. ACM. (Cited on page 53.)

[Andries 1992] Marc Andries, Marc Gemis, Jan Paredaens, Inge Thyssens and Jan Van den Bussche. *Concepts for Graph-Oriented Object Manipulation.* In Proceedings of the 3rd International Conference on Extending Database Technology: Advances in Database Technology, EDBT '92, pages 21–38, London, UK, 1992. Springer-Verlag. (Cited on pages 10, 23, 42 and 71.)

[Angles 2008] Renzo Angles and Claudio Gutierrez. *Survey of graph database models.* ACM Comput. Surv., vol. 40, no. 1, pages 1–39, 2008. (Cited on page 10.)

[Baeza-Yates 2000] R. Baeza-Yates and G. Valiente. *An Image Similarity Measure Based on Graph Matching.* In Proceedings of the Seventh International Symposium on String Processing Information Retrieval (SPIRE'00), pages 28–, Washington, DC, USA, 2000. IEEE Computer Society. (Cited on page 19.)

[Balog 2007] K. Balog, K. Hofmann, W. Weerkamp and M. de Rijke. *The University of Amsterdam at the TREC 2007 Enterprise Track.* In TREC 2007 Working Notes, pages 248–251, November 2007. (Cited on page 2.)

[Banâtre 2001] Jean-Pierre Banâtre, Pascal Fradet and Daniel Le Métayer. *Gamma and the Chemical Reaction Model: Fifteen Years After.* In Proceedings of the Workshop on Multiset Processing: Multiset Processing, Mathematical, Computer Science, and Molecular Computing Points of View, WMP '00, pages 17–44, London, UK, 2001. Springer-Verlag. (Cited on page 31.)

[Bardohl 1999] Roswitha Bardohl, Mark Minas, Andy Schürr and Gabi Taentzer. *Application of Graph Transformation to Visual Languages.* In Hartmut Ehrig, Gregor Engels, Hans-Jörg Kreowski and Grzegorz Rozenberg, editeurs, Handbook of Graph Grammars and Computing by Graph Transformation, volume II: Applications, Languages and Tools, pages 105–180. World Scientific, 1999. (Cited on page 31.)

[Barendsen 1999] E. Barendsen and S. Smetsers. Handbook of graph grammars and computing by graph transformation: Vol. 2: Applications, languages, and tools, chapitre Graph rewriting aspects of functional programming, pages 63–102. World Scientific Publishing Co., Inc., River Edge, NJ, USA, 1999. (Cited on page 31.)

[Baresi 2002] Luciano Baresi and Reiko Heckel. *Tutorial Introduction to Graph Transformation: A Software Engineering Perspective.* pages 402–429. 2002. (Cited on page 32.)

[Beckett 2003] D. Beckett and J. Grant. *SWAD-Europe Deliverable 10.2: Mapping Semantic Web Data with RDBMSes.*, January 2003. (Cited on page 35.)

[Bengoetxea 2002] E. Bengoetxea. *Inexact Graph Matching Using Estimation of Distribution Algorithms.* PhD thesis, Ecole Nationale Supérieure des Télécommunications, Paris, France, Dec 2002. (Cited on pages 15 and 17.)

[Berge 1985] C. Berge. Graphs and hypergraphs. Elsevier Science Ltd, 1985. (Cited on page 9.)

[Berners-Lee 1998] Â T. Berners-Lee. *Relational Databases on the Semantic Web.*, 1998. (Cited on page 35.)

[Berners-lee 2001] Hendler J. Berners-lee T. and O. Lassila. *The semantic web.* Scientific American, 2001. (Cited on page 46.)

[Billig 2008] Andreas Billig and Kurt Sandkuhl. *Enterprise Ontology based Artefact Management.* In GI Jahrestagung (2), pages 681–687, 2008. (Cited on page 54.)

[Bisson 2000] G. Bisson, C. Nédellec and D. Ca namero. *Designing clustering methods for ontology building: The MoâK workbench.* pages 13–19, 2000. (Cited on page 52.)

[Bizer 2007] Christian Bizer and Richard Cyganiak. *D2RQ - Lessons Learned.* W3C Workshop on RDF Access to Relational Databases, October 2007. (Cited on pages 35 and 36.)

[Blakeley 2007] Â C Blakeley. *RDFÂ ViewsÂ ofÂ SQLÂ DataÂ (DeclarativeÂ SQLÂ SchemaÂ toÂ RDFÂ Mapping).* OpenLinkÂ Software, 2007. (Cited on page 35.)

[Blau 2002] H. Blau, N. Immerman and D. Jensen. *A visual language for querying and updating graphs.* Technical Report 2002-037, University of Massachusetts Amherst Computer Science, 2002. (Cited on pages 20 and 22.)

[Blomqvist 2006] E. Blomqvist and A. Ohgren. *Constructing an Enterprise Ontology for an Automotive Supplier.* In Proceedings of the 12th IFAC Symposium on Information Control Problems in Manufacturing, Saint- Etienne, France, 2006. (Cited on pages 55 and 68.)

[Blomqvist 2007] Eva Blomqvist. *OntoCase: a pattern-based ontology construction approach.* In Proceedings of the 2007 OTM Confederated international conference on On the move to meaningful internet systems: CoopIS, DOA, ODBASE, GADA, and IS - Volume Part I, OTM'07, pages 971–988, Berlin, Heidelberg, 2007. Springer-Verlag. (Cited on page 54.)

[Bollegala 2007] Danushka Bollegala, Yutaka Matsuo and Mitsuru Ishizuka. *Measuring semantic similarity between words using web search engines.* In WWW, pages 757–766, 2007. (Cited on page 65.)

[Bonifati 2000] Angela Bonifati and Stefano Ceri. *Comparative analysis of five XML query languages.* SIGMOD Rec., vol. 29, no. 1, pages 68–79, March 2000. (Cited on page 29.)

[Borgwardt 2005] Karsten M. Borgwardt, Cheng Soon Ong, Stefan SchÃ¶nauer, S. V. N. Vishwanathan, Alex J. Smola and Hans-Peter Kriegel. *Protein function prediction via graph kernels*, 2005. (Cited on page 15.)

[Borkar 2004] V. Borkar and M. Carey. *Extending xquery for grouping, duplicate elimination, and outer joins.* In XML Conference and Expo, 2004. (Cited on page 30.)

[Brandes 2001] Ulrik Brandes, Markus Eiglsperger, Ivan Herman, Michael Himsolt and M. Scott Marshall. *GraphML Progress Report.* In Graph Drawing'01, pages 501–512, 2001. (Cited on page 12.)

[Bray 1997] Tim Bray, Jean Paoli and C. M. Sperberg-McQueen. *Extensible Markup Language (XML).* World Wide Web Journal, vol. 2, no. 4, pages 27–66, 1997. (Cited on page 12.)

[Broekstra 2003] J. Broekstra. *SeRQL: Sesame RDF Query Language.* In M. Ehrig, editeur, SWAP Deliverable 3.2 Method Design, pages 55–68. 2003. `http://swap.semanticweb.org/public/Publications/swap-d3.2.pdf`. (Cited on page 28.)

