UNIVERSIDAD DE CHILE
FACULTAD DE CIENCIAS FÍSICAS Y MATEMÁTICAS
DEPARTAMENTO DE CIENCIAS DE LA COMPUTACIÓN

MODELOS DE BASE DE DATOS DE GRAFO Y RDF

TESIS PARA OPTAR AL GRADO DE DOCTOR EN CIENCIAS,
MENCIÓN COMPUTACIÓN

RENZO ANGLES ROJAS

PROFESOR GUÍA:
CLAUDIO GUTIERREZ GALLARDO

MIEMBROS DE LA COMISIÓN:
PABLO BARCELÓ BAEZA
BENJAMIN BUSTOS CÁRDENAS
GONZALO NAVARRO BADINO
AXEL POLLERES

SANTIAGO DE CHILE
AGOSTO 2009

Universidad de Chile

Facultad de Ciencias Físicas y Matemáticas

Escuela de Postgrado

Modelos de Base de Datos de Grafo y RDF

Por

Renzo Angles Rojas

Tesis para optar el grado de

Doctor en Ciencias mención Computación

Profesor Guía	:	Claudio Gutierrez Gallardo
Comité	:	Pablo Barceló Baeza
	:	Benjamin Bustos Cárdenas
	:	Gonzalo Navarro Badino
	:	Axel Polleres (Profesor Externo)

Esta tesis ha recibido el apoyo de: Programa MECESUP (Chile) proyecto No. UCH0109; NIC Chile; Núcleo Milenio Centro de Investigación de la Web (Chile), proyecto No. P04-067-F; y FONDECYT (Chile), proyectos No. 1030810 y No. 1070348.

Departamento de Ciencias de la Computación - Universidad de Chile

Santiago - Chile

Agosto de 2009

Resumen

En el año 2004 el Consorcio de la *World Wide Web* (W3C) estandarizó un lenguaje de metadatos para la descripción de recursos en la Web denominado *Resource Description Framework* (RDF). La motivación fue el definir un lenguaje que sirva de base para el modelado extensible de dominios altamente interconectados y redes de información.

La especificación de RDF puede ser vista, desde un punto de vista de base de datos, como un modelo de base de datos. En efecto, subyacente a RDF se encuentra un modelo el cual trae a la mente la noción de datos con estructura de grafo. Por otra parte, la cifra creciente de información representada en el lenguaje RDF ha sido acompañada de varias propuestas sobre como almacenar y consultar datos RDF. En consecuencia, resulta natural el estudio de RDF desde un punto de vista de base de datos.

El objetivo principal de esta tesis es el estudio de RDF desde una perspectiva de base de datos. Nuestro estudio se concentra en el modelado de base de datos, enfocándonos en modelos particulares llamados modelos de base de datos de grafo, los cuales parecen estar más relacionados a RDF desde un punto de vista teórico. Asimismo, esta tesis sigue de cerca los desarrollos del W3C en el sentido de entregar a RDF el soporte de un modelo de base de datos, en particular desarrollar los aspectos de consulta de datos.

La principal contribución de la tesis es haber clarificado y desarrollado la relación entre RDF y los modelos de base de datos de grafo. Para esto, primero se estudió RDF como un modelo de base de datos. Segundo, se realizó una conceptualización del área de modelos de base de datos de grafo. Tercero, se propuso un conjunto deseable de propiedades de grafo las cuales deberían ser soportadas por un lenguaje de consulta ideal para RDF. Cuarto, se demostró que el lenguaje de consulta estándar definido por el W3C, denominado SPARQL, no soporta las consultas de grafo esenciales que se esperaría encontrar en un lenguaje de consulta para bases de datos de grafo. Esto fue conseguido al estudiar y determinar el poder expresivo de SPARQL. Finalmente, y debido a los resultados anteriores, nos concentramos en la aplicación de los modelos de bases de datos de grafo a la visualización de datos, esto al definir formalmente y caracterizar un modelo de visualización de grafo para RDF.

A mis padres, Dalton y Betty, quienes sacrificaron todo por entregarme mucho más.

Agradecimientos

Nunca consideres el estudio como una obligación, sino como una oportunidad para ingresar en el bello y maravilloso mundo del saber.

Albert Einstein

Esta es la última página que escribo de esta tesis, quizás la más difícil porque no hay espacio suficiente para describir los acontecimientos ni agradecer a todas las personas que de un modo u otro formaron parte de esta etapa de mi vida.

Recuerdo cuando llegué a Chile en agosto del 2002, sin conocer a nadie, incluso sin tener una clara idea de lo que estaba por empezar, pero eso sí, con la firme certeza de poder enorgullecer a mi familia entregando lo mejor de mí. No fue fácil iniciar una vida en otro país, pero afortunadamente pude rodearme de personas que me permitieron olvidar este detalle y trabajar en mis metas. Ahora, miro hacia atrás, me doy cuenta de todas las cosas que hice y, aunque creo que pude hacer mucho más, me alienta saber que dí un gran paso en aras de cumplir mis sueños.

Antes que todo quiero agradecer a mis padres quienes me enseñaron que el mejor regalo para un hijo es la educación. A mi padre Dalton, mi ejemplo de esfuerzo y superación profesional (el siguiente Doctor eres tu). A mi madre Betty, cuyo recuerdo llevo en mi corazón y revive con mis mejores acciones.

Agradecimientos a los miembros de mi familia y amigos de juventud. A mi mamita Lelita, por su rol de segunda madre. A mi tía Olinda, por ser la mejor cocinera de la familia. A mi novia Yenny, por brindarme el amor de pareja y el apoyo de compañera. A mi mejor amigo Juan Daniel, por sus consejos y comentarios imparciales. A los amigos de Arequipa, mi tierra, Aldo, Gerardo, Javier, Jessica, Oskhar, Welton y William.

A los amigos chilenos y extranjeros. A mis “patas” del alma, Andrés Neyem, Carlos Acosta y Christián Bridevaux, por las sesiones de comida, conversación y diversión. A los compañeros de estudio, Felipe Aguilera, Diego Arroyuelo, Cesar Collazos, Gilberto Gutierrez, Marcelo Mendoza y Rodrigo Paredes. Al grupo de chilenos más divertidos que conozco, Marcelo Arenas, Pablo Barceló y Jorge Pérez.

Al DCC de la Universidad de Chile en pleno, por brindar al extranjero un lugar donde pueda sentirse como en casa. A los profesores, por el apoyo y los conocimientos entregados. Un agradecimiento especial al profesor Sergio Ochoa, por la amistad y las oportunidades brindadas. A los funcionarios y demás trabajadores, gracias por los momentos de diversión y alegría.

Por último, un agradecimiento especial a Claudio Gutierrez, profesor guía, amigo y consejero. Siempre recordaré sus primeras palabras de advertencia: ´´*Soy partidario de desarrollar trabajos serios y relevantes. Deberás ser paciente ya que los buenos resultados toman su tiempo*". Estas palabras marcaron el desarrollo de esta tesis y marcarán mi carrera como investigador.

Renzo

Universidad de Chile

Faculty of Physics and Mathematics

Postgraduate School

Graph Database Models and RDF

by

Renzo Angles Rojas

Submitted to the Universidad de Chile in fulfillment
of the thesis requirement to obtain the degree of
Ph.D. in Computer Science

Advisor	:	Claudio Gutierrez Gallardo
Committee	:	Pablo Barceló Baeza
	:	Benjamin Bustos Cárdenas
	:	Gonzalo Navarro Badino
	:	Axel Polleres (External)

This work has been supported by: MECESUP (Chile) project No. UCH0109; NIC Chile; Millenium Nucleus, Center for Web Research (Chile) project No. P04-067-F; and FONDECYT (Chile) projects No. 1030810 and No. 1070348.

Department of Computer Science - Universidad de Chile
Santiago - Chile
August 2009

Abstract

In 2004 the World Wide Web Consortium (W3C) standardized a metadata language for describing resources on the Web, called the Resource Description Framework (RDF). The motivation was to have a language to serve as the basis for extensible modeling of highly interconnected domains and networks of information.

The RDF specification can be viewed, from a database point of view, as a database model. In fact, underlying RDF resides a model which brings to mind data with a graph-like structure. Moreover, due to the increasing amount of information defined in the RDF language, several proposals for storing and querying RDF data have been presented. Hence, it becomes natural to study RDF from the point of view of databases.

The main goal of this thesis is the study of RDF from a database perspective. We concentrate our study in the area of database modeling and focus our efforts in particular models called graph database models, which from a theoretical point of view seem more related to RDF. This thesis also follows closely the W3C developments in the direction of giving database model support to RDF, particularly on query features.

The main contribution of the thesis is to have clarified and developed the relationship between RDF and graph database models. This has been done first, by studying RDF as a database model. Second, by conceptualizing the area of graph database models. Third by proposing a desiderata of graph features which an ideal query language for RDF should support. Fourth, by showing that the standard query language defined by the W3C, SPARQL, does not support essential graph features expected in a query language for graph databases. This was done by characterizing the expressive power of SPARQL. Finally, due to the above results, we concentrate on the application of graph database models to data visualization, by formally defining a graph visualization model for RDF.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

The World Wide Web (or just Web) has changed the way people communicate with each other, how information is disseminated and retrieved, and how business is conducted. Typical uses of the Web today involve seeking and making use of information, searching for and getting in touch with other people, data sharing, etc.[1]

Unfortunately, the Web has developed most rapidly as a medium of documents for people rather than of information that can be manipulated automatically [2]. In this direction, an approach to build up the Web is to represent content in a form that is more easily machine-processable and to use intelligent techniques to take advantage of these representations [1]. We refer to this plan as the *Semantic Web*.

A first step for a Semantic Web is putting data on the Web in a form that can be processed directly or indirectly by machines [3]. In this direction, the *Resource Description Framework* (RDF) [4] is a recommendation of the World Wide Web Consortium (W3C), which proposes a framework for representing Web information in a form that machines can naturally understand.

RDF is based on the idea of identifying resources using Web identifiers, and describing such resources in terms of simple properties and property values. This results in a data model that allows modeling information with graph-like structure.

There are many projects that use RDF for representing data from social networks [5], lexical references [6], on-line encyclopedias [7, 8], digital library collections [9], etc. Therefore a basic requirement to develop the Semantic Web is to be able to manage large volumes of RDF data. This brings us to the realm of databases, and particularly the question: what database model is more suitable for modeling and querying RDF data?

The term *"data model"* has been used in the information management community widely and diversely, thus it covers various meanings. In the most general sense, a data model is a collection of conceptual tools used to model representations of real-world entities and the relationships among them [10]. This term is also often used to refer to a collection of data structure types. Mathematical frameworks for representing knowledge have also been called

data models [11].

From a database point of view, the conceptual tools that make up a data model (also called database model) should at least address data structuring, description and maintenance, and a way to retrieve or query the data. According to these criteria, a database model consists of three components: a set of data structure types, a set of operators or inference rules, and a set of integrity rules [12].

Due to the philosophical and practical importance of conceptual modeling, database models have become essential abstraction tools. Database models can be used for: specifying permitted data types, supporting a general database design methodology, coping with database evolution, developing families of high-level languages for query and data manipulation, focusing on database management systems architecture, and being a vehicle for research into the behavioral properties of alternate data organization structures [12].

Beginning in the Seventies numerous database models have been proposed, each of them with their own concepts and terminology (e.g. the Relational Model [13]). A particular type of models, called *Graph Database Models*, can be characterized as those where data structures for the schema and instances are modeled as graphs or generalizations of them, and data manipulation is expressed by graph-oriented operations and type constructors.

In brief, a graph database model introduces a level of abstraction that allows: a more natural modeling of graph data, querying directly the graph structure, and special structures and algorithms for storing and querying graph data.

1.1 Motivations and Contents

From a research point of view, the RDF model can be considered from at least two perspectives:

(1) From a *logical perspective*, as a minimal fragment of logic that includes all relevant features needed as representation language for metadata, or as the W3C recommendation says: RDF is an assertional language intended to be used to express propositions using precise formal vocabularies [14].

(2) From a *database perspective*, as an extension in expressive power of data models used in the database community.

The former point of view has been an active area of research. This does not come as surprise knowing that RDF emerged as a language to represent metadata on the Web, distilling the experience of the community of knowledge representation and Web researchers and developers [15]. The latter point of view has received less attention and is the focus of this thesis. Currently there is ample research work on storing and querying RDF data, but none of these works defines a database model.

The consideration of RDF as a database model puts forward the issue of developing coherently all its database features. Particular attention is paid to the query language, which should address the type of queries and problems of the application domains of RDF. This motivated us to study RDF from a database perspective, and to exploit this approach to improve the design of RDF databases. Therefore, the overall development of this thesis is driven by the goal of applying the theory and practice of databases to the design of RDF databases, an almost untouched area when we started the research in this thesis.

One of the main features of RDF is its ability to interconnect information resources. This results in a graph-like structure for which connectivity is a central notion [16]. As we will argue, basic concepts and problems of graph modeling play an important role for applications that involve RDF querying. Considering that the most representative RDF query languages (when we started this study) exploited the underlying graph structure of RDF only to a limited extent, we started studying and proposing - based in the experience of databases - a list of basic features that RDF query languages should ideally support to take advantage of the graph-like nature of RDF data.

Following this direction, if one would abstract from the real world developments, the ideal proposal for an RDF database model, because its closeness to graphs, would be a graph database model. This motivated us to survey in depth the work that has been done in the area by the community of databases.

In the meantime, the W3C issued a candidate recommendation (standard) of query language for RDF, called SPARQL [17], which definitively oriented the path that would follow the community. It was approved as an official standard by the W3C in January 2008, although its semantics and expressiveness generated ample debate, and many of the problems raised remained open due to lack of understanding of its theoretical expressive power. These reasons and our main goal of developing the basis of RDF databases let us to partially deviate from our goal in order to study this new specification, in particular how it is related with our graph database goal. The result was surprising as it showed that SPARQL is equivalent to relational algebra and thus the approved standard query language is no capable of expressing basic graph query features like connectivity, paths, etc.

Nevertheless, the very nature of RDF forced the community to continue investigating graph features in other scenarios. In this context, the project Linked Open Data includes the development of applications to publish existing open RDF datasets and to interlink resources between them. Particularly important turns out to be the visualization level, where, interestingly, surfaced again the graph nature of RDF. Thus we applied the background and designs of graph database models we had been studied in this area, formalizing a visualization data model based on the notion of nested graphs (i.e. graphs whose nodes can also be graphs).

The thesis is thus organized around these developments. In order for the reader to get a better grasp of the historical developments around this thesis, we present a roadmap of the main related events that together with our original goals guided our main focus of research.

February 1999. *The World Wide Consortium (W3C) publishes the initial document about RDF*, "Resource Description Framework (RDF) Model and Syntax Specification" [15]. The use of RDF data lead to the development of several RDF query languages.

February 2004. *The RDF model is endorsed as a W3C standard [4].* The research on RDF query languages is accompanied with the development of RDF repositories (the concept of RDF database will appear later).

February 2004. *Formation of the W3C RDF Data Access Working Group (DAWG).* Its mission is the design of a standard query language for RDF. A systematic race for defining the standard RDF query language begins.

June 2004. *Publication of the paper "Foundations of Semantic Web Databases [18]".* This article marks the beginning of the theoretical research on RDF databases.

Late 2004. *We begin our study around RDF query languages.* This research marks the beginning of this thesis.

March 2005. *The proposal of this thesis is approved.* The goal is the design of a graph database model oriented to support the RDF model.

May 2005. *Article "Querying RDF Data From a Graph Database Perspective".* In this article, presented as research paper in the 2nd European Semantic Web Conference, we propose the introduction of graph operators in the design of RDF query languages based on real-life use-cases. This is the first result of this thesis.

June 2005. *We begin a systematic review on graph database models.* Our "Survey of Graph Database Models" is sent for publication to ACM Surveys in November 2005 (it will appear in January 2008). In this work, we present a conceptualization and classification of the area of graph database models. This is the second contribution of this thesis .

Early 2006. *Begins our research on RDF databases.* It includes reviewing current RDF repositories and their support for storing huge RDF datasets. We study the application of graph database models in the design of RDF databases.

July 2006. *Tim Berners-Lee outlines the Web of Linked Open Data (LOD)*. The vision includes principles and mechanisms for large scale interlinking of open RDF datasets.

September 2006. *Franz Inc. contact us to test its RDF database, AllegroGraph*. We experience about issues in the design of real-life RDF databases, particularly the support for graph queries.

February 2007. *The W3C SWEO community founds the Linking Open Data project [19]*. It increases the research on browsers and search engines for Linked Data. We start a formal study on models for RDF visualization, particularly the Tabulator [20], one of the first Semantic Web data browser whose development follows the Linked Data principles.

Mid 2007. *We incorporate to our research the RDF query language SPARQL*. The importance gained by SPARQL, as the W3C proposal, motivates us to partially deviate our focus to study some open issues in its specification.

January 2008. *SPARQL is endorsed as the standard RDF query language [4]*. The SPARQL specification does not include graph operators, however there are many implementations that support them. This motivate us to concentrate our research around the recent specification and the expressive power of SPARQL.

Mid 2008. After finishing the SPARQL study, *we focus our research to RDF visualization*. Considering the current trends in querying RDF data (led by SPARQL), we concentrate on models for visualizing RDF data where a graph-based approach seems to have a direct application.

October 2008. *Publication of the article "The Expressive Power of SPARQL"*. In this article, presented in the 7th International Semantic Web Conference, we show that SPARQL has the same expressive power as Relational Algebra hence, not allowing graph features.

December 2008. *Acceptation of the article "A Nested Graph Model for Visualizing RDF Data"*. This article, accepted in the 3rd International Workshop on Foundations of Data Management, is the result of our research on RDF graph visualization and presents the application of graph database models in the context of RDF visualization.

January 2009. *This thesis is submitted for review.*

1.2 Outline of the Thesis

The organization of this thesis is as follows: In Chapter 2 the subject areas, database models and RDF, are briefly introduced and related. In Chapter 3 a survey of graph database models is presented. In Chapter 4 a study of the support for querying graph properties in RDF query languages is presented. In Chapter 5 the expressive power of SPARQL is studied. In Chapter 6 a graph model for visualizing RDF data is defined. Finally, in Chapter 7 we give a summary of the main results and indicate directions for further research.

Chapter 2

RDF and Database Models

The *Semantic Web* is an evolving extension of the Web that allows data to be shared and reused across application, enterprise, and community boundaries.[1] On this line, the *Resource Description Framework (RDF)* provides the technology for expressing the meaning of terms and concepts in a form that computers can readily process [2]. Recently, the increasing amount of RDF data has motivated the development of *RDF databases*.

Underlying the structure of a database is a *data(base) model*, a collection of conceptual tools for describing the real-world entities to be modeled in the database and the relationships among these entities [10]. The Relational Model is a paradigmatic example.

In this chapter we introduce RDF and database models as subject areas of this thesis. Additionally, we study RDF from a database modeling perspective by comparing it with database models.

2.1 RDF and the Semantic Web

Most of the Web content today is designed for humans to read, not for computer programs to manipulate meaningfully. In fact, computers can adeptly parse Web pages for layout and routine processing but in general, computers have no reliable way to process and understand the relations and semantics of Web pages [2].

The main obstacle to providing better support to Web users is that, at present, the meaning of Web content is not machine-accessible [1]. Of course, there are tools that can retrieve texts, split them into parts, check the spelling, count their words. But when it comes to interpreting sentences and extracting useful information for users, the capabilities of current software are still limited. Therefore, despite improvements in search engine technologies, the difficulties remain essentially the same, “data on the Web is not understandable for machines”.

[1] `http://www.w3.org/2001/sw/`

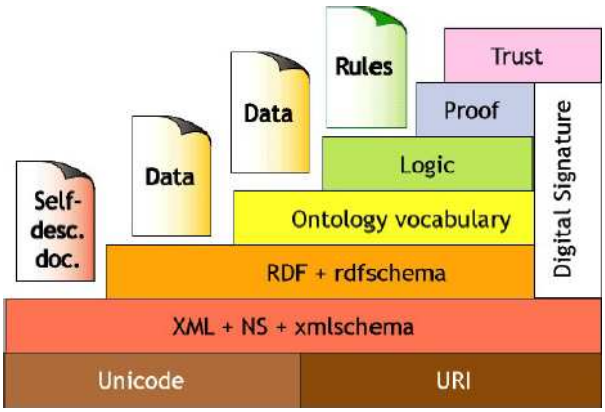


Figure 2.1: A layered approach to the Semantic Web [1].

2.1.1 The Semantic Web

The *Semantic Web* is not a separate Web but an extension of the current one, in which information is given well-defined meaning, better enabling computers and people to work in cooperation [2]. It derives from the Tim Berners-Lee's vision[2] (the same person who invented the Web in the late 1980s), and is promoted by the *World Wide Web Consortium* (W3C), an international standardization body for the Web.

The vision of a semantic Web involves more than retrieving HTML pages from Web servers. Such vision concentrates in the meaning of the information and implies a way to process automatically, for instance, relationships between resources in the Web (e.g. people, Web pages, organizations, etc.).

The development of the Semantic Web proceeds in steps, each step building a layer on top of another. Figure 2.1 shows the "layer cake" of the Semantic Web, which describes the main layers of its design and vision. Next we will describe the technologies associated to each layer. In order to visualize the meaning of each layer consider a running example consisting in modeling information about people (i.e. social network data).

- *Unicode* [21] is an industry standard for encoding symbols in text documents. In the context of the Semantic Web, Unicode is the universal standard allowing computers to consistently represent and manipulate text expressed in most of the writing systems.

- A *Uniform Resource Identifier* (URI) [22] is a compact string of characters used to identify or name a resource. A *Uniform Resource Locator* (URL) is a popular example of a URI that identifies, the location of, a Web retrievable document or service. Additionally, an *Internationalized Resource Identifier* (IRI) [23] is a generalization of a URI which extends the syntax of URIs to a much wider repertoire of characters.

[2] `http://www.w3.org/2000/Talks/1206-xml2k-tbl`

The use of URIs provides a system of machine-processable identifiers for identifying uniquely resources and avoid confusions. A *URI reference* (or URIref) is a URI, together with an optional fragment identifier at the end. For example, the URI reference `http://socialdata.org/example#person1` consists of the URI `http://socialdata.org/example` and (separated by the "#" character) the fragment identifier `person1`.

- The *Extensible Markup Language* (XML) [24] is a simple, very flexible text format that lets one write structured documents with a user-defined vocabulary. XML has become the standard format for serializing (using Unicode) and sharing data across different information systems. It is particularly suitable for encoding and sending documents across the Web. XML is the syntactic foundation layer of the Semantic Web [1]. In our example, social network data can be stored and shared by using XML documents.

- *XML Namespaces* (NS) [25] provide a means to qualify the elements and attributes (the vocabulary) of XML documents. Each term used in an XML document, is identified by an URI that makes the term truly unique on the Web and thus, universal (among other things). XML namespaces are used to avoid ambiguity among vocabularies.

  For instance, the URI `http://www.w3.org/1999/02/22-rdf-syntax-ns#"` (associated with the prefix `rdf`) identifies the XML namespace for RDF. In our example, assume that the URI `http://socialdata.org/vocabulary#` (and prefix `sd`) is the XML namespace that identifies the terms used in our social network data descriptions. Thus consider that the property "is relation of" is identified by the (abbreviated) URIref `sd:isFriendOf`.

- The *Resource Description Framework* (RDF) [4] defines a data model for writing simple statements about (Web) resources. An RDF document contains assertions that particular resources (e.g. a person) have properties (e.g. "is friend of") with certain values (e.g. another person). For example, consider that *John Lennon* and *Paul McCartney* are resources identified by URIrefs `ex:person1` and `ex:person2` respectively, where the prefix `ex` references the URI `http://socialdata.org/example#`. The RDF description for the assertion "*John Lennon is friend of Paul McCartney*" could be serialized in the XML-based syntax [26] as

```
<rdf:Description rdf:about="http://socialdata.org/example#person1">
  <sd:name>John Lennon</sd:name>
  <sd:isFriendOf>
    <rdf:Description rdf:about="http://socialdata.org/example#person2">
      <sd:name>Paul McCartney</sd:name>
    </rdf:Description>
  </sd:isFriendOf>
</rdf:Description>
```

A more intuitive syntax can be given by the N3 format [27]:

```
<ex:person1> <sd:name> "John Lennon" .
<ex:person1> <sd:isFriendOf> <ex:person2> .
<ex:person2> <sd:name> "Paul McCartney" .
```

- *RDF Schema* (RDFS) [28] is a language for describing properties and classes (a vocabulary) of RDF resources, with a semantics for generalization hierarchies of such properties and classes. RDF schema allows one to define classes and properties, subclass and sub-property relationships, and domain and range restrictions. For example, consider the RDF statement `<sd:musician rdfs:subClassOf sd:rockmusician>`. where the term `rdfs:subClassOf` represents the subclass relation in the RDFS vocabulary.

- *Ontology Vocabulary:* RDF Schema can be viewed as a primitive language for writing simple ontologies. The *Web Ontology Language (OWL)* [29] is a family of powerful knowledge representation languages for ontology definition. OWL expands RDF Schema by allowing the representation of more complex relationships between RDF resources. For example, the intersections of named classes is represented in OWL by using the term `owl:intersectionOf`.

- The *Logic layer* is used to enhance the ontology language further and to allow the writing of application-specific declarative knowledge.

- The *Proof layer* involves the actual deductive process as well as the representation of proofs in Web languages (from lower levels) and proof validation.

- The *Trust layer* will emerge through the use of digital signatures and other kinds of knowledge, based on recommendations by trusted agents or on rating and certification agencies and consumer bodies. Sometimes "Web of Trust" [30] is used to indicate that trust will be organized in the same distributed and chaotic way as the Web itself. Being located at the top of the pyramid, trust is a high-level and crucial concept: the Web will only achieve its full potential when users have trust in its operations (security) and in the quality of information provided.

Some elements of the semantic Web are expressed as prospective future possibilities that are yet to be implemented or realized. Other elements are expressed in formal specifications (e.g. XML, RDF and OWL). These technologies are combined in order to provide a formal description of concepts, terms, and relationships within a given knowledge domain.

In summary, the term *Semantic Web* comprises techniques that promise to dramatically improve the current Web and its use. The key idea is the use of machine-processable Web information. Key technologies include metadata, ontologies, logic and inferencing, and intelligent agents [1].

2.1.2 The Resource Description Framework (RDF)

The *Resource Description Framework* (RDF) [4] is a W3C recommendation intended to provide a simple way to describe Web resources. The development of RDF has been motivated by the following uses, among others [4]:

- *Web metadata*: providing information about Web resources and the systems that use them (e.g. content rating, capability descriptions, privacy preferences, etc.)
- Applications that require open rather than constrained information models (e.g. scheduling activities, describing organizational processes, annotation of Web resources, etc.)
- *To do for machine processable information (application data) what the Web has done for hypertext*: to allow data to be processed outside the particular environment in which it was created, in a fashion that can work at Internet scale.
- *Interworking among applications*: combining data from several applications to arrive at new information.
- *Automated processing of Web information by software agents*: the Web is moving from having just human-readable information to being a world-wide network of cooperating processes. RDF provides a world-wide lingua franca for these processes.

Although it was developed with the objective of describing Web resources, the broad goal of RDF is to define a mechanism for describing resources that makes no assumptions about a particular application domain. Examples of its use are lexical references [6], on-line encyclopedias [7, 8] digital library collections [9], social nets [5], on-line communities [31] and biological knowledge [32].

RDF is based on the idea that the resource being described have properties which have values, and that resources can be described by making statements that specify those properties and values. Hence, the fundamental concepts in RDF are *resources*, *properties* and *statements*.

Resources. We can think of a *resource* as an object (or a "thing") we want to describe. Resources may be authors, books, Web pages, and so on. Every resource in RDF is uniquely identified by a URI.

RDF is an open-world framework that allows anyone to make statements about any resource. Hence, the idea of using URIs to identify resources in the Web give us a global, worldwide, unique naming scheme which greatly reduces the homonym problem in the context of distributed data representation (note that an identifier does not necessarily enable access to a resource).

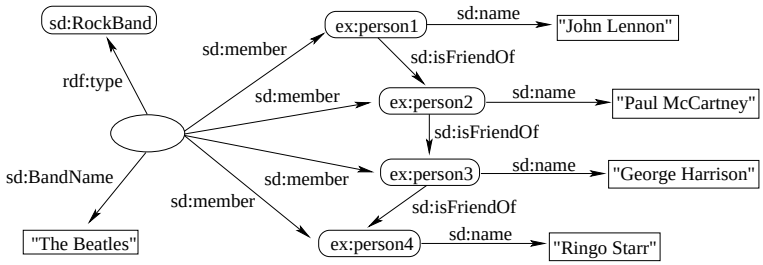
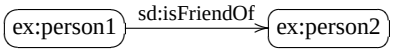


Figure 2.2: Example of an RDF graph. It shows URIs (rounded nodes and edge labels), a blank node (oval node) and literals (rectangular nodes).

Additionally, RDF allows us the use of anonymous resources called *blank nodes*. A blank node is a special kind of resource having no intrinsic name (like an existential variable in logic). A blank node could have a *node ID* which is limited in scope to an RDF document.

Properties. A *property* may be thought of as a characteristic or relation of a resource. For example "creator" and "title" can be properties of a Web page. Properties in RDF are also identified by URIs (i.e. they are a special kind of resource), thus properties can also have properties (this issue will be discussed in Section 2.3).

Statements. A *statement* asserts a property of a resource. For example "*John Lennon is friend of Paul McCartney*". In RDF, an statement is called an RDF triple. An *RDF triple* consists of a *subject*, a *predicate* and an *object*. The subject identifies the resource the statement is about (e.g. "John Lennon"). The predicate identifies the property or characteristic of the subject that the statement specifies (e.g. "is friend of"). The object identifies the value of that property (e.g. "Paul McCartney"). Values can either be resources or literals. Literals are atomic values (strings), such as numbers and dates, which may be plain (a string combined with an optional language tag) or typed (a string combined with a datatype URI). An RDF triple is graphically represented as a node-arc-node link:



The different syntactic forms of names are treated in particular ways. URI references are treated simply as logical constants. The presence of blank nodes is indicated by blank node identifiers. Plain literals are considered to denote themselves, so have a fixed meaning. An URI can occur in any part of an RDF triple, a blank node may be the subject or the object but not the predicate, whereas a literal may be the object but neither the subject nor the predicate.

A *general RDF expression* is a set of RDF triples, which can be intuitively considered as a labeled graph, called an *RDF Graph*. For example Figure 2.2 shows an RDF graph containing social network data. An RDF graph is usually called an *RDF database*.

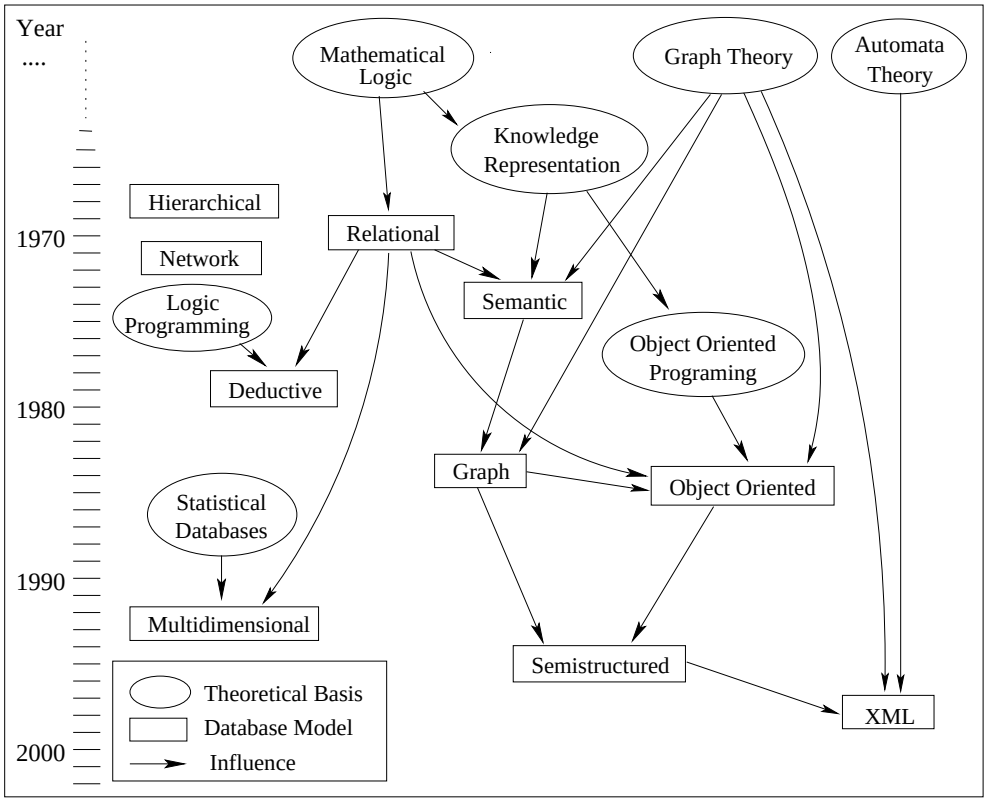


Figure 2.3: Evolution of database models. Rectangles denote database models, arrows indicate influences, and circles denote theoretical developments. A time-line in years is shown on the left. (based on a diagram by A. O. Mendelzon.)

2.2 Database Models

Since the emergence of database management systems, there has been an ongoing debate about what the database model (in what follows *db-model*) for such a system should be. The evolution and diversity of existent db-models shows that there are many factors that influence the development of these db-models. Some of the most important factors are the following: the characteristics or structure of the domain to be modeled, the type of theoretical tools that interest the expected users, and of course, the hardware and software constraints. Additionally, each db-model proposal is based on certain theoretical principles, and serves as basis for the development of related models. Figure 2.3 shows the evolution of database models and their influences.

Initial db-models focused essentially on the specification of data structures on actual file systems. Two representative db-models are the *hierarchical* [33] and the *network* [34] models, both of which place emphasis on the physical level. Kerschberg et al. [35] developed a taxonomy of db-models prior to 1976, comparing their mathematical structures and foundation, and the levels of abstraction used.

The relational model was introduced by Codd [13, 36] and highlights the concept of ab-

straction levels by introducing the idea of separation between physical and logical levels. It is based on the notions of sets and relations. Due to its ease of use, it gained wide popularity among business applications.

As opposed to previous models, *Semantic db-models* [37] allow database designers to represent objects and their relations in a natural and clear manner to the users, providing them with tools that can be used to faithfully capture the desired domain semantics. A well-known example is the entity-relationship model [38].

Object-oriented models [39] appeared in the eighties, when most of the research was concerned with so-called "advanced systems for new types of applications [40]". These models are based on the object-oriented paradigm and their goal is to represent data as a collection of objects, which are organized into classes, and are assigned complex values.

Graph database models made their appearance alongside object-oriented models. These models attempt to overcome the limitations imposed by traditional models with respect to capturing the inherent graph structure of data occurring in applications such as hypertext or geographic information systems, where the interconnectivity of data is an important aspect. These models are studied in depth in Chapter 3.

Semistructured models [41] are designed to model data with a flexible structure (e.g. documents and Web pages). Semistructured data are neither raw nor strictly typed as in conventional database systems. These models appeared in the nineties.

Closely related to semistructured models is the *eXtensible Markup Language (XML)* [42], which did not originate in the database community. Although originally introduced as a document exchange standard, it soon became a general purpose model, focusing on information with tree-like structure.

For a global view of database models see the works of Silberschatz et al. [10], Navathe [43], Beeri [40] and Kerschberg et al. [35].

2.3 RDF from a Database Modeling Perspective

The term data model has been used in the database community with different meanings and in diverse contexts. Here we will use it in two senses. In a *broad or abstract sense*, a data model is a collection of conceptual tools for describing the real-world entities to be modeled in the database and the relationships among these entities [10]. In a *strict or concrete sense*, a data model, as defined by Codd [12], is as a combination of three components: (a) a collection of data structure types; (b) a collection of transformation operators and query language and; (c) a collection of general integrity rules.

In the broad sense, RDF can be considered a data model: a collection of conceptual tools for describing real-world entities, namely metadata on the Web. But also in the strict sense of the term, RDF qualifies as well: Point (a) has been more or less addressed at a basic level. One

Model	Level	Data Complex.	Connectivity	Type of Data
Network/Hierarchical	physical	simple	high	homogeneous
Relational	logical	simple	low	homogeneous
Semantic	user	simple/medium	high	homogeneous
Object-Oriented	logical/physical	complex	medium	heterogeneous
Semistructured	logical	medium	medium	heterogeneous
Graph	logical/user	medium	high	heterogeneous
RDF	logical	medium	high	heterogeneous

Table 2.1: Summary of the comparison of RDF with database models. The parameters are: abstraction level, complexity of the data items modeled, degree of connectivity among the data and support to get this information, and finally, flexibility to store different types of data.

of the documents of the RDF suite [4] speaks of *graph data model* meaning by this concept the data structure implicitly defined by sets of triples. Although one can discuss the precise meaning of this concept [44], the graph-like nature of RDF data is clear. Point (b) has been covered by many proposals of query languages for RDF. Nowadays, it has been consolidated in SPARQL [17], the RDF query language recommended by the W3C. Point (c), namely the integrity constraints, is an open issue for RDF, especially when considering the duality of RDF as an open world specification of distributed resources on the Web versus RDF as data model for large single-source repositories with all the issues of a standard database system. An initial step about integrity constraints in RDF is presented in the work of Lausen [45], who studies the problem of how to represent the original key and foreign key constraints when mapping relational databases into an RDF graph. The topic of constraints is outside the scope of this thesis.

Comparison of RDF with Database Models. Following we will compare RDF with the database models presented in Section 2.2. We restrict this study to the logical level, i.e., avoiding –when possible– physical implementation and indexing considerations. A summary of this comparison is presented in Table 2.1.

- *Physical models.* The data-structuring in this models is not flexible and not apt to model non-traditional applications. In contrast, RDF was designed to model information without making any assumption about the application domain. For our discussion they do not have much relevance.