[Buchanan 2006] Leigh Buchanan and Andrew OÊ$\frac{1}{4}$Connell. *A brief history of decision making.* Harvard Business Review, vol. 84, no. 1, pages 32–41, 132, 2006. (Cited on page 1.)

[Buitelaar 2004] Paul Buitelaar, Daniel Olejnik and Michael Sintek. *A Protege Plug-in for Ontology Extraction from Text Based on Linguistic Analysis.* In In Proceedings of the 1st European Semantic Web Symposium (ESWS, 2004. (Cited on page 52.)

[Buitelaar 2005] Paul Buitelaar and Bernardo Magnini. *Ontology Learning from Text: An Overview.* In In Paul Buitelaar, P., Cimiano, P., Magnini B. (Eds.), Ontology Learning from Text: Methods, Applications and Evaluation, pages 3–12. IOS Press, 2005. (Cited on page 50.)

[Carey 2000] Michael J. Carey, Jerry Kiernan, Jayavel Shanmugasundaram, Eugene J. Shekita and Subbu N. Subramanian. *XPERANTO: Middleware for Publishing Object-Relational Data as XML Documents.* In Proceedings of the 26th International Conference on Very Large Data Bases, VLDB '00, pages 646–648, San Francisco, CA, USA, 2000. Morgan Kaufmann Publishers Inc. (Cited on page 36.)

[Casanova 1983] M.A. Casanova and J.E.A. de Sa. *Designing Entity-Relationship schemas for conventional information systems.* In 3rd Int. Conf. on the Entity-Relationship Approach, pages 265–278, 1983. (Cited on page 35.)

[Casanova 1984] Marco A. Casanova, Ronald Fagin and Christos H. Papadimitriou. *Inclusion dependencies and their interaction with functional dependencies.* Journal of Computer and System Sciences, vol. 28, no. 1, pages 29 – 59, 1984. (Cited on page 34.)

[Ceri 1999] Stefano Ceri, Sara Comai, Ernesto Damiani, Piero Fraternali, Stefano Paraboschi and Letizia Tanca. *XML-GL: a graphical language for querying and restructuring XML documents.* Computer Networks, vol. 31, no. 11Ãâ¬16, $pages1171 - -1187, 1999.(Citedonpage$ 29.)

[Chartrand 1985] G. Chartrand. Introductory graph theory, chapitre Directed Graphs as Mathematical Models, pages 16–19. Courier Dover Publications, 1985. (Cited on page 8.)

[Chen 1976] Peter Pin-Shan Chen. *The entity-relationship modeltoward a unified view of data.* ACM Trans. Database Syst., vol. 1, pages 9–36, March 1976. (Cited on page 34.)

[Chen 2005] Li Chen, Amarnath Gupta and M. Erdem Kurul. *A Semantic-aware RDF Query Algebra.* In In COMAD 2005, 2005. (Cited on page 30.)

[Cheng 2000] Josephine Cheng and Jane Xu. *XML and DB2.* Data Engineering, International Conference on, vol. 0, page 569, 2000. (Cited on page 36.)

[Chiang 1994] Roger H. L. Chiang, Terence M. Barron and Veda C. Storey. *Reverse engineering of relational databases: extraction of an EER model from a relational database.* Data Knowl. Eng., vol. 12, pages 107–142, March 1994. (Cited on page 35.)

[Chiang 1995] Roger H. L. Chiang. *A knowledge-based system for performing reverse engineering of relational databases.* Decis. Support Syst., vol. 13, pages 295–312, March 1995. (Cited on page 35.)

[Chong 2005] Eugene Inseok Chong, Souripriya Das, George Eadon and Jagannathan Srinivasan. *An Efficient SQL-based RDF Querying Scheme.* In VLDB, pages 1216–1227, 2005. (Cited on page 29.)

[Cimiano 2005a] P. Cimiano and J. VÃ¶lker. *Text2Onto A Framework for Ontology Learning and Data-driven Change Discovery.* In Proc. of the Int. Conf. on Applications of Natural Language to Information Systems (NLDB'2005), pages 227–238, 2005. (Cited on page 53.)

[Cimiano 2005b] Philipp Cimiano, Andreas Hotho and Steffen Staab. *Learning concept hierarchies from text corpora using formal concept analysis.* J. Artif. Int. Res., vol. 24, pages 305–339, August 2005. (Cited on page 52.)

[Cimiano 2006] Philipp Cimiano. Ontology learning and population from text: Algorithms, evaluation and applications. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2006. (Cited on pages 50, 53 and 64.)

[Codd 1980] E. F. Codd. *Data Models in Database Management.* In Michael L. Brodie and Stephen N. Zilles, editeurs, Proceedings of the Workshop on Data Abstraction, Databases and Conceptual Modelling, Pingree Park, Colorado, June 23-26, 1980, volume 11, pages 112–114. ACM Press, 1980. (Cited on page 19.)

[Coffman 2004] Thayne Coffman, Seth Greenblatt and Sherry Marcus. *Graph-based technologies for intelligence analysis.* Commun. ACM, vol. 47, no. 3, pages 45–47, March 2004. (Cited on page 19.)

[Cohen 2003] William W. Cohen, Pradeep D. Ravikumar and Stephen E. Fienberg. *A Comparison of String Distance Metrics for Name-Matching Tasks.* In IIWeb, pages 73–78, 2003. (Cited on pages 85 and 95.)

[Conrad 2000] Rainer Conrad, Dieter Scheffner and J. Christoph Freytag. *XML Conceptual Modeling using UML.* In In Proceedings of the Ninteenth International Conference on Conceptual Modeling (ER2000, pages 558–571, 2000. (Cited on page 37.)

[Consens 1989] M. P. Consens and A. O. Mendelzon. *Expressing structural hypertext queries in graphlog.* In Proceedings of the second annual ACM conference on Hypertext, HYPERTEXT '89, pages 269–292, New York, NY, USA, 1989. ACM. (Cited on pages 21 and 30.)

[Consortium 2010a] W3C Consortium. *XML Path Language (XPath) 2.0*, 2010. (Cited on page 29.)

[Consortium 2010b] W3C Consortium. *XQuery 1.0: An XML Query Language*, 2010. (Cited on page 29.)

[Conte 2004] Donatello Conte, Pasquale Foggia, Carlo Sansone and Mario Vento. *Thirty Years Of Graph Matching In Pattern Recognition.* IJPRAI, vol. 18, no. 3, pages 265–298, 2004. (Cited on page 15.)

[Cook 2006] Diane J. Cook and Lawrence B. Holder. Mining graph data. John Wiley & Sons, 2006. (Cited on page 15.)

[Coulet 2008] Adrien Coulet, Malika Smaïl-Tabbone, Amedeo Napoli and Marie-Dominique Devignes. *Role Assertion Analysis: a proposed method for ontology refinement through assertion learning.* In STAIRS, pages 47–58, 2008. (Cited on page 52.)