- *Relational data model.* Although RDF data can be logically viewed as a set of binary relations, the differences with the relational model are manifold. Among the most relevant ones are: the relational model was directed to simple record-type data with a structure known in advance (airline reservations, accounting, etc.); the schema is fixed and

extensibility is a difficult task; integration of different schemas is not easy nor machine-manipulatable; the query language does not support paths, neighborhoods and queries that address transitivity; there are no objects identifiers, but values. Several applications use relational databases for RDF data storage [46].

- *Semantic data models.* For RDF database research, semantic models are relevant because they are based on a graph-like structure which highlights the relations between the entities to be modeled. However, semantic models are more expressive to represent relations because the use of high-level abstraction concepts such as classification and aggregation. Some of such abstractions can be represented in RDF by using RDF Schema terms (e.g. classification can be represented by using the term *subClassOf*).

- *Object oriented data models.* These models are related to RDF because the explicit or implicit graph structure in their definitions [47, 48, 49]. For example O_2 [50] defines basic, tuple-structured, and set-structured types (the first type is similar to an RDF blank node and the remainder two can be modeled as relations in RDF). Nevertheless, there remain important differences rooted in the form that each of them models the world. O-O models view the world as a set of objects having certain state (data) and interacting among them by methods. On the contrary, RDF models the world as a network of relations. The emphasis in RDF is on the interconnection of the data, the network of relations among the data and the properties of these relations. The emphasis of O-O is on the objects, their values and methods. However, there are proposals to store RDF data in an O-O database [51].

- *Semistructured data models.* In agreement to the RDF philosophy, semistructured models provide a natural graph representation: objects are nodes, and values are labeled arcs (see for example OEM [52]). The main differences with RDF are: the lightweight inferencing available, the existence of blank nodes, the stronger typing system, and the fact that labels are also nodes in RDF.

- *XML.* We found substantial differences between RDF and XML. First, RDF has a higher abstraction level; in fact RDF is an application of XML to represent metadata. Structurally XML has a ordered-tree-like structure as opposed to the graph structure of RDF. At the semantic level, in XML the information about the data is part of the data (in other words XML is self-describing); in contrast, RDF expresses explicitly the information about the data using relations between entities. An important advantage of RDF is its extensibility in both schema and instance level. See the works of Gil and Ratnakar [53] and Arroyo et al. [54] for a major comparison of these models.

- *Graph database models.* These are the closest to the RDF model regarding the data type used. However, in spite of the graph nature of the abstract triple syntax of RDF,

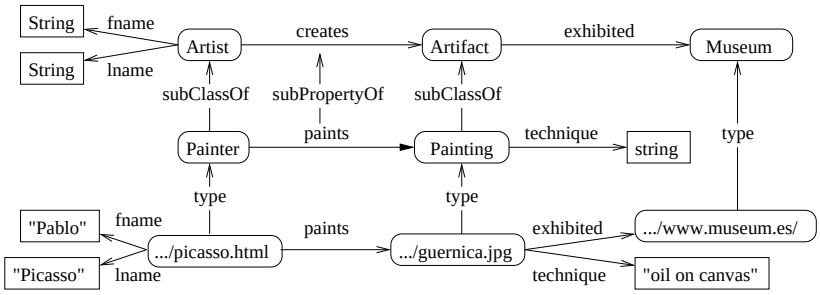


Figure 2.4: The subset of the Museum database [55] used for our study.

the RDF model presents features (some of which are not explicitly treated in the RDF specification) that, although appealing to the basic concept of graph, prevent the direct application of a graph database model. In this sense we have that:

- Formally an "RDF graph" is not a graph. In an RDF graph a resource may occur simultaneously as the predicate of one RDF triple and as the subject or object of another, a feature that breaks the classical notion of a graph. For example in Figure 2.4, the edges `creates` and `paints` are linked by predicate `subPropertyOf`.
- Technically an RDF graph is a directed hypergraph with ternary edges. Then RDF triples establish ternary relations which cannot be truly represented by the binary edges of classic graphs. A discussion of the graph-like nature of RDF is presented in the work of Hayes and Gutierrez [44], which proposes the application of bipartite graphs as an intermediate model between the abstract triple syntax and data structures used by applications.
- The schema of the data (i.e. the terms from the RDF Vocabulary) is itself part of the RDF graph. Therefore schema and data are mixed. For example, in Figure 2.4 predicates `subClassOf`, `subPropertyOf` and `type` are keywords of the RDFS Vocabulary that are used to define the schema. Additionally, note that edges labeled `type` disconnect the instance from the schema.
- RDF data can be extended with formal semantics [14] that adds explicit or implicit meaning to the data and schema. For example in Figure 2.4, the RDFS term `subClassOf`, which relates the nodes `Artist` and `Painter` let us infer that the resource `.../picasso.html` is of type `Artist` too.

From the above features and considering the differences with database models, we aim the development of a database model for RDF. In order to enhance the comparison with RDF, in the next chapter we will study in depth graph database models.

Chapter 3

Graph Database Models

Graph database models can be defined as those in which data structures for the schema and instances are modeled as graphs or generalizations of them, and data manipulation is expressed by graph-oriented operations and type constructors. In this chapter we survey the work on graph database models in order to provide a useful background that enables us to study RDF from a graph database perspective.

This chapter begins with a brief historical overview of the main proposals of graph database models. Then we conceptualize the notion of a graph database model by analyzing diverse definitions, motivations and applications. Finally, a comparative study of data structures, query languages, and integrity constraints for graph database models is presented.

3.1 Brief Historical Overview

Activity around graph databases flourished in the first half of the nineties and then the topic almost disappeared. The reasons for this decline are manifold: the database community moved toward semistructured data (a research topic which did not have links to the graph database work in the nineties); the emergence of XML captured all the attention of those working on hypertext and the Web (the tree-like structure was enough for most applications); people working on graph databases moved to particular applications like spatial data and biology. Figure 3.1 reflects the evolution of graph database models by means of papers published.

In an early approach, facing the failure of the systems of their time to take into account the semantics of the database, Roussopoulos and Mylopoulos [74] proposed a semantic network to store data about the database. An implicit structure of graphs for the data itself was presented in the Functional Data Model [78], whose goal was to provide a "conceptually natural" database interface. A different approach proposed the Logical Data Model (LDM) [71], where an explicit graph db-model intended to generalize the relational, hierarchical and network models. Later Kunii [65] proposed G-Base, a graph db-model for representing complex structures of knowledge.

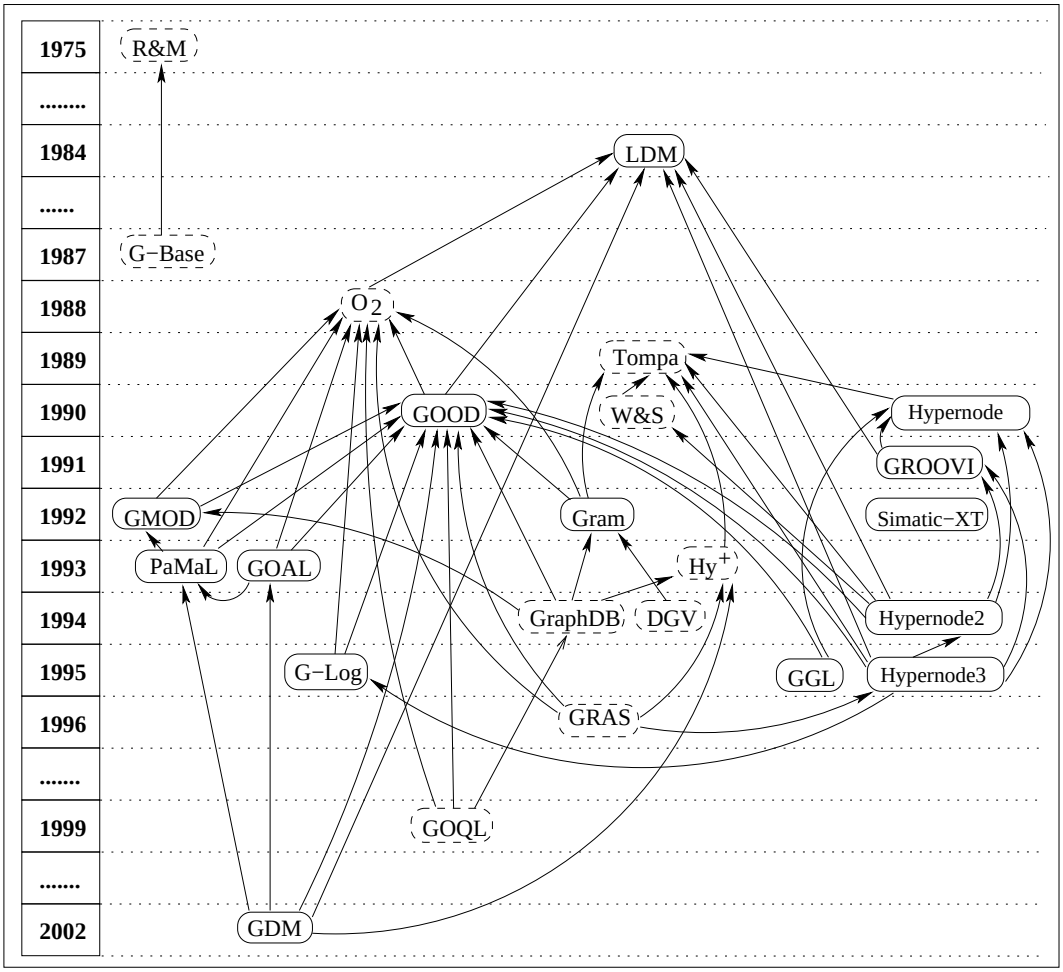


Figure 3.1: Graph database models development. Nodes indicate models and arrows citations. Dashed nodes represent related works in graph database models. DGV [56], GDM [57], GGL [58], GMOD [48], GOAL [59], GOOD [49, 60], GOQL [61], Gram [62], GRAS [63], GraphDB [64], GROOVY [47], G−Base [65], G−Log [66], Hypernode [67], Hypernode2 [68], Hypernode3 [69], Hy+ [70], LDM [71, 72], O_2 [50], PaMaL [73], R&M [74], Simatic-XT [75], Tompa [76], W&S [77].

In the late eighties, Lécluse et al. [50] introduced O_2, an object-oriented db-model based on a graph structure. Along the same lines, GOOD [49] is an influential graph-oriented object model, intended to be a theoretical basis for a system in which manipulation as well as representation are transparently graph-based.

Among the subsequent developments based on GOOD are: GMOD [48] that proposes a number of concepts for graph-oriented database user interfaces; Gram [62] which is an explicit graph db-model for hypertext data; PaMaL [73] which extends GOOD with explicit representation of tuples and sets; GOAL [59] that introduces the notion of association nodes; G-Log [66] which proposed a declarative query language for graphs; and GDM [57] that incorporates representation of n-ary symmetric relationships.

There were proposals that used generalizations of graphs with data modeling purposes. Levene and Poulovassilis [67] introduced a db-model based on nested graphs, called the Hypernode Model, on which subsequent work was developed [68, 69]. The same idea was used for modeling multi-scaled networks [75] and genome data [58]. GROOVY [47] is an object-oriented db-model which is formalized using hypergraphs. This generalization was used in other contexts: query and visualization in the Hy+ system [70]; modeling of data instances and access to them [77]; representation of user state and browsing [76].

There are several other proposals that deal with graph data models. Güting proposed GraphDB [64] intended for modeling and querying graphs in object-oriented databases and motivated by managing information in transport networks. Database Graph Views [56] proposed an abstraction mechanism to define and manipulate graphs stored in either relational object-oriented or file systems. The project GRAS [63] uses attributed graphs for modeling complex information from software engineering projects. The well known OEM [52] model aims at providing integrated access to heterogeneous information sources, focusing on information exchange.

Another important and recent line of development has to do with data representation models and the World Wide Web. Among them are data exchange models like XML [42], metadata representation models like RDF [4] and ontology representation models like OWL [79].

3.2 What is a Graph Database Model?

Although most papers on graph db-models use the term “graph data[base] model”, few of them define the notion explicitly. Nevertheless, their views on what a graph db-model is do not differ substantially. In some cases, an implicit definition is given and compared to other models that involve graphs, like the semantic, object-oriented and semi-structured models.

In what follows, we will conceptualize the notion of graph db-model with respect to three basic components, namely data structures, transformation language, and integrity constraints. Hence, a graph db-model is characterized as follows:

- *Data and/or the schema* are represented by graphs, or by data structures generalizing the notion of graph (hypergraphs or hypernodes). There is a wide consensus on this, except for slight variations. Let us review different authors' opinions on this issue:
  - One approach is to encode the whole database using graphs [64, 62].
  - Levene and Loizou [69] proposed a labeled directed graph as the single underlying data structure; the database consists of a single digraph.
  - According to Kuper and Vardi [71], a database schema for graph db-models is a directed graph, where leaves represent data and internal nodes represent connections between the data.
  - In another approach, database schemas, instances and rules are formalized using directed labeled graphs [66].
  - Kunii [65] describes databases using a schema graph, which is a labeled directed graph.
  - Another graph db-model formalized the representation of data structures using graphs, and stored these graphs in a database [58].
  - Gyssens et al. [49] treat object-oriented database schemas and instances as graphs. Nodes of the instance graph represent the database objects.
  - Hidders [57] describes database instances and schemas using certain types of labeled graphs.
  - Finally, Hidders and Paredaens [59] use labeled graphs to represent schemas and instances.

  An issue that concerns all graph db-models is the level of separation between schema and data (instances). In most cases, schema and instances can be clearly distinguished.

- *Data manipulation* is expressed by graph transformations [49], or by operations whose main primitives are on graph features like paths, neighborhoods, subgraphs, graph patterns, connectivity, and graph statistics (diameter, centrality, etc.). Some db-models define a flexible collection of type constructors and operations, which are used to create and access the graph data structures [58]. Another approach is to express all queries using a few powerful graph manipulation primitives [64]. The operators of the language can be based on pattern matching (i.e. finding all occurrences of a prototypical piece of an instance graph [59]).

- *Integrity constraints*, that enforce data consistency, can be grouped in schema-instance consistency, identity and referential integrity, and functional and inclusion dependencies. Examples of these are: labels with unique names [80], typing constraints on nodes [72], functional dependencies [47], domain and range of properties [4].

In summary, a graph db-model is a model 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 appropriate integrity constraints can be defined over the graph structure.

3.3 Why a Graph Database Model?

Graph db-models are applied in areas where information about data interconnectivity or topology is more important, or as important as the data itself. In these applications, the data and relations amongst the data are usually at the same level. In fact, introducing graphs as a modeling tool has several advantages for this type of data.

- *It allows for a more natural modeling of data.* Graph structures are visible to the user and they allow a natural way of handling applications data, for example hypertext or geographic data. 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 [66]. Graph objects (like paths and neighborhoods) may have first class citizenship; a user can define some part of the database explicitly as a graph structure [64], allowing encapsulation and context definition [67].

- *Queries can refer directly to this graph structure.* Associated with graphs are specific graph operations in the query language algebra, such as finding shortest paths, determining certain subgraphs, and so forth. Explicit graphs and graph operations allow users to express a query at a high level of abstraction. To some extent, this is the opposite of graph manipulation in deductive databases, where often, fairly complex rules need to be written [64]. It is not necessary to require full knowledge of the structure to express meaningful queries [81]. Finally, for the purposes of browsing it may be convenient to forget the schema [82].

- *Implementation-wise.* Graph databases may provide special graph storage structures, and efficient graph algorithms available for realizing specific operations [64].

3.4 Comparison with Other Database Models

In this section we compare graph db-models against the most influential database models. Table 3.1 presents a coarse-grained overview of this comparison. In the following text, we present the details of this comparison.

- *Physical db-models* were the first to successfully manage large collections of data. Among the most important ones are the hierarchical [33] and network [34] models. These models lack a good abstraction level, and it is difficult to separate the db-model from the

Database model	Abstraction level	Base data structure	Information focus
Network/Hierarchical	physical	pointers + records	records
Relational	logical	relations	data + attributes
Semantic	user	graph	schema + relations
Object-oriented	physical/logical	objects	object + methods
Semistructured	logical	tree	data + components
Graph	logical/user	graph	data + relations

Table 3.1: A coarse-grained comparison of the most influential database models. We are comparing the following aspects: abstraction level, base data structure used, and the types of information objects the db-model focuses on.

actual implementation. The data structures provided are not flexible, and not apt for modeling non-traditional applications. They permit database navigation at the record level by providing low-level operations which can be used to derive more abstract structures.

- The *Relational db-model* [36, 13] highlights the concept of abstraction levels by introducing a separation between the physical and logical levels. Gradually the focus shifted to modeling data as seen by applications and users [43]. This was the strength of the relational model, at a time when application domains managed relatively simple data (financial, commercial and administrative applications).

  The relational model was a landmark development because it gave the data modeling discipline a mathematical foundation. It is based on the simple notion of relation, which together with its associated algebra and logic, made the relational model a primary model for database research. In particular, its standard query and transformation language, SQL, became a paradigmatic language for querying.

  The differences between graph db-models and the relational db-model are manifold. For example, the relational model is geared towards simple record-type data, where the data structure is known in advance (airline reservations, accounting, inventories, etc.). The schema is fixed, which makes it difficult to extend these databases. It is not easy to integrate different schemas, nor this process is automatable. The query language cannot explore the underlying graph of relationships among the data, such as paths, neighborhoods and patterns.

- *Semantic db-models* [37] appeared as there was a need to incorporate a richer and more expressive set of semantics into the database, from a user's viewpoint. Database designers can represent objects and their relations in a natural and clear manner (similar to the way users view an application) by using high-level abstraction concepts such as aggre-

gation, classification and instantiation, sub- and super-classing, attribute inheritance and hierarchies [43].

In general, the extra semantics supports database design and evolution [83]. A well-known example is the entity-relationship model [38], which has become a basis for the early stages of database design. Other examples of semantic db-models are: IFO [84], a model which combines three basic constructs, namely Objects, Fragments and ISA relationships; and the Semantic Data Model (SDM) [85]. Semantic db-models are relevant to graph db-model research because the semantic db-models reason about the graph-like structure generated by the relationships between the modeled entities.

- *Object-oriented (O-O) db-models* [39] appeared in the eighties, when the database community realized that the relational model was inadequate for data intensive domains (Knowledge bases, engineering applications). This research was motivated by the appearance of non-conventional database applications, involving complex data objects and complex object interactions, such as CAD/CAM software, computer graphics and information retrieval.

  According to the O-O programming paradigm on which these models are based, they represent data as a collection of objects that are organized into classes, and have complex values and methods. Although O-O db-models permit much richer structures than the relational db-model, they still require that all data conform to a predefined schema [81].

  O-O db-models are similar to semantic models in that they provide mechanisms for constructing complex data by interrelating objects, and are fundamentally different in that they support forms of local behavior in a manner similar to object-oriented programming languages. For example, in O-O db-models identifiers are external to the objects and they remain invariant, whereas semantic models make up identifiers or keys based on internal attributes or internal properties; O-O support information-hiding and encapsulation [43].

  O-O db-models are related to graph db-models because of their explicit or implicit use of graph structures in definitions [47, 48, 49]. Nevertheless, there are important differences with respect to how each of them models the world. O-O db-models view the world as a set of complex objects having certain state (data), where interaction is via method invoking. On the other hand, graph db-models view the world as a network of relations, emphasizing data interconnection, and the properties of these relations. O-O db-models focus on object dynamics, their values and methods. Graph db-models focus instead on the interconnection, while maintaining the structural and semantic complexity of the data. Further comparison between O-O and graph db-models may be founded in the works of Beeri [40], Kerschberg et al. [35], Navathe [43] and Silberschatz et al. [10].

- *Semistructured db-models* [41, 86] were motivated by the increased existence of semi-structured data (also called unstructured data), data exchange, and data browsing [41]. In semistructured data, the structure is irregular, implicit and partial; the schema does not restrict the data, it only describes it, a feature that allows extensible data exchanges; the schema is large and constantly evolving; the data are self-describing, as it contains schema information [86].

  Among the most representative semistructured models are the Object Exchange Model (OEM) [52], UnQL [82], ACeDB [87], and Strudel [88]. Generally, semistructured data are represented using a tree-like structure. However, cycles between data nodes are possible, which leads to graph-like structures like in graph db-models. Some authors characterize semistructured data as rooted directed connected graphs [82].

3.5 Motivations and Applications

Graph db-models are motivated by real-life applications where component interconnectivity is a key feature. We will divide these application areas into classical applications and complex networks.

Classical Applications. Many applications played a part in motivating the need for graph databases:

- Generalizations of the classical db-models [71]. Classical models were criticized for their lack of semantics, the flatness of the permitted data structures, the difficulties the user has to "see" the data connectivity, and how difficult is to model complex objects [67].

- Applications where data complexity exceeded the capabilities of the relational db-model also motivated the introduction of graph databases. For instance, managing transport networks (train, plane, water, telecommunications) [89], spatially embedded networks like highways and public transport [64]. Several of these applications are now in the field of GIS and spatial databases.

- The limited expressive power of current query languages motivated the search for models that allowed better representation of more complex applications [66].

- Limitations (at the time) of knowledge representation systems [65], and the need for intricate but flexible knowledge representation and derivation techniques [66].

- After observing that graphs have been an integral part of the database design process in semantic and object-oriented db-models, the idea of having a model where both data manipulation and representation are graph-based emerged [49].

- The need for improving the functionality offered by object-oriented db-models [68]. For example, CASE, CAD, image processing and scientific data analysis software all fall into this category.
- Graphical and visual interfaces, geographical and multimedia systems [90, 70, 61].
- Software systems and integration [63].
- The appearance of on-line hypertext evidenced the need for other db-models, like the ones suggested by [76, 77, 62]. Together with hypertext, the Web created the need for a more apt model than classical ones for information exchange.

Complex Networks. Several areas have witnessed the emergence of huge data networks with interesting mathematical properties, called complex networks [91, 92, 93]. The need for database management for certain types of these networks has been recently highlighted [94, 95, 96, 80]. Although it is not yet evident whether we can group these databases into one category, we will present them in this manner.

As in Newman's survey [91], we will subdivide this category into four subcategories: social networks, information networks, technological networks and biological networks. We describe each subcategory via an example.

- In *social networks* [97], nodes are people or groups, while links show relationships or flows between nodes. Some examples are friendships, business relationships, sexual contact patterns, research networks (collaboration, co-authorship), terrorist networks [98], communication records (mail, telephone calls, email) and computer networks [99]. There is growing activity in the area of social network analysis [100], and also in visualization and data processing techniques for these networks.
- *Information networks* model relations representing information flow, such as citations between academic papers [101], World Wide Web (hypertext, hypermedia) [102, 103, 104], peer-to-peer networks [105], relations between word classes in a thesaurus and preference networks [106].
- In *technological networks*, the spatial and geographical aspects of the structure are dominant. Some examples are the Internet (as a computer network), electric power grids, airline routes, telephone networks and delivery networks (e.g. post offices). The area of *Geographic Information Systems (GIS)* is today covering a big part of this area (roads, railways, pedestrian traffic, rivers) [107, 108].
- *Biological networks* represent biological information whose volume, management and analysis has become an issue due to the automation of the process of data gathering.

A good example is the area of *Genomics*, where networks occur in gene regulation, metabolic pathways, chemical structure, map order and homology relationships between species [109].

There are other kinds of biological networks, such as food webs, neural networks, etc. The reader can consult database proposals for genomics [80, 109, 110], an overview of models for biochemical pathways [111], a tutorial on graph data management for biology [94], and a model for chemistry [112].

It is important to stress that classical query languages offer little help when dealing with the type of queries needed in the above areas. For example:

- Data processing in GIS includes geometric operations (area or boundary, intersection, inclusions, etc), topological operations (connectedness, paths, neighbors, etc) and metric operations (distance between entities, diameter of the network, etc).
- In genetic regulatory networks, examples of measures are connected components (interactions between proteins), nearest neighbor degrees (strong pair correlations).
- In social networks, some measures are: distance, neighborhoods, clustering coefficient of a vertex, clustering coefficient of a network, betweenness, size of giant connected components, size distribution of finite connected components [93].
- Similar problems occur in the Semantic Web, where RDF queries would benefit from being able to reason about graph aspects [113].

3.6 Comparison Among Graph Database Models

Comparison among db-models are typically done by either using a set of common features [114, 33, 37] or by defining a general model used as a comparison basis [83]. The evaluation presented in this chapter follows a conceptual organization of modeling features by considering three general components in the database modeling process: (a) Basic foundations of the model and the data structures available at schema and instance level; (b) Approaches to enforce data consistency (integrity constraints); and (c) Languages for querying and manipulating the database. The study emphasizes conceptual modeling issues rather than implementation aspects. The table in Figure 3.2 summarizes the set of features and other information about graph db-models. A complete description of each graph db-model is presented in Appendix A.

3.6.1 Data Structures

The representation of entities and relations are identified as being fundamental to graph db-models. An entity or object represents something that exists as a single and complete unit. A

Characteristics | Database Model		LDM	Hypernode	GOOD	GROOVY	GMOD	Simatic-XT	Gram	Pamal	GOAL	Hypernode2	Hypernode3	GGL	GDM
		1984	1990	1990	1991	1992	1992	1992	1993	1993	1994	1995	1995	2002
Basic Foundation	Graph model	√		√		√		√	√	√			√	√
	Hypergraphs				√									
	Hypernodes		√				√				√	√	√	
	Object Oriented model	√		√	√	√			√	√				√
Digraph	Node labeled	√	√	√	√	√	√	√	√	√	√	√	√	√
	Edge labeled			√	√	√	√	√	√	√			√	√
Support:	Schema	√		√	√	√		√	√	√	√	√		√
	Complex objects	√	√		√		√		√	√	√	√	√	√
	Higher-order relations		√		√					±	√	√	√	√
	Tuples	√							√					√
	Sets	√							√					
	N-ary relations									√				√
	Grouping		√		√		√				√	√	√	
	Derivation and Inheritance				√				√	√	±			√
	Nested relations		√		√		√				√	√	√	
Query Language	Algebraic - Procedural	√					√	√				√		
	Logic - Declarative	√	√		√						√		√	
	Graphical			√		√			√	√				√
	Query	√	√		√		√	√	√		√	√	√	
	Transformation	√		√		√				√				√
	Path queries		±	±			√	√	±	±	±	±	±	
Integrity Constraints	Schema-Instance Consistency	√		√	√	√		√	√	√	√	√		√
	Identity and Referential Integrity		√	±	√	±		√	√	±	√	√	√	±
	Functional Dependencies				√							√		
Implementation			√	√			√	√			√	√	√	√
Motivation		Complex objects	Complex objects	Graphical Interfaces	General	Hypermedia	Transport networks	Hypertext	Graphical Interfaces	Graphical Interfaces	Complex objects	Hypertext	Genomics	Complex objects

Figure 3.2: Main proposals on Graph Database Models and their characteristics ("√" indicates support and "±" partial support). LDM [71, 72], Hypernode [67], GOOD [49, 60], GROOVI [47], GMOD [48], Simatic-XT [75], Gram [62], PaMaL [73], GOAL [59], Hypernode2 [68], Hypernode3 [69], GGL [58], GDM [57]. A working description of each model is given in Appendix A.

relation is a property or predicate that establishes a connection between two or more entities. In this section we analyze the data structures used for modeling entities and relations in graph db-models.

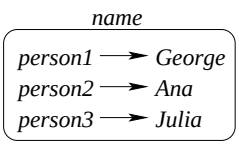
One of the most distinctive characteristic of a graph db-model is a framework for the representation of connectivity among entities, distinct from attributes (relational model), standard abstractions (semantic models), complex objects (O-O models), or composition relations (semistructured models).

All graph db-models have as their formal foundation variations on the basic mathematical definition of a graph, for example: directed or undirected graphs; labeled or unlabeled edges and nodes; hypergraphs; and hypernodes. On top of this basic layer, models present diverse features influenced by the semantic or object-oriented approaches. For example, the data representation of GOOD, GMOD, G-Log and Gram is simply a digraph with labeled nodes and labeled edges between them. The Hypernode Model, Simatic-XT and GGL emphasize the use of hypernodes for representing nested complex objects. GROOVY is centered in the use of hypergraphs as a formalism for modeling complex objects, sub-object sharing, integrity constraints and structural inheritance.

Representing the database as a simple flat graph (with many interconnected nodes) has the drawback that, in practice, it is difficult to present the information to the user in a clear way. In contrast, a hypernode database consists of a set of nested graphs, which provides inherent support for data abstraction and the ability to represent each real-world object as a separate database entity [68]. Additionally, the use of hypernodes and hypergraphs allows a concept to grow from a simple undefined concept (primitive object) to one defined by multiple complex relations (complex object). In contrast, a pure-graph approach (where nodes and edges are different structures) does not provide such expressiveness and extensibility.

Note that hypergraphs can be modeled by hypernodes by (i) encapsulating the contents of each undirected hyperedge within a further hypernode and (ii) replacing each directed hyperedge by two hypernodes related by a labeled edge. In contrast, multi-level nesting provided by hypernodes cannot be easily captured by hypergraphs.

Most models have explicit labels on edges, except LDM, GROOVY and the Hypernode model. LDM has an order among the edges which induces implicit labeling. Although the Hypernode model and GROOVY do not use labeled edges, the task of representing relations (and its names) can be attained by encapsulating edges which represent the same relation (same label edges), within one hypernode (or hyperedge) labeled with the relation-name. For example, we can represent the set of labeled arcs $person1 \xrightarrow{name} George$, $person2 \xrightarrow{name} Ana$ and $person3 \xrightarrow{name} Julia$, by the hypernode:



Next, we will study in more detail the representation of entities and relations.

Representation of entities

Models represent entities at both, instance and schema level. Specifically, entity types, relation types and schema restrictions are related to the definition of the database schema, and instances of entities and relations conform the database instances.

Several models (e.g. GOOD, GMOD, G-Log and Gram) represent both schema and instance as a labeled digraph. An exception is LDM, whose schemas are digraphs where leaves represent data and internal nodes represent structured data. LDM instances consist of two-column tables, each of which associates entities of a particular type (primitive, tuple or set).

A *schema* graph defines: *entity types* represented as nodes labeled with zero (GMOD) or one type name; *primitive entities* represented as nodes labeled with basic types; and *relations* represented as edges labeled with relation-names. Relations are only defined for entity types (primitive entities have no properties) and each primitive entity has an associated domain of constants.

An *instance* graph contains: *concrete entities* represented as nodes labeled by either an entity type name or an object identifier; *primitive values* represented as nodes labeled with a value from the domain of a primitive entity; and *relations* represented as edges labeled with the corresponding relation-name according to the schema.

The above approach was extended in other models by including nodes for explicit representation of tuples and sets (PaMaL, GDM) and n-ary relations (GOAL, GDM). These types of nodes allow the definition of complex structures. Tuple and set nodes of PaMaL are unlabeled, allowing to define (instantiate) more than one entity class (respectively concrete entity) using the same tuple and set node, providing data reduction. Association nodes of GOAL provide simple definition of multi-attribute and multi-valued n-ary relations. Composite-value entities in GDM are used for representing both tuples and n-ary relations.

The basic structure of a graph (nodes and edges) is complemented with the use of hypernodes (Hypernode model, Simatic-XT and GGL) and hypergraphs (GROOVY), extensions that provide support for nested structures. A novel feature of GROOVY is the use of hypergraphs for defining value functional dependencies. The hypernode model is characterized by using nested graphs at the schema and instance level. A database consists of a set of hypernodes defining types and their respective instances. GGL introduces, in addition to its support for hypernodes (called Master-nodes), the notion of *Master-edge* for encapsulation of paths.

Although hypernodes are used in several models, they are used in different forms. Simatic-XT and GGL use hypernodes as an abstraction mechanism consisting in packaging other graphs as an encapsulated vertex, whereas the Hypernode model additionally uses hypernodes to represent other abstractions, for example complex objects, relations, etc.

A common characteristic of models based on simple and based on extended graph structures is the support for defining non-traditional data types, feature procured by the definition of complex objects. In this sense, LDM, PaMaL, GOAL and GDM enable the representation of complex objects by defining special constructs (tuple, set or association nodes). Hypernodes and Hypergraphs are flexible data structures that support the representation of arbitrarily complex objects and present inherent ability to encapsulate information.

Representation of Relations

Roughly speaking, two types of relations occurring in graph db-models can be distinguished: Simple relations, which connect two entities under a simple semantics (e.g. attributes), and are easily represented in graphs as edge labels; and complex relations, which conform networks of relations (e.g. hierarchical) with additional semantics (e.g. composition), and whose representation depends of the data structures provided by each model.

In what follows, we will discuss specific types of relations supported by graph db-models.

- *Attributes*. The relation represents a property (mono-valued or multi-valued) directly linked to an entity. Most graph db-models represent attributes by using labeled edges directly related to nodes.

  LDM and PaMaL define tuple nodes to describe a set of attributes that are (re)used to define an entity. In the case of the Hypernode model and GROOVY, attributes are triples $< node, edge, node >$ inside hypernodes or hyperedges which represent complex objects. The (unlabeled) edge establishes the relation between the attribute-name (first node) and the attribute-value (second node). GOOD and GOAL define edges (called functional and non-functional) to distinguish between mono-valued and multi-valued attributes.

- *Entities*. If the relation of two or more objects conceptually describes a distinct model object, such relation is considered an entity. This approach implies support for higher-order relations (relations between relations). Most models do not support this feature because relations are represented as simple labeled edges.

  A partial support is presented in GOAL, where association nodes may have properties. The Hypernode model and GROOVY have inherent support because property-names can be complex objects represented as hypernodes or hyperedges. GGL supports this feature by labeling edges with types.

- *Neighborhood relations*. Models with a basic graph structure provide simple support and visualization of neighborhood relations, although special structures used by some of these models (e.g. tuple-nodes in PaMaL) can obscure the simplicity of representation. In the case of hypergraph and hypernode base models, neighborhood relations

are translated into nested relation. A particular modeling structure is the Master-edge of Simatic-XT, which is used for representing *Path relations*.

- *Standard abstractions*. The most frequently used are aggregation (is-part-of relation) and its opposite, composition (is-composed-by relation), association (n-ary relations) and grouping or sets. Some models provide explicit representation of sets (LDM,PaMaL), tuples (LDM,PaMaL,GDM) and n-ary relations (GOAL, GDM). Models based on hypernodes and hypergraphs support grouping by joining entities within a hypernode or a hyperedge.

- *Derivation and inheritance*. These abstractions are represented at the schema level by relations of subclass and superclass (ISA) and at the instance level by relations of instantiation (is-of-type). The notion of inheritance (only structural-attribute inheritance) is supported using these relations, allowing entity classes with overlapping structure to share their semantic content.

  ISA relations are explicitly supported in PaMaL, GOAL, and GDM by considering ISA-labeled edges as class-hierarchy links. However, we should note that this type of relation can be implicitly considered as an attribute relation. The Hypernode and GROOVY models show how structural inheritance is supported naturally by nested-graph structures. The representation of object-class schemas by means of hypergraphs leads to a natural formalization of the notion of object sharing and structural inheritance.

- *Nested relations*. They are recursively specified relations defined by nested objects. This feature is naturally supported by using hypernode or hypergraph structures.

3.6.2 Integrity Constraints

Integrity constraints are general statements and rules, which define the set of consistent database states or changes of state or both [12]. Integrity constraints have been studied for the relational [115, 116], semantic [117, 118], object-oriented [119], and semistructured [120, 121] db-models. Thalheim [118] presented a unifying framework for integrity constraints.

In the case of graph db-models, examples of integrity constraints include schema-instance consistency, identity and referential integrity constraints, functional and inclusion dependencies. Next we study each of them.

Schema-instance consistency

Entity types and type checking constitute a powerful data-modeling and integrity checking tool, since they allow database schemas to be represented and enforced. Some graph db-models do not define a schema, for example the first version of Hypernode, initial versions of GGL and Simatic-XT.

The approach applied in most models (GOOD, GMOD, PaMaL, GOAL, G-log, GDM and Gram) to confront the issue of schema-instance consistency follow in general two guidelines: (i) the instance should contain only concrete entities and relations from entity types and relations that were defined in the schema; (ii) an entity in the instance may only have those relations or properties defined for its entity type (or for a super-type in the case of inheritance). Then, all node and edge labels occurring in the instance must occur in the scheme too, but the opposite is not required in some models (GOOD, GMOD, GOAL and G-Log). In this way, incomplete or non-existing information can be incorporated in the database. PaMaL, GDM and Gram do not support incomplete information because they establish an instantiation function between schema and instance.

LDM defines a logic (similar to relational tuple calculus) used to specify integrity constraints on LDM schemas and to define views. Integrity constraints are LDM formulas, which enforce that instance objects satisfy certain conditions (satisfaction of LDM formulas). That is, given a database and a sentence in the logic, one can test effectively whether the sentence is true in the database or not.

GROOVY introduces the notion of object-class schemas over which objects are defined. An object schema defines valid objects (value-schemas), value functional dependencies, and valid shared values between objects (sub-object schemas). These restrictions are formalized using a hypergraph representation. Indeed, there is a one-to-one correspondence between each object schema and a hypergraph, where objects are vertices, value functional dependencies are directed hyperedges, and sub-object schemas are undirected hyperedges.

Additionally, conflicts of inheritance are discussed in some models. The general approach is the notion of consistent schema to prevent problems. GOAL establishes that objects only belong to the class they are labeled with and their super-classes. This implies that an object can only belong to two classes at once if these classes have a common subclass. PaMaL does not provide a conflict resolution mechanism, since the authors consider that this has to be part of the implementation.

Poulovassilis and Levene [68] showed that testing a hypernode repository for well-typedness can be performed in polynomial time with respect to the size of the repository. GDM presents a systematic study of the complexity of well-typedness checking: deciding well-typedness of a pattern with no ISA edges under a schema graph with no implicit object class nodes is in PTIME; deciding well-typedness of a pattern under a schema graph with no implied object class nodes is co-NP complete; deciding well-typedness of an addition/deletion under a schema graph with no implied object class nodes is in PSPACE; deciding well-typedness of an addition under a schema graph with no implied object class nodes and no composite-value classes is in PTIME.