[Craven 2000] DiPasquo D. Freitag-D. McCallum A. Mitchell T. Nigam K. Slattery S. Craven M. *Learning to construct knowledge bases from the World Wide Web.* Artificial Intelligence, pages 69–113, 2000. (Cited on page 54.)

[Cruz 1987] Isabel F. Cruz, Alberto O. Mendelzon and Peter T. Wood. *A graphical query language supporting recursion.* In Proceedings of the 1987 ACM SIGMOD international conference on Management of data, SIGMOD '87, pages 323–330, New York, NY, USA, 1987. ACM. (Cited on page 20.)

[Cruz 1988] Isabel F. Cruz, Alberto O. Mendelzon and Peter T. Wood. *G+: Recursive Queries Without Recursion.* In Expert Database Conf., pages 645–666, 1988. (Cited on pages 20 and 30.)

[Cunningham 2002] Hamish Cunningham. *GATE, a General Architecture for Text Engineering.* Computers and the Humanities, vol. 36, no. 2, pages 223–254, 2002. (Cited on page 138.)

[Dar 1998] Shaul Dar, Gadi Entin, Shai Geva and Eran Palmon. *DTL's DataSpot: Database Exploration Using Plain Language.* In Proceedings of the 24rd International Conference on Very Large Data Bases, VLDB '98, pages 645–649, San Francisco, CA, USA, 1998. Morgan Kaufmann Publishers Inc. (Cited on page 38.)

[Depiero 1996] Fred Depiero, Mohan Trivedi and Steven Serbin. *Graph matching using a direct classification of node attendance.* Pattern Recognition, vol. 29, no. 6, pages 1031 – 1048, 1996. (Cited on page 19.)

[Deutsch 1998] Alin Deutsch, Mary F. Fernandez, Daniela Florescu, Alon Y. Levy and Dan Suciu. *XML-QL: A Query Language for XML*, 1998. (Cited on page 29.)

[Dijkstra 1959] E.W. Dijkstra. *A Note on Two Problems in Connexion with Graphs.* Numerische Mathematik, vol. 1, pages 269–271, 1959. (Cited on page 87.)

[Din 2007] Finding top-k Min-Cost connected trees in databases, April 2007. (Cited on page 37.)

[Dobbie 2001] Gillian Dobbie, Wu Xiaoying, Tok Wang Ling and Mong Li Lee. *ORA-SS: An Object-Relationship-Attribute Model for Semi-Structured Data.* Rapport technique, 2001. (Cited on page 37.)

[Dragut 2004] Eduard C. Dragut and Ramon Lawrence. *Composing Mappings Between Schemas Using a Reference Ontology.* In CoopIS/DOA/ODBASE (1), pages 783–800, 2004. (Cited on page 35.)

[Du 2001] Wenyue Du, Mong Li, Lee Tok and Wang Ling. *XML Structures for Relational Data.* In Proceedings of the Second International Conference on Web Information Systems Engineering (WISE'01) Volume 1 - Volume 1, WISE '01, pages 151–, Washington, DC, USA, 2001. IEEE Computer Society. (Cited on page 37.)

[Duerst 2005] M. Duerst and M. Suignard. *Internationalized Resource Identifiers (IRIs*, 2005. (Cited on pages 11 and 116.)

[Ehrig 2004] Hartmut Ehrig, Ulrike Prange and Gabriele Taentzer. *Fundamental Theory for Typed Attributed Graph Transformation.* In ICGT, pages 161–177, 2004. (Cited on page 33.)

[Ehrig 2006a] H. Ehrig, K. Ehrig, U. Prange and G. Taentzer. Fundamentals of algebraic graph transformation (monographs in theoretical computer science. an eatcs series). Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2006. (Cited on pages 8, 9 and 31.)

[Ehrig 2006b] Hartmut Ehrig, Karsten Ehrig, Ulrike Prange and Gabriele Taentzer. *Fundamental Theory for Typed Attributed Graphs and Graph Transformation based on Adhesive HLR Categories.* Fundam. Inf., vol. 74, no. 1, pages 31–61, October 2006. (Cited on page 33.)

[Ermel 1999] C. Ermel, M. Rudolf and G. Taentzer. The agg approach: language and environment, pages 551–603. World Scientific Publishing Co., Inc., River Edge, NJ, USA, 1999. (Cited on page 31.)

[Etzioni 2004] Oren Etzioni, Michael Cafarella, Doug Downey, Stanley Kok, Ana-Maria Popescu, Tal Shaked, Stephen Soderland, Daniel S. Weld and Alexander Yates. *Web-Scale Information Extraction in KnowItAll*, 2004. (Cited on page 53.)

[Fahrner 1995] Christian Fahrner and Gottfried Vossen. *A survey of database design transformations based on the Entity-Relationship model.* Data Knowl. Eng., vol. 15, pages 213–250, June 1995. (Cited on page 34.)

[Faure 2000] D. Faure and T. Poibeau. *First experiments of using semantic knowledge learned by ASIUM for information extraction task using INTEX.* In C. Nedellec P. S. Staab A. Maedche, editeur, Proceedings of the Workshop on Ontology Learning, 14th European Conference on Artificial Intelligence ECAIâ00, Berlin, Germany., 2000. 7-12. (Cited on page 52.)

[Fischler 1973] M. A. Fischler and R. A. Elschlager. *The Representation and Matching of Pictorial Structures.* IEEE Trans. Comput., vol. 22, pages 67–92, January 1973. (Cited on page 18.)

[Fleischman 2001] Michael Fleischman. *Automated Subcategorization of Named Entities.* In Meeting of the Association for Computational Linguistics, pages 25–30, 2001. (Cited on page 54.)

[Fleischman 2004] Michael Ben Fleischman and Eduard Hovy. *Multi-document person name resolution.* In In Proceedings of ACL-42, Reference Resolution Workshop, 2004. (Cited on page 54.)

[Flesca 2000] Sergio Flesca and Sergio Greco. *Querying Graph Databases.* In Carlo Zaniolo, Peter C. Lockemann, Marc H. Scholl and Torsten Grust, editeurs, EDBT, volume 1777 of *Lecture Notes in Computer Science*, pages 510–524. Springer, 2000. (Cited on page 23.)

[Fomin 2005] Fedor Fomin, Pinar Heggernes and Dieter Kratsch. *Exact Algorithms for Graph Homomorphisms.* In Maciej Liskiewicz and RÃÂ$\frac{1}{4}$diger Reischuk, editeurs, Fundamentals of Computation Theory, volume 3623 of *Lecture Notes in Computer Science*, pages 161–171. Springer Berlin / Heidelberg, 2005. 10.1007/11537311$_1$5.(*Citedonpage* 16.)

[Fong 2005] Joseph Fong and San Kuen Cheung. *Translating relational schema into XML schema definition with data semantic preservation and XSD graph.* Inf. Softw. Technol., vol. 47, pages 437–462, May 2005. (Cited on page 36.)

[Fong 2006] Fong A. Wong H. Fong J. and P. Yu. *Translating Relational Schema with Constraints into XML Schema.* Journal of Software Engineering and Knowledge Engineering„ vol. 16, no. 2, page 201â244, 2006. (Cited on page 37.)

[Fonkam 1992] M. Fonkam and W. Gray. *An approach to eliciting the semantics of relational databases.* In Pericles Loucopoulos, editeur, Advanced Information Systems Engineering, volume 593 of *Lecture Notes in Computer Science*, pages 463–480. Springer Berlin / Heidelberg, 1992. 10.1007/BFb0035148. (Cited on page 35.)

[Fox 1995] M. S. Fox, M. Barbuceanu and M. Gruninger. *An organisation ontology for enterprise modelling: preliminary concepts for linking structure and behaviour.* In Proceedings of the 4th Workshop on Enabling Technologies: Infrastructure for Collaborative Enterprises (WET-ICE'95), WET-ICE '95, pages 71–, Washington, DC, USA, 1995. IEEE Computer Society. (Cited on page 54.)

[Freeman 1977] L. C. Freeman. *A set of measures of centrality based on betweenness.* Sociometry, vol. 40, no. 1, pages 35–41, 1977. (Cited on pages 2 and 124.)

[Gemis 1993] Marc Gemis and Jan Paredaens. *An Object-Oriented Pattern Matching Language.* In JSSST, International Symposium on Object Technologies for Advanced Software, pages 339–355. Springer-Verlag, 1993. (Cited on pages 10 and 23.)

[Glauert 1997] John R. W. Glauert, Richard Kennaway, George A. Papadopoulos and M. Ronan Sleep. *Dactl: an experimental graph rewriting language.* J. Prog. Lang., vol. 5, no. 1, pages 85–108, 1997. (Cited on page 31.)

[Golfarelli 2004] Matteo Golfarelli, Stefano Rizzi and Iuris Cella. *Beyond data warehousing: what's next in business intelligence?* In DOLAP, pages 1–6, 2004. (Cited on page 1.)

[Gomez-Perez 1996] Fernandez-Lopez M. de Vicente A. Gomez-Perez A. *Towards a Method to Conceptualize Domain Ontologies,.* In ECAI96 Workshop on Ontological Engineering, 1996. (Cited on page 49.)

[Gomez-Perez 2003] Manzano-Macho-D. Gomez-Perez. *A survey of ontology learning methods and techniques.* Deliverable 1.5. ontoweb., 2003. (Cited on page 51.)

[Gomez-Perez 2004] Asuncion Gomez-Perez and David Manzano-Macho. *An overview of methods and tools for ontology learning from texts.* Knowl. Eng. Rev., vol. 19, pages 187–212, September 2004. (Cited on page 47.)

[Graves 1995] Bergeman-E. R. Lawrence C. B Graves M. *Graph database systems for genomics.* IEEE Eng.Medicine Biol, vol. 14, pages 737–745, 1995. (Cited on pages 11 and 71.)

[Green 2008] Jennifer Green, Glen Hart, Catherine Dolbear, Paula C. Engelbrecht and John Goodwin. *Creating a Semantic Integration System using Spatial Data.* In Christian Bizer and Anupam Joshi, editeurs, International Semantic Web Conference (Posters & Demos), volume 401 of *CEUR Workshop Proceedings*. CEUR-WS.org, 2008. (Cited on page 36.)

[Gruber 1993] T.R Gruber. *Toward principles for the design of ontologies used for knowledge sharing.* In Poli-R. Guarino N., editeur, International Workshop on Formal Ontology in Conceptual Analysis and Knowledge Representation, pages 907–928, 1993. (Cited on page 46.)

[Gruninger 1995] Fox-M.S. Gruninger M. *Methodology for the design and evaluation of ontologies,.* In Workshop on Basic Ontological Issues in Knowledge Sharing, Montreal, 1995. (Cited on page 56.)

[Guarino 1998] N. Guarino. *Formal Ontology in Information Systems.* In Guarino N., editeur, International Conference on Formal Ontology in Information Systems (FOISâ98), pages 3–15, Trento, Italy, 1998. IOS Press. (Cited on pages 46 and 47.)

[Güting 1994] Ralf Hartmut Güting. *GraphDB: Modeling and Querying Graphs in Databases.* In Proceedings of the 20th International Conference on Very Large Data Bases, VLDB '94, pages 297–308, San Francisco, CA, USA, 1994. Morgan Kaufmann Publishers Inc. (Cited on page 24.)

[Gyssens 1990a] Marc Gyssens, Jan Paredaens and Dirk Van Gucht. *A graph-oriented object model for database end-user interfaces.* SIGMOD Rec., vol. 19, pages 24–33, May 1990. (Cited on pages 10 and 42.)

[Gyssens 1990b] Marc Gyssens, Jan Paredaens and Dirk Van Gucht. *A graph-oriented object model for database end-user interfaces.* In Proceedings of the 1990 ACM SIGMOD international conference on Management of data, SIGMOD '90, pages 24–33, New York, NY, USA, 1990. ACM. (Cited on page 22.)

[Gyssens 1990c] Marc Gyssens, Jan Paredaens and Dirk Van Gucht. *A graph-oriented object model for database end-user interfaces.* In Proceedings of the 1990 ACM SIGMOD international conference on Management of data, SIGMOD '90, pages 24–33, New York, NY, USA, 1990. ACM. (Cited on page 71.)

[Hahn 2000] Udo Hahn and Stefan Schulz. *Towards Very Large Terminological Knowledge Bases: A Case Study from Medicine.* In Proceedings of the 13th Biennial Conference of the Canadian Society on Computational Studies of Intelligence: Advances in Artificial Intelligence, AI '00, pages 176–186, London, UK, 2000. Springer-Verlag. (Cited on page 52.)

[hai Do 2002] Hong hai Do and Erhard Rahm. *COMA - A system for flexible combination of Schema Matching Approaches.* In In VLDB, pages 610–621, 2002. (Cited on page 35.)

[Harchaoui 2007] Z. Harchaoui and F. Bach. *Image classification with segmentation graph kernels.* In Proceedings of the Conference on Computer Vision and Pattern Recognition, 2007. (Cited on page 15.)

[Hart 1968] Peter Hart, Nils Nilsson and Bertram Raphael. *A Formal Basis for the Heuristic Determination of Minimum Cost Paths.* IEEE Transactions on Systems Science and Cybernetics, vol. 4, no. 2, pages 100–107, February 1968. (Cited on page 17.)

[Hart 1972] Peter E. Hart, Nils J. Nilsson and Bertram Raphael. *A Formal Basis for the Heuristic Determination of Minimum Cost Paths.* SIGART Bull., pages 28–29, December 1972. (Cited on page 122.)

[Hart 2007] Glen Hart, Catherine Dolbear and John Goodwin. *Lege Feliciter: Using Structured English to represent a Topographic Hydrology Ontology.* In Christine Golbreich, Aditya Kalyanpur and Bijan Parsia, editeurs, OWLED, volume 258 of *CEUR Workshop Proceedings.* CEUR-WS.org, 2007. (Cited on page 36.)