Checking consistency is also related to restructuring, querying and updating the database. For example, deletion of entity types or relations in the schema implies the deletion of concrete

entities in the instance. Given an arbitrary GOOD program (i.e. a sequence of GOOD operations), statically checking the consistency of an edge addition in the program is undecidable in general. In GMOD the notion of schema-instance consistency is referenced to consistency of schema transformations in object-oriented databases [122]. In PaMal an addition or deletion operation specifies a valid graph transformation between two instance graphs (schema transformations are not included).

Schema-instance separation. Another aspect to consider is the degree to which schema and instance are different objects in the database (an issue discussed largely in semistructured db-models). Representation of (un)structured entities means that the data (do not) respect a previously defined data type. In this sense, a data model can be classified as structured or non-structured depending on whether it allows or does not allow the definition of a schema to restrict the database instance to well-typed data. In most models there is a separation between the database schema and the database instance. An exception is the hypernode model [69], where the lack of typing constraints on hypernodes has the advantage that changes to the database can be dynamic.

Redundancy of data. Because an instance graph can contain many instances of entities (for example non-printable nodes in GOOD, set/tuple nodes in PaMaL, association nodes in GOAL or composite-value nodes in GDM), the database presents redundant information. One possible solution is to introduce an operation which is used to group entities on the basis of some of their relations. This abstraction creates a unique entity for each equivalence class of duplicate entities; as such, it acts as a duplicate eliminator. The correspondent operations are Abstraction (GOOD), Reduce (PaMaL) and Reduction (GDM). This operation is informally mentioned in GOAL as a merging of nodes representing the same value.

Object identity and referential integrity

Set-based data models such as the relational model and the nested relational model are value-based; that is, tuples in relations (respectively nested relations) are identified by the values of their attributes. On the other hand, in O-O db-models object identity is independent of attribute values and is achieved by equipping each database object with a unique identifier, for example a label. Graph db-models implement the two types of identification. The use of identifiers has several advantages: arbitrarily complex objects can be identified and referred; objects can share common sub-objects thus making possible the construction of general networks; query and update operation can be simplified.

In some models, concrete entities like instance nodes (PaMaL, Gram), edges (GGL), hypernodes (Hypernode based models), hyperedges (GROOVY) are labeled with entity identifiers, whereas in models like GOOD, GMOD, GOAL, G-Log and GDM each entity is identified by its attributes and it exists independently of their properties (it is considered as object identity).

The Hypernode Model [68] defines two integrity constraints: *Entity Integrity* makes sure that each hypernode is a unique real world entity identified by their content; *Referential Integrity* requires that only existing entities be referenced. Similarly, GGL [58] establishes that: labels in a graph are uniquely named; edges are composed of the labels and vertices of the graph in which the edge occurs. These constraints are similar to primary key and foreign key (referential) integrity constraints in the relational db-model.

Functional dependencies

In the Hypernode model [69] the notion of semantic constraints was considered. The concept of *Hypernode functional dependency*, denoted by $A \rightarrow B$, where A and B are sets of attributes, let us express that the value of A determines the value of B in all hypernodes of the database.

GROOVY [47] uses *directed hyperedges* in order to represent *Value Functional Dependencies (VFDs)*. They are used in the value schema level to establish semantic integrity constraints. A VFD asserts that the object value restricted to a set of attributes uniquely determines the object value restricted to a further attribute.

Finally, let us remark that the notion of integrity constraint in graph db-models is concentrated in the creation of consistent instances and the correct identification and reference of entities. The notion of functional dependency is a feature taken from the relational model that cannot be presented easily in all graph db-models.

3.6.3 Query and Manipulation Languages

A query language is a collection of operators or inferencing rules which can be applied to any valid instances of the data structure types of the model, with the objective of manipulating and querying data in those structures in any combination desired [12].

A great deal of papers discuss the problems concerning the definition of a query language for a db-model [123, 83, 124, 125, 86, 126]. Also a variety of query languages and formal frameworks for studying them have been proposed and developed, including the relational db-model [127], semantic databases [128, 129], object-oriented databases [130], semistructured data [82, 86, 81] and the Web [126, 102].

Among graph db-models, there is substantial work focused in query languages, the problem of querying graphs, the visual presentation of results, and graphical query languages. Due to the volume of the research done, this specific area by itself deserves a thorough survey. Considering the scope of this chapter, in this section we limit ourselves to describe the most important graph query languages to serve as reference resources.

The Logical Database Model [71, 72] presents a logic language very much in the spirit of relational tuple calculus, which uses fixed sort variables and atomic formulas to represent queries over a schema using the power of full first order languages. The result of a query

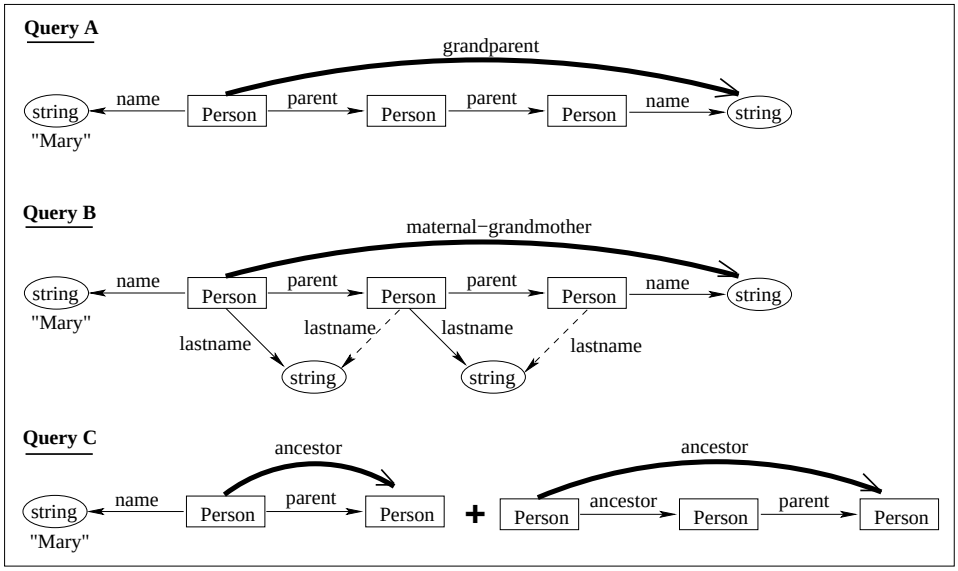


Figure 3.3: Queries in G-Log. *Query A* asks for the names of Mary's grandparents (fixed path query). *Query B* asks for the name of the maternal grandmother of Mary (tree-like query). *Query C* calculates Mary's Ancestors (transitive closure).

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.

The query languages G, G+ y GraphLog integrate a family of related graphical languages defined over a general simple graph model.

- The graphical query language G [131] is based on regular expressions that allow simple formulation of recursive queries. A *graphical query* in G is a set of labeled directed multigraphs where nodes may be either variables or constants, and edges can be labeled with regular expressions. The result of a query is the union of all query graphs which match subgraphs from the instance.

- G evolved into a more powerful language called G+ [132] where 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 for and a summary graph that represents how to restructure the answer obtained by the query graph.

- GraphLog [133] is a query language for hypertext that extends G+ by adding negation and unifying the concept of a query graph. A query is now only one graph pattern containing one distinguished edge, which corresponds to the restructured edge of the summary graph in G+. The effect of the query is to find all instances of the pattern that occur in the database graph and for each one of them to define a virtual link represented by the distinguished edge. GraphLog includes an implicit transitive closure operator,

which replaces the usual recursion mechanism. The algorithms used in the GraphLog implementation are discussed in the work of Mendelzon and Wood [134].

The proposal G-Log [66] includes a declarative language for complex objects with identity. It uses the logical notion of *rule satisfaction* to evaluate queries which are expressed as G-Log programs. *G-Log programs* are sets of graph-based rules, which specify how the schema and instance of the database will change. G-Log is a graph-based, declarative, nondeterministic, and computationally complete query language that does not suffer from the copy-elimination problem. G-Log is a good example of graphical query language (see Figure 3.3).

In the context of graph-oriented object models, there are query languages that regard database transformations as *graph transformations* (which can be interpreted as database queries and updates). They are based on *graph-pattern matching* and allow the user to specify node insertions and deletions in a graphical way. GOOD [49] presented a graph-based language that is shown to be able to express all constructive database transformations. This language was followed by the proposals GMOD [48], PaMaL [73], GOAL [59], and GUL [57]. Additionally, GOAL includes the notion of *fixpoints* in order to handle the recursion derived from a finite list of additions and deletions. PaMaL proposed the inclusion of *Loop, Procedure and Programs constructs*, and PaMaL and GUL presented an operator that reduces instance graphs by deleting repeated data. We should note that graph-oriented manipulation formalisms based on patterns allow a syntax-directed way of working much more natural than text-based interfaces.

GROOVY [47] introduces a Hypergraph Manipulation Language (HML) for querying and updating labeled hypergraphs. It defines two basic operators for querying hypergraphs by identifier or by value, and eight operators for manipulation (addition and deletion) of hypergraphs and hyperedges.

Watters and Shepherd [77] presented a framework for general data access based in hypergraphs that include operators for creation of edges and set operators like intersection, union and difference. In a different context, Tompa [76] introduced basic operations over hypergraph structures representing user state and views in page-oriented hypertext data.

The literature also include proposals of query languages that deal with *hypernodes*:

- Levene and Poulovassilis [67] defined a logic-based query and update language, where queries are expressed as sets of hypernode rules (h-rules) that are called *hypernode programs*. The query language defines an operator which infers new hypernodes from the instance using the set of rules in a hypernode program.

- This query language was extended by Hyperlog [68, 135] including deletions as well as insertions, and discussing in more detail the implementation issues. A full Turing-machine capability is obtained by adding composition, conditional constructs and iteration. Hyperlog is computationally and update complete, although the evaluation of Hyperlog programs is intractable in the general case.

- In a procedural style, HNQL [69] defines a set of operators for declarative querying and updating of hypernodes. It also includes assignment, sequential composition, conditional (for making inferences), for loop, and while loop constructs.

Glide [136] is a graph query language where queries are expressed using a linear notation formed by labels and wild-cards (regular expressions). Glide uses a method called GraphGrep based on subgraph matching to answer the queries.

Oriented to search the Web, Flesca and Greco [137] proposed the use of partially ordered languages to define path queries to search databases and present results on their computational complexity. A query language based on the previous ideas was proposed later [138].

Cardelli et al. [139] introduced a spatial logic for reasoning about graphs and defined a query language based in pattern matching and recursion. This Graph Logic combines first-order logic with additional structural connectives. A query asks for a substitution of variables such that a *satisfaction relation* determines which graph satisfy which formulae. The query language is based on *queries* that build new graphs from old and *transducers* that relate input graphs with output graphs.

In the area of Geographic information Systems, the Simatic-XT model [140] defines a query language. It includes basic operators that deal with encapsulated data (nesting of hypernodes), set operators (union, concatenation, selection and difference) and high level operators (paths, inclusion and intersections).

WEB [141, 142] is a declarative programming language based on a graph logic and oriented to querying genome data. WEB programs define graph templates for creating, manipulating and querying objects and relations in the database. These operations are answered by matching graphs in valid instances.

Models like Gram [62] and GOQL [61] propose SQL-Style query languages with explicit path expressions. *Gram* presents a query algebra where 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 a hyperwalk algebra with operations closed under the set of hyperwalks.

Models like DGV [56] and GraphDB [64] define special operators for functional definition and querying of graphs. For example, a query in GraphDB consists of several steps, each of which computes operations that specify argument subgraphs in the form of regular expressions over edges that extend or restrict dynamically the database graph. GraphDB includes a class of objects called path class, which are used to represent several paths in the database.

One of the most fundamental graph problems in graph query languages is to compute reachability of information, which reduced essentially to path problems characterized and expressed by recursive queries. For example, path queries are relevant in GraphLog, Gram,

Simatic-XT, DGV, GOQL, Flesca and Grego, Cardelli et al., and in less degree treated in Hypernode, GOAL, Hyperlog, GraphDB, G-Log, and HNQL. The notion of shortest path is considered in Flesca and Greco, G+, GraphLog, and DGV. Path and other relevant graph queries for RDF will be discussed in Section 4.2.

The importance and computational complexity of path-based queries is studied in several works [143, 144, 145, 16]. Finding simple paths with desired properties in direct graphs is difficult, and essentially every nontrivial property gives rise to an NP-complete problem [146]. Yannakakis [147] surveyed a set of path problems relevant to the database area including computing transitive closures, recursive queries and the complexity of path searching. Mannino and Shapiro [148] presented a survey of extensions to database query languages that solve graph traversal problems.

3.7 Conclusions

In this chapter we have surveyed the research in the area of graph database models. The main contribution of this chapter is the integration of several concepts and definitions used in graph database modeling. Specific contributions are:

- The conceptualization of the notion of a graph database model by proposing a definition which encompasses implicit and informal notions used in the literature.
- A comparison of graph database models with other database models, highlighting their importance as an area with its own motivations, applications and characteristic problems.
- The identification of a set of typical graph features provided by graph database models and a comparative study of existing proposals based on these characteristics. We concentrate in presenting the main aspects of modeling, that are data structures, query languages and integrity constraints.
- The presentation of a uniform description for most relevant graph database models.

In the context of this thesis, this chapter presents a useful background for studying RDF from a graph database perspective.

We would like to warn the reader that there are several related topics that fall out of the scope of the work in this chapter. Some of these topics are: graph visualization; graph data structures and algorithms for secondary memory; and in general, graph database system implementation.

On a side note, there are other important database models as well as modeling frameworks that concern graph modeling, which are not covered here. Some related database models are: temporal database models [149, 150], spatial database models [151, 152], multidimensional database models [153], and database models for geographical information systems

(GIS) [107, 154]. Frameworks related to our topic, but not directly focusing in database issues are: semantic networks [155, 156], conceptual graphs [157, 158], knowledge representation systems [159], topic maps [160, 161], Hypertext [162], and OWL [79] (the recent family of models for representing ontologies on the Web).

Several interesting lines of work work can be expected in the area. We can mention the application of graph database models in the development of emergent domains that need graph database support. Regarding the Semantic Web topics, the survey shows that there is potential for useful results in the area of RDF databases. In what follows, we will concentrate on RDF query languages an their capabilities for querying graph properties.

Chapter 4

Querying RDF Data from a Graph Database Perspective

One of the main features of RDF is its ability to interconnect information resources, resulting in a graph-like structure for which connectivity is a central notion [16]. As was showed in the above chapter, the data structure of the model influences the set of operations that should be provided by the query language, hence the support for graph operations in RDF query languages becomes a fundamental requirement.

The objective of this chapter is to show that basic concepts of graph theory such as degree, path and diameter play an important role for applications that involve querying RDF data. We motivate our study by presenting some examples of real-life queries involving graph notions in RDF data. Then a brief review of RDF query languages and their support for graph properties is presented. The same graph properties are then summarized for a subset of the graph query languages presented in Chapter 3. The results for both sets of query languages are then compared and discussed. Finally, we propose a set of basic features that we consider desirable to be included in the design of RDF query languages.

4.1 Introduction

Several languages for querying RDF data have been proposed, most of them on the lines of classical database query languages like SQL and OQL, and some following the ideas of XPath [163] for XML. There are several surveys about them [164, 165, 19]. Nowadays, SPARQL [17] is the RDF query language recommended by the W3C.

There are also works addressing foundational issues. For instance see the works of Horrocks and Tessaris [166], Karvounarakis et al. [55], Gutierrez et al. [18]. Nevertheless, the discussion about RDF as a full fledged strict database model and the design of a query language for such a model is probably one of the most needed features if we want to take advantage

of all the potential of the RDF data model (e.g. query optimization, query rewriting, views, update, etc.).

One of the motivations of this chapter, is to study those application domains where interconnection at large scale and navigation of a network is the main modeling theme. Examples of this are social networks [5], on-line communities [31], terrorist networks [98], bibliographic databases [101], etc.

To give a flavor of the type of queries in RDF, consider questions like: "are suspects A and B related?", submitted to a police database, or "what is the Erdős number of author X?", submitted to a bibliographic data (e.g. an RDF version of DBLP). The first asks for "relevant" paths connecting these resources in the (RDF) police database, and the second asks simply for the length of the shortest path between the nodes representing Erdős and X.

Current proposals of RDF query languages do not support[1] queries like the ones mentioned above. To address these type of problems, the notions and techniques of *graph query languages* (See Section 3.6.3) can be very valuable.

There are several comparative studies [164, 165, 168, 169, 170] of RDF query languages, considering features such as their expressiveness, robustness, and syntax. One of them [19] devotes a small section to graph properties of these languages, but considers only path expressions of fixed length.

4.2 Queries Involving Graph Notions

In this section we present a set of motivation query examples concerning graph properties. It is based on applications over real-life RDF data for which graph querying is relevant.

EX 1. (*Bibliographical information*) An RDF knowledge base containing bibliographic information on scientific publications (such as Citeseer [171] and DBLP [172]) would give rise to a number of interesting queries, such as: "what is the relation between scientists A and B?", which amounts to the computation of paths between resources of a RDF graph, possibly restricted to properties (edges) such as *cites* and *isCoauthor*; "What is the *influence* of article D?", which requires the computation of the transitive closure of the *isReferencedBy* (the inverse of *cites*) relation from the root node D.

EX 2. (*Lexical information*) Wordnet [6] is an on-line lexical reference system whose data are available in the RDF format [106]. Words are organized into synonym sets, which are ordered by the hyponym (roughly: sub-concept) relation. One common use of these data is to find meaning-related clusters of words, such as the query for the *coordinate terms* for a word (i.e. the "sister words", all immediate sub-concepts of the super-concept of a word)—see figure 4.1. This actually corresponds to a path (`tree`, `hyponymOf`, `X`), (`Y`, `hyponymOf`, `X`), what

[1] A language is said to *support* a feature if it provides facilities that make it convenient (reasonable easy, safe and efficient) to use that feature [167].

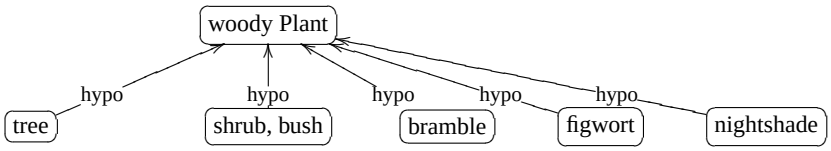


Figure 4.1: Example of data in WordNet: coordinate terms of “tree”.

could be easily done in SPARQL [17]. However, for the arbitrary length of such a pattern (*k* hyponyms up, *k* down again) it or any other current RDF query language is not sufficient. Similarly, a simple query such as “are the terms *professor* and *master* related?” could not be answered.

EX 3. (*Social Networks*) FOAF (an acronym of Friend of a Friend) defines a vocabulary for describing people, their activities and their relations to other people and objects. FOAF allows groups of people to describe social networks without the need for a centralized database. Some interesting queries on this data are: “the friends of a person” (i.e. its neighborhood); “the acquaintance of two people” (i.e. the paths between them); “the most popular people” (i.e. resources with a higher connection degree).

EX 4. (*Biology*) Excellent examples for the need for complex queries in large datasets come from biology. These include bio-pathways data, protein interaction networks, taxonomies and phylogenetic trees, chemical structure graphs, food webs, laboratory protocols, genetic maps, and multiple sequence alignments. Queries in such a data context often include various types of path queries where regular expressions, shortest paths, and matching of subgraphs play a central role [94].

More examples can be found in W3C Working Draft: “RDF Data Access Use Cases and Requirements [173]”. This document presents uses cases that characterize some of the most important and most common motivations behind the development of existing RDF query languages.

4.3 Graph Properties Support in RDF Query Languages

Several languages for querying RDF data have been proposed and implemented, some in the lines of traditional database query languages (e.g. SQL, OQL), others based on logic and rule languages. Among them: *RQL* [55] is a typed language for querying RDF repositories; *RDQL* [174] is an implementation of *SquishQL* [175], a SQL-style query language that permits simple graph navigation in RDF sources; *RDFQL* [176] is a statement-based query language with a SQL-style to perform queries, inference operations, and construction of views on RDF structured data; *TRIPLE* [177] is a language that allows rule definition, inference and transformation of RDF models; *Notation 3* (N3) [27] provides a text-based syntax for RDF; *Versa* [178]

PROPERTY	SPARQL	RQL	SeRQL	RDQL	Triple	N3	Versa	RxPath
Adjacent nodes	$\surd$	$\pm$	$\pm$	$\pm$	$\pm$	$\pm$	$\pm$	$\times$
Adjacent edges	$\surd$	$\pm$	$\pm$	$\pm$	$\pm$	$\times$	$\times$	$\times$
Degree of a node	$\pm$	$\pm$	$\times$	$\times$	$\times$	$\times$	$\times$	$\times$
Path	$\times$	$\times$	$\times$	$\times$	$\times$	$\times$	$\times$	$\pm$
Fixed-length path	$\pm$	$\pm$	$\pm$	$\pm$	$\pm$	$\pm$	$\times$	$\pm$
Distance between two nodes	$\times$	$\times$	$\times$	$\times$	$\times$	$\times$	$\times$	$\times$
Diameter	$\times$	$\times$	$\times$	$\times$	$\times$	$\times$	$\times$	$\times$

Table 4.1: Support of some RDF query languages for graph properties ("$\surd$" indicates support, "$\pm$" indicates partial support and "$\times$" no support).

is a graph-based language with some support for rules; *SeRQL* [179] combines characteristics of languages like *RQL*, *RDQL*, *TRIPLE*, *N3* plus some new features; *RxPath* [29] is a query language that allows one to address parts of an RDF graph by using XPath [163] expressions. Good surveys are the works of Haase et al. [19] and Magkanaraki et al. [164].

W3C members that conform the RDF Data Access Working Group (DAWG) produced *SPARQL* [17], a query language with a SQL-like style, where a simple query is based on query patterns, and query processing consists of binding of variables to generate pattern solutions.

In order to illustrate the problems for querying graph-like properties in RDF data, we compare eight RDF query languages with respect to their support for seven graph properties one would like to retrieve (as can be seen in the following list of query examples). As RDF dataset to query against we chose the well-known museum example showed in Figure 2.4. Table 4.1 gives the results of our comparison.

1. *Adjacent nodes.* Two nodes are called adjacent if they share a common edge.

   Query: "All resources adjacent to the resource `Guernica`".

   Expected result: `Painting`, `www.museum.es`, `"oil on canvas"` and `picasso.html` . Not all languages support this feature. The problem is that this query can only be expressed as a union of two queries: one for outgoing edges from `Guernica` (e.g.`type`), another for ingoing edges (e.g. `paints`). Some languages do not support the union operator.

2. *Adjacent edges.* The adjacent edges of a node is the set of outgoing edges plus the incoming edges of the node.

   Graph query: "All predicates of statements involving `Guernica`".

   Expected result: `type`, `exhibited`, `technique` and `paints`. The problems faced are similar to the previous case. Note that here we probably would like to differentiate schema predicates from data predicates.

3. *Degree of a node.* The degree of a node is the number of edges incident to the node.

   Graph query: "Number of predicates involving `Guernica`"

   Expected result: 4. Same problems as above plus the fact that most languages do not support aggregation at this level. SeRQL for example returns the number, but not as part of the answer.

4. *Path.* A path in a graph is a sequence of nodes such that from each of its nodes there is an edge to the next node in the sequence.

   Query: "Find paths between `picasso.html` and `www.museum.es`"

   Expected results: There are several, for example

   $$\texttt{picasso.html} \xrightarrow{\texttt{paints}} \texttt{guernica.jpg} \xrightarrow{\texttt{exhibited}} \texttt{www.museum.es}$$

   and

   $$\texttt{picasso.html} \xrightarrow{\texttt{type}} \texttt{Painter} \xrightarrow{\texttt{paints}} \texttt{Painting} \xrightarrow{\texttt{subClassOf}} \texttt{Artifact} \xrightarrow{\texttt{exhibited}} \texttt{Museum} \xleftarrow{\texttt{type}} \texttt{www.museum.es}.$$

   None of the languages studied support arbitrary paths like the ones needed for this case. Note that it must be considered whether paths via the schema are regarded as relevant.

5. *Fixed-length path:* The length of a path is the number of edges in the path.

   Query: "Find paths of length 2 between `picasso.html` and `www.museum.es`"

   Expected results: There is only one, the former exhibited above. This notion is partially supported by RDF query languages.

6. *Distance between two nodes.* That is the length of the shortest path between the nodes.

   Query: "How far is `picasso.html` from `www.museum.es`?"

   Expected result: 2. Not supported by any language.

7. *Diameter of a graph.* That is the distance between the two node which are furthest from each other

   Query: "Diameter of the museum graph"

   Expected result: 5. Not supported. It is based on distance and paths.

Summary. An RDF graph can be considered as a directed graph. This view produces problems in languages that do not have a union operator when retrieving neighborhoods, e.g. "all statements involving a given resource". Only SPARQL supports querying of adjacent nodes and edges. In this context, some query results violate the query language property of closure [19] by returning results which are not in the RDF format.

PROPERTY	G	G+	GraphLog	Gram	GraphDB	Lorel	F-G
Adjacent nodes	$\pm$	$\surd$	$\surd$	$\surd$	$\pm$	$\surd$	$\pm$
Adjacent edges	$\pm$	$\surd$	$\surd$	$\surd$	$\pm$	$\surd$	$\pm$
Degree of a node	$\times$	$\surd$	$\surd$	$\times$	?	$\times$	$\times$
Path	$\surd$	$\surd$	$\surd$	$\surd$	$\surd$	$\surd$	$\surd$
Fixed-length path	$\surd$	$\surd$	$\surd$	$\surd$	$\surd$	$\surd$	$\surd$
Distance between two nodes	$\times$	$\surd$	$\surd$	$\times$	?	$\times$	$\times$
Diameter	$\times$	$\surd$	$\surd$	$\times$	?	$\times$	$\times$

Table 4.2: Support of some graph database query languages for the example graph properties of Table 4.1 ("$\surd$" indicates support, "$\pm$" partial support, "$\times$" no support, and "?" indicates there is no information available).

There are two main problems concerning paths: (a) most languages support only querying for patterns of paths which are limited in length and form (the issue of edge direction blows up the size of the query exponentially); (b) only one language, RxPath, is able to retrieve paths starting from a fixed node and with some other restrictions.

Aggregate functions like COUNT, MIN, MAX applied to paths could be used to answer queries as for the degree of a node, the distance between nodes, and the diameter of a graph. None of these functions is systematically supported, even though some of them are listed as issues by the RDF DAWG [173].

4.4 Graph Primitives for RDF Query Languages

In this section we propose a set of graph-theoretical notions to be supported by RDF query languages. We stress the graph-like features that in our opinion are missing in them.

Before discussing the primitives in detail, let us enumerate desirables features for an RDF query language. They are very much inspired by a similar wish-list stated by Abiteboul [86] for semi-structured data. They are: Standard database-style query primitives; Navigation in the style of semi-structured data or Web-style browsing; Searching for patterns in an information-retrieval style [180]; Temporal queries [181], including versioning; Querying both the data and the schema in the same query; Incorporating transparently the lightweight inferencing of RDF Schema and relevant polynomial-time extensions [182]; and Sound theoretical foundation.

Additionally, Table 4.2 shows the support that selected graph query languages give to the properties in Table 4.1. These graph query languages constitute a good reference for studying the inclusion of graph properties in the design of RDF query languages.

The following groups of graph primitives comprise features of graph query languages (see Section 3.6.3), real-life queries (see Section 4.2) and those found in the DAWG use cases [173]. They reflect in a fair manner the problems needed for querying RDF data.

Neighborhoods. The neighborhood of a node *n* in a graph *G* is given by all nodes adjacent to *n* and all edges connecting two such nodes. Hence the primary notion is *adjacency* (both node and edge adjacency). The notion of neighborhood is relevant for RDF in various contexts (e.g. use cases 2.2, 2.7, 2.9 and 2.15 in the DAWG Draft).

> Use case 2.7: Exploring the Neighborhood (Tourism))
> *... José attends a conference in Washington, DC, at the new convention center, and he stays in a hotel nearby. José wants to find out the latitude, longitude, name, and type of everything within one mile of the convention center, as well as all events occurring during his stay, ...*

A more advanced notion of adjacency, like the *k-neighborhood* of a node, is necessary in several contexts. In the RDF context, inference of new triples is relevant in node and edge adjacency queries. The need of 1-neighborhood retrieval in an RDF Graph is argued in the works of Sayers [183] and Guha et al. [184].

To the best of our knowledge, the notion of neighborhood as a primitive for query languages has not been studied systematically in the database literature.

Paths and connectedness. One of the most fundamental graph problems is to compute reachability information. In fact, many of the recursive queries that arise in relational databases and, more generally, in data with graph structure, are in practice graph traversals characterized by path problems (e.g. use cases 2.5 and 2.17 in the DAWG Draft).

> Use case 2.5: Avoiding Traffic Jams (Transportation)
> *Niel has to drive every day from home to his office ... Using his cell phone, Niel requests that his car query public RDF storage servers on the Web for a description of current Atlanta road construction projects, traffic jams, and roads affected by inclement weather. ...Niel uses the mapping program in his cell phone to plan a different route to work...*

The importance of path queries is studied in several works [143, 144, 145, 16]. One of the challenges to incorporate such notion into a query language is its computational complexity. Finding simple paths with desired properties in direct graphs is very difficult, and essentially every nontrivial property gives rise to an NP-complete problem [146].

Yannakakis [147] surveyed a set of path problems relevant to the database area including computing transitive closures, recursive queries, and the complexity of path searching. Extensions to database query languages to solve graph traversal problems are surveyed in the work of Mannino and Shapiro [148].

In what follows, we describe the support that the graph query languages described in Section 3.6.3 give to path problems.

A initial implementation of G translates the graphical queries into C-Prolog programs. Simple paths are traversed using certain non-Horn clause constructs available in Prolog. Although it does not support cycles or finding the shortest path, it is a good approximation to a graph query language.

The evaluation of path queries in G+ is a two-stage process consisting of a depth-first search of the graph database, and the use of a nondeterministic finite state automaton to control the search. In addition, path queries are a subset of the class of linear chain queries and hence can be evaluated rapidly in parallel. The evaluation algorithm can be shown to compute the identity query in O(e) time and the transitive closure in O(ne) time (assuming that the graph database has n nodes and e edges). G+ was implemented in the HyperG system providing primitive operators like depth-first search, shortest path, transitive closure and connected components.

Motivated by the implementation of G+, Mendelzon and Wood [134] studied the problem of finding all pairs of nodes connected by a simple path such that the concatenation of the labels along the path satisfies a regular expression. Although the regular simple path problem is in general NP-complete, the paper presents an algorithm that runs in polynomial time in the size of the graph when some conditions are fulfilled: the graph is acyclic, the regular expression is restricted (according to the definition in the paper), or the graph complies with a cycle constraint compatible with the regular expression. The evaluation algorithm uses a deterministic finite automaton to traverse paths in the graph. They also prove the intractability of certain types of simple paths in a particular class of direct graphs and characterize a class of queries about regular simple paths which can be evaluated in polynomial time. The analysis and implementation presented in the paper assumes that the graph can be entirely stored in main memory.

The expressive power of GraphLog is characterized by establishing the equivalence between GraphLog, stratified linear Datalog (a language of function-free Horn clauses), nondeterministic logarithmic space, and transitive closure. The queries expressible in the language are exactly those that can be computed in logarithmic space in the size of the database.

To implement graph operations in GraphDB, efficient graph algorithms are used. Shortest path and cycle were both implemented using an algorithm called A*. Moreover, nodes, paths and subgraphs are indexed using path classes and index structures like B-Tree and LSD-Tree.

Lorel presents a SQL-style query language that supports two types of path expressions: simple path expressions, which allow one to obtain the set of objects reachable by following a sequence of labels starting from a named object in the OEM graph; and a more powerful syntax for path expressions, called general path expressions, based on wildcards and regular expressions. To improve query execution, the Lore DBMS [185] implements the query lan-

guage *Lore* and uses two kinds of indexes, a link (edge) index called Lindex, and a value index called Vindex. A *Lindex* takes an object identifier and a label, and returns the object identifiers of all parents via the specified label. A *Vindex* takes a label, operator, and a value, and returns all atomic objects having an incoming edge with the specific label and a value satisfying the specific operator and value. Vindexes and Lindexes are implemented using B+ trees and linear hashing, respectively.

In graph databases, where the number of nodes is very large (e.g. the Web), is useful to subdivide the domain of evaluation by selecting subsets of the domain on the basis of some criteria. With this objective, Flesca and Greco [137] introduced partially ordered regular languages based on some order on the nodes. Such languages are an extension of regular languages where strings are partially ordered, for example, two strings s_1 and s_2, such that $s_1 > s_2$, denote two paths in the graph with the constraint that the path s_1 should be preferred to the path s_2. In a later work [138], they present an algebra for partially ordered relations, an algorithm for the computation of path queries and show that computing an instance of a graph query can be done in polynomial time. Also, they present a SQL-like language that considers general paths and extended regular expressions, and show how extended regular expressions can be used to search the Web.

With similar motivations, but in the context of RDF, Anyanwu and Sheth [186] introduced a path operator ρ to address relevant relationships between entities called semantic associations. Semantic associations are represented in a RDF graph as sequences (i.e. edges, paths) between entities or more complex structures of sequences, and a notion of similarity between them is defined. The implementation of the ρ-operator is evaluated on two strategies, first implementing a processing layer on existing RDF data storage technologies, and second the use of a memory resident graph representation of the RDF model along with the use of efficient graph traversal algorithms (e.g. transitive closure and isomorphism of paths).

Pattern matching. This consists in determining if there exists a mapping (or isomorphism) between a graph pattern and a subgraph of a database graph (e.g. use cases 2.1, 2.9, 2.10, 2.12, 2.13 and 2.15 in the DAWG Draft).

> Use case 2.12: Browsing Patient Records (Health Care)
> *Peter is developing a medical knowledge base using OWL/RDF in collaboration with medical domain experts... Peter enters a few parameters and issues a query... It returns a large number of matching results...*

Pattern matching deals with two problems: the graph isomorphism problem that has unknown computational complexity, and the subgraph isomorphism problem which is known to be NP-complete. Pattern matching has attracted a great deal of attention especially on data mining (see the work of Washio and Motoda [187] for a survey), update [73, 59], querying [131,

132, 133] and visualization [48]. Shasha et al. [146] presented a survey of pattern-matching based algorithms for fast searching in trees and graphs.

PaMal use graph patterns to describe the part of the database instance that are affected by an operation (addition and deletion of nodes and edges). In the case of GraphDB, the subgraph problem is solved by moving the conditions into subsequent graph operations or other database access.

Aggregate functions. These are operations non related to the data model. They permit one to summarize (e.g. COUNT) or operate (e.g. SUM) on the query results (e.g. use cases 2.3, 2.4, 2.6, 2.8, 2.11, 2.14, 2.16 and 2.19 in the DAWG Draft).

> Use case 2.19: Building a Table of Contents (Publishing)
> *Leigh, a programmer for a large publishing house, uses RDF to store data and metadata about books and journals. Leigh uses an RDF query language to retrieve the first three articles associated with an issue and sort them by page number...*

Such functions are oriented to deal directly with the structure of the underlying graph, such as the degree of a node, the diameter of the graph (or a set of nodes), the distance between nodes, etc.

With the purpose of performing computations on retrieved subgraphs product of a query operation, G+ defines two types of summary operators: path operators which summarize on the values of the attributes along paths, and set operators which summarize on the values of the attributes on a set of paths. The set of such operators include sum, products, maximum and count.

GraphLog becomes more expressive that relational algebra and calculus with aggregates, adding aggregate operators (e.g. MAX, SUM, etc.) and path summarization. The implementation of GraphLog use algorithms discussed in the work of Mendelzon and Wood [134].

Gram, consistent with its SQL-like syntax, defines two types of algebraic operations: unary (projection, selection, renaming) and binary (join, concatenation, set operations) which are closed under the set of hyperwalks.

PaMal provides a reduce-operation to work with a special group of instances called reduced instances and programming constructs (loop, procedure and program). Finally, the query language of GraphDB supports further operations for sorting, grouping, and aggregate functions (e.g. SUM).

4.5 Conclusions.

In this chapter we have evaluated RDF query languages regarding their graph query capabilities. Particularly we found that they give little or no support for querying graph properties.

The main contribution of this chapter is a list of desirable graph primitives that should ideally be supported by RDF query languages. The specific contributions of this chapter are:

- A review of RDF query languages and a summary of their support for querying graph properties;
- A comparison between RDF query languages and graph query languages, with respect to their capabilities to support graph queries;
- A suggested set of graph-theoretical notions that should be supported, directly or indirectly, when RDF data are queried.

From this study, it follows the need to research and experiment in the design of RDF query languages and their support for graph properties, taking advantage of the experience on graph query languages developed by the database community.

Chapter 5

The Expressive Power of SPARQL

SPARQL [17] became in 2008 the query language recommended for querying RDF data. In the above chapter we have studied SPARQL with respect to a set of desirable graph properties. In this chapter we study in depth the expressive power of SPARQL, i.e., we determine the set of all queries expressible in this language, and thus clarify completely its graph query capabilities.

First we study some issues of the SPARQL specification (e.g. the simulation of negation). Then we show that the operational(W3C) [17] and a compositional [188] semantics of SPARQL are equivalent. In order to determine the expressive power of SPARQL, we present transformations from/to non-recursive safe Datalog with negation. Finally, we present the implications of the results.

5.1 Introduction

By the *expressive power* of a query language, we understand the set of all queries expressible in that language [189, 190]. The W3C recommendation SPARQL is from January 2008. Hence, it is no surprise that little work has been done in the formal study of its expressive power. Several conjectures were raised during the WG sessions [1]. Furche et al. [191] surveyed expressive features of query languages for RDF (including old versions of SPARQL) in order to compare them systematically. But there is no particular analysis of the expressive power of SPARQL.