[He 2007] Hao He, Haixun Wang, Jun Yang and Philip S. Yu. *BLINKS: ranked keyword searches on graphs.* In Proceedings of the 2007 ACM SIGMOD international conference on Management of data, SIGMOD '07, pages 305–316, New York, NY, USA, 2007. ACM. (Cited on pages 37 and 38.)

[He 2008] Huahai He and Ambuj K. Singh. *Graphs-at-a-time: query language and access methods for graph databases.* In Proceedings of the 2008 ACM SIGMOD international conference on Management of data, SIGMOD '08, pages 405–418, New York, NY, USA, 2008. ACM. (Cited on pages 16, 23, 25, 126, 127 and 170.)

[Hearst 1992] M.A. Hearst. *Automatic acquisition of hyponyms from large text corpora.* In Proceedings of the 14th International Conference on Computational Linguistics (COLING-92)., page 539â545, Nantes, France, 1992. (Cited on pages 51 and 53.)

[Heckel 2002] Reiko Heckel, Jochen Malte Küster and Gabriele Taentzer. *Confluence of Typed Attributed Graph Transformation Systems.* In Proceedings of the First International Conference on Graph Transformation, ICGT '02, pages 161–176, London, UK, UK, 2002. Springer-Verlag. (Cited on page 33.)

[Hert 2011] Matthias Hert, Gerald Reif and Harald C. Gall. *A Comparison of RDB-to-RDF Mapping Languages.* In Proceedings of the 7th International Conference on Semantic Systems (I-Semantics), 2011. (Cited on page 35.)

[Hidders 2002] Jan Hidders. *Typing Graph-Manipulation Operations.* In Proceedings of the 9th International Conference on Database Theory, ICDT '03, pages 394–409, London, UK, 2002. Springer-Verlag. (Cited on pages 10 and 23.)

[Himsolt 1996] Michael Himsolt. *The Graphlet System.* In Graph Drawing, pages 233–240, 1996. (Cited on page 13.)

[Holt 2000] Richard C. Holt, Andreas Winter and Andy SchÃ¼rr. *GXL: Toward a Standard Exchange Format.* In 9/99 HASSAN DIAB, ULRICH FURBACH, HASSAN TABBARA. ON THE USE OF FUZZY TECHNIQUES IN CACHE MEMORY MANAGAMENT. 8/99 JENS WOCH, FRIEDBERT WIDMANN. IMPLEMENTATION OF A SCHEMA-TAG-PARSER, pages 162–171, 2000. (Cited on page 14.)

[Hristidis 2002] Vagelis Hristidis and Yannis Papakonstantinou. *Discover: keyword search in relational databases.* In Proceedings of the 28th international conference on Very Large Data Bases, VLDB '02, pages 670–681. VLDB Endowment, 2002. (Cited on page 38.)

[Hristidis 2003] Vagelis Hristidis, Luis Gravano and Yannis Papakonstantinou. *Efficient IR-style keyword search over relational databases.* In Proceedings of the 29th international conference on Very large data bases - Volume 29, VLDB '2003, pages 850–861. VLDB Endowment, 2003. (Cited on page 39.)

[Hu 2007] Wei Hu and Yuzhong Qu. *Discovering Simple Mappings Between Relational Database Schemas and Ontologies.* In ISWC/ASWC, pages 225–238, 2007. (Cited on page 35.)

[Huet 1999] Benoit Huet and Edwin R. Hancock. *Shape recognition from large image libraries by inexact graph matching.* Pattern Recogn. Lett., vol. 20, pages 1259–1269, November 1999. (Cited on page 18.)

[Inmon 1992] W. H. Inmon. Building the data warehouse. QED Information Sciences, Inc., Wellesley, MA, USA, 1992. (Cited on page 1.)

[Irniger 2007] Christophe Irniger and Horst Bunke. *Decision trees for filtering large databases of graphs.* Int. J. Intell. Syst. Technol. Appl., vol. 3, pages 166–187, June 2007. (Cited on page 16.)

[ISO9075:1999 1999] ISO9075:1999. *Information Technology-Database Language SQL.* Standard No. ISO/IEC 9075:1999, International Organization for Standardization (ISO), 1999. (Available from American National Standards Institute, New York, NY 10036, (212) 642-4900.). (Cited on page 41.)

[Ji 1991] Wenguang Ji, C. Robert Carlson and David Dreyer. *An Algorithm Converting Relational Schemas to Nested Entity-Relationship Schemas.* In Toby J. Teorey, editeur, Proceedings of the 10th International Conference on Entity-Relationship Approach (ER'91), 23-25 October, 1991, San Mateo, California, USA, pages 231–246. ER Institute, 1991. (Cited on page 35.)

[Jiang 1997] J.J. Jiang and D.W. Conrath. *Semantic similarity based on corpus statistics and lexical taxonomy.* In Proc. of the Int'l. Conf. on Research in Computational Linguistics, pages 19–33, 1997. (Cited on page 85.)

[Jones 1998] Dean Jones, Trevor Bench-capon and Pepijn Visser. *Methodologies For Ontology Development.* In Proceedings of IT&KNOWS Conference of the 15 th IFIP World Computer Congress, pages 62–75, 1998. (Cited on page 49.)

[Kacholia 2005] Varun Kacholia, Shashank Pandit, Soumen Chakrabarti, S. Sudarshan, Rushi Desai and Hrishikesh Karambelkar. *Bidirectional expansion for keyword search on graph databases.* In Proceedings of the 31st international conference on Very large data bases, VLDB '05, pages 505–516. VLDB Endowment, 2005. (Cited on page 37.)

[Kaplan 2006] I. Kaplan. *A Semantic Graph Query Language.* Ucrl-tr-255447, Lawrence Livermore National Laboratory, 17 October 2006. (Cited on page 27.)

[Karvounarakis 2002] Gregory Karvounarakis, Sofia Alexaki, Vassilis Christophides, Dimitris Plexousakis and Michel Scholl. *RQL: a declarative query language for RDF.* In Proceedings of the 11th international conference on World Wide Web, WWW '02, pages 592–603, New York, NY, USA, 2002. ACM. (Cited on page 28.)

[Kietz 2000] Maedche-A. Kietz J.U. and R.: Volz. *A Method for Semi-Automatic Ontology Acquisition from a Corporate Intranet.* In EKAWâ00 Workshop on Ontologies and Texts., pages 37–50, 2000. (Cited on page 52.)

[Klyne 2004] Graham Klyne and Jeremy J. Carroll. *Resource Description Framework (RDF): Concepts and Abstract Syntax.* World Wide Web Consortium, Recommendation REC-rdf-concepts-20040210, February 2004. (Cited on page 11.)

[Knublauch 2004] Holger Knublauch, Ray W. Fergerson, Natalya F. Noy and Mark A. Musen. *The ProtÃ©gÃ© OWL plugin: An open development environment for semantic web applications.* pages 229–243. Springer, 2004. (Cited on page 138.)

[Kowalski 2010] Gerald Kowalski. Information retrieval architecture and algorithms. Springer-Verlag New York, Inc., New York, NY, USA, 1st édition, 2010. (Cited on page 2.)

[Kuper 1993] Gabriel M. Kuper and Moshe Y. Vardi. *The logical data model.* ACM Trans. Database Syst., vol. 18, pages 379–413, September 1993. (Cited on pages 10 and 25.)

[Laforest 2003] F. Laforest and M. Boumédiene. *Study of the Automatic Construction of XML Documents Models from a Relational Data Model.* In Proceedings of the 14th International Workshop on Database and Expert Systems Applications, DEXA '03, pages 566–, Washington, DC, USA, 2003. IEEE Computer Society. (Cited on page 37.)

[Larrosa 2002] Javier Larrosa and Gabriel Valiente. *Constraint satisfaction algorithms for graph pattern matching.* Mathematical. Structures in Comp. Sci., vol. 12, pages 403–422, August 2002. (Cited on page 16.)

[Lazarescu 2000] Mihai Lazarescu, Horst Bunke and Svetha Venkatesh. *Graph Matching: Fast Candidate Elimination Using Machine Learning Techniques.* In SSPR/SPR, pages 236–245, 2000. (Cited on page 16.)

[Lenat 1989] Douglas B. Lenat and R. V. Guha. Building large knowledge-based systems; representation and inference in the cyc project. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1st édition, 1989. (Cited on page 49.)

[Levene 1991] M. Levene and A. Poulovanssilis. *An object-oriented data model formalised through hypergraphs.* Data Knowl. Eng., vol. 6, pages 205–224, May 1991. (Cited on pages 11 and 21.)

[Levene 1995] Mark Levene and George Loizou. *A Graph-Based Data Model and its Ramifications.* IEEE Trans. on Knowl. and Data Eng., vol. 7, no. 5, pages 809–823, 1995. (Cited on pages 9, 11, 27 and 71.)

[Li 2008] 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 Proceedings of the 2008 ACM SIGMOD international conference on Management of data, SIGMOD '08, pages 903–914, New York, NY, USA, 2008. ACM. (Cited on page 37.)

[Lin 2010] Jimmy Lin and Michael Schatz. *Design patterns for efficient graph algorithms in MapReduce.* In Proceedings of the Eighth Workshop on Mining and Learning with Graphs, MLG '10, pages 78–85, New York, NY, USA, 2010. ACM. (Cited on page 168.)

[Liu 2004] Shaorong Liu, Qinghua Zou and Wesley W. Chu. *Configurable indexing and ranking for XML information retrieval.* In Proceedings of the 27th annual international ACM SIGIR conference on Research and development in information retrieval, SIGIR '04, pages 88–95, New York, NY, USA, 2004. ACM. (Cited on page 30.)

[Lopez 1999] M. FernandÃ©z Lopez. *Overview of Methodologies for Building Ontologies.* In Proceedings of the IJCAI-99 Workshop on Ontologies and Problem Solving Methods (KRR5), Stockholm, Sweden,, August 2 1999. (Cited on page 49.)

[Luxburg 2007] Ulrike Luxburg. *A tutorial on spectral clustering.* Statistics and Computing, vol. 17, pages 395–416, December 2007. (Cited on page 133.)

[Maatuk 2008] M. Abdelsalam Maatuk, M. Akhtar Ali and B. Nick Rossiter. *Relational Database Migration: A Perspective.* In DEXA, pages 676–683, 2008. (Cited on pages 74 and 150.)

[MacQueen 1967] J. B. MacQueen. *Some Methods for Classification and Analysis of MultiVariate Observations.* In L. M. Le Cam and J. Neyman, editeurs, Proc. of the fifth Berkeley Symposium on Mathematical Statistics and Probability, volume 1, pages 281–297. University of California Press, 1967. (Cited on page 133.)

[Madhavan 2001] Jayant Madhavan, Philip A. Bernstein and Erhard Rahm. *Generic Schema Matching with Cupid.* In Proceedings of the 27th International Conference on Very Large Data Bases, VLDB '01, pages 49–58, San Francisco, CA, USA, 2001. Morgan Kaufmann Publishers Inc. (Cited on page 85.)

[Maedche 2000] A. Maedche and Staab S. *Discovering Conceptual Relations from Text.* In IOS Press, editeur, Proceedings of the 14th European Conference on Artificial Intelligence, pages 321–325, Amesterdam, 2000. (Cited on page 51.)

[Maedche 2001] Alexander Maedche and Steffen Staab. *Ontology Learning for the Semantic Web.* IEEE Intelligent Systems, vol. 16, pages 72–79, March 2001. (Cited on pages 50, 51 and 53.)

[Maedche 2002] Alexander Maedche. Ontology learning for the semantic web. Kluwer Academic Publishers, 2002. (Cited on page 50.)

[Maedche 2003] Alexander Maedche, Boris Motik, Ljiljana Stojanovic, Rudi Studer and Raphael Volz. *Ontologies for Enterprise Knowledge Management.* IEEE Intelligent Systems, vol. 18, pages 26–33, March 2003. (Cited on page 54.)

[Mannila 1992] Heikki Mannila and Kari-Jouko Räihä. The design of relational databases. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1992. (Cited on page 35.)

[Manolescu 2000] Ioana Manolescu, Daniela Florescu, Donald Kossmann, Florian Xhumari and Dan Olteanu. *Agora: Living with XML and Relational.* In Proceedings of the 26th International Conference on Very Large Data Bases, VLDB '00, pages 623–626, San Francisco, CA, USA, 2000. Morgan Kaufmann Publishers Inc. (Cited on page 36.)

[McKay 1981] Brendan McKay. *Practical graph isomorphism.* Congressus Numerantium, 1981. (Cited on page 16.)

[Messmer 1998] Bruno T. Messmer and Horst Bunke. *A New Algorithm for Error-Tolerant Subgraph Isomorphism Detection.* IEEE Trans. Pattern Anal. Mach. Intell., vol. 20, pages 493–504, May 1998. (Cited on page 19.)

[Miller 1995] George A. Miller. *WordNet: a lexical database for English.* Commun. ACM, vol. 38, pages 39–41, November 1995. (Cited on page 50.)

[Minas 2002] Mark Minas. *Concepts and realization of a diagram editor generator based on hypergraph transformation.* Sci. Comput. Program., vol. 44, pages 157–180, August 2002. (Cited on page 31.)

[Morin 1999] E. Morin. *Automatic acquisition of semantic relations between terms from technical corpora.* In Proceedings of the fifth international congress on terminology and knowledge engineering (TKE-99), pages 268–278, Vienna, 1999. (Cited on page 51.)

[Navathe 1988] Shamkant B. Navathe and A. M. Awong. *Abstracting Relational and Hierarchical Data with a Semantic Data Model.* In Proceedings of the Sixth International Conference on Entity-Relationship Approach, pages 305–333, Amsterdam, The Netherlands, The Netherlands, 1988. North-Holland Publishing Co. (Cited on page 35.)

[Navigli 2004] Roberto Navigli and Paola Velardi. *Learning Domain Ontologies from Document Warehouses and Dedicated Web Sites.* Comput. Linguist., vol. 30, pages 151–179, June 2004. (Cited on page 52.)