In this chapter we study in depth the expressive power of SPARQL. A first issue addressed is the incorporation of negation. The W3C specification of SPARQL provides explicit operators for join and union of graph patterns, even for specifying optional graph patterns, but it does not define explicitly the difference of graph patterns. Although intuitively it can be emulated via a combination of optional patterns and filter conditions (like negation as failure in logic programming), we show that there are several non-trivial issues to be addressed if one likes to define the difference of patterns inside the language.

[1] See `http://lists.w3.org/Archives/Public/public-rdf-dawg-comments/`, especially the years 2006 and 2007.

A second expressiveness issue refers to graph patterns with non-safe filter, i.e., graph patterns $(P \text{ FILTER} C)$ for which there are variables in the filter constraint C not present in the graph pattern P. It turns out that these type of patterns, which have non-desirable properties, can be simulated by safe ones (i.e. patterns where every variable occurring in C also occurs in P). This simple result has important consequences for defining a clean semantics, in particular a compositional and context-free one.

A third topic of concern was the presence of non desirable features in the W3C semantics like its operational character. We show that the W3C specification of the semantics of SPARQL is equivalent to a well behaved and studied compositional semantics for SPARQL [188], which we will denote in this thesis $\text{SPARQL}_{\text{COMP}}$.

Using the above results, we are able to determine the expressive power of SPARQL. We will prove that $\text{SPARQL}_{\text{COMP}}$ and non-recursive safe Datalog with negation (nr-Datalog$^\neg$) are equivalent in their expressive power. For this, first we show that $\text{SPARQL}_{\text{COMP}}$ is contained in nr-Datalog$^\neg$ by defining transformations (for databases, queries, and solutions) from $\text{SPARQL}_{\text{COMP}}$ to nr-Datalog$^\neg$, and we prove that the result of evaluating a $\text{SPARQL}_{\text{COMP}}$ query is equivalent, via the transformations, to the result of evaluating (in nr-Datalog$^\neg$) the transformed query. Second, we show that nr-Datalog$^\neg$ is contained in $\text{SPARQL}_{\text{COMP}}$ using a similar approach. It is important to remark that the transformations used are explicit and simple, and in all steps bag semantics is considered.

Finally, and by far, the most important result of this chapter is the proof that SPARQL has the same expressive power of Relational Algebra under bag semantics (which is the one of SPARQL). This follows from the well known fact that Relational Algebra has the same expressive power as nr-Datalog$^\neg$ [189].

Related Work. Cyganiak [192] presented a translation of SPARQL into Relational Algebra considering only a core fragment of SPARQL. His work is extremely useful to implement and optimize SPARQL in SQL engines. At the level of analysis of expressive issues it presented a list of problems that should be solved (many of which still persist), like the filter scope problem and the nested optional problem.

Polleres [193] proved the inclusion of the fragment of SPARQL patterns with safe filters into Datalog by giving a precise and correct set of rules. Schenk [194] proposed a formal semantics for SPARQL based on Datalog, but concentrated on complexity more than expressiveness issues. Both works do not consider bag semantics of SPARQL in their translations.

The work of Pérez et al. [188], that gave the formal basis for $\text{SPARQL}_{\text{COMP}}$ compositional semantics, addressed several expressiveness issues, but no systematic study of the expressive power of SPARQL was done.

5.2 Preliminaries

Preliminary concepts introduced in this chapter are the RDF model, RDF datasets, Datalog and the notion of expressive power.

5.2.1 The RDF Model

Assume there are pairwise disjoint infinite sets **U**, **B**, **L** such that, **U** is the set of *RDF URI references*, **L** is the set of *RDF literals*, and **B** is the set of *Blank nodes*. We denote by **T** the union $\mathbf{U} \cup \mathbf{B} \cup \mathbf{L}$, and each element in **T** is called a *term*. A *name* is an element in the set $\mathbf{U} \cup \mathbf{L}$. A set of names is referred to as a *vocabulary*.

A tuple $(v_1, v_2, v_3) \in (\mathbf{U} \cup \mathbf{B}) \times \mathbf{U} \times \mathbf{T}$ is called an *RDF triple*, where v_1 is the *subject*, v_2 the *predicate*[2], and v_3 the *object*. An *RDF Graph* [4] is a set of RDF triples.

Let G be an RDF graph. We denote by $\text{term}(G)$ the set of elements of **T** occurring in the triples of G, that is the *universe* of G. The *vocabulary* of G is the set $\text{term}(G) \cap (\mathbf{U} \cup \mathbf{L})$. The set of blank nodes in G is denoted $\text{blank}(G)$. We say that G is *ground* if $\text{blank}(G)$ is empty.

Let G_1 and G_2 be RDF graphs. We say that G_1 is a *subgraph* of G_2, denoted $G_1 \subseteq G_2$ if the set of triples of G_1 is a subset of the triples of G_2. A *proper subgraph* is a proper subset of the triples in the graph. We denote $G_1 \subset G_2$ when $G_1 \subseteq G_2$ and $G_1 \neq G_2$. The *union* of G_1 and G_2, denoted $G_1 \cup G_2$, is the set theoretical union of their sets of triples. The *merge* of G_1 and G_2, denoted $G_1 + G_2$, is the graph $G_1 \cup G_2'$, where G_2' is the RDF graph obtained from G_2 by consistently renaming its blank nodes to avoid clashes with those in G_1.

5.2.2 RDF Datasets

Assume that **I** is the set of IRI References. An *RDF dataset* D is a set $\{G_0, \langle u_1, G_1\rangle, \ldots, \langle u_n, G_n\rangle\}$ where each G_i is a graph and each u_j is an IRI. G_0 is called the *default graph* of D and each pair $\langle u_j, G_j\rangle$ is called a *named graph* [195]. Every dataset satisfies that: (i) it always contains one default graph (which could be empty); (ii) there may be no named graphs; (iii) $u_i \neq u_j$ for each i, j; and without loss of generality we assume - unless stated otherwise - that (iv) $\text{blank}(G_i) \cap \text{blank}(G_j) = \emptyset$ for $i \neq j$.

Given the dataset D, we denote by $\text{term}(D)$ the set of terms occurring in the graphs of D. The default graph of D is denoted $\text{dg}(D)$. For a named graph $\langle u_i, G_i\rangle$ in D, define $\text{name}(G_i)_D = u_i$ and $\text{gr}(u_i)_D = G_i$. The set of IRIs $\{u_1, \ldots, u_n\}$ is denoted $\text{names}(D)$. Finally, the *active graph* of D is a distinguished graph G_i (used for querying the dataset).

[2]The predicate is also known as the *property* of the triple.

5.2.3 Datalog

We will briefly review notions of Datalog (For further details see the books of Abiteboul et al. [189], Levene and Loizou [190]).

A *term* is either a variable or a constant. An *atom* is either a *predicate formula* $p(x_1,...,x_n)$ where p is a predicate name and each x_i is a term, or an *equality formula* $t_1 = t_2$ where t_1 and t_2 are terms. A *literal* is either an atom (a *positive literal* L) or the negation of an atom (a *negative literal* $\neg L$).

A Datalog *rule* is an expression of the form $L \leftarrow L_1,\ldots,L_n$ where L is a positive literal called the *head*[3] of the rule and $L_1,\ldots,L_n$ is a set of literals called the *body*. A rule is *ground* if it does not have any variables. A ground rule with empty body is called a *fact*.

A *Datalog program* Π is a finite set of Datalog rules. The set of facts occurring in Π, denoted facts(Π), is called the *initial database* of Π. A predicate is *extensional* in Π if it occurs only in facts(Π), otherwise it is called *intensional*.

A variable x occurs positively in a rule r if and only if x occurs in a positive literal L in the body of r such that: (1) L is a predicate formula; (2) if L is $x = c$ then c is a constant; (3) if L is $x = y$ or $y = x$ then y is a variable occurring positively in r. A Datalog rule r is said to be *safe* if all the variables occurring in the literals of r (including the head of r) occur positively in r. A Datalog program Π is *safe* if all the rules of Π are safe. The safety restriction provides a syntactic restriction of programs which enforces the finiteness of derived predicates.

The *dependency graph* of a Datalog program Π is a digraph (N,E) where the set of nodes N is the set of predicates that occur in the literals of Π, and there is an arc (p_1,p_2) in E if there is a rule in Π whose body contains predicate p_1 and whose head contains predicate p_2. A Datalog program is said to be *recursive* if its dependency graph is cyclic, otherwise it is said to be *non-recursive*.

Hence, a Datalog program is *non-recursive* and *safe* if it does not contain any predicate that is recursive in the program and it can only generate a finite number of answers. In what follows, we only consider non-recursive and safe Datalog programs.

A *substitution* θ is a set of assignments $\{x_1/t_1,\ldots,x_n/t_n\}$ where each x_i is a variable and each t_i is a term. Given a rule r, we denote by $\theta(r)$ the rule resulting of substituting the variable x_i for the term t_i in each literal of r. A substitution is *ground* if every term t_i is a constant.

A rule r in a Datalog program Π is true with respect to a ground substitution θ, if for each literal L in the body of r one of the following conditions is satisfied:
(i) $\theta(L) \in$ facts(Π); (ii) $\theta(L)$ is an equality $t = t$ where t is a constant;
(iii) $\theta(L)$ is a literal of the form $\neg p(c_1,...,c_n)$ and $p(c_1,...,c_n) \notin$ facts(Π);
(iv) $\theta(L)$ is a literal of the form $\neg(c_1 = c_2)$ and c_1 and c_2 are distinct constants.

[3] We may assume that all heads of rules have only variables by adding the corresponding equality formula to its body.

The *meaning* of a Datalog program Π, denoted facts$^*(\Pi)$, is the database resulting from adding to the initial database of Π as many new facts of the form $\theta(L)$ as possible, where θ is a substitution that makes a rule r in Π true and L is the head of r. Then the rules are applied repeatedly and new facts are added to the database until this iteration stabilizes, i.e., until a *fixpoint* is reached.

A *Datalog query* Q is a pair (Π, L) where Π is a Datalog program and L is a positive (goal) literal. The *answer* to Q over database $D = \text{facts}(\Pi)$, denoted $\text{ans}_d(Q, D)$ is defined as the set of substitutions $\{\theta \mid \theta(L) \in \text{facts}^*(\Pi)\}$.

5.2.4 Expressive Power of Query Languages

The *expressive power* of a query language means the set of all queries expressible in that language [189, 190]. In order to determine the expressive power of a query language L, usually one chooses a well-studied query language L' and compares L and L' in their expressive power. Two query languages have the same expressive power if they express exactly the same set of queries.

A given query language is defined as a quadruple $(\mathcal{Q}, \mathcal{D}, \mathcal{S}, \text{eval})$, where $\mathcal{Q}$ is a set of queries, $\mathcal{D}$ is a set of databases, $\mathcal{S}$ is a set of solutions, and $\text{eval} : \mathcal{Q} \times \mathcal{D} \to \mathcal{S}$ is the evaluation function. The evaluation of a query $Q \in \mathcal{Q}$ on a database $D \in \mathcal{D}$ is denoted $\text{eval}(Q, D)$ (usually $\text{eval}(Q, D)$ is simply denoted $Q(D)$ if no confusion arises). Two queries $Q_1, Q_2 \in \mathcal{Q}$ are *equivalent*, denoted $Q_1 \equiv Q_2$, if $\text{eval}(Q_1, D) = \text{eval}(Q_2, D)$ for every $D \in \mathcal{D}$, i.e., they return the same answer for all input databases.

Let $L_1 = (\mathcal{Q}_1, \mathcal{D}_1, \mathcal{S}_1, \text{eval}_1)$ and $L_2 = (\mathcal{Q}_2, \mathcal{D}_2, \mathcal{S}_2, \text{eval}_2)$ be two query languages. We say that L_1 is *contained* in L_2 if and only if there are bijective data transformations $\mathcal{T}_D : \mathcal{D}_1 \to \mathcal{D}_2$ and $\mathcal{T}_S : \mathcal{S}_1 \to \mathcal{S}_2$, and query transformation $\mathcal{T}_Q : \mathcal{Q}_1 \to \mathcal{Q}_2$, such that for all $Q \in \mathcal{Q}_1$ and $D \in \mathcal{D}_1$ it satisfies that $\mathcal{T}_S(\text{eval}_1(Q, D)) = \text{eval}_2(\mathcal{T}_Q(Q), \mathcal{T}_D(D))$. We say that L_1 and L_2 are *equivalent* if and only if L_1 is contained in L_2 and L_2 is contained in L_1. (Note that, if L_1 and L_2 are subsets of a language L, then $\mathcal{T}_D$, $\mathcal{T}_S$ and $\mathcal{T}_Q$ are the identity.)

5.3 SPARQL

In this section we formally present the SPARQL query language. First we introduce general concepts of its syntax and semantics. Then two versions of SPARQL are presented: $\text{SPARQL}_{\text{COMP}}$, the version with compositional semantics [188]; and $\text{SPARQL}_{\text{OP}}$, the W3C version with operational semantics [17].

5.3.1 Syntax and Semantics of SPARQL (General Concepts)

A SPARQL query is syntactically represented by a block consisting of a *query form* (SELECT, CONSTRUCT or DESCRIBE), zero o more *dataset clauses* (FROM and FROM NAMED), a WHERE *clause*, and possibly *solution modifiers* (e.g. DISTINCT). The WHERE clause provides a *graph pattern* to match against the RDF dataset constructed from the dataset clauses.

There are two formalizations of SPARQL which will be used throughout this chapter: $\text{SPARQL}_{\text{OP}}$, the W3C recommendation language SPARQL [17] and $\text{SPARQL}_{\text{COMP}}$, the formalization of SPARQL given in the work of Pérez et al. [188]. We will need some general definitions before describe briefly both languages.

Assume that $\mathbf{I}$ is the set of IRI References. We extend the set of terms $\mathbf{T}$ to the union $\mathbf{I} \cup \mathbf{B} \cup \mathbf{L}$. Additionally, assume the existence of an infinite set $\mathbf{V}$ of variables disjoint from $\mathbf{T}$. We denote by $\text{var}(\alpha)$ the set of variables occurring in the structure α.

A tuple from $(\mathbf{I} \cup \mathbf{L} \cup \mathbf{V}) \times (\mathbf{I} \cup \mathbf{L} \cup \mathbf{V}) \times (\mathbf{I} \cup \mathbf{V})$ is called a *triple pattern*. A *basic graph pattern* is a finite set of triple patterns.

A *filter constraint* is defined recursively as follows: (i) if $?X, ?Y \in \mathbf{V}$ and $u \in \mathbf{I} \cup \mathbf{L}$ then $?X = u$, $?X = ?Y$, $\text{bound}(?X)$, $\text{isIRI}(?X)$, $\text{isLiteral}(?X)$, and $\text{isBlank}(?X)$ are *atomic filter constraints*[4]; (ii) if C_1 and C_2 are filter constraints then $(\neg C_1)$, $(C_1 \wedge C_2)$, and $(C_1 \vee C_2)$ are *complex filter constraints*.

A *mapping* μ is a partial function $\mu : \mathbf{V} \to \mathbf{T}$. The domain of μ, $\text{dom}(\mu)$, is the subset of $\mathbf{V}$ where μ is defined. The *empty mapping* μ_0 is a mapping such that $\text{dom}(\mu_0) = \emptyset$. Two mappings μ_1, μ_2 are *compatible*, denoted $\mu_1 \sim \mu_2$, when for all $?X \in \text{dom}(\mu_1) \cap \text{dom}(\mu_2)$ it satisfies that $\mu_1(?X) = \mu_2(?X)$, i.e., when $\mu_1 \cup \mu_2$ is also a mapping. The expression $\mu_{?X \to v}$ denote a mapping such that $\text{dom}(\mu) = \{?X\}$ and $\mu(?X) = v$

Let C_1 and C_2 be filter constrains. The evaluation of a filter constraint C against a mapping μ, denoted $\mu(C)$, is defined in a three value logic with values $\{true, false, error\}$ as follows:

- if C is an atomic filter constraint, then
  - if $\text{var}(C) \subseteq \text{dom}(\mu)$ then
    $\mu(C) = true$ when
    - C is $?X = u$ and $\mu(?X) = u$; or
    - C is $?X = ?Y$ and $\mu(?X) = \mu(?Y)$; or
    - C is $\text{isIRI}(?X)$ and $\mu(?X) \in \mathbf{I}$; or
    - C is $\text{isLiteral}(?X)$ and $\mu(?X) \in \mathbf{L}$; or
    - C is $\text{isBlank}(?X)$ and $\mu(?X) \in \mathbf{B}$; or
    - C is $\text{bound}(?X)$;

    and $\mu(C) = false$ otherwise.

[4] For a complete list of atomic filter constraints see the SPARQL Specification [17].

- if $\text{var}(C) \nsubseteq \text{dom}(\mu)$ then
  - if C is bound($?X$) then $\mu(C) = \textit{false}$ else $\mu(C) = \textit{error}$.[5]

- if C is a complex filter constraint, then $\mu(C)$ is defined as follows:

$\mu(C_1)$	$\mu(C_2)$	$\mu(C_1) \wedge \mu(C_2)$	$\mu(C_1) \vee \mu(C_2)$
true	*true*	*true*	*true*
true	*false*	*false*	*true*
true	*error*	*error*	*true*
false	*true*	*false*	*true*
false	*false*	*false*	*false*
false	*error*	*false*	*error*
error	*true*	*error*	*true*
error	*false*	*false*	*error*
error	*error*	*error*	*error*

$\mu(C_1)$	$\neg\mu(C_1)$
true	*false*
false	*true*
error	*error*

A mapping μ satisfies a filter constraint C, denoted $\mu \models C$, iff $\mu(C) = \textit{true}$.

To define the semantics of the SPARQL operators, we need to introduce the following operations between two sets of mappings Ω_1, Ω_2:

$\Omega_1 \bowtie \Omega_2 = \{\mu_1 \cup \mu_2 \mid \mu_1 \in \Omega_1, \mu_2 \in \Omega_2 \text{ and } \mu_1 \sim \mu_2\}$

$\Omega_1 \bowtie_C \Omega_2 = \{\mu_1 \cup \mu_2 \mid \mu_1 \in \Omega_1, \mu_2 \in \Omega_2, \mu_1 \sim \mu_2 \text{ and } (\mu_1 \cup \mu_2) \models C\}$

$\Omega_1 \cup \Omega_2 = \{\mu \mid \mu \in \Omega_1 \text{ or } \mu \in \Omega_2\}$

$\Omega_1 \setminus \Omega_2 = \{\mu_1 \in \Omega_1 \mid \text{ for all } \mu_2 \in \Omega_2, \mu_1 \text{ and } \mu_2 \text{ are not compatible } \}$

$\Omega_1 \setminus_C \Omega_2$ = $\{\mu_1 \in \Omega_1 \mid \text{for all } \mu_2 \in \Omega_2, \mu_1 \text{ and } \mu_2 \text{ are not compatible } \} \cup$
$\{\mu_1 \in \Omega_1 \mid \text{for all } \mu_2 \in \Omega_2 \text{ compatible with } \mu_1, (\mu_1 \cup \mu_2) \not\models C\}$

$\Omega_1 \text{⟕} \Omega_2 = (\Omega_1 \bowtie \Omega_2) \cup (\Omega_1 \setminus \Omega_2)$

$\Omega_1 \text{⟕}_C \Omega_2 = (\Omega_1 \bowtie_C \Omega_2) \cup (\Omega_1 \setminus_C \Omega_2)$

Intuitively, $\Omega_1 \bowtie \Omega_2$ is the set of mappings that results from extending mappings in Ω_1 with their compatible mappings in Ω_2. A mapping is in $\Omega_1 \bowtie_C \Omega_2$ if it is a result mapping in $\Omega_1 \bowtie \Omega_2$ which satisfy the filter constraint C. The result of $\Omega_1 \cup \Omega_2$ is the usual set theoretical union. As opposed to $\Omega_1 \bowtie \Omega_2$, $\Omega_1 \setminus \Omega_2$ is the set of mappings in Ω_1 that cannot be extended with any mapping in Ω_2. A mapping of Ω_1 is in $\Omega_1 \setminus_C \Omega_2$ if it cannot be extended with any mapping in Ω_2, or each time it is extended with any mapping in Ω_2, the result mapping does not satisfy the filter constraint C. A mapping is in $\Omega_1 \text{⟕} \Omega_2$ if it is the extension of a mapping of Ω_1 with a compatible mapping of Ω_2, or if it belongs to Ω_1 and cannot be extended with any mapping of Ω_2.

[5] Functions invoked with an argument of the wrong type are evaluated to *error*.

Graph pattern P	Evaluation $[\![P]\!]_G^D$
$(P_1 \text{ AND } P_2)$	$[\![P_1]\!]_G^D \bowtie [\![P_2]\!]_G^D$
$(P_1 \text{ OPT } P_2)$	$[\![P_1]\!]_G^D \mathbin{⟕} [\![P_2]\!]_G^D$
$(P_1 \text{ UNION } P_2)$	$[\![P_1]\!]_G^D \cup [\![P_2]\!]_G^D$
$(P_1 \text{ FILTER } C)$	$\{\mu \mid \mu \in [\![P_1]\!]_G^D \text{ and } \mu \models C\}$
$(u \text{ GRAPH } P_1)$	$[\![P_1]\!]^D_{\text{gr}(u)_D}$
$(?X \text{ GRAPH } P_1)$	$\bigcup_{v \in \text{names}(D)} ([\![P_1]\!]^D_{\text{gr}(v)_D} \bowtie \{\mu_{?X \to v}\})$

Table 5.1: Semantics of $\text{SPARQL}_{\text{COMP}}$ graph patterns. P_1, P_2 are $\text{SPARQL}_{\text{COMP}}$ graph patterns, C is a filter constraint, u is an IRI and $?X$ is a variable.

5.3.2 $\text{SPARQL}_{\text{COMP}}$: Compositional SPARQL.

A $\text{SPARQL}_{\text{COMP}}$ graph pattern `P` is defined recursively by the following grammar:

```
P  ::= t | "(" GP ")"
GP ::= P "AND" P | P "UNION" P | P "OPT" P | P "FILTER" C | n "GRAPH" P
```

where `t` denotes a triple pattern, `C` denotes a filter constraint, and `n` $\in \mathbf{I} \cup \mathbf{V}$.

The evaluation of a $\text{SPARQL}_{\text{COMP}}$ graph pattern P over an RDF dataset D having active graph G, denoted $[\![P]\!]_G^D$ (or $[\![P]\!]$ where D and G are clear from the context), is defined recursively as follows:

- if P is a triple pattern t, $[\![P]\!]_G^D = \{\mu \mid \text{dom}(\mu) = \text{var}(t) \text{ and } \mu(t) \in G\}$ where $\mu(t)$ is the triple obtained by replacing the variables in t according to mapping μ.
- if P is a complex graph pattern then $[\![P]\!]_G^D$ is defined as given in Table 5.1.

5.3.3 $\text{SPARQL}_{\text{OP}}$: Operational SPARQL (W3C).

A $\text{SPARQL}_{\text{OP}}$ graph pattern `GroupGP` is defined by the following grammar[6]:

```
GroupGP        ::= "{" TB? ((GPNotTriples | Filter) "."? TB?)* "}"
GPNotTriples   ::= OptionalGP | GroupOrUnionGP | GraphGP
OptionalGP     ::= "OPTIONAL" GroupGP
GraphGP        ::= "GRAPH" VarOrIRIref GroupGP
GroupOrUnionGP ::= GroupGP ( "UNION" GroupGP )*
Filter         ::= "FILTER" Constraint
```

where `TB` denotes a basic graph pattern (a set of triple patterns), `VarOrIRIref` denotes a term in the set $\mathbf{I} \cup \mathbf{V}$ and `Constraint` denotes a filter constraint. Note that the operator `{A . B}` represents the AND but it has not fixed arity.

[6] `http://www.w3.org/TR/rdf-sparql-query/#grammar`. We use `GP` and `TB` to abbreviate `GraphPattern` and `TriplesBlock` respectively.

The evaluation of a SPARQL$_{\text{OP}}$ graph pattern `GroupGP` is defined by a series of steps, starting by transforming `GroupGP`, via a function T, into an intermediate algebra expression E (with operators BGP, Join, Union, LeftJoin, Graph and Filter), and finally evaluating E on an RDF dataset D.

The transformation $T(\texttt{GroupGP})$ is given by Algorithm 1. The evaluation of $E = T(\texttt{GroupGP})$ over an RDF dataset D having active graph G, which we will denote $\langle\langle E \rangle\rangle_G^D$ (or $\langle\langle E \rangle\rangle$ where D and G are clear from the context) is defined recursively as follows [7]:

- if E is BGP(`TB`), $\langle\langle E \rangle\rangle_G^D = \{\mu \mid \text{dom}(\mu) = \text{var}(E) \text{ and } \mu(E) \subseteq G\}$ where $\mu(E)$ is the set of triples obtained by replacing the variables in the triple patterns of `TB` according to mapping μ.
- if E is a complex expression then $\langle\langle E \rangle\rangle_G^D$ is defined as given in Table 5.2.

Note. In the definition of graph patterns, we avoided blank nodes, because this restriction does not diminish the generality of our study. In fact, each SPARQL query Q can be simulated by a SPARQL query Q' without blank nodes in its pattern. It follows from the definitions of RDF instance mapping, solution mapping, and the order of evaluation of solution modifiers (see the SPARQL Specification [17]), that if Q is a query with graph pattern P, and Q' is the same query where each blank node b in P has been replaced by a fresh variable $?X_b$ then Q and Q' give the same results. (Note that, if Q has the query form SELECT or DESCRIBE, the "$*$" parameter is –according to the specification of SPARQL– an abbreviation for all variables occurring in the pattern. In this case the query Q' should explicit in the SELECT clause all variables of the original pattern P.)

Note. SPARQL$_{\text{COMP}}$ follows a compositional semantics, whereas SPARQL$_{\text{OP}}$ follows a mixture of compositional and operational semantics where the meaning of certain patterns depends on their context, e.g., lines 8 to 11 in Algorithm 1.

Note. In this study we will follow the simpler syntax of SPARQL$_{\text{COMP}}$, better suited to do formal analysis and processing than the syntax presented by SPARQL$_{\text{OP}}$. There is an easy and intuitive way of translating back and forth between both syntax formalisms, which we will not detail here.

5.4 Expressing Difference of Patterns in SPARQL$_{\text{OP}}$

The SPARQL$_{\text{OP}}$ specification indicates that it is possible to test if a graph pattern does not match a dataset, via a combination of optional patterns and filter conditions, like negation as failure in logic programming (see Sec. 11.4.1 of the SPARQL specification[17]). In this section we analyze in depth the scope and limitations of this approach.

[7] The evaluation function $\langle\langle E \rangle\rangle_G^D$ is originally denoted eval$(D(G), E)$ in the SPARQL Specification [17].

Algorithm 1 Transformation of $\text{SPARQL}_{\text{OP}}$ patterns into algebra expressions.

1: // Input: a $\text{SPARQL}_{\text{OP}}$ graph pattern `GroupGP`
2: // Output: an algebra expression $E = T(\texttt{GroupGP})$
3: $E \leftarrow$ empty pattern; $FS \leftarrow \emptyset$
4: **for** each syntactic form f in `GroupGP` **do**
5: **if** f is `TB` **then**
6: $E \leftarrow \text{Join}(E, \text{BGP}(\texttt{TB}))$
7: **if** f is `OPTIONAL` $\texttt{GroupGP}_1$ **then**
8: **if** $T(\texttt{GroupGP}_1)$ is $\text{Filter}(F, E')$ **then**
9: $E \leftarrow \text{LeftJoin}(E, E', F)$
10: **else**
11: $E \leftarrow \text{LeftJoin}(E, T(\texttt{GroupGP}_1), \mathit{true})$
12: **if** f is $\texttt{GroupGP}_1$ `UNION` $\cdots$ `UNION` $\texttt{GroupGP}_n$ **then**
13: **if** $n > 1$ **then**
14: $E' \leftarrow \text{Union}(\cdots(\text{Union}(T(\texttt{GroupGP}_1), T(\texttt{GroupGP}_2))\cdots), T(\texttt{GroupGP}_n))$
15: **else**
16: $E' \leftarrow T(\texttt{GroupGP}_1)$
17: $E \leftarrow \text{Join}(E, E')$
18: **end if**
19: **if** f is `GRAPH VarOrIRIref` $\texttt{GroupGP}_1$ **then**
20: $E \leftarrow \text{Join}(E, \text{Graph}(\texttt{VarOrIRIref}, T(\texttt{GroupGP}_1)))$
21: **if** f is `FILTER constraint` **then**
22: $FS \leftarrow (FS \wedge \texttt{constraint})$
23: **end for**
24: **if** $FS \neq \emptyset$ **then**
25: $E \leftarrow \text{Filter}(FS, E)$
26: **return** E

Algebra Expression E	Evaluation $\langle\langle E \rangle\rangle_G^D$
$\text{Join}(E_1, E_2)$	$\langle\langle E_1 \rangle\rangle_G^D \bowtie \langle\langle E_2 \rangle\rangle_G^D$
$\text{LeftJoin}(E_1, E_2, C)$	$\langle\langle E_1 \rangle\rangle_G^D \text{⟕}_C \langle\langle E_2 \rangle\rangle_G^D$
$\text{Union}(E_1, E_2)$	$\langle\langle E_1 \rangle\rangle_G^D \cup \langle\langle E_2 \rangle\rangle_G^D$
$\text{Filter}(C, E_1)$	$\{ \mu \mid \mu \in \langle\langle E_1 \rangle\rangle_G^D \text{ and } \mu \models C \}$
$\text{Graph}(u, E_1)$	$\langle\langle E_1 \rangle\rangle_{\text{gr}(u)_D}^D$
$\text{Graph}(?X, E_1)$	$\bigcup_{v \in \text{ names}(D)} (\langle\langle E_1 \rangle\rangle_{\text{gr}(v)_D}^D \bowtie \{\mu_{?X \to v}\})$

Table 5.2: Semantics of $\text{SPARQL}_{\text{OP}}$ graph patterns. A pattern `GroupGP` is transformed into an algebra expression E using Algorithm 1. Then E is evaluated as the table shows. E_1 and E_2 are algebra expressions, C is a filter constraint, u is an IRI and $?X$ is a variable.

We will introduce a syntax for the "difference" of two graph patterns P_1 and P_2, denoted $(P_1 \,\text{MINUS}\, P_2)$, with the intended informal meaning: "the set of mappings that match P_1 and does not match P_2". Formally:

Definition 5.4.1 *Let P_1, P_2 be graph patterns and D be a dataset with active graph G. Then*

$$\langle\langle (P_1 \,\text{MINUS}\, P_2) \rangle\rangle_G^D = \langle\langle P_1 \rangle\rangle_G^D \setminus \langle\langle P_2 \rangle\rangle_G^D.$$

A *naive implementation* of the MINUS operator in terms of the other operators would be the graph pattern $((P_1 \,\text{OPT}\, P_2) \,\text{FILTER}\, C)$ where C is the filter constraint $(\neg \text{bound}(?X))$ for some variable $?X \in \text{var}(P_2) \setminus \text{var}(P_1)$. This means that for each mapping $\mu \in \langle\langle (P_1 \,\text{OPT}\, P_2) \rangle\rangle_G^D$ at least one variable $?X$ occurring in P_2, but not occurring in P_1, does not match (i.e. $?X$ is unbounded). There are two problems with this solution:

- Variable $?X$ cannot be an arbitrary variable. For example, P_2 could be in turn an optional pattern $(P_3 \,\text{OPT}\, P_4)$ where only variables in P_3 are relevant.
- If $\text{var}(P_2) \setminus \text{var}(P_1) = \emptyset$ there is no variable $?X$ to check unboundedness.

The above two problems motivate the introduction of the notions of non-optional variables and copy patterns.

The set of *non-optional variables* of a graph pattern P, denoted $\text{nov}(P)$, is a subset of the variables of P defined recursively as follows: $\text{nov}(P) = \text{var}(P)$ when P is a basic graph pattern; if P is either $(P_1 \,\text{AND}\, P_2)$ or $(P_1 \,\text{UNION}\, P_2)$ then $\text{nov}(P) = \text{nov}(P_1) \cup \text{nov}(P_2)$; if P is $(P_1 \,\text{OPT}\, P_2)$ then $\text{nov}(P) = \text{nov}(P_1)$; if P is $(n \,\text{GRAPH}\, P_1)$ then either $\text{nov}(P) = \text{nov}(P_1)$ when $n \in \mathbf{I}$ or $\text{nov}(P) = \text{nov}(P_1) \cup \{n\}$ when $n \in \mathbf{V}$; and $\text{nov}(P_1 \,\text{FILTER}\, C) = \text{nov}(P_1)$. Intuitively $\text{nov}(P)$ contains the variables that necessarily must be bounded in any mapping of P.

Let $\phi : \mathbf{V} \to \mathbf{V}$ be a variable-renaming function. Given a graph pattern P, we say that $\phi(P)$ is a *copy pattern* of P, i.e., $\phi(P)$ is an isomorphic copy of P whose variables have been renamed according to ϕ and satisfying that $\text{var}(P) \cap \text{var}(\phi(P)) = \emptyset$.

Theorem 5.4.1 *Let P_1 and P_2 be graph patterns. Then:*

$$(P_1 \,\text{MINUS}\, P_2) \equiv ((P_1 \,\text{OPT}((P_2 \,\text{AND}\, \phi(P_2)) \,\text{FILTER}\, C_1)) \,\text{FILTER}\, C_2) \tag{5.1}$$

where:

- *C_1 is the filter constraint $(?X_1 = ?X_1' \wedge \cdots \wedge ?X_n = ?X_n')$ where $?X_i \in \text{var}(P_2)$ and $?X_i' = \phi(?X_i)$ for $1 \leq i \leq n$.*
- *C_2 is the filter constraint $(\neg\text{bound}(?X'))$ for some $?X' \in \text{nov}(\phi(P_2))$.*

Proof. Let P be the graph pattern $(P_1 \,\text{MINUS}\, P_2)$ and P' be the right hand side of (5.1). We will prove that for every dataset D with active graph G, it satisfies that $\langle\langle P \rangle\rangle_G^D = \langle\langle P' \rangle\rangle_G^D$.

(a) *Evaluation $\langle\langle P \rangle\rangle$:* By definition, $\langle\langle P \rangle\rangle = \langle\langle P_1 \rangle\rangle \setminus \langle\langle P_2 \rangle\rangle$. Then, a mapping μ is in $\langle\langle P \rangle\rangle$ if and only if $\mu \in \langle\langle P_1 \rangle\rangle$ and for every mapping $\mu' \in \langle\langle P_2 \rangle\rangle$, μ and μ' are not compatible.

(b) *Evaluation $\langle\langle P' \rangle\rangle$:* To simplify the idea of the proof, we reduce P' to the graph pattern $((P_1 \,\text{OPT}\, P_2) \,\text{FILTER}\, C_2)$ where C_2 is $(\neg\text{bound}(?X))$ for some $?X \in \text{nov}(P_2) \setminus \text{var}(P_1)$. Note that this reduction does not diminish the generality of the proof because $\phi(P_2)$ and C_1 were added into P' to solve the case when $\text{nov}(P_2) \setminus \text{var}(P_1) = \emptyset$ (see notes later). Here, a mapping μ is in $\langle\langle P' \rangle\rangle$ if and only if $\mu \in \langle\langle (P_1 \,\text{OPT}\, P_2) \rangle\rangle$ and $\mu \models C_2$. Given $\mu_1 \in \langle\langle P_1 \rangle\rangle$, it holds that $\mu \in \langle\langle (P_1 \,\text{OPT}\, P_2) \rangle\rangle$ iff either (i) $\mu = \mu_1 \cup \mu_2$ for some $\mu_2 \in \langle\langle P_2 \rangle\rangle$ compatible with μ_1; or (ii) $\mu = \mu_1$ and for every $\mu_2 \in \langle\langle P_2 \rangle\rangle$, μ_1 and μ_2 are not compatible. Note that, in case (i), $\mu(?X)$ is bounded for every variable $?X \in \text{nov}(P_2)$ and, in case (ii), $\mu(?X)$ is unbounded for every variable $?X \in \text{nov}(P_2) \setminus \text{var}(P_1)$. Given that C_2 contains the filter constraint $(\neg\text{bound}(?X))$ for some variable $?X \in \text{nov}(P_2) \setminus \text{var}(P_1)$, only case (ii) satisfies the condition $\mu \models C_2$ (Note that here is critical the fact that $?X$ is a safe variable occurring in P_2 but not in P_1). Then, $\langle\langle P' \rangle\rangle$ will only contain mappings from case (ii), that is each mapping $\mu \in \langle\langle P' \rangle\rangle$ satisfies that $\mu = \mu_1 \in \langle\langle P_1 \rangle\rangle$ and for every mapping $\mu_2 \in \langle\langle P_2 \rangle\rangle$, μ_1 and μ_2 are not compatible.

Therefore, $\langle\langle P' \rangle\rangle$ has exactly the same mappings as the evaluation of $\langle\langle P \rangle\rangle$ in (a), and we conclude the proof.

□

Note. Why the copy pattern $\phi(P)$ is necessary?
Consider the naive implementation for $(P_1 \,\text{MINUS}\, P_2)$, that is pattern $((P_1 \,\text{OPT}\, P_2) \,\text{FILTER}\, C)$ where C is the filter constraint $(\neg\text{bound}(?X))$ for some $?X \in \text{var}(P_2) \setminus \text{var}(P_1)$. Note that this implementation would fail when $\text{var}(P_2) \setminus \text{var}(P_1) = \emptyset$, because *there exist no variables* to check unboundedness. For example, consider the graph patterns $P_1 = (?X, \text{name}, ?N)$ and

$P_2 = (?X, \text{lastname}, \text{“Perez”})$. The naive implementation of $(P_1 \text{ MINUS } P_2)$ will give a pattern with filter condition $C = \emptyset$ because there are no variables in $\text{var}(P_2) \setminus \text{var}(P_1)$ (Note that it is not possible to use variable $?X$ to check unboundedness when evaluating P_2 because –to satisfy the entire pattern– variable $?X$ must have already been bound in the evaluation of pattern P_1).