[Neuhaus 2007] M. Neuhaus and H. Bunke. *Automatic learning of cost functions for graph edit distance.* Information Sciences, vol. 177, no. 1, pages 239–247, January 2007. (Cited on page 17.)

[Newman 2001] M. E. J. Newman. *Scientific collaboration networks. II. Shortest paths, weighted networks, and centrality.* Physical Review E, vol. 64, page 016132, 2001. (Cited on pages 130, 131 and 170.)

[Noy 2001] McGuinness L. Noy N. *Ontology Development 101: A Guide to Creating Your First Ontology.* Technical report ksl-01-05 and stanford medical informatics technical report smi-2001-0880, Stanford Knowledge Systems Laboratory, 2001. (Cited on pages 49 and 59.)

[Ohgren 2005] A. Ohgren and K. Sandkuhl. *Towards a Methodology for Ontology Development in Small and Medium-Sized Enterprises.* In IADIS Conference on Applied Computing, Algarve, Portugal, 2005. (Cited on page 49.)

[O'Reilly 2005] Tim O'Reilly. *What Is Web 2.0. Design Patterns and Business Models for the Next Generation of Software.* http://www.oreillynet.com/pub/a/oreilly/tim/news/2005/09/30/what-is-web-20.html, September 2005. Stand 12.5.2009. (Cited on page 2.)

[Paredaens 1995] Jan Paredaens, Peter Peelman and Letizia Tanca. *G-Log: A Graph-Based Query Language.* IEEE Trans. on Knowl. and Data Eng., vol. 7, pages 436–453, June 1995. (Cited on pages 7 and 26.)

[Park 2011] Jaehui Park and Sang goo Lee. *Keyword search in relational databases.* Knowl. Inf. Syst., vol. 26, no. 2, pages 175–193, 2011. (Cited on page 37.)

[Pérez 2009] Jorge Pérez, Marcelo Arenas and Claudio Gutierrez. *Semantics and complexity of SPARQL.* ACM Trans. Database Syst., vol. 34, pages 16:1–16:45, September 2009. (Cited on page 28.)

[Petros Papapanagiotou 2006] Vassileios Tsetsos Christos Anagnostopoulos Stathes Hadjiefthymiades Petros Papapanagiotou Polyxeni Katsiouli. *RONTO: RELATIONAL TO ONTOLOGY SCHEMA MATCHING.* AIS SIGSEMIS Bulletin, vol. 3, page 32â36, 2006. (Cited on page 35.)

[Power 2007] D.J. Power. *A Brief History of Decision Support Systems.* DSSResources.COM, March 2007. (Cited on page 1.)

[Punin 2001] John Punin and Mukkai Krishnamoorthy. *XGMML (eXtensible Graph Markup and Modeling Language) 1.0 Draft Specification*, 2001. (Cited on page 14.)

[Rania Soussi 2012] Marie-Aude Aufaure Amine Louati Rania Soussi Etienne Cuvelier and Yves Lechevallier. Social network analysis and mining, chapitre DB2SNA: an All-in-one Tool for Extraction and Aggregation of underlying Social Networks from Relational Databases. Springer LNSN, 2012. (Cited on page 133.)

[Riesen 2007] Kaspar Riesen, Stefan Fankhauser and Horst Bunke. *Speeding Up Graph Edit Distance Computation with a Bipartite Heuristic.* In MLG, 2007. (Cited on page 17.)

[Riesen 2010a] Kaspar Riesen and Horst Bunke. Graph classification and clustering based on vector space embedding. World Scientific Publishing Co., Inc., River Edge, NJ, USA, 2010. (Cited on page 17.)

[Riesen 2010b] Kaspar Riesen, Xiaoyi Jiang and Horst Bunke. *Exact and Inexact Graph Matching: Methodology and Applications.* In Managing and Mining Graph Data, pages 217–247. 2010. (Cited on page 15.)

[Rodgers 1998] P. J. Rodgers. *A Graph Rewriting Programming Language for Graph Drawing.* In Proceedings of the IEEE Symposium on Visual Languages, VL '98, pages 32–, Washington, DC, USA, 1998. IEEE Computer Society. (Cited on page 31.)

[Rodrigues Jr. 2006] Jose F. Rodrigues Jr., Agma J. M. Traina, Christos Faloutsos and Caetano Traina Jr. *SuperGraph Visualization.* In ISM '06: Proceedings of the Eighth IEEE International Symposium on Multimedia, pages 227–234, Washington, DC, USA, 2006. IEEE Computer Society. (Cited on page 133.)

[Ronen 2009] Royi Ronen and Oded Shmueli. *SoQL: A Language for Querying and Creating Data in Social Networks.* In Proceedings of the 2009 IEEE International Conference on Data Engineering, pages 1595–1602, Washington, DC, USA, 2009. IEEE Computer Society. (Cited on page 25.)

[Rozenberg 1997] Grzegorz Rozenberg. Handbook of graph grammars and computing by graph transformation: volume i. foundations. World Scientific Publishing Co., Inc., River Edge, NJ, USA, 1997. (Cited on page 31.)

[S. Paparizos 2002] H. V. Jagadish L. V. S. Lakshmanan A. Nierman D. Srivastava S. Paparizos S. Al-Khalifa and Y. Wu. *Grouping in xml.* In EDBT â02 Worshops, page 128â147, 2002. (Cited on page 30.)

[Sahoo 2008] Satya S. Sahoo, Olivier Bodenreider, Joni L. Rutter, Karen J. Skinner and Amit P. Sheth. *An ontology-driven semantic mashup of gene and biological pathway information: Application to the domain of nicotine dependence.* J. of Biomedical Informatics, vol. 41, pages 752–765, October 2008. (Cited on page 36.)

[Sahoo 2009] Satya S. Sahoo, Wolfgang Halb, Sebastian Hellmann, Kingsley Idehen, Ted Thibodeau Jr, SÃ¶ren Auer, Juan Sequeda and Ahmed Ezzat. *A Survey of Current Approaches for Mapping of Relational Databases to RDF*, 01 2009. (Cited on page 35.)

[Schenker 2005] Adam Schenker, Horst Bunke, Mark Last and Abraham Kandel. *Graph-Theoretic Techniques for Web Content Mining.* 2005. (Cited on page 15.)

[Schlobach 2004] Olsthoorn M. Schlobach S. and M. de Rijke. *Type Checking in Open-Domain Question Answering.* In Proceedings of European Conference on Artificial Intelligence 2004, 2004. (Cited on page 53.)

[Scowen 1998] R. S. Scowen. *Extended BNF/Ageneric base standard*, September 1998. Paper for SIGPLAN Notice. (Cited on page 105.)

[Seaborne 2004] Andy Seaborne. *RDQL - A Query Language for RDF (Member Submission).* Rapport technique, W3C, January 2004. (Cited on page 28.)

[Seaborne 2007] Â Steer Â D.-Â Williams Â S Seaborne Â A. *SQL-RDF*, 2007. (Cited on page 35.)