To solve this problem, P_2 is replaced by $((P_2 \text{ AND } \phi(P_2)) \text{ FILTER } C_1)$ where $\phi(P_2)$ is a copy of P_2 whose variables have been renamed and whose relations of equality with the original ones are in condition C_1. Then we can use some variable from $\phi(P_2)$ to check if graph pattern P_2 does not match. The copy pattern ensures that there will exist a variable to check unboundedness.

Then, the implementation of $(P_1 \text{ MINUS } P_2)$ in the example will be

$$\begin{aligned}&(((?X, \text{name}, ?N) \text{ OPT}\\ &\qquad (((?X, \text{lastname}, \text{“Perez”}) \text{ AND } (?X', \text{lastname}, \text{“Perez”}))\\ &\qquad\qquad \text{FILTER}(?X = ?X'))) \text{ FILTER}(\neg \text{bound}(?X'))),\end{aligned}$$

where variable $?X' \in \phi(P_2)$ has been selected to check unboundedness.

Note that the inclusion of copy patterns could introduce an exponential blow-up in the size of the pattern. A possible optimization (still inside the syntax of SPARQL) is to select a *safe triple pattern t* of P_2, i.e., a triple pattern having only safe variables (at least one), and using the copy pattern $\phi(t)$ instead of the entire copy pattern $\phi(P_2)$.

Note. Why non-optional variables?
Consider the graph pattern $P = ((?X, \text{name}, ?N) \text{ MINUS } ((?X, \text{knows}, ?Y) \text{ OPT } (?Y, \text{mail}, ?Z)))$. The naive implementation of P would be the pattern $P' = ((P_1 \text{ OPT } P_2) \text{ FILTER}(\neg \text{bound}(?Z)))$, where $P_1 = (?X, \text{name}, ?N)$, $P_2 = ((?X, \text{knows}, ?Y) \text{ OPT } (?Y, \text{mail}, ?Z))$ and $?Z$ is the variable selected to check unboundedness. (Note that variable $?Y$ could also have been selected because $?Y \in \text{var}(P_2) \setminus \text{var}(P_1)$.) Additionally, consider the RDF graph

$$\begin{aligned}G = \{ &\text{ (a,name,}n_a\text{), (b,name,}n_b\text{), (b,knows,c), (b,mail,}m_b\text{),}\\ &\text{(c,name,}n_c\text{), (c,knows,d), (d,name,}n_d\text{), (d,mail,}m_d\text{) }\}.\end{aligned}$$

Let $P_2 = (P_3 \text{ OPT } P_4)$ where $P_3 = (?X, \text{knows}, ?Y)$ and $P_4 = (?Y, \text{mail}, ?Z)$. Consider the following evaluations over graph G:

$[\![P_1]\!]_G =$

?X	?N
a	n_a
b	n_b
c	n_c
d	n_d

$[\![P_3]\!]_G =$

?X	?Y
b	c
c	d

$[\![P_4]\!]_G =$

?Y	?Z
b	m_b
d	m_d

$[\![P_2]\!]_G$ =

?X	?Y	?Z
b	c	
c	d	m_d

$[\![(P_1 \text{ OPT } P_2)]\!]_G$ =

?X	?N	?Y	?Z
a	n_a		
b	n_b	c	
c	n_c	d	m_d
d	n_d		

Then $P = (P_1 \text{ MINUS } P_2)$ and $P' = ((P_1 \text{ OPT } P_2) \text{ FILTER}(\neg \text{bound}(?Z)))$ are evaluated as follows:

$[\![P]\!]_G$ =

?X	?N
a	n_a
d	n_d

$[\![P']\!]_G$ =

?X	?N	?Y	?Z
a	n_a		
b	n_b	c	
d	n_d		

Note that the evaluation of graph pattern P' differs from that of pattern P. To see the problem recall the informal semantics: a mapping μ matches pattern P if and only if μ matches P_1 and μ does not match P_2. This latter condition means: it is false that every variable in P_2 (but not in P_1) is bounded. But to say "every variable" is not correct in this context because P_2 contains the optional pattern $(?Y, \text{mail}, ?Z)$, and its variables could be unbounded for some valid solutions of P_2. Hence the problem is produced by the expression $(\neg \text{bound}(?Z))$, because the bounding state of variable $?Z$ introduces noise when testing if pattern P_2 gets matched.

In fact, consider the mapping μ such that $\mu(?X) = b$, $\mu(?N) = n_b$ and $\mu(?Y) = c$. This mapping is not a solution for P because it matches P_2 since it matches $(?X, \text{knows}, ?Y)$ although it does not match the optional pattern $(?Y, \text{mail}, ?Z)$. On the other hand, we have that μ matches P' because it matches $(P_1 \text{ OPT } P_2)$ and μ satisfies the filter constraint $(\neg \text{bound}(?Z))$.

Now, if we ensure the selection of a "non-optional variable" to check unboundedness when transforming P, we have that $?Y$ is the unique non-optional variable occurring in P_2 but not occurring in P_1, i.e., variable $?Y$ works exactly as the test to check if a mapping matching P_1 matches P_2 as well. Hence, instead of P', the pattern $P'' = ((P_1 \text{ OPT } P_2) \text{ FILTER}(\neg \text{bound}(?Y)))$ is the one that expresses faithfully the graph pattern $(P_1 \text{ MINUS } P_2)$, and in fact, the evaluation of P'' gives exactly the same set of mappings as P.

5.5 Avoiding Unsafe Patterns in SPARQL$_{\text{OP}}$

One critical point in the evaluation of patterns in SPARQL$_{\text{OP}}$ is the behavior of *filters*. What is the scope of a filter? What is the meaning of a filter having variables that do not occur in the graph pattern to be filtered?

It was proposed in the work of Pérez et al. [188] that for reasons of simplicity for the user and cleanness of the semantics, the scope of filters should be the expression which they filter,

and free variables should be disallowed in the filter condition. Formally, a graph pattern of the form $(P \text{ FILTER } C)$ is said to be *safe* if $\text{var}(C) \subseteq \text{var}(P)$. In the work of Pérez et al. [188] only safe filter patterns were allowed in the syntax, and hence the scope of the filter C is the pattern P which defines the filter condition. This approach is further supported by the fact that non-safe filters are rare in practice.

The WG decided to follow a different approach, and defined the scope of a filter condition C to be a case-by-case and context-dependent feature:

1. The scope of a filter is defined as follows: a filter "is a restriction on solutions over the whole group in which the filter appears".

2. There is one exception, though, when filters combine with optionals. If a filter expression C belongs to the group graph pattern of an optional, the scope of C is local to the group where the optional belongs to. This is reflected in lines 7 and 8 of Algorithm 1.

The complexities that this approach brings were recognized in the discussion of the WG, and can be witnessed by the reader by following the evaluation of patterns in $\text{SPARQL}_{\text{OP}}$.

Let $\text{SPARQL}_{\text{OP}}^{\text{Safe}}$ be the subset of queries of $\text{SPARQL}_{\text{OP}}$ having only filter-safe patterns. In what follows, we will show that, in $\text{SPARQL}_{\text{OP}}$, non-safe filters are superfluous, and hence its non-standard and case-by-case semantics can be avoided. In fact, we will prove that non-safe filters do not add expressive power to the language, or in other words, that $\text{SPARQL}_{\text{OP}}$ and $\text{SPARQL}_{\text{OP}}^{\text{Safe}}$ have the same expressive power, that is, for each graph pattern P there is a filter-safe graph pattern $P' = \text{safe}(P)$ which computes exactly the same mappings as P.

The transformation $\text{safe}(P)$ is given by Algorithm 2. This algorithm works as the identity for most patterns. The key part is the treatment of patterns which combine filters and optionals. Line 9 is exactly the codification of the $\text{SPARQL}_{\text{OP}}$ evaluation of filters inside optionals. For non-safe filters (see lines 15-20), it replaces each atomic filter condition C', where a free variable occurs, by either an expression *false* when C' is $\text{bound}(\cdot)$; or an expression $\text{bound}(a)$ otherwise. (Note that $\text{bound}(a)$ is evaluated to *error* because a is a constant.)

Note. On Algorithm 2.
The expression in line 9 must be refined for bag semantics to the expression:

$$
\begin{aligned}
P' \leftarrow (\ & \text{safe}((P_1 \text{ AND } P_3) \text{ FILTER } C) \text{ UNION} \\
& (\text{safe}(P_1) \text{ MINUS } \text{safe}(P_3)) \text{ UNION} \\
& ((\text{safe}(P_1) \text{ MINUS } (\text{safe}(P_1) \text{ MINUS } \text{safe}(P_3))) \\
& \qquad \text{MINUS } \text{safe}((P_1 \text{ AND } P_3) \text{ FILTER } C))\ )
\end{aligned}
$$

Lemma 5.5.1 *For every SPARQL_{OP} graph pattern P, the pattern* $\text{safe}(P)$ *is filter-safe and it holds* $\langle\langle P \rangle\rangle = \langle\langle \text{safe}(P) \rangle\rangle$.

Algorithm 2 Transformation of a general graph pattern into a safe pattern.

1: // Input: a $\text{SPARQL}_{\text{OP}}$ graph pattern P
2: // Output: a safe graph pattern $P' \leftarrow \text{safe}(P)$
3: $P' \leftarrow \emptyset$
4: **if** P is $(P_1 \text{ AND } P_2)$ **then** $P' \leftarrow (\text{safe}(P_1) \text{ AND safe}(P_2))$
5: **if** P is $(P_1 \text{ UNION } P_2)$ **then** $P' \leftarrow (\text{safe}(P_1) \text{ UNION safe}(P_2))$
6: **if** P is $(n \text{ GRAPH } P_1)$ **then** $P' \leftarrow (n \text{ GRAPH safe}(P_1))$
7: **if** P is $(P_1 \text{ OPT } P_2)$ **then**
8: **if** P_2 is $(P_3 \text{ FILTER } C)$ **then**
9: $P' \leftarrow (\text{safe}(P_1) \text{ OPT}(\text{safe}((P_1 \text{ AND } P_3) \text{ FILTER } C)))$
10: **else** $P' \leftarrow (\text{safe}(P_1) \text{ OPT safe}(P_2))$
11: **end if**
12: **if** P is $(P_1 \text{ FILTER } C)$ **then**
13: **if** $\text{var}(C) \subseteq \text{var}(\text{safe}(P_1))$ **then** $P' \leftarrow (\text{safe}(P_1) \text{ FILTER } C)$
14: **else**
15: **for all** $?X \in \text{var}(C)$ and $?X \notin \text{var}(\text{safe}(P_1))$ **do**
16: **for all** atomic filter constraint C' in C
17: **if** C' is $(?X = u)$ or $(?X = ?Y)$ or $\text{isIRI}(?X)$ or $\text{isBlank}(?X)$ or $\text{isLiteral}(?X)$
18: Replace in C the constraint C' by $\text{bound}(a)$ //where a is a constant
19: **else if** C' is $\text{bound}(?X)$ **then**
20: Replace in C the constraint C' by *false*
21: **end for**
22: **end for**
23: $P' \leftarrow (\text{safe}(P_1) \text{ FILTER } C)$
24: **end if**
25: **end if**
26: **return** P'

Proof. We present the proof for the most relevant cases presented in Algorithm 2, that is, (a) transformation in line 9 and (b) rewriting of filters in lines 17-20.

(a) Let $P = (P_1 \text{ OPT } (P_2 \text{ FILTER } C))$.
Here $T(P) = \text{LeftJoin}(T(P_1), T(P_2), C)$ and $\langle\langle T(P)\rangle\rangle = \langle\langle T(P_1)\rangle\rangle \rtimes_C \langle\langle T(P_2)\rangle\rangle$.

Suppose that $\Omega_1 = \langle\langle T(P_1)\rangle\rangle$ and $\Omega_2 = \langle\langle T(P_2)\rangle\rangle$. Then $\langle\langle T(P)\rangle\rangle$ is given by the expression $(\Omega_1 \bowtie_C \Omega_2) \cup (\Omega_1 \setminus_C \Omega_2)$ where:

$$^{(\star)}(\Omega_1 \bowtie_C \Omega_2) = \{\mu_1 \cup \mu_2 \mid \mu_1 \in \Omega_1, \mu_2 \in \Omega_2, \mu_1 \sim \mu_2, \text{ and } (\mu_1 \cup \mu_2) \models C\}$$

$$^{(\star\star)}(\Omega_1 \setminus_C \Omega_2) = \{\mu_1 \in \Omega_1 \mid \text{for all } \mu_2 \in \Omega_2, \mu_1 \text{ and } \mu_2 \text{ are not compatible}\} \cup \{\mu_1 \in \Omega_1 \mid \text{for all } \mu_2 \in \Omega_2 \text{ compatible with } \mu_1, (\mu_1 \cup \mu_2) \not\models C\}$$

(i) Let $P' = (P_1 \text{ OPT } ((P_1 \text{ AND } P_2) \text{ FILTER } C))$. We will prove that, *under set semantics*, $\langle\langle P\rangle\rangle_G^D = \langle\langle P'\rangle\rangle_G^D$ for every dataset D with active graph G.

Consider that $P_3 = (P_1 \text{ AND } P_2)$. Then $T(P') = \text{LeftJoin}(T(P_1), T(P_3), C)$ and $\langle\langle T(P')\rangle\rangle = \langle\langle T(P_1)\rangle\rangle \rtimes_C \langle\langle T(P_3)\rangle\rangle$. If we assume that $\Omega_1 = \langle\langle T(P_1)\rangle\rangle$, $\Omega_2 = \langle\langle T(P_2)\rangle\rangle$ and $\Omega_3 = \langle\langle T(P_3)\rangle\rangle$, then $\langle\langle T(P')\rangle\rangle = (\Omega_1 \bowtie_C \Omega_3) \cup (\Omega_1 \setminus_C \Omega_3)$ where:

$$^{(1)}(\Omega_1 \bowtie_C \Omega_3) = \{\mu_1 \cup \mu_3 \mid \mu_1 \in \Omega_1, \mu_3 \in \Omega_3, \mu_1 \sim \mu_3 \text{ and } (\mu_1 \cup \mu_3) \models C\}$$

and

$$^{(2)}(\Omega_1 \setminus_C \Omega_3) = \{\mu_1 \in \Omega_1 \mid \text{for all } \mu_3 \in \Omega_3, \mu_1 \text{ and } \mu_3 \text{ are not compatible}\} \cup \{\mu_1 \in \Omega_1 \mid \text{for all } \mu_3 \in \Omega_3 \text{ compatible with } \mu_1, (\mu_1 \cup \mu_3) \not\models C\}.$$

Assume $\Omega_3 = \Omega_1 \bowtie \Omega_2 = \{\mu_1 \cup \mu_2 \mid \mu_1 \in \Omega_1, \mu_2 \in \Omega_2 \text{ and } \mu_1 \sim \mu_2\}$. If we rewrite (1) by solving μ_3, we will have the set

$$^{(1.1)}\{\mu_1 \cup \mu_2 \mid \mu_1 \in \Omega_1, \mu_2 \in \Omega_2, \mu_1 \sim \mu_2 \text{ and } (\mu_1 \cup \mu_2) \models C\}.$$

In the former set of (2): by definition of Ω_3, it applies that μ_1 is not compatible with every mapping $(\mu_1' \cup \mu_2) \in \Omega_3$ such that $\mu_1' \in \Omega_1$, $\mu_2 \in \Omega_2$ and $\mu_1' \sim \mu_2$. This condition is true if and only if $\mu_1' \neq \mu_1$. Consequently μ_1 is not compatible with every $\mu_2 \in \Omega_2$. Then, we can simplify the former set in (2) as:

$$^{(2.1)}\{\mu_1 \in \Omega_1 \mid \text{for all } \mu_2 \in \Omega_2, \mu_1 \text{ and } \mu_2 \text{ are not compatible}\}$$

In the latter set of (2): by definition of Ω_3 we have that each mapping $(\mu_1' \cup \mu_2) \in \Omega_3$ satisfying that $\mu_1' \in \Omega_1$, $\mu_2 \in \Omega_2$, $\mu_1' \sim \mu_2$ and $\mu_1 \sim (\mu_1' \cup \mu_2)$, it holds that $(\mu_1 \cup (\mu_1' \cup \mu_2)) \not\models C$. The condition $\mu_1 \sim (\mu_1' \cup \mu_2)$ is true if and only if $\mu_1' = \mu_1$.

Consequently μ_1 is compatible with some $\mu_2 \in \Omega_2$ and $(\mu_1 \cup \mu_2) \not\models C$. Then, we can simplify the latter set in (2) to the set:

$$^{(2.2)}\{\mu_1 \in \Omega_1 \mid \text{for all } \mu_2 \in \Omega_2 \text{ compatible with } \mu_1, (\mu_1 \cup \mu_2) \not\models C\}$$

Finally, we have that (1.1) corresponds to $(\star)$, (2.1) is the former set in $(\star\star)$ and (2.2) is the latter set in $(\star\star)$.

Then, we have proved that $\langle\langle P \rangle\rangle = \langle\langle P' \rangle\rangle$.

(ii) Let P' be the graph pattern

$$\begin{array}{l}(\ ((P_1 \text{ AND } P_2) \text{ FILTER } C) \text{ UNION} \\ \quad (P_1 \text{ MINUS } P_2) \text{ UNION} \\ \quad ((P_1 \text{ MINUS } (P_1 \text{ MINUS } P_2)) \\ \qquad \text{MINUS } ((P_1 \text{ AND } P_2) \text{ FILTER } C)) \ )\end{array}$$

We will prove that, *under bag semantics*, $\langle\langle P \rangle\rangle_G^D = \langle\langle P' \rangle\rangle_G^D$ for every dataset D with active graph G.

Consider that $P_3 = ((P_1 \text{ AND } P_2) \text{ FILTER } C)$, $P_4 = (P_1 \text{ MINUS } P_2)$ and $P_5 = ((P_1 \text{ MINUS } (P_1 \text{ MINUS } P_2)) \text{ MINUS } ((P_1 \text{ AND } P_2) \text{ FILTER } C))$.
We have that $T(P') = \text{Union}(\text{Union}(T(P_3), T(P_4)), T(P_5))$ where

$T(P_3) = \text{Filter}(C, \text{Join}(T(P_1), T(P_2)))$,
$T(P_4) = \text{Diff}(T(P_1), T(P_2), \mathit{true})$, and
$T(P_5) = \text{Diff}(\text{Diff}(T(P_1), T(P_2), \mathit{true}), T(P_3), \mathit{true})$

Suppose that $\Omega_1 = \langle\langle T(P_1) \rangle\rangle$ and $\Omega_2 = \langle\langle T(P_2) \rangle\rangle$. Then $\langle\langle T(P') \rangle\rangle$ is given by the expression $\langle\langle T(P_3) \rangle\rangle \cup \langle\langle T(P_4) \rangle\rangle \cup \langle\langle T(P_5) \rangle\rangle$ where

$$\langle\langle T(P_3) \rangle\rangle = \{\mu_1 \cup \mu_2 \mid \mu_1 \in \Omega_1, \mu_2 \in \Omega_2, \mu_1 \sim \mu_2, \text{ and } (\mu_1 \cup \mu_2) \models C\}$$

$$\begin{aligned}\langle\langle T(P_4) \rangle\rangle = &\{\mu_1 \in \Omega_1 \mid \text{for all } \mu_2 \in \Omega_2, \mu_1 \text{ and } \mu_2 \text{ are not compatible}\} \cup \\ &\{\mu_1 \in \Omega_1 \mid \text{for all } \mu_2 \in \Omega_2 \text{ compatible with } \mu_1, (\mu_1 \cup \mu_2) \not\models \mathit{true}\}\end{aligned}$$

$$\begin{aligned}\langle\langle T(P_5) \rangle\rangle &= (\langle\langle P_1 \rangle\rangle \setminus_{\mathit{true}} \langle\langle T(P_4) \rangle\rangle) \setminus_{\mathit{true}} \langle\langle T(P_3) \rangle\rangle \\ &= \{\mu_1 \in \Omega_1 \mid \text{for all } \mu_2 \in \Omega_2 \text{ compatible with } \mu_1, (\mu_1 \cup \mu_2) \not\models C\}\end{aligned}$$

From the above sets we can state that:

- $\langle\langle T(P_3) \rangle\rangle$ correspond to the set in $^{(\star)}$.
- $\langle\langle T(P_4) \rangle\rangle$ correspond to the former set in $(\star\star)$. Note that the second set will always be empty because condition $(\mu_1 \cup \mu_2) \not\models \mathit{true}$ is false in any case.
- The expression $(\langle\langle P_1 \rangle\rangle \setminus_{\mathit{true}} \langle\langle T(P_4) \rangle\rangle)$ returns the subset of mappings in $\langle\langle P_1 \rangle\rangle$ which are compatible with some mapping in $\langle\langle P_2 \rangle\rangle$; from this set we subtract mappings from $\langle\langle P_3 \rangle\rangle$ (i.e. such mappings that satisfies condition C); Then $\langle\langle T(P_5) \rangle\rangle$ returns mappings in $\langle\langle P_1 \rangle\rangle$ that are compatible with some mapping

in $\langle\langle P_2 \rangle\rangle$ but not satisfying condition C, that is $\langle\langle T(P_5) \rangle\rangle$ corresponds to the latter set in $(\star\star)$.

Then we have proved that $\langle\langle P \rangle\rangle = \langle\langle P' \rangle\rangle$.

(b) Consider the following semantics defined in the SPARQL Specification [17]:

- Apart from bound$(\cdot)$, all functions and operators operate on RDF Terms and will produce a type *error* if any arguments are unbound (Sec. 11.2).
- Function bound(var) returns true if *var* is bound to a value, and returns false otherwise (Sec. 11.4.1).

Let P be the non-safe graph pattern $(P_1 \text{ FILTER} C)$, $?X$ be a variable in $\text{var}(C) \setminus \text{var}(P_1)$ and μ be a mapping in $\langle\langle P_1 \rangle\rangle$. The evaluation $\mu(C')$ of an atomic filter constraint C' in C which contains variable $?X$, will be given (according to the above semantics) as follows:

(i) if C' is $(?X = u)$ or $(?X = ?Y)$ or isIRI$(?X)$ or isBlank$(?X)$ or isLiteral$(?X)$ then $\mu(C') = error$;

(ii) else if C' is bound$(?X)$ then $\mu(C') = false$.

To attain the same results, we can replace C' in C by either

- the filter expression bound(a) with $a \in \mathbf{I} \cup \mathbf{L}$ in case (i); or
- the filter expression *false* in case (ii).

Applying the above procedure to each atomic filter condition in C having a variable in $\text{var}(C) \setminus \text{var}(P_1)$, we will transform P in a safe filter pattern.

□

Thus we proved:

Theorem 5.5.2 *$SPARQL_{OP}$ and $SPARQL_{OP}^{Safe}$ have the same expressive power.*

5.6 Equivalence of $SPARQL_{OP}$ and $SPARQL_{COMP}$

As we have been showing, the semantics that the WG gave to SPARQL departed in some aspects from a compositional semantics. We also indicated that there is an alternative formalization, with a standard compositional semantics, which was called $SPARQL_{COMP}$ [188].

The good news is that, albeit apparent differences, these languages are equivalent in expressive power, that is, they compute the same class of queries.

Theorem 5.6.1 *$SPARQL_{OP}^{Safe}$ is equivalent to $SPARQL_{COMP}$ under bag semantics.*

Proof. The proof of this theorem is an induction on the structure of patterns. The only non-evident case is the particular evaluation of filters inside optionals where the semantics of $\text{SPARQL}_{\text{OP}}^{\text{Safe}}$ and $\text{SPARQL}_{\text{COMP}}$ differ.

Specifically, given a graph pattern $P = (P_1 \,\text{OPT}(P_2 \,\text{FILTER}\, C))$, we have that $\text{SPARQL}_{\text{OP}}^{\text{Safe}}$ evaluates the algebra expression $T(P) = \text{LeftJoin}(T(P_1), T(P_2), C)$, whereas $\text{SPARQL}_{\text{COMP}}$ evaluates the expression $[\![P_1]\!] \bowtie [\![(P_2 \,\text{FILTER}\, C)]\!]$, which is the same as the $\text{SPARQL}_{\text{OP}}$ algebra expression $\text{LeftJoin}(T(P_1), \text{Filter}(C, T(P_2)), \mathit{true})$. Note that the scope of filter condition C in $\text{SPARQL}_{\text{OP}}$ is the entire pattern P, whereas in $\text{SPARQL}_{\text{COMP}}$ the scope of C is the pattern P_2.

Let P be the graph pattern $(P_1 \,\text{OPT}(P_2 \,\text{FILTER}\, C))$ where $\text{var}(C) \subseteq \text{var}(P_2)$ (i.e. P is filter safe). We will show that for every dataset D with active graph G, it satisfies that $\langle\langle P \rangle\rangle_G^D = [\![P]\!]_G^D$.

- *Evaluation* $\langle\langle P \rangle\rangle_G^D$: Following the steps of evaluation in $\text{SPARQL}_{\text{OP}}$, we have that $T(P) = \text{LeftJoin}(T(P_1), T(P_2), C)$ and $\langle\langle T(P) \rangle\rangle = \langle\langle T(P_1) \rangle\rangle \bowtie_C \langle\langle T(P_2) \rangle\rangle$.

  Now suppose that $\Omega_1 = \langle\langle T(P_1) \rangle\rangle$ and $\Omega_2 = \langle\langle T(P_2) \rangle\rangle$. Then $\langle\langle T(P) \rangle\rangle$ is given by the expression $(\Omega_1 \bowtie_C \Omega_2) \cup (\Omega_1 \setminus_C \Omega_2)$ where:

  $$^{(\star)}(\Omega_1 \bowtie_C \Omega_2) = \{\mu_1 \cup \mu_2 \mid \mu_1 \in \Omega_1, \mu_2 \in \Omega_2, \mu_1 \sim \mu_2 \text{ and } (\mu_1 \cup \mu_2) \models C\}$$

  and

  $$^{(\star\star)}(\Omega_1 \setminus_C \Omega_2) = \{\mu_1 \in \Omega_1 \mid \text{for all } \mu_2 \in \Omega_2, \mu_1 \text{ and } \mu_2 \text{ are not compatible}\} \cup \{\mu_1 \in \Omega_1 \mid \text{for all } \mu_2 \in \Omega_2 \text{ compatible with } \mu_1, (\mu_1 \cup \mu_2) \not\models C\}.$$

- *Evaluation* $[\![P]\!]_G^D$: We have that $[\![P]\!] = [\![P_1]\!] \bowtie [\![(P_2 \,\text{FILTER}\, C)]\!]$.

  Suppose that $\Omega_1 = [\![P_1]\!]$, $\Omega_2 = [\![P_2]\!]$ and $\Omega_3 = [\![(P_2 \,\text{FILTER}\, C)]\!]$. Then $[\![P]\!]$ is given by the expression $(\Omega_1 \bowtie \Omega_3) \cup (\Omega_1 \setminus \Omega_3)$ where

  $$^{(1)}(\Omega_1 \bowtie \Omega_3) = \{\mu_1 \cup \mu_3 \mid \mu_1 \in \Omega_1, \mu_3 \in \Omega_3 \text{ and } \mu_1 \sim \mu_3\}$$

  and

  $$^{(2)}(\Omega_1 \setminus \Omega_3) = \{\mu_1 \in \Omega_1 \mid \text{ for all } \mu_3 \in \Omega_3, \mu_1 \text{ and } \mu_3 \text{ are not compatible}\}.$$

  Considering that $\Omega_3 = \{\mu_2 \in \Omega_2 \mid \mu_2 \models C\}$. If we redefine (1) by solving $\mu_3 \in \Omega_3$, we will have the set

  $$^{(1.1)}\{\mu_1 \cup \mu_2 \mid \mu_1 \in \Omega_1, \mu_2 \in \Omega_2, \mu_1 \sim \mu_2 \text{ and } (\mu_1 \cup \mu_2) \models C\}.$$

  Additionally, consider to change the universal quantifier in (2) by an existential one. That is $(\Omega_1 \setminus \Omega_3) = \{\mu_1 \in \Omega_1 \mid \nexists \mu_3 \in \Omega_3 \text{ such that } \mu_1 \sim \mu_3\}$. Here we have two cases:

– When $\Omega_3 = \emptyset$. In this case, there exists no mapping $\mu_2 \in \Omega_2$ satisfying that $\mu_2 \models C$. Then this case encodes the set

$$^{(2.1)}\{\mu_1 \in \Omega_1 \mid \text{ for all } \mu_2 \in \Omega_2 \text{ compatible with } \mu_1, (\mu_1 \cup \mu_2) \not\models C\}.$$

– When $\Omega_3 \neq \emptyset$. In this case, for each mapping $\mu_2 \in \Omega_2$ satisfying that $\mu_2 \models C$, it applies that μ_1 and μ_2 are not compatible. Then this case encodes the set

$$^{(2.2)}\{\mu_1 \in \Omega_1 \mid \text{ for all } \mu_2 \in \Omega_2 \text{ such that } \mu_2 \models C, \mu_1 \text{ and } \mu_2 \text{ are not compatible}\}$$

Note that (1.1) corresponds to ($\star$), (2.1) corresponds to the latter set in ($\star\star$), and (2.2) corresponds to the former set in ($\star\star$).

Then we have proved that $\langle\langle P \rangle\rangle_G^D = [\![P]\!]_G^D$.

□

5.7 Expressive Power of SPARQL$_{\text{COMP}}$

In this section we study the expressive power of SPARQL$_{\text{COMP}}$ by comparing it against non recursive safe Datalog with negation (just Datalog from now on). Note that, because SPARQL$_{\text{COMP}}$ and Datalog programs have different type of input and output formats, we have to normalize them to be able to do the comparison.

Following definitions in Section 5.2.4, let $L_s = (\mathcal{Q}_s, \mathcal{D}_s, \mathcal{S}_s, \text{ans}_s)$ be the SPARQL$_{\text{COMP}}$ language, and $L_d = (\mathcal{Q}_d, \mathcal{D}_d, \mathcal{S}_d, \text{ans}_d)$ be the Datalog language. In this comparison we restrict the notion of *SPARQL$_{COMP}$ Query* to a pair (P, D) where P is a graph pattern and D is an RDF dataset.

5.7.1 From SPARQL$_{\text{COMP}}$ to Datalog

To prove that the SPARQL$_{\text{COMP}}$ language $L_s = (\mathcal{Q}_s, \mathcal{D}_s, \mathcal{S}_s, \text{ans}_s)$ is contained in the Datalog language $L_d = (\mathcal{Q}_d, \mathcal{D}_d, \mathcal{S}_d, \text{ans}_d)$, we define transformations $\tau_Q : \mathcal{Q}_s \to \mathcal{Q}_d$, $\tau_D : \mathcal{D}_s \to \mathcal{D}_d$, and $\tau_S : \mathcal{S}_s \to \mathcal{S}_d$. That is, τ_Q transforms a SPARQL$_{\text{COMP}}$ query into a Datalog query, τ_D transforms an RDF dataset into a set of Datalog facts, and τ_S transforms a set of SPARQL$_{\text{COMP}}$ mappings into a set of Datalog substitutions.

RDF datasets as Datalog facts.
Given an RDF dataset $D = \{G_0, \langle u_1, G_1\rangle, \ldots, \langle u_n, G_n\rangle\}$, the transformation $\tau_D(D)$ works as follows: each term t in D is encoded by a fact $iri(t)$, $blank(t)$ or $literal(t)$ when t is an IRI, a blank node or a literal respectively; the set of terms in D is defined by the set of rules $term(X) \leftarrow iri(X)$, $term(X) \leftarrow blank(X)$, and $term(X) \leftarrow literal(X)$; the fact $Null(null)$ encodes the *null* value[8]; each triple (v_1, v_2, v_3) in the default graph G_0 is encoded by a fact

[8]We use the term *null* to represent an unbounded value.

$triple(g_0, v_1, v_2, v_3)$; each named graph $\langle u_i, G_i \rangle$ is encoded by a fact $graph(u)$ and each triple (v_1, v_2, v_3) in G_i is encoded by a fact $triple(u_i, v_1, v_2, v_3)$.

$SPARQL_{COMP}$ mappings as Datalog substitutions.
Given a graph pattern P, an RDF dataset D with default graph G, and the set of mappings $\Omega = [\![P]\!]_G^D$. The transformation $\tau_S(\Omega)$ returns a set of substitutions defined as follows: for each mapping $\mu \in \Omega$ there exists a substitution θ in $\tau_S(\Omega)$ satisfying that, for each $x \in \text{var}(P)$ there exists $x/t \in \theta$ such that $t = \mu(x)$ when $\mu(x)$ is bounded and $t = null$ otherwise.

Graph patterns as Datalog rules.
Let P be a graph pattern to be evaluated against an RDF graph identified by g which occurs in dataset D. We denote by $\delta(P,g)_D$ the function which transforms P into a set of Datalog rules. Table 5.3 shows the transformation rules defined by the function $\delta(P,g)_D$, where:

- The notion of compatible mappings is implemented by the rules:

  $comp(X,X,X) \leftarrow term(X)$,
  $comp(X,null,X) \leftarrow term(X)$
  $comp(null,X,X) \leftarrow term(X)$ and
  $comp(X,X,X) \leftarrow Null(X)$.

- Let $?X, ?Y \in \mathbf{V}$ and $u \in \mathbf{I} \cup \mathbf{L}$. An atomic filter condition C is encoded by a literal L as follows:
  - if C is either $(?X = u)$ or $(?X = ?Y)$ then L is C;
  - if C is isIRI$(?X)$ then L is $iri(?X)$;
  - if C is isLiteral$(?X)$ then L is $literal(?X)$;
  - if C is isBlank$(?X)$ then L is $blank(?X)$;
  - if C is bound$(?X)$ then L is $\neg Null(?X)$.

The transformation follows essentially the intuitive transformation presented by Polleres [193] with the improvement of the necessary code to support faithful translation of bag semantics. Specifically, we changed the transformations for complex filter expressions by simulating them with double negation.

Remark. The semantics of the *error* value is not taken into account in the transformations for filter constrains, i.e., the *error* value is treated as *false*.

$SPARQL_{COMP}$ queries as Datalog queries.
Given a $\text{SPARQL}_{\text{COMP}}$ query $Q = (P,D)$ where P is a graph pattern and D is an RDF dataset. The function $\tau_Q(Q)$ returns the Datalog query $(\Pi, p(\overline{\text{var}}(P)))$ where Π is the Datalog program

$\mathcal{T}_D(D) \cup \delta(P, g_0)_D$, the identifier g_0 references the default graph of D, and p is the goal literal related to P.

The following theorem states that the above transformations work well.

Theorem 5.7.1 *$SPARQL_{COMP}$ is contained in non-recursive safe Datalog with negation.*

Proof. We need to prove that for every $\text{SPARQL}_{\text{COMP}}$ query $Q = (P, D)$ it satisfies that

$$\mathcal{T}_S(\text{ans}_s(Q, D)) = \text{ans}_d(\mathcal{T}_Q(Q), \mathcal{T}_D(D)),$$

where $\text{ans}_s(Q, D)$ denotes the evaluation function $[\![P]\!]^D_{\text{dg}(D)}$.

Considering that $\mathcal{T}_Q(Q)$ is the Datalog query $(\Pi, p(\overline{\text{var}}(P)))$ where Π is the Datalog program $\mathcal{T}_D(D) \cup \delta(P, g_0)_D$. We need to show that for each mapping $\mu \in [\![P]\!]^D_{\text{dg}(G)}$ there exists substitution θ such that $\theta(p(\overline{\text{var}}(P))) \in \text{facts}^*(\Pi)$ and $\theta = \mathcal{T}_S(\mu)$. The proof is by induction on the structure of P.

Base case:

(1) P is a triple pattern (x_1, x_2, x_3).
In this case $\delta(P, g)$ returns the rule $p(\overline{\text{var}}(P)) \leftarrow \mathit{triple}(g, x_1, x_2, x_3)$.
Given a substitution θ, it satisfies that $\theta(p(\overline{\text{var}}(P))) \in \text{facts}^*(\Pi)$ iff there is a substitution $\theta = \{x_i/v_i \mid x_i \in \text{var}(P)\}$ such that $\theta(\mathit{triple}(g, x_1, x_2, x_3)) \in \mathcal{T}_D(D)$.
On the other hand, a mapping μ is in $[\![P]\!]^D_G$ iff $\text{dom}(\mu) = \text{var}(P)$ and $\mu((x_1, x_2, x_3)) = (v_1, v_2, v_3) \in G$. Then $\mu(x_i) = v_i$ when $x_i \in \text{var}(P)$. If we transform μ into a substitution, that is $\mathcal{T}_S(\mu) = \{x_i/v_i \mid x_i \in \text{var}(P)\}$. Then $\theta = \mathcal{T}_S(\mu)$ and we are done.

Inductive case: Let P_1 and P_2 be graph patterns. We consider several cases:

(2) P is $(P_1 \text{ AND } P_2)$.
In this case $\delta(P, g)$ returns the set of rules

$$\{\ p(\overline{\text{var}}(P)) \leftarrow \nu_1(p_1(\overline{\text{var}}(P_1))) \wedge \nu_2(p_2(\overline{\text{var}}(P_2))) \bigwedge_{x \in \text{var}(P_1) \cap \text{var}(P_2)} \mathit{comp}(\nu_1(x), \nu_2(x), x),\ \delta(P_1, g),\ \delta(P_2, g)\ \}$$

where $\text{dom}(\nu_1) = \text{dom}(\nu_2) = \text{var}(P_1) \cap \text{var}(P_2)$ and $\text{range}(\nu_1) \cap \text{range}(\nu_2) = \emptyset$.
Note that we use functions ν_1 and ν_2 to rename common variables between patterns P_1 and P_2, and we use the renamed variables to simulate the notion of compatible mappings through the predicate *comp*.

Given a substitution θ, it satisfies that a fact $\theta(p(\overline{\text{var}}(P)))$ is in $\text{facts}^*(\Pi)$ if and only if $\theta(\nu_1(p_1(\overline{\text{var}}(P_1)))) \in \text{facts}^*(\Pi)$, $\theta(\nu_2(p_2(\overline{\text{var}}(P_2)))) \in \text{facts}^*(\Pi)$, and for each variable $x_i \in \text{var}(P_1) \cap \text{var}(P_2)$, $\theta(\mathit{comp}(\nu_1(x_i), \nu_2(x_i), x_i)) \in \text{facts}^*(\Pi)$ i.e., $\theta(x_i) = \theta(\nu_1(x_i)) =$

$\theta(\nu_2(x_i))$, or $\theta(\nu_1(x_i)) = null$ and $\theta(x_i) = \theta(\nu_2(x_i))$, or $\theta(\nu_2(x_i)) = null$ and $\theta(x_i) = \theta(\nu_1(x_i))$.

On the other hand, a mapping μ is in $[\![(P_1 \text{ AND } P_2)]\!]_G^D$ iff $\mu = \mu_1 \cup \mu_2$ such that $\mu_1 \in [\![P_1]\!]_G^D$, $\mu_2 \in [\![P_2]\!]_G^D$, and μ_1 is compatible with μ_2 i.e, for each $x \in \text{var}(P_1) \cap \text{var}(P_2)$ it applies that $\mu_1(x) = \mu_2(x)$ or $\mu_1(x)$ is unbounded or $\mu_2(x)$ is unbounded.

For induction hypothesis, we have substitutions $\theta_1 = \mathcal{T}_S(\mu_1)$ and $\theta_2 = \mathcal{T}_S(\mu_2)$ such that $\theta_1(p_1(\overline{\text{var}}(P_1))) \in \text{facts}^*(\Pi)$, $\theta_2(p_2(\overline{\text{var}}(P_2))) \in \text{facts}^*(\Pi))$, and for each $x \in \text{var}(P_1) \cap \text{var}(P_2)$ we have that $\theta_1(x) = \theta_2(x)$, or $\theta_1(x)$ is null, or $\theta_2(x)$ is null. Considering that $\mathcal{T}_S(\mu) = \theta_1 \cup \theta_2$ we have that $\theta = \mathcal{T}_S(\mu)$ and we are done.

(3) If P is $(P_1 \text{ UNION } P_2)$.

In this case $\delta(P, g)$ returns the set of rules

$$\{ p(\overline{\text{var}}(P)) \leftarrow p_1(\overline{\text{var}}(P_1)) \bigwedge_{x \in \text{var}(P_2) \wedge x \notin \text{var}(P_1)} Null(x),$$
$$p(\overline{\text{var}}(P)) \leftarrow p_2(\overline{\text{var}}(P_2)) \bigwedge_{x \in \text{var}(P_1) \wedge x \notin \text{var}(P_2)} Null(x),$$
$$\delta(P_1, g), \delta(P_2, g) \}$$

Given a substitution θ, it satisfies that $\theta(p(\overline{\text{var}}(P))) \in \text{facts}^*(\Pi)$ iff either

(a) $\theta(p_1(\overline{\text{var}}(P_1))) \in \text{facts}^*(\Pi)$ and x is null for each $x \in \text{var}(P) \setminus \text{var}(P_1)$, i.e, $\theta = \{x/v \mid x \in \text{var}(P_1)\} \cup \{x/null \mid x \in \text{var}(P) \setminus \text{var}(P_1)\}$; or

(b) $\theta(p_2(\overline{\text{var}}(P_2))) \in \text{facts}^*(\Pi)$ and x is null for each $x \in \text{var}(P) \setminus \text{var}(P_2)$, i.e, $\theta = \{x/v \mid x \in \text{var}(P_2)\} \cup \{x/null \mid x \in \text{var}(P) \setminus \text{var}(P_2)\}$.

On the other hand, a mapping μ is in $[\![(P_1 \text{ UNION } P_2)]\!]_G^D$ iff either (a) $\mu = \mu_1 \in [\![P_1]\!]_G^D$ or (b) $\mu = \mu_2 \in [\![P_2]\!]_G^D$. For induction hypothesis, there are substitutions $\theta_1 = \mathcal{T}_S(\mu_1)$ and $\theta_2 = \mathcal{T}_S(\mu_2)$ satisfying that $\theta_1(p_1(\overline{\text{var}}(P_1))) \in \text{facts}^*(\Pi)$ and $\theta_2(p_2(\overline{\text{var}}(P_2))) \in \text{facts}^*(\Pi)$. Assuming that $\theta_1 = \{x/v \mid x \in \text{var}(P_1)\}$ and $\theta_2 = \{x/v \mid x \in \text{var}(P_2)\}$, it holds that $\mathcal{T}_S(\mu) = \theta_1 \cup \{x/null \mid x \in \text{var}(P) \setminus \text{var}(P_1)\}$ in case (a), and $\mathcal{T}_S(\mu) = \theta_2 \cup \{x/null \mid x \in \text{var}(P) \setminus \text{var}(P_2)\}$ in case (b). Then, we have that $\theta = \mathcal{T}_S(\mu)$ and we are done.

(4) P is $(P_1 \text{ OPT } P_2)$.

In this case $\delta(P, g)$ returns the set of rules

$$\{ p(\overline{\text{var}}(P)) \leftarrow p_3(\overline{\text{var}}(P_3)),$$
$$p(\overline{\text{var}}(P)) \leftarrow p_1(\overline{\text{var}}(P_1)) \wedge \neg p_1'(\overline{\text{var}}(P_1)) \bigwedge_{x \in \text{var}(P_2) \wedge x \notin \text{var}(P_1)} Null(x),$$
$$p_1'(\overline{\text{var}}(P_1)) \leftarrow p_3(\overline{\text{var}}(P_3)),$$
$$\delta(P_1, g), \delta(P_2, g), \delta(P_3, g) \},$$

where $P_3 = (P_1 \text{ AND } P_2)$.

Given a substitution θ, we have that $\theta(p(\overline{\text{var}}(P))) \in \text{facts}^*(\Pi)$ iff either

(i) $\theta(p_3(\overline{\text{var}}(P_3))) \in \text{facts}^*(\Pi)$; or

(ii) $\theta(p_1(\overline{\text{var}}(P_1))) \in \text{facts}^*(\Pi)$ and is false that $\theta(p_1'(\overline{\text{var}}(P_1))) \in \text{facts}^*(\Pi)$; that is, if $\theta = \theta_1$ such that $\theta_1(p_1(\overline{\text{var}}(P_1))) \in \text{facts}^*(\Pi)$, then for all $\theta_2(p_2(\overline{\text{var}}(P_2))) \in \text{facts}^*(\Pi)$ it is false that $comp(\theta_1(x), \theta_2(x), \theta(x))$, i.e., it applies that $\theta_1(x) \neq \theta_2(x)$ for each variable $x \in \text{var}(P_1) \cap \text{var}(P_2)$. In this case, $\theta(x)$ is null for each variable $x \in \text{var}(P) \setminus \text{var}(P_1)$.

On the other hand, a mapping μ is in $[\![(P_1 \,\text{OPT}\, P_2)]\!]_G^D$ iff either:

(a) $\mu \in [\![P_3]\!]_G^D$ where $P_3 = (P_1 \,\text{AND}\, P_2)$; or

(b) $\mu = \mu_1 \in [\![P_1]\!]_G^D$ such that for all $\mu_2 \in [\![P_2]\!]_G^D$ it satisfies that μ_1 and μ_2 are not compatible. Here $\mu(x)$ is unbounded for each $x \in \text{var}(P) \setminus \text{var}(P_1)$.

For induction hypothesis, we have substitutions $\theta_1 = \tau_S(\mu_1)$ and $\theta_2 = \tau_S(\mu_2)$ satisfying that $\theta_1(p_1(\overline{\text{var}}(P_1))) \in \text{facts}^*(\Pi)$ and $\theta_2(p_2(\overline{\text{var}}(P_2))) \in \text{facts}^*(\Pi)$.

Suppose that $\theta' = \tau_S(\mu)$. Following definition of μ, we have that:

- In case (a), $\theta'(p_3(\overline{\text{var}}(P_3))) \in \text{facts}^*(\Pi)$ (as was showed in (2)).
- In case (b), $\theta' = \theta_1$ and θ_1 is not compatible with every θ_2, that is $\theta_1(x) \neq \theta_2(x)$ for each variable $x \in \text{var}(P_1) \cap \text{var}(P_2)$. Additionally, $x/null \in \theta'$ for each $x \in \text{var}(P) \setminus \text{var}(P_1)$.

Considering that (a) corresponds to (i), and (b) corresponds to (ii), then $\theta = \theta' = \tau_S(\mu)$ and we are done.

(5) P is $(u \,\text{GRAPH}\, P_1)$ where $u \in \mathbf{I}$.
In this case $\delta(P, g)$ returns the set of rules

$$\{\ p(\overline{\text{var}}(P)) \leftarrow p_1(\overline{\text{var}}(P_1)),\ \delta(P_1, u)\ \}.$$

Given a substitution θ, we have that $\theta(p(\overline{\text{var}}(P))) \in \text{facts}^*(\tau_D(D) \cup \delta(P, g))$ if and only if $\theta(p_1(\overline{\text{var}}(P_1))) \in \text{facts}^*(\tau_D(D) \cup \delta(P_1, u))$. On the other hand, a mapping μ is in $[\![P]\!]_G^D$ if and only if $\mu \in [\![P_1]\!]_{G'}^D$ such that $G' = \text{gr}(u)_D$. In both cases, the active graph identified g has been changed by the graph identified u. Then by induction hypothesis we have that $\theta = \tau_S(\mu)$.

(6) P is $(?X \,\text{GRAPH}\, P_1)$ where $?X \in \mathbf{V}$.
In this case, for each named graph identified u_i in dataset D, we have that $\delta(P, g)$ returns the set of rules:

$$\{\ p(\overline{\text{var}}(P)) \leftarrow p_{1i}(\overline{\text{var}}(P_{1i}))\ \wedge\ graph(?X) \wedge ?X = u_i,\\ \delta(P_{1i}, u_i)_D\ \}.$$

Considering that P_{1i} is a copy of P_1 and using result (5), we can prove that the rule $p(\overline{\text{var}}(P)) \leftarrow p_{1i}(\overline{\text{var}}(P_{1i}))$ is correct for each named graph u_i in dataset D. Additionally, given that $\text{var}(P)$ is $?X \cup \text{var}(P_{1i})$, we use the literals $graph(?X)$ and $?X = u_i$ to assign the respective IRI u_i to variable $?X$, then we are changing the active graph to the graph identified by u_i. As result, a substitution θ is in $\delta(P,g)$ iff θ is a substitution for a some $\delta(P_{1i}, u_i)$ where u_i identifies a graph in D. Then we have proved the case.

(7) If P is $(P_1 \text{ FILTER } C)$ and C is an atomic filter constraint.
In this case $\delta(P,g)$ returns the set of rules

$$\{\ p(\overline{\text{var}}(P)) \leftarrow p_1(\overline{\text{var}}(P_1)) \wedge cond,\\ \delta(P_1, g)\ \},$$

where *cond* is a Datalog literal encoding the filter condition C.
Given a substitution θ, it holds that $\theta(p(\overline{\text{var}}(P)))$ is in $\text{facts}^*(\Pi)$ iff $\theta(p_1(\overline{\text{var}}(P_1))) \in \text{facts}^*(\Pi)$ and $\theta(cond)$ is true.
On the other hand, a mapping μ is in $[\![P]\!]_G^D$ iff $\mu \in [\![P_1]\!]_G^D$ and μ satisfies C.
By induction hypothesis and considering that *cond* is a Datalog literal equivalent to C, it applies that there exists substitution $\theta = \tau_S(\mu)$ satisfying that $\theta(p_1(\overline{\text{var}}(P_1))) \in \text{facts}^*(\Pi)$ and $\theta(cond)$ is true.

(8) If P is $(P_1 \text{ FILTER } C)$ and C is $(\neg(C_1))$.
In this case $\delta(P,g)$ returns the set of rules

$$\{\ p(\overline{\text{var}}(P)) \leftarrow p_1(\overline{\text{var}}(P_1)) \wedge \neg p_4(\overline{\text{var}}(P_1)),\\ \delta(P_1, g), \delta(P_4, g)\ \},$$

where $P_4 = (P_1 \text{ FILTER } C_1)$.
Given a substitution θ, it holds that $\theta(p(\overline{\text{var}}(P)))$ is in $\text{facts}^*(\Pi)$ iff $\theta(p_1(\overline{\text{var}}(P_1))) \in \text{facts}^*(\Pi)$ and is false that $\theta(p_4(\overline{\text{var}}(P_1))) \in \text{facts}^*(\Pi)$. The last condition implies that, if $cond_1$ is the Datalog literal encoding C_1 then, $\theta(cond_1)$ is not true.
On the other hand, we have that a mapping μ is in $[\![P]\!]_G^D$ if and only if $\mu \in [\![P_1]\!]_G^D$ and it is not true that $\mu \models C_1$.
By induction hypothesis and considering that $cond_1$ is the Datalog literal equivalent to C_1, we have that there exists substitution $\theta = \tau_S(\mu)$ satisfying that $\theta(p_1(\overline{\text{var}}(P_1))) \in \text{facts}^*(\Pi)$ and $\theta(cond_1)$ is not true.

(9) If P is $(P_1 \text{ FILTER } C)$ and C is $(C_1 \wedge C_2)$.
In this case $\delta(P,g)$ returns the set of rules

$$\{\ p(\overline{\text{var}}(P)) \leftarrow p_1(\overline{\text{var}}(P_1)) \wedge \neg p'(\overline{\text{var}}(P_1)),$$
$$p'(\overline{\text{var}}(P_1)) \leftarrow p_1(\overline{\text{var}}(P_1)) \wedge \neg p''(\overline{\text{var}}(P_1)),$$
$$p''(\overline{\text{var}}(P_1)) \leftarrow p_4(\overline{\text{var}}(P_1)) \wedge p_5(\overline{\text{var}}(P_1)),$$
$$\delta(P_1,g),\ \delta(P_4,g),\ \delta(P_5,g)\ \},$$

where $P_4 = (P_1 \text{ FILTER} C_1)$ and $P_5 = (P_1 \text{ FILTER} C_2)$.
Note that the graph pattern $(P_1 \text{ FILTER}(C_1 \wedge C_2))$ can be rewritten as the graph pattern $((P_1 \text{ FILTER} C_1) \text{ AND} (P_1 \text{ FILTER} C_2))$ (it is showed in the rule for predicate p'' and by the patterns P_4 and P_5). This transformation is true under set-semantics, but it fails when we consider bag-semantics because it duplicates the bag of solutions. To solve this problem, we consider a double negation of the filter condition, that is we rewrite C to $(\neg(\neg C))$ (as is showed by the rules for predicates p and p'). Given that negated literals does not increase solutions, we will have only solutions from predicate p_1. Then we have proved the case.

(10) If P is $(P_1 \text{ FILTER} C)$ and C is $(C_1 \vee C_2)$.
In this case $\delta(P,g)$ returns the set of rules

$$\{\ p(\overline{\text{var}}(P)) \leftarrow p_1(\overline{\text{var}}(P_1)) \wedge \neg p'(\overline{\text{var}}(P_1)),$$
$$p'(\overline{\text{var}}(P_1)) \leftarrow p_1(\overline{\text{var}}(P_1)) \wedge \neg p''(\overline{\text{var}}(P_1)),$$
$$p''(\overline{\text{var}}(P_1)) \leftarrow p_4(\overline{\text{var}}(P_1)),$$
$$p''(\overline{\text{var}}(P_1)) \leftarrow p_5(\overline{\text{var}}(P_1)),$$
$$\delta(P_1,g),\ \delta(P_4,g),\ \delta(P_5,g)\ \}$$

where $P_4 = (P_1 \text{ FILTER} C_1)$ and $P_5 = (P_1 \text{ FILTER} C_2)$.
Note that the graph pattern $(P_1 \text{ FILTER}(C_1 \vee C_2))$ can be rewritten as the graph pattern $((P_1 \text{ FILTER} C_1) \text{ UNION} (P_1 \text{ FILTER} C_2))$ (it is showed by the rules for predicate p'' and by the patterns P_4 and P_5). Similar to (9), we apply a double negation of the filter condition C (as is showed by the rules for predicates p and p') to solve the problem for bag-semantics. This proved the case.

□

Note. Given a graph pattern P, the transformation $\delta(P,g)$ preserves the bag semantics of the SPARQL WG specification. Consider the cardinality m of a solution s for P (and the equivalent solution for $\delta(P,g)$). It can be checked that:

- in case (1), the value of m is 1 because each triple occurs once in the active graph;
- in case (2), m is the product of the cardinalities for s in the bags for $\langle\langle P_1 \rangle\rangle$ and $\langle\langle P_2 \rangle\rangle$;
- in case (3), m is the sum of the cardinalities for s in the bags for $\langle\langle P_1 \rangle\rangle$ and $\langle\langle P_2 \rangle\rangle$;

- in case (4), m is given by either the product of cardinalities for s in the bags for $\langle\langle P_1 \rangle\rangle$ and $\langle\langle P_2 \rangle\rangle$, or the cardinalities for s in the bag for $\langle\langle P_1 \rangle\rangle$;
- in case (5), m is given by the cardinality of s in the bag of solutions for named graph u;
- in case (6), m is given by the sum of cardinalities for s in the bag for each named graph in the dataset;
- in cases (7), (8), (9) and (10), m is given by the cardinality of s in the bag for P_1.

5.7.2 From Datalog to SPARQL$_{\text{COMP}}$

To prove that the Datalog language $L_d = (\mathcal{Q}_d, \mathcal{D}_d, \mathcal{S}_d, \text{ans}_d)$ is contained in the SPARQL$_{\text{COMP}}$ language $L_s = (\mathcal{Q}_s, \mathcal{D}_s, \mathcal{S}_s, \text{ans}_s)$, we define transformations $\tau'_Q : \mathcal{Q}_d \to \mathcal{Q}_s$, $\tau'_D : \mathcal{D}_d \to \mathcal{D}_s$, and $\tau'_S : \mathcal{S}_d \to \mathcal{S}_s$. That is, τ'_Q transforms a Datalog query into an SPARQL$_{\text{COMP}}$ query, τ'_D transforms a set of Datalog facts into an RDF dataset, and τ'_S transforms a set of Datalog substitutions into a set of SPARQL$_{\text{COMP}}$ mappings.

Datalog facts as an RDF Dataset.
Given a Datalog fact $f = p(c_1, ..., c_n)$, define function desc(f) which returns the set of triples { (_:b,predicate,p), (_:b,rdf:_1,c_1),...,(_:b,rdf:_n,c_n) }, where _:b is a fresh blank node. Given a set of Datalog facts F, the function $\tau'_D(F)$ returns an RDF dataset with default graph $G_0 = \{\text{desc}(f) \mid f \in F\}$, where $\text{blank}(\text{desc}(f_i)) \cap \text{blank}(\text{desc}(f_j)) = \emptyset$ for each $f_i, f_j \in F$ with $i \neq j$.

Datalog substitutions as SPARQL$_{COMP}$ mappings.
Given a set of substitutions Θ, the transformation $\tau'_S(\Theta)$ returns a set of mappings defined as follows: for each substitution $\theta \in \Theta$ there exists a mapping $\mu \in \tau'_S(\Theta)$ satisfying that, if $x/t \in \theta$ then $x \in \text{dom}(\mu)$ and $\mu(x) = t$.

Datalog rules as SPARQL$_{COMP}$ graph patterns.
Let Π be a Datalog program, and L be a literal $p(x_1, \ldots, x_n)$ where p is a predicate in Π and each x_i is a variable. We define the function $\text{gp}(L)_\Pi$ which returns a graph pattern encoding the program (Π, L), that is, the fragment of the program Π used for evaluating literal L.

The translation works intuitively as follows:

(a) If predicate p is extensional, then $\text{gp}(L)_\Pi$ returns the graph pattern
$((?Y, \text{predicate}, p)\,\text{AND}(?Y, \text{rdf:\_1}, x_1)\,\text{AND}\cdots\text{AND}(?Y, \text{rdf\_n}, x_n))$,
where $?Y$ is a fresh variable.

(b) If predicate p is intensional, then for each rule in Π of the form

$$L \leftarrow L_1 \wedge \cdots \wedge L_s \wedge \neg L_{s+1} \wedge \cdots \wedge \neg L_t \wedge L_1^{eq} \wedge \cdots \wedge L_u^{eq},$$

where each L_i is a predicate formula and each L_k^{eq} is a literal either of the form $t_1 = t_2$ or $\neg(t_1 = t_2)$, it applies that $\text{gp}(L)_\Pi$ returns a graph pattern with the structure

$$(((\cdots((\text{gp}(L_1)_\Pi \,\text{AND}\cdots\text{AND}\,\text{gp}(L_s)_\Pi) \\ \text{MINUS}\,\text{gp}(L_{s+1})_\Pi)\cdots)\,\text{MINUS}\,\text{gp}(L_t)_\Pi) \\ \text{FILTER}(L_1^{eq}\wedge\cdots\wedge L_u^{eq})). \quad (5.2)$$

The formal definition of $\text{gp}(L)_\Pi$ is Algorithm 3.

Datalog queries as SPARQL$_{COMP}$ queries.
Given a Datalog query $Q = (\Pi, L)$ where Π is a Datalog program and L is the goal literal. The function $\tau'_Q(Q)$ returns the SPARQL$_{\text{COMP}}$ query (P, D) where P is the graph pattern $\text{gp}(L)_\Pi$ and D is an RDF dataset with default graph $G_0 = \tau'_D(\text{facts}(\Pi))$.

The following theorem states that the above transformations work well.

Theorem 5.7.2 *nr-Datalog$^\neg$ is contained in SPARQL$_{COMP}$.*

Proof. We need to prove that for every Datalog query $Q = (\Pi, L)$ it satisfies that:

$$\tau'_S(\text{ans}_d(Q, \text{facts}(\Pi)) = \text{ans}_s(\tau'_Q(Q), \tau'_D(\text{facts}(\Pi))).$$

Considering that $\text{ans}_s(\cdots)$ denotes function $[\![\cdot]\!]$, we will show that $\tau'_S(\text{ans}_d(Q, \text{facts}(\Pi)) = [\![\text{gp}(L)_\Pi]\!]^D_{\text{dg}(D)}$ where $\text{dg}(D) = \tau'_D(\text{facts}(\Pi))$.

The proof is by induction on the level l of the program (Π, L). The level of a program (Π, L) is the number $l(L)$ where: $l(\neg L) = l(L)$; $l(L) = 0$ if L contains an extensional predicate; $l(L) = 1 + \max_i(l(L_i))$ if L contains an intensional predicate and L_i are all literals which occur in the body of any rule with head L. (Note that the function is well defined because the Datalog programs considered are not recursive.)

Base case: $l(\Pi, L) = 0$.
Let $L = p(x_1, \ldots, x_n)$. In this case p is extensional and L matches line 4 of Algorithm 3. Hence $\text{gp}(L)_\Pi$ returns the graph pattern

$$P = ((?Y, \text{predicate}, p)\,\text{AND}(?Y, \text{rdf:\_1}, x_1)\,\text{AND}\cdots\text{AND}(?Y, \text{rdf:\_n}, x_n)).$$

Now, a mapping μ is in $[\![P]\!]^D_{\text{dg}(D)}$ if and only if for every triple pattern t in P it satisfies that $\mu(t) \in \text{dg}(D)$.

On the other hand, a substitution θ is in $\text{ans}_d((\Pi, L), \text{facts}(\Pi))$ if and only if $\theta(L) \in \text{facts}(\Pi)$ (Note that we only consider the initial database $\text{facts}(\Pi)$ because predicate p is extensional).

Note that τ'_S maps bijectively substitutions from $\text{ans}_d((\Pi, L), \text{facts}(\Pi))$ to mappings in $[\![\text{gp}(L)_\Pi]\!]^D_{\text{dg}(D)}$. Specifically, for each variable $v \in L$ it satisfies that $\theta(v) = \mu(v)$. This proves the basic case.

Inductive step: $l(\Pi, L) = n > 0$.

Recall that $L = p(x_1, \ldots, x_n)$ and assume that Π_p denotes the set of rules of Π having predicate p in the head. In this case, L matches line 7 of Algorithm 3 and $\mathrm{gp}(L)_\Pi$ returns the pattern

$$(\ \mathrm{gp}(L^{r_1})_\Pi \,\mathrm{UNION} \cdots \mathrm{UNION}\, \mathrm{gp}(L^{r_m})_\Pi \ ), \tag{5.3}$$

where $\mathrm{gp}(L^{r_i})_\Pi$ returns the graph pattern corresponding to rule $r_i \in \Pi_p$. In this case it clearly holds that $[\![\mathrm{gp}(L)_\Pi]\!]^D_{\mathrm{dg}(D)} = \bigcup_i [\![\mathrm{gp}(L^{r_i})_\Pi]\!]^D_{\mathrm{dg}(D)}$.

On the other hand, a substitution θ is in $\mathrm{ans}_d((\Pi, L), \mathrm{facts}(\Pi))$ iff there is a rule $r_i \in \Pi_p$ such that $\theta'(r_i)$ is true in Π. Considering (5.3), it is enough to prove that for each particular rule $r_i \in \Pi_p$ it satisfies that:

$$\mathcal{T}'_S(\mathrm{ans}((\Pi, L^{r_i}), \mathrm{facts}(\Pi))) = [\![\mathrm{gp}(L^{r_i})]\!]^D_{\mathrm{dg}(D)}. \tag{5.4}$$

To prove this, assume that the rule r_i has the following general structure:

$$L \ \leftarrow \ L_1 \ \wedge \ \cdots \ \wedge \ L_s \ \wedge \ \neg L_{s+1} \ \wedge \ \cdots \ \wedge \ \neg L_t \ \wedge \ L_1^{eq} \ \wedge \ \cdots \ \wedge \ L_u^{eq}, \tag{5.5}$$

where each L_j is a predicate formula (positive or negative) and each L_k^{eq} is a literal of the form $t_1 = t_2$ or $\neg(t_1 = t_2)$.

Let us compute the $\mathrm{SPARQL}_{\mathrm{COMP}}$ evaluation first. We have that $\mathrm{gp}(L)_\Pi$ returns a graph pattern with the structure

$$\begin{aligned}(((\cdots((\mathrm{gp}(L_1)_\Pi \,\mathrm{AND} \cdots \mathrm{AND}\, \mathrm{gp}(L_s)_\Pi) \\ \mathrm{MINUS}\, \mathrm{gp}(L_{s+1})_\Pi) \cdots) \,\mathrm{MINUS}\, \mathrm{gp}(L_t)_\Pi) \\ \mathrm{FILTER}(L_1^{eq} \wedge \cdots \wedge L_u^{eq})),\end{aligned} \tag{5.6}$$

Observe that a mapping μ is in $[\![\mathrm{gp}(L)_\Pi]\!]^D_{\mathrm{dg}(D)}$ if and only if:

(i) for each L_i with $1 \leq i \leq s$, there exists a mapping $\mu'_i \in [\![\mathrm{gp}(L_i)_\Pi]\!]^D_{\mathrm{dg}(D)}$ satisfying that μ and μ'_i are compatible;

(ii) for each L_j with $s < j \leq t$, there exists no mapping $\mu''_j \in [\![\mathrm{gp}(L_j)_\Pi]\!]^D_{\mathrm{dg}(D)}$ satisfying that μ and μ''_j are compatible; and

(iii) for each literal L_k^{eq}, it satisfies that $\mu(t_1) = \mu(t_2)$ when L_k^{eq} is $t_1 = t_2$, and $\mu(t_1) \neq \mu(t_2)$ when L_k^{eq} is $\neg(t_1 = t_2)$ (suppose that $\mu(t_i) = t_i$ where t_i is a constant).

Now, let us compute the Datalog evaluation. A substitution θ is in $\mathrm{ans}_d((\Pi, L), \mathrm{facts}(\Pi))$ if and only if $\theta(L) \in \mathrm{facts}^*(\Pi)$. This means that:

(a) for each L_i with $1 \leq i \leq s$, there is a substitution θ'_i in $\mathrm{ans}_d((\Pi, L_i), \mathrm{facts}(\Pi))$ satisfying that $\theta(x) = \theta'(x)$ for each variable $x \in \mathrm{var}(\theta') \cap \mathrm{var}(\theta'_i)$, .

(b) for each L_j with $s < j \leq t$, there is not substitution θ''_j in $\text{ans}_d((\Pi, L_j), \text{facts}(\Pi))$ satisfying that $\theta(x) = \theta''(x)$ for each variable $x \in \text{var}(\theta) \cap \text{var}(\theta''_j)$.

(c) for each literal L_k^{eq}, it satisfies that $\theta'(t_1) = \theta'(t_2)$ when L_k^{eq} is $t_1 = t_2$, and $\theta'(t_1) \neq \theta'(t_2)$ when L_k^{eq} is $\neg(t_1 = t_2)$ (assume that $\theta'(t_i) = t_i$ where t_i is a constant).

Note that, because Π is not recursive, for each pair of literal L_i, L_j in rule r_i, it holds that $l(\Pi, L_i) < l(\Pi, L)$ and $l(\Pi, L_j) < l(\Pi, L)$.
Hence, by induction hypothesis:

$$\mathcal{T}'_S(\text{ans}_d((\Pi, L_i), \text{facts}(\Pi))) = [\![\text{gp}(L_i)_\Pi]\!]^D_{\text{dg}(D)} \text{ and}$$
$$\mathcal{T}'_S(\text{ans}_d((\Pi, L_j), \text{facts}(\Pi))) = [\![\text{gp}(L_j)_\Pi]\!]^D_{\text{dg}(D)}.$$

These identities plus the conditions (i), (ii), (iii) and (a), (b), (c) above, show the bijections between maps $\mu \in [\![\text{gp}(L)_\Pi]\!]^D_{\text{dg}(D)}$ and substitutions $\theta \in \text{ans}((\Pi, L), D_d)$, that is:

$$\mathcal{T}'_S(\text{ans}((\Pi, L), \text{facts}(\Pi))) = [\![\text{gp}(L)_\Pi]\!]^D_{\text{dg}(D)}.$$

This concludes the proof.

□

5.8 Conclusions

In this chapter we have studied in depth the expressive power of SPARQL. Among the most important findings are: the definition of negation, the proof that non-safe filter patterns are superfluous, the proof of the equivalence between the operational (W3C) and a compositional semantics of SPARQL, and the proof that the compositional version of SPARQL and non-recursive safe Datalog with negation are equivalent in their expressive power.

From these results we can state the most relevant result of this chapter:

Theorem 5.8.1 (Main) *SPARQL has the same expressive power as Relational Algebra under bag semantics.*

This result follows from the well known fact (for example, see the works of Abiteboul et al. [189], Levene and Loizou [190]) that relational algebra and non-recursive safe Datalog with negation have the same expressive power, and from theorems 5.5.2, 5.6.1, 5.7.1 and 5.7.2.

Relational Algebra is probably one of the most studied query languages, and has become a favorite by theoreticians because of a proper balance between expressiveness and complexity. Hence, the result that SPARQL is equivalent in its expressive power to Relational Algebra has important implications. Particularly, we can say that, as Relational Algebra, SPARQL: is essentially equivalent in expressive power to relational calculus; is powerful enough for most

practical purposes; and, one of the goals of the study, it does not support classic graph queries (e.g. transitive closure).

Future work includes to propose some extensions to the language which take advantage of this result. Particularly, the support for nested queries in SPARQL. Also, after these results remain open the goal of extending SPARQL with features for graph-like query capabilities.

Algorithm 3 Transformation of Datalog rules into $\text{SPARQL}_{\text{COMP}}$ graph patterns

1: //Input: a literal $L = p(x_1, \ldots, x_n)$ and a Datalog program Π
2: //Output: a $\text{SPARQL}_{\text{COMP}}$ graph pattern $P = \text{gp}(L)_\Pi$
3: $P \leftarrow \emptyset$
4: **if** predicate p is extensional in Π **then**
5: Let $?Y$ be a fresh variable
6: $P \leftarrow ((?Y, \text{predicate}, p)\,\text{AND}(?Y, \text{rdf:\_1}, x_1)\,\text{AND}\cdots\text{AND}(?Y, \text{rdf\_n}, x_n))$
7: **else if** predicate p is intensional in Π **then**
8: **for** each rule $r \in \Pi$ with head $p(x'_1, \ldots, x'_n)$ **do**
9: $P' \leftarrow \emptyset$
10: $C \leftarrow \emptyset$
11: Let $r' = \nu(r)$ where ν is a substitution such that $\nu(x'_i) = x_i$
12: **for** each positive literal $q(y_1, \ldots, y_m)$ in the body of r' **do**
13: **if** $P' = \emptyset$ **then** $P' \leftarrow \text{gp}(q)_\Pi$
14: **else** $P' \leftarrow (P'\,\text{AND}\,\text{gp}(q)_\Pi)$
15: **end for**
16: **for** each negative literal $\neg q(y_1, \ldots, y_m)$ in the body of r' **do**
17: $P' \leftarrow (P'\,\text{MINUS}\,\text{gp}(q))$
18: **end for**
19: **for** each equality formula $t_1 = t_2$ in r' **do**
20: **if** $C = \emptyset$ **then** $C \leftarrow (t_1 = t_2)$
21: **else** $C \leftarrow C \wedge (t_1 = t_2)$
22: **end for**
23: **for** each negative literal $\neg(t_1 = t_2)$ in r' **do**
24: **if** $C = \emptyset$ **then** $C \leftarrow \neg(t_1 = t_2)$
25: **else** $C \leftarrow C \wedge \neg(t_1 = t_2)$
26: **end for**
27: **if** $C \neq \emptyset$ **then** $P' \leftarrow (P'\,\text{FILTER}\,C)$
28: **if** $P = \emptyset$ **then** $P \leftarrow P'$
29: **else** $P \leftarrow (P\,\text{UNION}\,P')$
30: **end for**
31: **end if**
32: **return** P

Pattern P	$\delta(P,g)_D$
(x_1,x_2,x_3)	$p(\overline{\text{var}}(P)) \leftarrow triple(g,x_1,x_2,x_3)$
$(P_1 \text{ AND } P_2)$	$p(\overline{\text{var}}(P)) \leftarrow \nu_1(p_1(\overline{\text{var}}(P_1))) \wedge \nu_2(p_2(\overline{\text{var}}(P_2)))$ $\bigwedge_{x\in\text{var}(P_1)\cap\text{var}(P_2)} comp(\nu_1(x),\nu_2(x),x)$, $\delta(P_1,g)_D$, $\delta(P_2,g)_D$
	$\text{dom}(\nu_1)=\text{dom}(\nu_2)=\text{var}(P_1)\cap\text{var}(P_2)$, $\text{range}(\nu_1)\cap\text{range}(\nu_2)=\emptyset$.
$(P_1 \text{ UNION } P_2)$	$p(\overline{\text{var}}(P)) \leftarrow p_1(\overline{\text{var}}(P_1)) \bigwedge_{x\in\text{var}(P_2)\wedge x\notin\text{var}(P_1)} Null(x)$, $p(\overline{\text{var}}(P)) \leftarrow p_2(\overline{\text{var}}(P_2)) \bigwedge_{x\in\text{var}(P_1)\wedge x\notin\text{var}(P_2)} Null(x)$, $\delta(P_1,g)_D$, $\delta(P_2,g)_D$
$(P_1 \text{ OPT } P_2)$	$p(\overline{\text{var}}(P)) \leftarrow p_3(\overline{\text{var}}(P_3))$, $p(\overline{\text{var}}(P)) \leftarrow p_1(\overline{\text{var}}(P_1)) \wedge \neg p_1'(\overline{\text{var}}(P_1)) \bigwedge_{x\in\text{var}(P_2)\wedge x\notin\text{var}(P_1)} Null(x)$, $p_1'(\overline{\text{var}}(P_1)) \leftarrow p_3(\overline{\text{var}}(P_3))$, $\delta(P_1,g)_D$, $\delta(P_2,g)_D$, $\delta(P_3,g)_D$
$(u \text{ GRAPH } P_1)$ and $u \in \mathbf{I}$	$p(\overline{\text{var}}(P)) \leftarrow p_1(\overline{\text{var}}(P_1))$, $\delta(P_1,u)_D$
$(?X \text{ GRAPH } P_1)$ and $?X \in \mathbf{V}$	$p(\overline{\text{var}}(P)) \leftarrow p_{11}(\overline{\text{var}}(P_{11})) \wedge graph(?X) \wedge ?X = u_1$, $\delta(P_{11},u_1)_D$, $\ldots$ $p(\overline{\text{var}}(P)) \leftarrow p_{1n}(\overline{\text{var}}(P_{1n})) \wedge graph(?X) \wedge ?X = u_n$, $\delta(P_{1n},u_n)_D$
$(P_1 \text{ FILTER } C)$ C is atomic	$p(\overline{\text{var}}(P)) \leftarrow p_1(\overline{\text{var}}(P_1)) \wedge cond$ $\delta(P_1,g)_D$
$(P_1 \text{ FILTER } C)$ C is $(\neg(C_1))$	$p(\overline{\text{var}}(P)) \leftarrow p_1(\overline{\text{var}}(P_1)) \wedge \neg p_4(\overline{\text{var}}(P_1))$, $\delta(P_1,g)_D$, $\delta(P_4,g)_D$
$(P_1 \text{ FILTER } C)$ C is $(C_1 \wedge C_2)$	$p(\overline{\text{var}}(P)) \leftarrow p_1(\overline{\text{var}}(P_1)) \wedge \neg p'(\overline{\text{var}}(P_1))$, $p'(\overline{\text{var}}(P_1)) \leftarrow p_1(\overline{\text{var}}(P_1)) \wedge \neg p''(\overline{\text{var}}(P_1))$, $p''(\overline{\text{var}}(P_1)) \leftarrow p_4(\overline{\text{var}}(P_1)) \wedge p_5(\overline{\text{var}}(P_1))$, $\delta(P_1,g)_D$, $\delta(P_4,g)_D$, $\delta(P_5,g)_D$
$(P_1 \text{ FILTER } C)$ C is $(C_1 \vee C_2)$	$p(\overline{\text{var}}(P)) \leftarrow p_1(\overline{\text{var}}(P_1)) \wedge \neg p'(\overline{\text{var}}(P_1))$, $p'(\overline{\text{var}}(P_1)) \leftarrow p_1(\overline{\text{var}}(P_1)) \wedge \neg p''(\overline{\text{var}}(P_1))$ $p''(\overline{\text{var}}(P_1)) \leftarrow p_4(\overline{\text{var}}(P_1))$, $p''(\overline{\text{var}}(P_1)) \leftarrow p_5(\overline{\text{var}}(P_1))$, $\delta(P_1,g)_D$, $\delta(P_4,g)_D$, $\delta(P_5,g)_D$

Table 5.3: Transforming $\text{SPARQL}_{\text{COMP}}$ graph patterns into Datalog Rules. D is a dataset having active graph identified by g. $\overline{\text{var}}(P)$ denotes the tuple of variables obtained from a lexicographical ordering of the variables in the graph pattern P. Each p_i is a predicate identifying the graph pattern P_i. If L is a literal, then $\nu_j(L)$ denotes a copy of L with its variables renamed according to a variable renaming function $\nu_j : \mathbf{V} \rightarrow \mathbf{V}$. *cond* is a literal encoding the filter condition C. Each P_{1i} is a copy of P_1 and $u_i \in \text{names}(D)$. $P_3 = (P_1 \text{ AND } P_2)$, $P_4 = (P_1 \text{ FILTER } C_1)$ and $P_5 = (P_1 \text{ FILTER } C_2)$.