[Shamsfard 2003] Mehrnoush Shamsfard and Ahmad Abdollahzadeh Barforoush. *The state of the art in ontology learning: a framework for comparison.* Knowl. Eng. Rev., vol. 18, pages 293–316, December 2003. (Cited on page 52.)

[Sheng 1999] L. Sheng and G. Ãzsoyoglu. *A Graph Query Language and Its Query Processing.* In In ICDE, pages 572–581, 1999. (Cited on page 24.)

[Shonkry 1996] A. Shonkry and M. Aboutabl. *Neural network approach for solving the maximal common subgraph problem.* Systems, Man and Cybernetics, Part B, IEEE Transactions on, vol. 26 - Issue 5, pages 785–790, October 1996. (Cited on page 19.)

[Silverston 2001] Len Silverston. The data model resource book, volume 1. John Wiley & Sons, 2001. (Cited on pages xi, 57 and 59.)

[Sintek 2002] Michael Sintek and Stefan Decker. *TRIPLE - A Query, Inference, and Transformation Language for the Semantic Web.* In Proceedings of the First International Semantic Web Conference on The Semantic Web, ISWC '02, pages 364–378, London, UK, UK, 2002. Springer-Verlag. (Cited on page 28.)

[Sperduti 1997] A. Sperduti and A. Starita. *Supervised Neural Networks for the Classification of Structures.* IEEE Transactions on Neural Networks, vol. 8, no. 3, pages 714–735, 1997. (Cited on page 18.)

[Studer 1998] Rudi Studer, V. Richard Benjamins and Dieter Fensel. *Knowledge Engineering: Principles and Methods.* Data and Knowledge Engineering, vol. 25, no. 1-2, pages 161–197, March 1998. (Cited on page 46.)

[Suarez-Figueroa 2007] G. Aguado de Cea C. Buil C. Caracciolo M. Dzbor A. Gomez-PÃ©rez G. Herrrero H. Lewen E. Montiel-Ponsoda Suarez-Figueroa M. C. and V. Presutti. *D5.3.1 neon development process and ontology life cycle.* Neon deliverable, August 2007. (Cited on page 49.)

[Suarez-Figueroa 2008] M. C. Suarez-Figueroa and A. Gomez-PÃ©rez. *Building Ontology Networks: How to Obtain a Particular Ontology Network Life Cycle?* In International Conference on Semantic Systems (ISEMANTICS08), Graz, Austria, September 2008. (Cited on page 47.)

[Suganthan 2002] P. N. Suganthan. *Structural pattern recognition using genetic algorithms.* Pattern Recognition, vol. 35, pages 1883–1893, 2002. (Cited on page 19.)

[Tanev 2006] Hristo Tanev and Bernardo Magnini. Weakly supervised approaches for ontology population. 2006. (Cited on pages 53 and 54.)

[Tian 2008] Yuanyuan Tian, Richard A. Hankins and Jignesh M. Patel. *Efficient aggregation for graph summarization.* In Proceedings of the 2008 ACM SIGMOD international conference on Management of data, SIGMOD '08, pages 567–580, New York, NY, USA, 2008. ACM. (Cited on pages 110 and 133.)

[Tran 2011] Thanh Tran, Peter Mika, Haofen Wang and Marko Grobelnik. *SemSearch'11: the 4th semantic search workshop.* In WWW (Companion Volume), pages 315–316, 2011. (Cited on page 2.)

[Ullmann 1976] J. R. Ullmann. *An Algorithm for Subgraph Isomorphism.* J. ACM, vol. 23, pages 31–42, January 1976. (Cited on page 16.)

[Umeyama 1988] S. Umeyama. *An Eigendecomposition Approach to Weighted Graph Matching Problems.* IEEE Trans. Pattern Anal. Mach. Intell., vol. 10, pages 695–703, September 1988. (Cited on page 18.)

[Uschold 1995] Mike Uschold and Martin King. *Towards a Methodology for Building Ontologies.* In In Workshop on Basic Ontological Issues in Knowledge Sharing, held in conjunction with IJCAI-95, 1995. (Cited on pages 50, 54, 55 and 56.)

[Uschold 1998] Mike Uschold, Martin King, Stuart Moralee and Yannis Zorgios. *The Enterprise Ontology.* Knowl. Eng. Rev., vol. 13, pages 31–89, March 1998. (Cited on pages 55, 57 and 68.)

[Varró 2005] Gergely Varró, Katalin Friedl and Dániel Varró. *Graph Transformation in Relational Databases.* Electron. Notes Theor. Comput. Sci., vol. 127, pages 167–180, March 2005. (Cited on page 32.)

[Velardi 2005] Navigli-R. Cuchiarelli A. Velardi P. and Neri F. Ontology learning from text: Methods, evaluation and applications., chapitre Evaluation of Ontolearn, a Methodology for Automatic Population of Domain Ontologies. IOS Press, 2005. (Cited on page 53.)

[Wang 2005] Chunyan Wang, Anthony Lo, Reda Alhajj and Ken Barker. *Novel Approach for Reengineering Relational Databases into XML.* In Proceedings of the 21st International Conference on Data Engineering Workshops, ICDEW '05, pages 1284–, Washington, DC, USA, 2005. IEEE Computer Society. (Cited on page 36.)

[Watkins 2000] Chris Watkins. *Dynamic Alignment Kernels.* In A. Smola and P. Bartlett, editeurs, Advances in Large Margin Classifiers, chapitre 3, pages 39–50. MIT Press, Cambridge, MA, USA, 2000. (Cited on page 18.)

[Wilson 1997] Richard C. Wilson and Edwin R. Hancock. *Structural Matching by Discrete Relaxation.* IEEE Trans. Pattern Anal. Mach. Intell., vol. 19, pages 634–648, June 1997. (Cited on page 18.)

[Wilson 2005] Richard C. Wilson, Edwin R. Hancock and Bin Luo. *Pattern Vectors from Algebraic Graph Theory.* IEEE Trans. Pattern Anal. Mach. Intell., vol. 27, pages 1112–1124, July 2005. (Cited on page 18.)

[Winkler 1999] William E. Winkler. *The state of record linkage and current research problems.* Rapport technique, Statistical Research Division, U.S. Bureau of the Census, 1999. (Cited on page 85.)

[Wong 1990] M.; Chan S.C.; Wong A.K.C.; You. *An algorithm for graph optimal monomorphism.* IEEE Transactions on Systems, Man and Cybernetics,, vol. 20, pages 628–638, 1990. (Cited on page 16.)

[Xu 2001] Lei Xu and Irwin King. *A PCA Approach for Fast Retrieval of Structural Patterns in Attributed Graphs.* In Humboldt University Berlin, 2001. (Cited on page 18.)

[Zouaq 2009] A. Zouaq and R. Nkambou. *Evaluating the Generation of Domain Ontologies in the Knowledge Puzzle Project.* IEEE Trans. Knowledge and Data Eng, vol. 21, no. 11, pages 1559–1572, 2009. (Cited on page 53.)