Chapter 6

A Graph Visualization Model for RDF

Data visualization involves selecting, transforming and representing data in a form that facilitates human interaction and understanding. A graph visualization model provides an abstraction mechanism based on a graph structure.

The main objective of this chapter is the definition of a nested graph model for visualizing RDF data (a nested graph is a graph whose nodes can also be graphs). First we present a brief review and study of RDF visualization models. Then we define a general framework for nested graphs. Finally, the visualization model based on nested graphs is defined and applied to RDF.

6.1 Introduction

The term *data exploration* refers to the process by which the user is able to visualize, browse and query the data. The primary objectives of *visualization* are to present, transform and convert data into a visual representation, so that humans can analyze and query the data efficiently [196].

A *visualization model* provides a formalism that enables us to study the structure, properties and behavior of a visual interface. A *graph visualization model* is characterized by its abstraction mechanism, which follows the structure of a graph.

RDF data are typically very large, highly interconnected, and heterogeneous without following a fixed schema [113]. Any technique for visualizing RDF data should therefore be scalable; should support graph-based navigation; should be generic, not depending on a fixed schema; and should allow exploration of the RDF descriptions without a-priori knowledge of its structure [197].

Current applications for exploring RDF data use different interfaces, each having its advantages and disadvantages. For example, circle-and-arrow diagrams (e.g. IsaViz [198]) are intuitive and useful to show the graph structure of RDF, but they are not adequate to visualize large datasets [199]. On the other hand, a complex object-based interface (e.g. Tabulator [200, 20]) provides a view where the user is able to see the data as a nested structure of resources.

The first motivation of this chapter is to show that underlying complex object-based interfaces there is a powerful data model which needs to be studied formally to take advantage of its properties. The second motivation is the application of our research on graph database models in the context of RDF visualization. In this sense, we will study complex object-based interfaces in terms of nested graphs (a concept introduced in the area of database modeling by the Hypernode Model [67]).

A *nested graph* is a simple and flexible data structure [201] that: extends the plain structure of a graph to a nested structure (i.e. a graph whose nodes can themselves be graphs); supports the representation of arbitrarily complex objects; provides inherent support for data abstraction and encapsulation via nesting of graphs.

6.2 RDF Data Visualization

As Semantic Web data emerges, applications for exploring RDF data are necessary. This applications, commonly named *RDF Browsers*[1], use different interfaces each one having its advantages and disadvantages. Among these we can mention:

- *Keyword Search:* It suffices for simple information lookup, but not for higher abstraction search activities such as browsing and querying. This is commonly used by retrieval systems for the Semantic Web (e.g. Swoogle [202]).

- *Explicit queries:* It consists in using a query language for querying the data. It has the advantage that the language can be powerful enough to express any query, however writing explicit queries is difficult and requires schema knowledge.

- *Graph visualization:* It is the most basic interface and is based on circle-and-arrow diagrams (e.g. IsaViz [198]). It provides an intuitive and useful interface when trying to understand the structure of the data (a graph in the case of RDF). However it is not an appropriate way to look at data when a big number of nodes are present or for comparing objects of the same class. Moreover, graph visualization does not scale to large datasets ([199], Chapter 42. Scalable Network Visualization).

- *Object-based:* In this kind of interface, the user is able to see the description of a resource (i.e. its outgoing and incoming properties) as being an object which encapsulates its information. The basic approach shows the data of a unique resource in each moment (for example Disco [203], Marbles [204], Gruff [205], BrownSauce [206] and DAML Viewer [207]). A more complex approach consists in showing a nested structure of resources (e.g. The Tabulator [200, 20] and Zitgist [208]). Examples of the basic and complex approach are presented in Figures 6.1 and 6.2 respectively.

[1] `http://esw.w3.org/topic/SemanticWebTools`

Figure 6.1: Example of a simple object-based interface. The user interface of Disco[203] displays the information about (just) one resource as a property-value table. The identifiers *G*1, *G*2, .. refer to the list of all data sources.

Figure 6.2: Example of a complex object-based interface. In the outliner view of the Tabulator [200, 20], the user can explore an RDF graph as a tree of object descriptions, expanding nodes to get more information about them.

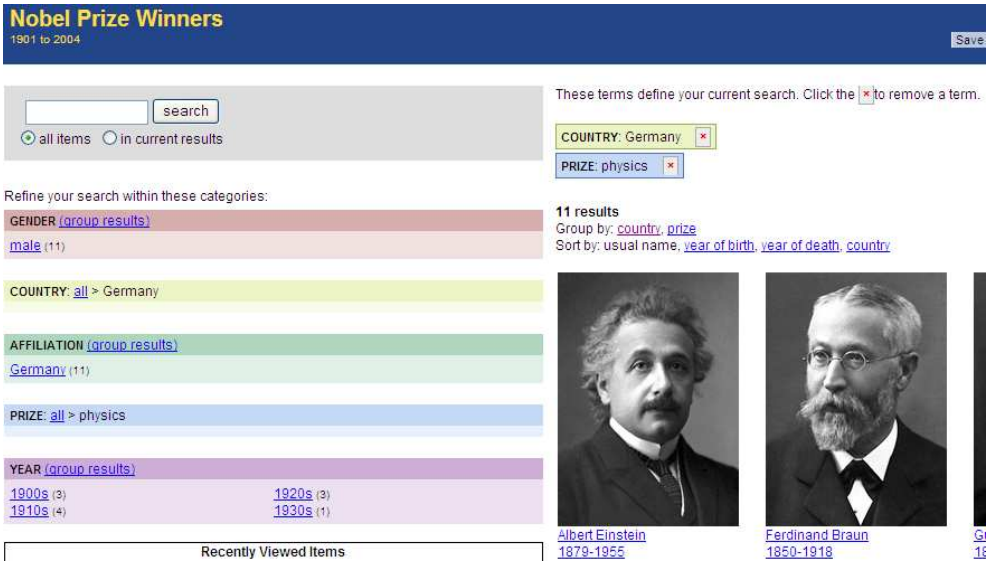


Figure 6.3: Example of a facet-based interface. The user interface of Flamenco [211] shows several facets or categories (on the left side) which are used to filter the data space (in this case, a list of Nobel Prize Winners).

- *Faceted Browsing:* In this model the information is faceted, that is, composed of orthogonal sets of categories [209]. Facets allow the user to restrict the information space to be visualized and to find information without an a-priori knowledge of its schema. For example, in Figure 6.3 the set of Nobel Prize Winners has been restricted by using facets `COUNTRY: Germany` and `PRIZE: physics`.

  Faceted browsing has become popular as a user-friendly way to navigate through a wide range of data collections [210], however existing faceted interfaces are manually constructed, do not fully support graph-based navigation and are domain-dependent. Among the applications using facets we can mention: Flamenco [211], BrowserRDF [212, 197], Longwell [213], /facet [214, 210], Zitgist [208] and the WIQA data browser [215, 216].

In terms of usability, and considering the graph structure of RDF data, an application for visualizing RDF should provide a user-friendly interface for selecting and navigating resources and their relationships with other resources (through their predicates). Additionally, based on the *Linked Data Principles* [217], an end-user interface for exploring RDF linked data should provide the following features [218]:

- *The description:* When someone looks up a resource, the representation should include all triples from the source dataset that have the resource as the subject (i.e. the *outgoing properties* of the resource). This is the immediate description of the resource.

- *Backlinks:* The representation should also include all triples from the source dataset that have the resource as the object (i.e. the *incoming properties* of the resource). It follows

the independence of direction for properties in RDF, where the direction one choses for a given property is arbitrary (it does not matter whether one defines “parent” or “child”). This is redundant, but it allows the user to traverse links in either direction.

- *Related descriptions:* The interface may include any additional information about related resources that may be of interest in typical usage scenarios.
- *Metadata:* The representation should contain any metadata the user want to attach to the published data, such as provenance and trustworthiness.
- *End user friendly views on the data:* It allows dealing with information overflow. For example ordering and merging of properties.
- *More advanced data analysis features:* For example aggregation, drill-down calculations and query-by-example views.

Considering the above requirements, we found that complex object-based interfaces are a good approach for exploring RDF data. It can be shown by the following list of features:

- The resource’s description includes outgoing and (possibly) incoming properties, which are presented as a list of property-value pairs. Recall that an outgoing / incoming property comes from a statement where the resource occurs as the subject / object respectively. An outgoing property is commonly represented as a statement “`is` *property* `of`” (e.g. see Figure 6.2).
- The nesting of resources is not necessarily hierarchical and cyclic references are possible. For example, in Figure 6.2 the node titled `Tim Berners-Lee` contains the property `is acquaintance of` which introduces a cyclic reference to the root resource `David Li`.
- Encapsulation of information is allowed because a non-expanded resource hides its properties. For example, the node `James Hollenbach` is not expanded in Figure 6.2.
- There are redundant data, in that whenever an object is expanded more than one level, for an outer property we will also found a “dual inner property” in the opposite direction. For example see properties `acquaintance` and `is acquaintance of` in Figure 6.2.
- The exploration begins in a root node (a resource that acts as the container), and browsing is achieved by selecting a property-value which either shows a new description (in the basic approach) or expands a node (in the complex approach).
- Queries can be defined by either writing a query expression or constructing graphically a graph pattern in a query-by-example style (e.g. see The Tabulator [20]).

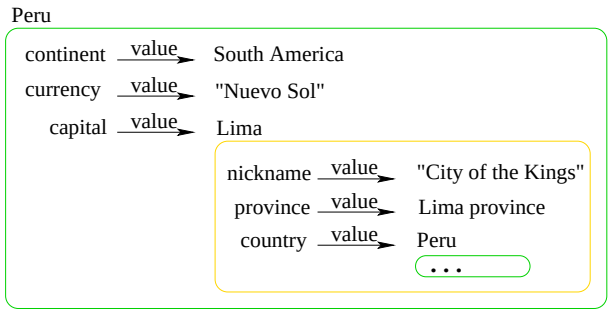


Figure 6.4: Example of a nested graph. The graph named `Lima` is nested in the graph named `Peru`.

In what follows, we concentrate our efforts in studying complex object-based interfaces. In fact, we will show that underlying these interfaces there is a powerful visualization model which deserves to be studied formally to take advantage of its properties and features.

6.3 Nested Graph Model

In this section we will define a data model based on the notion of *nested graphs*, a concept that was introduced in the area of graph database models by the Hypernode Model (see Chapter 3).

6.3.1 Nested Graphs

Definition 6.3.1 *(Nested Graph) Assume that Σ is an infinite set of labels. A nested graph is defined recursively as follows:*

(i) a triple (u,N,E) such that $u \in \Sigma$, $N \subset \Sigma$ and $E \subseteq N \times \Sigma \times N$ is a nested graph;

*(ii) let **NG** be a set of nested graphs, then any triple (u,N,E) for which $u \in \Sigma$, $N \subset \Sigma \cup \mathbf{NG}$ and $E \subseteq N \times \Sigma \times N$ is also a nested graph.*

Given a nested graph $G = (u,N,E)$, u is called the *name* (or label) of G, N is the set of *nodes* of G, and E is the set of *edges* of G. Given a node $n \in N$, if $n \in \Sigma$ then n is called a *primitive node* and $\text{name}(n) = n$; otherwise, if $n = (u',N',E') \in \mathbf{NG}$ then n is called a *complex node* and $\text{name}(n) = u'$.

Given a set of nested graphs $M \subset \mathbf{NG}$, we will assume that for each pair of nested graphs G_1 and G_2 in M, it holds that $\text{name}(G_1) \neq \text{name}(G_2)$ (i.e. we restrict nested graphs to be be uniquely labeled).

For example, Figure 6.4 presents graphically a nested graph named `Peru`. It contains primitive nodes (e.g. `continent` and `"Nuevo Sol"`), complex nodes (e.g. `Lima`) and edges (e.g. `continent` $\overset{value}{\rightarrow}$ `South America`). The complex node `Lima` is a nested graph which is nested inside the nested graph `Peru`.

Definition 6.3.2 *Let $G = (u, N, E)$ be a nested graph and $k > 1$. The operator* $\text{nodes}^k(G)$ *is defined recursively as follows:*

(i) $\text{nodes}^1(G) = N$*; and*

(ii) $\text{nodes}^k(G) = \text{nodes}^1(G) \cup \bigcup_{G' \in N \cap \mathbf{NG}} \text{nodes}^{k-1}(G')$.

Additionally, we define $\text{nodes}^*(G) = \bigcup_{j \geq 1} \text{nodes}^j(G)$.

Then, $\text{nodes}^k(G)$ extracts nodes from G by examining recursively each nested graph in G until reaching the k^{th} level of nesting. For example, let G be the nested graph presented in Figure 6.4, then $\text{nodes}^2(G)$ is given by the set of nodes of the nested graph `Peru` plus the nodes of the nested graph `Lima`. Additionally, $\text{nodes}^*(G)$ extracts nodes from G until a fixpoint is reached. In our example, $\text{nodes}^*(G) = \text{nodes}^2(G)$.

A nested graph G is *cyclic* if $G \in \text{nodes}^*(G)$ or if there exists $G' \in \text{nodes}^*(G)$ such that G' is cyclic. If G is not cyclic then it is a *hierarchical* nested graph. A node n is *encapsulated* in G if $n \in \text{nodes}^*(G)$. For example, the nested graph of Figure 6.4 is cyclic because the nested graph `Peru` occurs as node in the nested graph `Lima`, i.e., the nested graph `Peru` is encapsulated in itself.

6.3.2 Graphsets

In this section we present the notion of *graphset*, a plain (or unnested) representation for a nested graph. The complexity of working with a nested structure is usually reduced by transforming it to a plain structure. In this sense, we define additional operators and properties for nested graphs in terms of graphsets.

Definition 6.3.3 *(Plain Graph) A* Plain Graph *is a nested graph (u, N, E) satisfying that $N \subset \Sigma$, i.e., a plain graph has no nested graphs as nodes.*

Given two plain graphs $G_1 = (u_1, N_1, E_1)$ and $G_2 = (u_2, N_2, E_2)$, we say that G_1 is a subgraph of G_2, denoted $G_1 \subseteq G_2$, iff $N_1 \subseteq N_2$ and $E_1 \subseteq E_2$. We say that G_1 and G_2 are *isomorphic*, denoted $G_1 \approx G_2$, if and only if $G_1 \subseteq G_2$ and $G_2 \subseteq G_1$. Additionally, consider the following operations between G_1 and G_2:

Union: $G_1 \cup G_2 = (u_3, N_1 \cup N_2, E_1 \cup E_2)$
Intersection: $G_1 \cap G_2 = (u_3, N_1 \cap N_2, E_1 \cap E_2)$
Difference: $G_1 - G_2 = (u_3, N_1 \setminus N_2, \{(n_1, n_2, n_3) \in N_1 \mid n_i \in N_1 \setminus N_2\})$

where $u_3 \in \Sigma$ is a fresh label. If $\text{name}(G_1) = \text{name}(G_2)$ then $u_3 = \text{name}(G_1)$.

Definition 6.3.4 *(Graphset) A* Graphset *S is a set of plain graphs satisfying that, for each two plain graphs G_1 and G_2 in S,* $\text{name}(G_1) \neq \text{name}(G_2)$ *, i.e., a label $u \in \Sigma$ identifies at most one plain graph in S. We denote by* ***GS*** *the set of graphsets.*

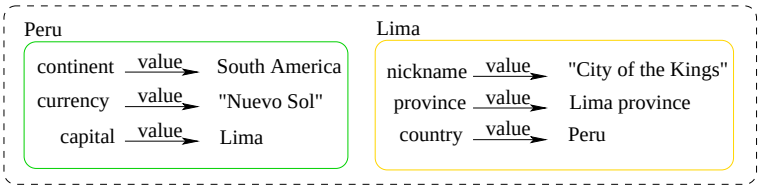


Figure 6.5: Example of a graphset which contains two plain graphs called `Peru` and `Lima` respectively.

Figure 6.5 shows an example of a graphset.

Given two graphsets S_1 and S_2, we say that S_1 is a *sub-graphset* of S_2, denoted $S_1 \subseteq S_2$, if and only if for each $G_1 \in S_1$ there exists $G_2 \in S_2$ satisfying that $\text{name}(G_1) = \text{name}(G_2)$ and $G_1 \subseteq G_2$. Then, S_1 and S_2 are *equal*, denoted $S_1 = S_2$, if and only if $S_1 \subseteq S_2$ and $S_2 \subseteq S_1$.

Additionally, define the *maximal-intersection* and *union* between two graphsets S_1 and S_2:

$$S_1 \sqcap S_2 = \{G_1 \cup G_2 \mid G_1 \in S_1, G_2 \in S_2 \text{ and } \text{name}(G_1) = \text{name}(G_2)\}$$

$$S_1 \cup S_2 = (S_1 \sqcap S_2) \cup \{G_1 \in S_1 \mid \text{for all } G_2 \in S_2, \text{name}(G_1) \neq \text{name}(G_2)\} \cup \{G_2 \in S_2 \mid \text{for all } G_1 \in S_1, \text{name}(G_2) \neq \text{name}(G_1)\}$$

6.3.3 Information Capacity of Nested Graphs

The *information capacity* of a representation is given by the set of objects modeled by such representation. Additionally, two complex object types are absolutely equivalent if and only if they can both be reduced to a normal form complex object types, which is based on some natural restructuring operators [219]. In this direction, the representation capacity of a nested graph will be defined in terms of graphsets.

From nested graphs to graphsets. First, we introduce operators for transforming nested graphs into graphsets.

Definition 6.3.5 *(Flattening of Nested Graphs) Let $G = (u, N, E)$ be a nested graph. We define the operators* flat *and* flat* *as follows:*

- $\text{flat}(G) = (u, N', E')$ *where* $N' = \{\text{name}(n) \mid n \in N\}$ *and* $E' = \{(\text{name}(n_1), \text{name}(n_2), \text{name}(n_3)) \mid (n_1, n_2, n_3) \in E\}$
- $\text{flat}^*(G) = \text{flat}(G) \cup \{\text{flat}^*(G') \mid G' \in N \cap \boldsymbol{NG}\}$

Then, $\text{flat}(G)$ transforms the nested graph G into a plain graph by flattening its first level of nesting. Additionally, $\text{flat}^*(G)$ flattens G and each nested graph G' in G until a fixpoint is reached, i.e., $\text{flat}^*(G)$ returns a graphset. For example, Figure 6.5 shows the graphset obtained by flattening the nested graph in Figure 6.4.

Note. Let us analyze the fixpoint of the recursive definition. Let G be a nested graph and S will be the graphset constructed from $\text{flat}^*(G)$. At each iteration of the recursion, we choose a

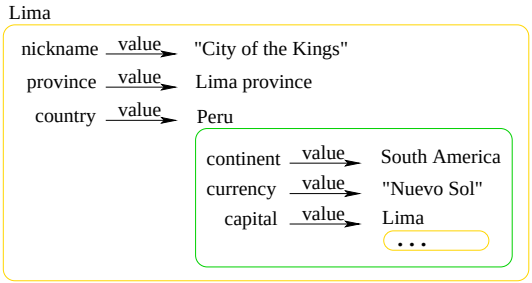


Figure 6.6: A nested graph obtained by changing the structure of the nested graph presented in Figure 6.4

nested graph G' encapsulated in G and check if $\text{flat}(G')$ is in S. If not, we add $\text{flat}(G')$ to S and we process $\text{flat}^*(G')$. If $\text{flat}(G')$ is currently in S we can avoid a second evaluation of $\text{flat}^*(G')$ by assuming that it has been evaluated in some point in the past. That is, the fixpoint of the algorithm is reached when no new plain graphs are added to S. Assuming this approach, we can see that the evaluation of $\text{flat}^*(G)$ is polynomial in the number of nodes encapsulated in G.

From graphsets to nested graphs. In the opposite direction, a graphset S can be transformed into a set of nested graphs by expanding the structure of each plain graph in S, i.e., by replacing simple nodes by complex nodes. The following definition formalizes this notion:

Definition 6.3.6 *(Expansion of Graphsets) Given a plain graph G and a graphset S, we define the function* $\text{integrate}(G,S)$ *recursively as follows: for each primitive node $n \in \text{nodes}^1(G)$, if there exists $G' \in S$ such that* $\text{name}(G') = n$*, then replace n by* $\text{integrate}(G',S)$*. This procedure is applied until a fixpoint is reached. Additionally, we define* $\text{integrate}^*(S) = \bigcup_{G \in S} \text{integrate}(G,S)$.

Note that the result of $\text{integrate}(G,S)$ is the (possibly expanded) node G. Additionally, $\text{integrate}^*(S)$ returns a set of nested graphs, i.e., one for each plain graph in S. For example, if we expand the graphset in Figure 6.5, we will obtain the nested graph in Figure 6.4 plus the nested graph in Figure 6.6.

Note. Let G be a plain graph and S be a graphset. The evaluation of $\text{integrate}(G,S)$ is polynomial in the number of plain graphs in S (in the worst case we will execute $\text{integrate}(G',S)$ for each plain graph G' in S). Considering that $\text{integrate}^*(S)$ evaluates $\text{integrate}(G'',S)$ for each plain graph G'' in S, we conclude that $\text{integrate}^*(S)$ can be evaluated in polynomial time.

The following lemma defines the equivalence, in terms of representation, between nested graphs and graphsets.

Lemma 6.3.1 *Let **NG** be the set of nested graphs and **GS** be the set of graphsets. Then:*

*(i) for each nested graph $G \in$ **NG**, $G \in \text{integrate}^*(\text{flat}^*(G))$; and*

*(ii) for each graphset $S \in$ **GS**, $S = \bigcup_{G \in \text{integrate}^*(S)} \text{flat}(G)$.*

Proof.

- Case (i): Let G be a nested graph and S be the grapset returned by $\text{flat}^*(G)$. We have that S is given the plain graph $\text{flat}(G)$ plus a plain graph $\text{flat}(G')$ for each nested graph G' encapsulated in G, i.e., it holds that $G = \text{integrate}(\text{flat}(G), S)$. Considering that $\text{integrate}^*(S) = \bigcup_{G'' \in S} \text{integrate}(G'', S)$, then $\text{integrate}(\text{flat}(G), S)$ is in $\text{integrate}^*(S)$. Finally, we conclude that G is in $\text{integrate}^*(S)$.

- Case (ii): Let S be a graphset. A nested graph G is in $\text{integrate}^*(S)$ iff there is a plain graph G' in S such that $G = \text{integrate}(G', S)$. Considering that definition $G' = \text{flat}(G)$, it holds that $S = \bigcup_{G \in \text{integrate}^*(S)} \text{flat}(G)$.

□

Consider a set of nested graphs M starting from a graphset S (i.e. $M = \text{integrate}^*(S)$). Then it holds that M models the same set of objects than S (because $S = \bigcup_{G \in \text{integrate}^*(S)} \text{flat}^*(G)$). However, in some cases it can occur that a minimal subset M' of M is enough for modeling all the data modeled by the graphset S (i.e. we can obtain S by flattening each nested node in M'). This minimal subset will be called the *source* of a set of nested graphs.

Definition 6.3.7 *Let M be a set of nested graphs. A* source *of M is a minimal subset M' of M satisfying that $\bigcup_{G' \in M'} \text{flat}^*(G') = \bigcup_{G \in M} \text{flat}^*(G)$.*

For example, consider the nested graph presented in Figure 6.4 and call it G. If we flatten G, we have that $S = \text{flat}^*(G)$ is the graphset presented in Figure 6.5. Now, we have that $\text{integrate}^*(S)$ will contain the nested graphs of Figure 6.4 and Figure 6.6. Note that each of the nested graphs (`Peru` and `Lima`) is a source of the graphset S, because any of them contains all the data modeled by S.

If the source of a set of nested graphs M is a single nested graph G, then G is called the *single-source* of M. In the context of graphsets, the above property introduces the notion of a *single-source graphset*.

Definition 6.3.8 *(Single-source graphset) A graphset S is called a* single-source graphset *if the source of* $\text{integrate}^*(S)$ *is a singleton (i.e. has only one element).*

The notion of a source is useful for visualizing nested graphs. Consider the problem of selecting a good start point for navigation. If we use the source as a minimal set of navigation, we can reduce the number of visible or active nested graphs without losing data, such that all the data could be accessed by navigating through the nested graphs in the source. Clearly this problem could have a direct solution if we have a single-source graphset, then all the data can be accessed from a single nested graph.

Lemma 6.3.2 *Determining if a graphset S is single-source can be computed in polynomial time.*

Proof. Let S be a graphset. Construct a digraph $G = (N, E)$ where the set of nodes N is the set of names of the graphs in S, and there is an arc from u_1 to u_2 in E if the name u_2 occurs as a node in the graph named u_1. We have that S is single-source if G is *connected* and there is a *spanning tree* for G (i.e. there is a tree which connects all the nodes together). It is known that the problem of determining if a graph has a spanning tree can be computed in polynomial time ([220], Section 4).

□

Finally, the *information capacity of a nested graph G* is defined by the graphset $\text{flat}^*(G)$. Additionally, the equivalence of two nested graphs is given by the equivalence in their information capacities (i.e. they can be reduced to the same graphset).

Definition 6.3.9 *(Equivalence of nested graphs) Two nested graphs G_1 and G_2 are* equivalent, *denoted $G_1 \approx G_2$, if and only if* $\text{flat}^*(G_1) = \text{flat}^*(G_2)$.

For example, the nested graphs presented in Figure 6.4 and Figure 6.6 are equivalent.

6.4 A Graph Model for Visualizing RDF Data

In this section we define the graph model for visualizing RDF data based on nested graphs. Specifically, we select a subset of nested graphs whose particular structure is used for modeling RDF data. Moreover, we propose this structure as a formal representation for complex object-based interfaces.

Assume that $\mathbf{NG}^* \subset \mathbf{NG}$ is the set of nested graphs whose set of labels is given by $\mathbf{U} \cup \mathbf{L}$ (RDF URIs and labels), and each nested graph (u, N, E) in $\mathbf{NG}^*$ satisfies:

(i) $u \in \mathbf{U}$;

(ii) $N \subset \mathbf{U} \cup \mathbf{L} \cup \mathbf{NG}^*$;

(iii) $E \subseteq ((N \setminus \mathbf{L}) \times \{+\} \times N) \cup ((N \setminus \mathbf{L}) \times \{-\} \times (N \setminus \mathbf{L}))$; and

(iv) if $n \in N$ then n occurs in some edge of E.

We denote by $\mathbf{GS}^*$ the set of graphsets having only plain graphs from $\mathbf{NG}^*$.

Let G_{RDF} be an RDF graph[2]. we construct a graphset $S \in \mathbf{GS}^*$ from G_{RDF} as follows: for each resource $u \in \mathbf{U}$ occurring either as subject or object in some triple of G_{RDF}, we add a plain graph (u, N, E) in S such that E contains (i) an edge $(n_1, +, n_2)$ for each triple (u, n_1, n_2) in G_{RDF}, and (ii) an edge $(n_1, -, n_2)$ for each triple (n_2, n_1, u) in G_{RDF}.

The above construction is given by the following algorithm:

[2]We restrict our study to RDF graphs having no blank nodes.

Let S be an empty graphset
for each triple (v_1, v_2, v_3) in G_{RDF} **do**
  $G_1 = (v_1, N_1, E_1)$ where $N_1 = \{v_2, v_3\}$ and $E_1 = \{(v_2, +, v_3)\}$
  $S \leftarrow S \cup G_1$
  if $v_3 \in$ **U then**
    $G_2 = (v_3, N_2, E_2)$ where $N_2 = \{v_1, v_2\}$ and $E_2 = \{(v_2, -, v_1)\}$
    $S \leftarrow S \cup G_2$
  end if
end for
return S

Using this transformation, we will show how to use nested graphs for modeling complex object-based interfaces. Let G_{RDF} be an RDF graph and S be the graphset obtained from G_{RDF} (as defined above). Consider the characteristics of complex object-based interfaces presented in Section 6.2:

- *the user is able to see the description of a resource (i.e. its outgoing and incoming properties) as being an object which encapsulates its information.* A plain graph (u, N, E) in S is intended for modeling a resource identified u whose *outgoing* and *incoming properties* are encapsulated by edges labeled "$+$" and "$-$" respectively.

- *It consists in showing a nested structure of resources.* From Lemma 6.3.1, we have that the graphset S can be transformed in the set of nested graphs integrate$^*(S)$. Then, an RDF graph G_{RDF} can also be represented by the set of nested graphs integrate$^*(S)$.

- *There are redundant data, in that whenever an object is expanded more than one level, for an outer property we will also found a "dual inner property" in the opposite direction.* Let u_1, p and u_2 be URIs. For each RDF triple (u_1, p, u_2) in G_{RDF} it holds that integrate$^*(S)$ contains two nested graphs $G_1 = (u_1, N_1, E_1)$ and $G_2 = (u_2, N_2, E_2)$ such that G_1 represents the resource identified by URI u_1, G_2 represents the resource identified by URI u_2, E_1 contains the edge $(p, "+", G_2)$ (i.e. the outer property in G_1) and E_2 contains the edge $(p, "-", G_1)$ (i.e. the dual inner property in G_2).

- *Encapsulation of information is allowed because a non-expanded resource hides its properties.* We have shown that nested graphs enable us to model objects with an arbitrarily complex structure. Encapsulation of nested graphs was formally defined in Section 6.3.1.

- *The nesting of resources is not necessarily hierarchical and cyclic references are possible.* Cyclic references are modeled as cyclic nested graphs, i.e., when a nested graph is encapsulated in itself.

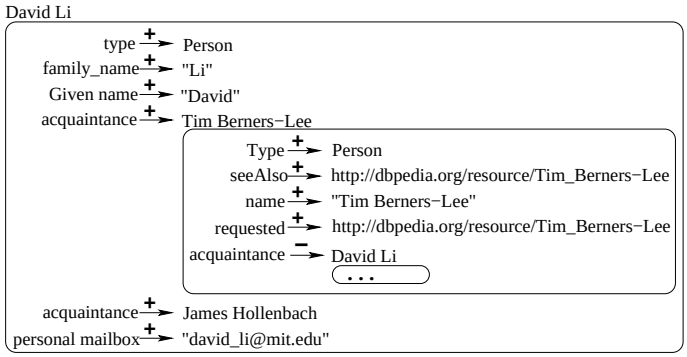


Figure 6.7: An abstract representation to the interface in Figure 6.2.

- *The exploration begins in a root node.* The selection of a suitable root node has been formalized as the source of a set of nested graphs (see Definition 6.3.7). A particular case is given by the notion of a single-source graphset (see Definition 6.3.8).

For example, consider the object-based interface presented in Figure 6.2. The expanded node `David Li` can be represented by the nested graph presented in Figure 6.7. If we compare both figures, we can see that an expanded node in the interface can be represented by a nested graph, and outgoing and incoming properties are represented by edges labeled "+" and "-" respectively. For example, the incoming link `is acquaintance of` → `David Li` in Figure 6.2 is represented by the edge $\texttt{acquaintance} \xrightarrow{-} \texttt{David Li}$ in Figure 6.7.

From the above comparison, we state the following fact:

The nested graph model can be thought of as an abstract representation for complex object-based interfaces.

The abstract representation based on nested graphs enable us to study formally object-based interfaces. For example, the notion of single-source graphset can be used to select an object as the start point of navigation. A language for querying object-based interfaces could be based on the operators defined for nested graphs and graphsets. Exploration states can be stored, serialized or shared in the form of nested graphs or graphsets.

6.5 Conclusions

The research represented in this chapter is firstly a helpful attempt in studying formally the visualization of RDF data. The main contribution of this chapter is the definition of a graph visualization model for RDF data. In particular we

- presented a general framework for studying nested graphs;
- studied the information capacity of a nested graph (i.e. the set of objects that can be modeled by it) in terms of its plain representation called a *graphset*;

- presented a form of representing RDF data using nested graphs;
- defined a data model for visualizing RDF data based on nested graphs.

Further work includes to study more properties of the model, to analyze some visualization parameters (e.g. to define a notion of relevance in order to reduce the number of visible properties for a given resource), the definition of a query language for the model and the implementation of the model as part of a Linked Data browser.

Chapter 7

Conclusions and Future Work

In this thesis we have advocated the research on RDF from a database point of view. In particular, we studied RDF in relation to graph database models.

A first motivation of this thesis was the management of huge collections of RDF data. This point of view evoked the notion of RDF databases, thus our decision to study the RDF specification from a database perspective. A second motivation was to incorporate the missing elements the RDF specification needed to become a database model. When the RDF specification was released, it contained the definition of the data structure (i.e. the RDF graph model), however neither query capabilities nor integrity constraints were included. In the meantime, a standard query language for RDF, SPARQL, was defined; however integrity constraints are still missing.

Considering the above motivations, we aimed at the definition of a database model for RDF. Moreover, considering the graph nature of RDF, it should ideally be a graph database model. Thus, the main goal of this thesis was the design of a graph database model oriented to support the RDF model. We achieved only partially this goal, for the following reasons. On the one hand, the juvenility of the area of RDF databases did not let researchers to follow an established and standard approach. Even though we participated actively in the W3C forums and discussions, at the time was not evident to what degree an RDF database model would incorporate graph features. On the other hand, the direction of this thesis was guided by the needs of the RDF community and its current research, developments, and standards. This is particularly clear from the apparent diversity of results we got.

A balance on the fulfillment of the specific objectives of the thesis proposal that guided our research shows the following results:

1. *Review of past data models addressing graph-like applications*. This objective was covered by our survey on graph database models (Section 3). It had a good reception in the community of databases and also called the attention of people working on the Semantic Web. We received good comments from referenced authors and short contacts with peo-

ple in the industry. For instance, H. S. Kunni, author of one of the first graph database models, G-Base, congratulates us about the survey.

This work establishes an asset framework of knowledge for the subsequent research in the thesis.

2. *Study and selection of RDF characteristics that the database model will support*. This objective was accomplished from two perspectives. First, we surveyed RDF query languages and we identified, based on the experience of the graph database community, a set of desirable missing graph operators that should be incorporated (Chapter 4). Secondly, we completely clarified the relationship of the W3C standard query language SPARQL with respect to its graph database functionalities. In fact, we proved that SPARQL has the same expressive power of Relational Algebra, whose limitations for supporting graph features are well known (Chapter 5).

3. *Definition of a graph database model that supports RDF*. As we stated above, this objective was only partially fulfilled. After the W3C released SPARQL, we concentrated our research in to study its (weak) relationship with graph query languages, and so focusing in what the community was pursing from a graph data model point of view: the definition of the graph visualization model for RDF (Chapter 6).

4. *Design and implementation of a prototype (proof of concept) for the proposed model*. When we started the development of this thesis, the creation of a prototype was a desirable goal because there were only experimental applications for storing RDF data, and no one with graph database features. For reasons far of our control, in a short time there showed up a whole new generation of RDF data store applications and databases. Summing the developments described above, this goal lost its meaning.

In contrast to the partially fulfilled goal of building a monolithic graph database model for RDF, this thesis presents relevant contributions regarding the relationships of the current standards (that were adopted while the thesis was being developed) with graph database models.

7.1 Contributions

Summarizing, in a broad sense, this thesis contributes to the general understanding of RDF databases by elaborating the connection between graph databases and RDF. Specifically:

- We studied RDF from a database modeling perspective (Section 2.3). We compared RDF with database models and discussed characteristics of RDF that difficult the application of a graph database model.

- The survey on graph database models (Chapter 3) provides a useful background for researchers whose concerns are the modeling, querying, and storing of data with graph structure. For example data from the Web, social networks, biology, etc.
- We discovered advantages and weakness in developing query languages for RDF, in particular the support for querying graph features (Chapter 4).
- We studied in depth the standard RDF query language, SPARQL (Chapter 5). Particularly, we clarified some open issues (e.g. its support for negation) and determined its expressive power. Combined with the above result, this study is useful for studying extensions to the language, for example the support for graph properties.
- We have defined a graph model for visualizing RDF data along the lines of the database perspective (Chapter 6). This work can spread the formal study of RDF visualization models in order to improve the development of applications for exploring RDF data.

As concrete bibliographic results of this thesis, we have the following publications in chronologic order:

- *"Querying RDF Data from a Graph Database Perspective"*
  Proceedings of the 2nd European Semantic Web Conference (2005)
  LNCS Number 3532
- *"Survey of graph database models"*
  ACM Computing Surveys (2008)
- *"The Expressive Power of SPARQL"*
  Proceedings of the 7th International Semantic Web Conference (2008)
  LNCS Number 5318
- *"A Nested Graph Model for Visualizing RDF Data"*
  III Alberto Mendelzon International Workshop on Foundations of Data Management (2009)

7.2 Future Work

This thesis posed several research problems which would be desirable to address, oriented to the development of RDF databases, thus promoting the study, definition, and application of the theoretical foundations for RDF databases.

On the area of graph databases, open lines of research include: the categorization and comparison of graph query languages (our ACM surveys is concentrated mainly in the data

structures); the definition of a standard query language for graph databases; a deep study of integrity constraints for graph data; and the implementation of a complete graph database model, i.e., the integration of graph data structures, a graph query language and graph integrity constraints.

We have brought more evidence, from the RDF field, to the known fact that a database model is an important building block in the development of databases. We have seen that in the context of RDF databases the definition of a database model is still missing, particularly regarding the inclusion of integrity constraints in RDF. Moreover, this thesis showed that there is plenty of work to improve the design of RDF databases by using results in graph databases.

There are positive results and conclusions about the semantics, complexity, and expressive power of SPARQL. Nonetheless a lot of questions remain open, leaving space for further research. For example, the SPARQL Working Group is intensively working in three aspects[1]: query language expressivity, update and service description. The first point presents a list of open issues, summarized as aggregate queries, sub-queries, negation and project expressions, that could increase the expressive power of the language. The second point concerns the definition of an update language for RDF datasets, including issues like concurrency and security. And the third point suggests the definition of a service description mechanism to be used by SPARQL endpoints.

Several proposals for extending RDF query languages with graph features have been presented recently after we finish this document [221]. Some of them extend SPARQL following ideas from the graph query languages presented in Chapter 4. For example:

- There are several extensions of SPARQL that use regular expressions for querying paths. PSPARQL [222, 223] allows graph patterns whose predicates (edges) are modeled with regular expressions. CPSPARQL [224] extends PSPARQL by allowing constraints on internal nodes that belong to a path. SPARQLeR [225] enables the discovery of semantic associations in RDF using path expressions. SPARQ2L [226] introduces the notions of path variables and path filter expressions, i.e., constraints on path variables into SPARQL. GLEEN [227] is a library that implements complex path expressions for SPARQL.

- Seid and Mehrotra [228] studied the problem of supporting grouping and aggregate queries over RDF databases. The authors define grouping and aggregate operators that return either RDF graphs or n-ary relations consisting of projected variable bindings. Such operators are then used to extend SPARQL with the corresponding constructs.

- nSPARQL [229] is a navigational language for RDF which extends SPARQL with nested regular expressions. It shows that the use of regular expressions is appropriate to navigate RDF data and allows support for graph queries (mainly path queries).

[1] `http://www.w3.org/2009/sparql/track/issues/open`

The integration of these results in the extended version of SPARQL could result in a very interesting work.

In general, several known database techniques for query optimization, cost estimation, and query rewriting could be extended to SPARQL. For example, consider the notion of database views (i.e., how SPARQL queries can be maintained, updated and propagated) and distributed queries (i.e. how to divide a SPARQL query in several queries that can be evaluated independently).

Nowadays, there is a growing community of researchers studying and developing Linked Data applications. Considering the goal of the Linked Open data project, RDF data extraction, conversion, transformation, and integration needs further research.

In relation to the visualization model presented in Chapter 6, further research includes the design of a visual query language, the support for SPARQL queries, the identification and study of visualization parameters (e.g. ranking of properties) and a combination with facet-based classification. Additionally, would be interesting that the proposed model be incorporated in the design and implementation of a Linked Data browser.

Appendix A

Representative Graph Database Models

In this appendix we describe most representative graph db-models and other related models that do not fit properly as graph db-models, but use graphs, for example, for navigation, for defining views, or as language representation. For each proposal, we present their data structures, query language and integrity constraint rules. In general, there are few implementations and no standard benchmarks, hence we avoid surveying implementations.

To give a flavor of the modeling in each proposal, we will use as a running example the toy genealogy shown in Figure A.1.

A.1 Logical Data Model (LDM)

Motivated by the lack of semantics in the relational db-model, Kuper and Vardi [71] proposed a db-model that generalizes the relational, hierarchical and network models. The model describes mechanisms to restructure data plus a logical and an algebraic query languages.

In LDM a *schema* is an arbitrary directed graph where each node has one of the following types: the *Basic* type $\square$ describes a node that contains the data stored, the *Composition* type $\owedge$ describes a node that contains tuples whose components are taken from its children, and the *Collection* type $\bigcirc$ describes a node that contains sets and whose elements are taken from its child. Hence, Basic nodes occur as terminal nodes and represent atomic data, Composition and Collection nodes occur as internal nodes and represent structured data, and edges represent connections between data.

A second version of the model [72], besides renaming the nodes Composition and Collection as *Product* $\otimes$ and *Power* $\circledast$ respectively, incorporates a new type, the *Union* type$\bigcirc\!\!\!\!\cup$, intended to represent a collection whose domain is the union of the domains of its children (see example in Figure A.2).

NAME	LASTNAME
George	Jones
Ana	Stone
Julia	Jones
James	Deville
David	Deville
Mary	Deville

PERSON	PARENT
Julia	George
Julia	Ana
David	James
David	Julia
Mary	James
Mary	Julia

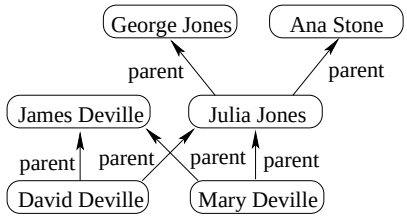
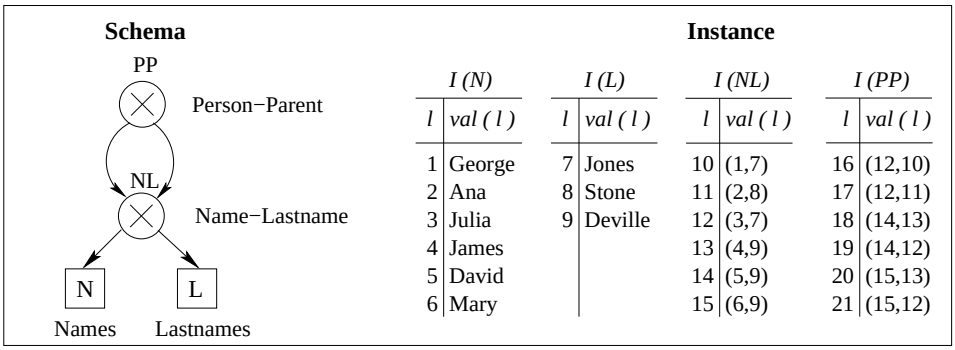


Figure A.1: A genealogy diagram (right-hand side) represented as two tables (left-hand side) NAME-LASTNAME and PERSON-PARENT (Children inherit the lastname of the father just for modeling purposes).



I (N)

l	*val (l)*
1	George
2	Ana
3	Julia
4	James
5	David
6	Mary

I (L)

l	*val (l)*
7	Jones
8	Stone
9	Deville

I (NL)

l	*val (l)*
10	(1,7)
11	(2,8)
12	(3,7)
13	(4,9)
14	(5,9)
15	(6,9)

I (PP)

l	*val (l)*
16	(12,10)
17	(12,11)
18	(14,13)
19	(14,12)
20	(15,13)
21	(15,12)

Figure A.2: Logical Data Model. The schema (on the left) uses two basic type nodes for representing data values (N and L), and two product type nodes (NL and PP) to establish relations between data values in a relational style. The instance (on the right) is a collection of tables, one for each node of the schema. Note that internal nodes use pointers (names) to make reference to basic and set data data values defined by other nodes.

An LDM database *instance* consists of an assignment of values to each node of the schema. The instance of a node is a set of elements from the underlying domain (for basic type nodes) and tuples or sets taken from the instance of the node's children (for $\otimes$, $\circledast$ and$\bigcirc\!\!\!\!\cup$ types). With the objective of avoiding cyclicity at the instance level, the model proposes to keep a distinction between memory locations and their content. Thus, instances consist of a set of *l-values* (the address space), plus an *r-value* (the data space) assigned to each of them. These features allow to model transitive relations like hierarchies and genealogies.

Over this structure a first order many-sorted language is defined. With this language, a query language and integrity constraints are defined. Finally, an algebraic language –equivalent to the logical one– is proposed, providing operations for node and relation creation, transformation and reduction of instances, and other operations like union, difference and projection.

LDM is a complete db-model (i.e. data structures plus query languages and integrity constraints) that supports modeling of complex relations (e.g. hierarchies, recursive relations). The notion of virtual records (pointers to physical records) proves useful to avoid redundancy of data by allowing cyclicity at the schema and instance level. Due to the fact that the model

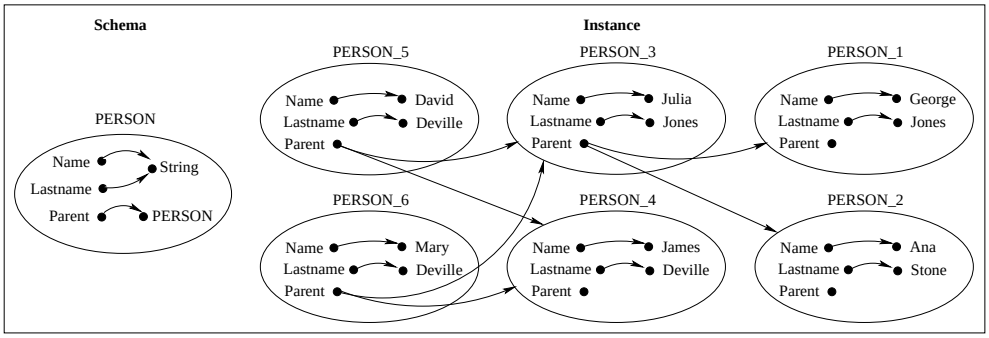


Figure A.3: Hypernode Model. The schema (left) defines a *person* as a complex object with the properties *name* and *lastname* of type string, and *parent* of type person (recursively defined). The instance (on the right) shows the relations in the genealogy among different instances of person.

is a generalization of other models (like the relational model), their techniques or properties can be translated into the generalized model. A relevant example is the definition of integrity constraints.

A.2 Hypernode Model

A hypernode is a directed graph whose nodes can themselves be graphs (or hypernodes), allowing nesting of graphs. Hypernodes can be used to represent *simple* (flat) and *complex objects* (hierarchical, composite, and cyclic) as well as mappings and records. A key feature is its inherent ability to *encapsulate information*.

The hypernode model was introduced by Levene and Poulovassilis [67], who define the model and a declarative logic-based language structured as a sequence of instructions (hypernode programs), used for querying and updating hypernodes. The implementation of a storage system based on the hypernode model is presented in the work of Tuv et al. [230].

In a second version [68] of the model, the notion of schema and type checking is introduced via the idea of types (primitive and complex), that are also represented by nested graphs (see an example in Figure A.3). The model is completed with entity and referential integrity constraints over a hypernode repository. Moreover it presents a rule-based query language called *Hyperlog*, which can support both querying and browsing with derivations as well as database updates, and is intractable in the general case.

A third version of the model [69] discusses a set of constraints (entity, referential and semantic) over hypernode databases and introduces the concept of *Hypernode functional dependency (HDF)*, denoted by $A \rightarrow B$, where A and B are sets of attributes, and A determines the value of B in all hypernodes of the database. In addition it presents another query and update language called HNQL, which use compounded statements to produce HNQL programs.

Summarizing, the main features of the Hypernode model are: it is based on a nested graph

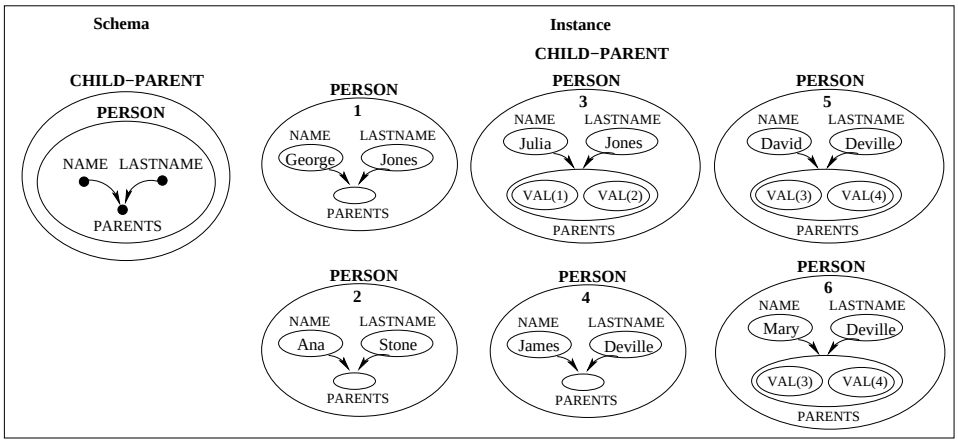


Figure A.4: GROOVY. At the schema level (left), we model an object *PERSON* as an hypergraph that relates the attributes *NAME*, *LASTNAME* and *PARENTS*. Note the value functional dependency (VDF) *NAME,LASTNAME* $\rightarrow$ *PARENTS* logically represented by the directed hyperedge ({NAME,LASTNAME} {PARENTS}). This VFD asserts that NAME and LASTNAME uniquely determine the set of PARENTS.

structure which is simple and formal; it has the ability to model arbitrary complex objects in a straightforward manner; it can provide the underlying data structure of an object-oriented data model; it can enhance the usability of a complex objects database system via a graph-based user interface. The drawbacks are that data redundancy can be generated by its basic value labels, and restrictions in the schema level are limited. For example, the specification of restrictions for missing information or multivalued relations is not possible.

A.3 Hypergraph-Based Data Model (GROOVY)

GROOVY (Graphically Represented Object-Oriented data model with Values) [47] is a proposal of object-oriented db-model which is formalized using *hypergraphs*. That is, a generalization of graphs where the notion of edge is extended to *hyperedge*, which relates an arbitrary set of nodes [231]. An example of hypergraph schema and instance is presented in Figure A.4.

The model defines a set of structures for an object data model: value schemas, objects over value shemas, value functional dependencies, object schemas, objects over objects schemas and class schemas. It is shown that these structures can be defined in terms of hypergraphs.

A *Value Schema* defines the attributes (atomic or multi-valued) that contain a class of objects. Attributes in value schemas can themselves be value schemas, allowing representation of complex objects and encapsulation of information. An *Object over a value schema* is a pair $O =< i, v >$ where i is the object-ID (identity) and v is the object value (properties). A *Value Functional Dependency* is used at the value schema level to assert that the value of a set of attributes uniquely determines the value of other attribute. The determined attribute can be

single-valued or multi-valued. An *Object Schema* is a triple $< N, F, S >$, where N is a value schema, F is a set of value functional dependencies over N, and S is a set of subsets of N including N itself. S represents sub-object schemas which describe the potential sharing between objects and subjects. A *Class Schema* is a triple $< N, F, H >$ such that $H \subseteq \mathcal{P}(N)$ and $N \in H$. H defines a super-class schema/sub-class schema relationship which induces a partial ordering of class schemas, the inheritance lattice. There is a one-to-one correspondence between each object schema $< N, F, S >$ and a hypergraph, interpreting N as vertices, F as directed hyperedges, and S as undirected hyperedges. The same approach is applied for representing class schemas and objects in the instance level.

A hypergraph manipulation language (HML) for querying and updating hypergraphs is presented. It has two operators for querying hypergraphs by identifier or by value, and eight operators for manipulation (insertion and deletion) of hypergraphs and hyperedges.

The use of hypergraphs has several advantages. It introduces a single formalism for both sub-object sharing and structural inheritance, avoiding redundancy of data (values of common sub-objects are shared by their super-objects). Hypergraphs allow the definition of complex objects (using undirected hyperedges), functional dependencies (using directed hyperedges), object-ID and (multiple) structural inheritance. Value functional dependences establish semantic integrity constraints for object schemas.

Let us mention that GROOVY influenced the development of the Hypernode model by providing another approach to modeling complex objects. If we compare both models, we can see that hypergraphs can be modeled by hypernodes by encapsulating the contents of each hyperedge within a further hypernode. In contrast, the multilevel nesting provided by hypernodes cannot easily be captured by hypergraphs [68].

The notion of hypergraphs is also used in other proposals. Consens and Mendelzon [70] present a query and visualization system based on the concept of *hygraphs*, a version of hypergraphs. Their model defines an special type of edge called *Blob*, which relates a node with a set of nodes. Tompa [76] proposes a model for hypertext where nodes represent Web pages and hyperedges represent user state and browsing. Watters and Shepherd [77] use hypergraphs to model data instances (in an existent database) and access to them. The model represents data instances as nodes in a hypergraph, and perform operations over both hyperedges and nodes representing data.

A.4 Simatic-XT: A Data Model to Deal with Multi-scaled Networks

Motivated by modeling of transport networks (train, plane, telecommunications), Mainguenaud [75] proposed a graph (object-oriented) db-model that merges the concepts of graph and object-oriented paradigm, focusing in the (graph) structure of the data but not on the behavior of entities to be modeled. An example is presented in Figure A.5.

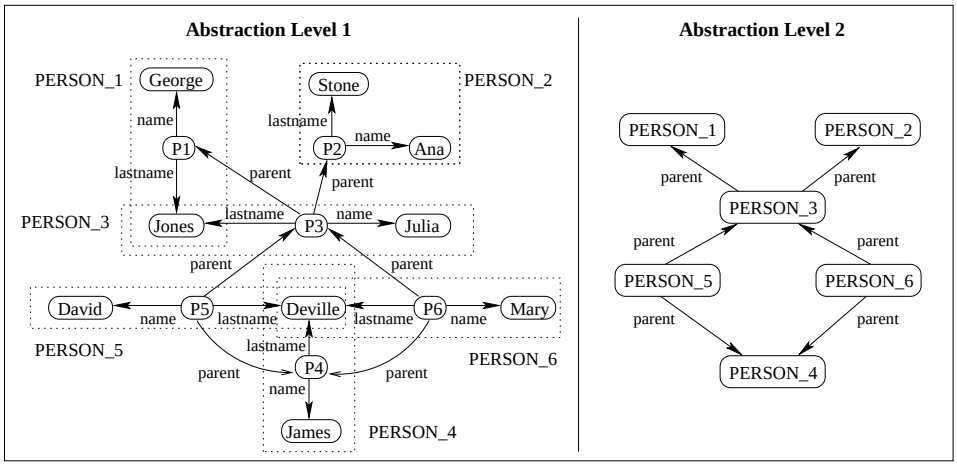


Figure A.5: Simatic-XT. The model does not define an schema. The relations Name-Lastname and Person-Parent are represented in two abstraction levels. In the first level (the most general), the graph contains the relations *name* and *lastname* to identify people (*P1*, ..., *P6*). In the second level we use the abstraction of *Person*, to compress the attributes *name* and *lastname* and represent only the relation *parent* between people.

The model represents a labeled directed multi-graph, defining three basic types: *Node type*, *Edge type*, and *Network type* (representing a graph). Additionally, the model introduces the notion of *Master Nodes* and *Master Edges*, to support levels of abstraction of sub-networks and paths respectively. Each object in the model has an object identifier (OID), that permits identification and referencing. The level of abstraction is given by the nested level of Master Nodes and Edges Nodes. The model defines the attribute *in_edges* to represent the set of edges arriving in the subgraph (resp. path) that the Master Node (resp. Master Edge) represents. In the same form *out_edges* represent the set of edges leaving the Master node or Master Edge.

A sequel paper [140] presents a set of graph operators divided into three classes: *Basic operators*, managing the notion of abstraction (the Develop and Undevelop operators); *Elementary operators*, managing the notion of graph and sub-graph (Union, Concatenation, Selection and, Difference) and; *high level operators* (Paths, Inclusions and Intersections).

This proposal allows simple modeling and abstraction of complex objects and paths, and encapsulation at node and edge levels. It improves the representation and querying of paths between nodes, and the visualization of complex nodes and paths. At its current state, it lacks definition of integrity constraints.

A.5 Graph Database System for Genomics (GGL)

This db-model comes from the biology community and highlights the advantage of storing Genome maps as graphs. GGL includes a graph-theoretic db-model [58], a genome graph language [80], and query operators for the model [142, 141]. The model is based on binary

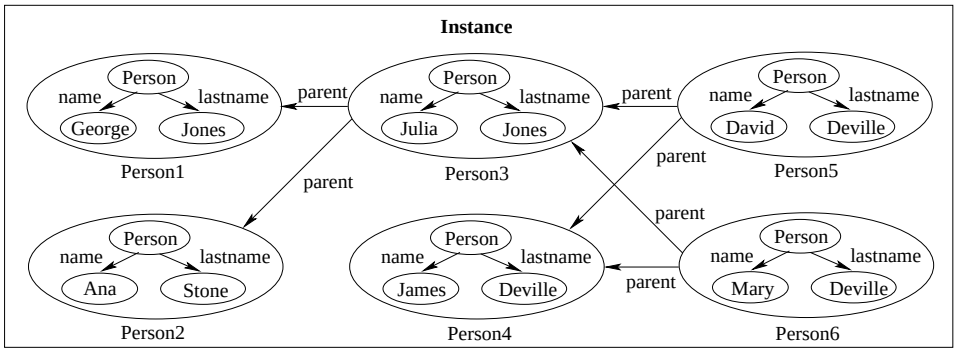


Figure A.6: GGL. Schema and instances are mixed. *Packaged graph vertices* (*Person1, Person2, ...*) are used to encapsulate information about the graph defining a Person. Relations between these packages are established using edges labeled with *parent*.

relationships between objects. It extends the basic notion of a graph by including vertices that represent edge types which allow one to specify relations between relations (higher-order relations), and encapsulated graphs as vertices. An example is presented in Figure A.6.

A instance graph in GGL is basically a collection of: *Simple vertices* which model simple concepts and can be labeled or unlabeled; *Symbols* that define nodes without outgoing edges; *Edges* that connect two vertices and are labeled with a relation name; *Packaged graph vertices* that represent graphs which are packaged (encapsulated) into vertices; *Relation type vertices* which are used to represent relations between relations (higher-order relations). According to this definition, the graph data structure consists of a directed labeled, possibly cyclic, which maintains hierarchically-ordered graphs.

For querying data two methods are proposed. The first [142], restricts the form of the query graph to be rooted directed acyclic graphs with equality constraints. The strategy is based in following the paths specified by the query-graph and returning the values that correspond to the end of the paths. The second, is a declarative programming language called WEB [141], that defines queries as graphs with the same structures of the model and return the graphs in the database which *match* the query-graph.

The model defines two integrity constraints: Labels in a graph are uniquely named; edges are composed of the labels and vertices of the graph in which the edge occurs.

The model was designed to support the requirements to model genome data, but also is generic enough to support complex interconnected structures. The distinction between schema and instance is blurred. Its nesting levels increase the complexity of modeling and processing.

A.6 Graph Object-Oriented Data Model (GOOD)

The Graph Object-Oriented Data Model [49] is a proposal oriented mainly to develop database end-user graphical interfaces [90]. In GOOD, schema and instances are represented by directed

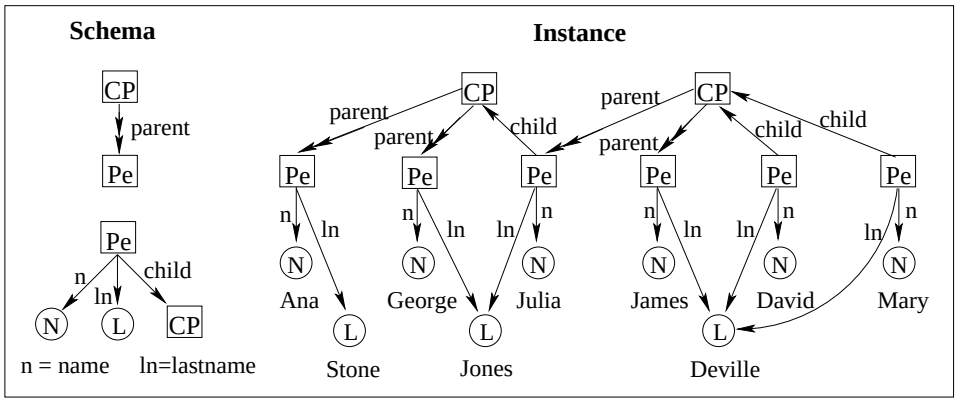


Figure A.7: GOOD. In the schema, we use printable nodes *N* and *L* to represent names and lastnames respectively and non-printable nodes *Pe*(rson) and *CP* to represent relations Name-Lastname and Child-Parent respectively. A double arrow indicates non-functional relationship, and a simple arrow indicates functional relationship. The instance is got by assigning values to printable nodes and instantiating the CP and PE nodes.

labeled graphs, and the data manipulation is expressed by graph transformations. An example of its application is the database management system presented in the work of Gemis et al. [232]. The GOOD schema and instance for the general example is presented in Figure A.7.

The model permits only two types of nodes, non-printable nodes (denoted by squares) and printable nodes (denoted by circles). There is no distinction between atomic, composed and set objects. There are two types of edges, functional (have a unique value, denoted by $\rightarrow$) and non-functional (multi-valued and denoted by $\twoheadrightarrow$). A more detailed version [60] added node and edges for representing set containment, object composition, generalization, and specialization.

GOOD includes a data transformation language with graphical syntax and semantics. It contains four elementary graph transformation operations: addition and deletion of nodes and edges, plus a fifth operation called abstraction, used to group nodes on the basis of common functional or non-functional properties. The specification of all these operations relies on the notion of pattern to describe subgraphs in the database instance. GOOD studies other issues like macros (for more succinct expression of frequent operations), computational-completeness of the query language, and simulation of object-oriented characteristics (i.e. inheritance).

The model presented introduced several useful features. The notion of printable and non-printable nodes is relevant for the design of graphical interfaces. It has a simple definition of multivalued relations and allows recursive relations. It solves in a balanced way the data redundancy problem.

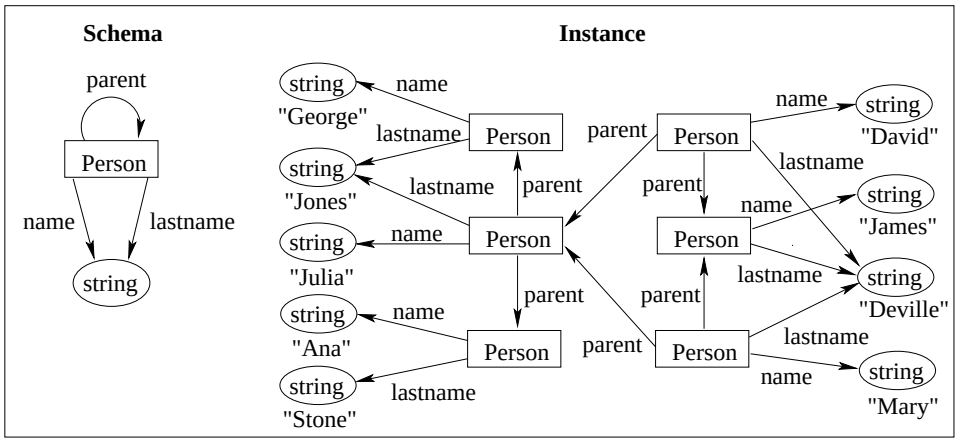


Figure A.8: GMOD. In the schema, nodes represent abstract objects (*Person*) and labeled edges establish relations with primitive objects (properties *name* and *lastname*) and other abstract objects (*parent* relation). For building an instance, the schema for each person needs to be instantiated by assigning values to oval nodes.

A.7 Graph-Oriented Object Manipulation (GMOD)

GMOD [48] is a proposal of a general model for object database concepts focused on graph-oriented database user interfaces. Schema and instance are labeled digraphs (see Figure A.8).

The *schema* graph has two class of nodes, *abstract objects* (rectangular shape) representing class names and, *primitive objects* (oval shape) representing basic types. Edges represent properties of abstract objects. Distinction between single-value and multi-value properties is not considered.

The *instance* graph contains the data and includes instance nodes for abstract and primitive objects (represented as in the schema level). The latter have an additional label indicating their value (according to the primitive object domain). The same edges defined in the schema are used to represent the properties of instance objects, but their use is not necessarily required (incomplete information is allowed). Formally, there is a graph morphism from the graph instance (without the labels indicating value) to the schema.

The model uses graph pattern matching as a uniform object manipulation primitive for querying, specification, updating, manipulation, viewing and browsing. G-Log [66] is a proposal of a declarative query language for graphs, which works on the data structures defined by GMOD. Queries in G-log are expressed by programs which consist of a number of rules and use patterns (denoted as graphs with variables in the nodes and predicates in the edges) to match subgraphs in the instance.

The model allows a simple representation of objects and relations, incomplete information, and permits avoiding redundancy of data. The issue of property-dependent identity and a not completely transparent notion of object-ID incorporates some complexities in the modeling.

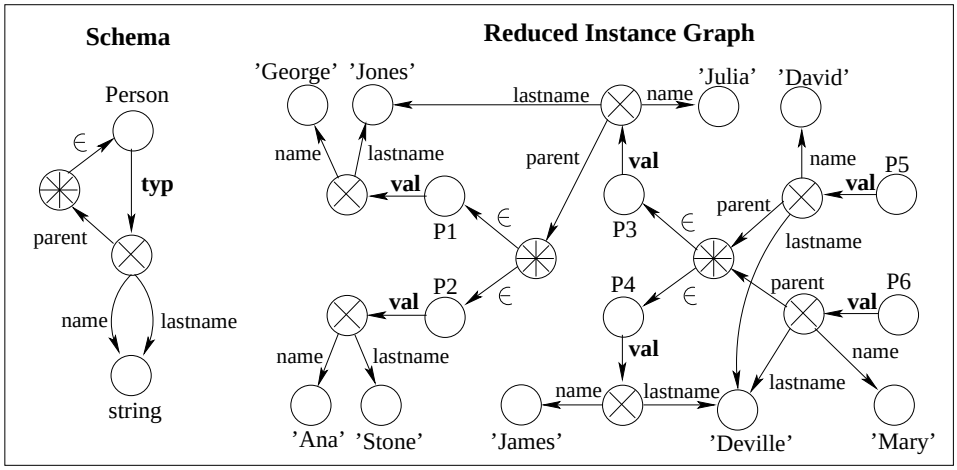


Figure A.9: PaMaL. The example shows all the nodes defined in PaMaL: basic type (*string*), class (*Person*), tuple (⊗), set (⊛) nodes for the schema level, and atomic (*George*, *Ana*, etc.), instance (*P1*, *P2*, etc), tuple and set nodes for the instance level. Note the use of edges ∈ to indicate elements in a set, and the edge **typ** to indicate the type of class *Person* (these edges are changed to **val** in the instance level).

A.8 Object-Oriented Pattern Matching Language (PaMaL)

PaMaL is a graphical data manipulation language that uses patterns (represented as graphs) to specify the parts of the instance on which the operation has to be executed. Gemis and Paredaens [73] proposed this pattern-based query language based on a graphical object-oriented db-model as an extension of GOOD by an explicit representation of tuples and sets. An example of PaMaL schema and instance is presented in Figure A.9.

The *schema* defines four types of Nodes: ◯ class nodes (upper-case labels), ◯ basic-type nodes (lower-case labels), ⊗ tuple nodes, and ⊛ set nodes. There are four kinds of edges, indicating attribute of a tuple, type of the elements in a set (labeled with ∈), type of the classes (labeled with **typ**), and hierarchical relationship (labeled with **isa**).

An *instance* graph may contain *atomic*, *instance*, *tuple* and *set* nodes (they are determined by the schema). *Atomic objects* are labeled with values from their domains and *instance objects* are labeled with object-ID's. *tuple* and *set* objects are identified by their outgoing edges, motivating the notion of *reduced instance graph* to merge nodes that represent the same set or tuple. To refer to the node that describes the properties (or content) of an object, an edge labeled **val** is used and represents the edge **typ** in the schema.

PaMaL presents operators for addition, deletion (of nodes and edges) and an special operation that reduces instance graphs. It incorporates loop, procedure and program constructs that makes it a computationally complete language. Among the highlights of the model are the explicit definition of sets and tuples, the multiple inheritance, and the use of graphics to describe queries.

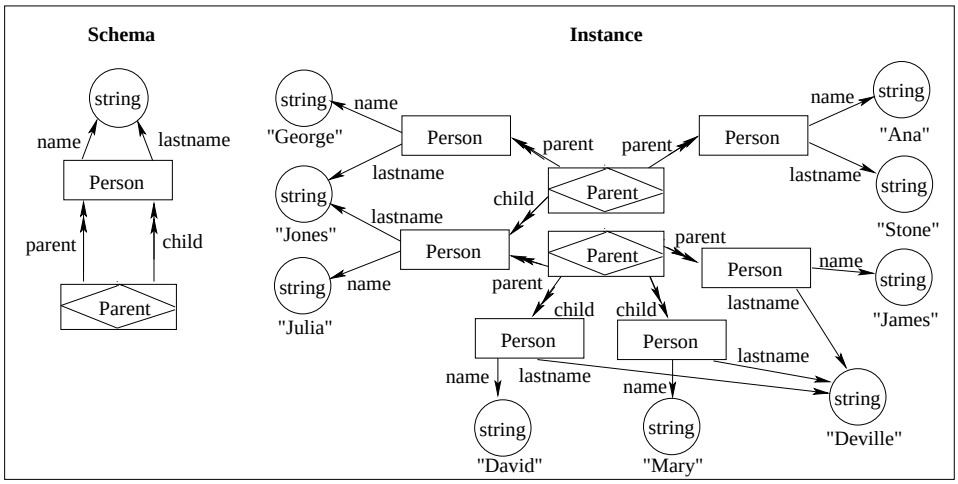


Figure A.10: GOAL: The schema presented in the example shows the use of the object node *Person* with properties Name and Lastname. The association node *Parent* and the double headed edges *parent* and *child* allow to express the relation Person-Parent. At the instance level, we assign values to value nodes (*string*) and create instances for object and association nodes. Note that nodes with same value were merged (e.g. *Deville*).

A.9 Graph-based Object and Association Language (GOAL)

Motivated by the introduction of more complex db-models like object-oriented ones, and directed to offer the user a consistent graphical interface, Hidders and Paredaens [59] proposed a graph based db-model for describing schemas and instances of object databases. GOAL extends the model of GOOD by adding the concept of *association* nodes (similar to the entity relationship model). The main difference between associations and objects is that the identity of objects is independent of their properties, whereas associations are considered identical if they have the same properties. An example is presented in Figure A.10.

Schema and instances in GOAL are represented as finite directed labeled graphs. A schema allows to define three types of nodes: *object nodes* that represent objects (rectangular nodes); *value nodes* that represent printable values such as string, integers or booleans (round nodes) and; *association nodes* that represent associations or relations among more than two nodes (diamond shape nodes). Objects and associations may have properties that are represented by edges. The model allows representation of functional properties (single headed edges) and multi-valued properties (double headed edges), as well as ISA relations (double unlabeled arrows). An instance in GOAL assigns values to value nodes and creates instances for object and association nodes.

GOAL introduces the notion of consistent schema to enforce that objects only belong to the class they are labeled with and its super-classes. In addition GOAL presents a graph data manipulation language with operations for addition and deletion based on pattern matching. The addition (deletion) operation adds (deletes) nodes and/or edges at the instance level. A

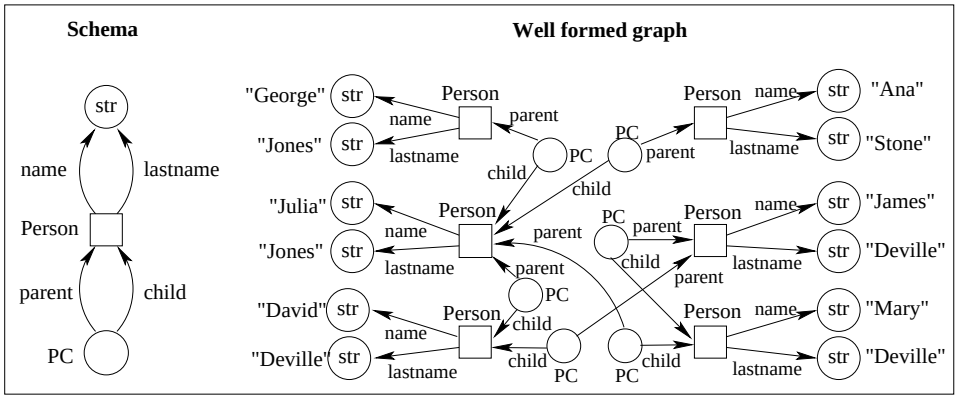


Figure A.11: GDM. In the schema each entity Person (object node represented as a square) has assigned the attributes *name* and *lastname* (basic value nodes represented round and labeled *str*). We use the composite-value node *PC* to establish the relationship *Parent-Child*. Note the redundancy introduced by the node *PC*. The instance is built by instantiating the schema for each person.

finite sequence of additions and deletions is called a *transformation*.

There are several novelties introduced by this model. Association nodes allow simple definition of multi-attribute and multi-valued relations. In contrast to the Entity Relationship model, GOAL supports relations between associations. Properties are optional, therefore it is possible to model incomplete information. Additionally, GOAL defines restrictions that introduce notions of consistent schema and weak instance.

A.10 Graph Data Model (GDM)

GDM [57, 233] is a graph-based db-model based on GOOD, that adds explicit complex values, inheritance and *n*-ary symmetric relationships. Schema and instances in GDM are described by labeled graphs called instance graph and schema graph respectively. An example is presented in Figure A.11.

A *schema* graph contains nodes that represent classes and edges labeled with attribute names indicating that entities in that class may have that attribute. Three types of class nodes are allowed: *object*, *composite-value* and, *basic value*. An edge denoted by a double-line arrow defines an *ISA* relation between class nodes.

In an *instance* graph, nodes represent entities and edges represent attributes of these entities. We can have *object nodes* (depicted squared), *composite-value nodes* (round empty) and *basic value nodes* (round labeled with a basic-type name). An object node is labeled with zero o more class names indicating their membership to certain classes. If several edges with the same label leave a node, then it is a single set-valued attribute.

GDM introduces the concept of well-formed graph defining four constraints: (I-BVA) no

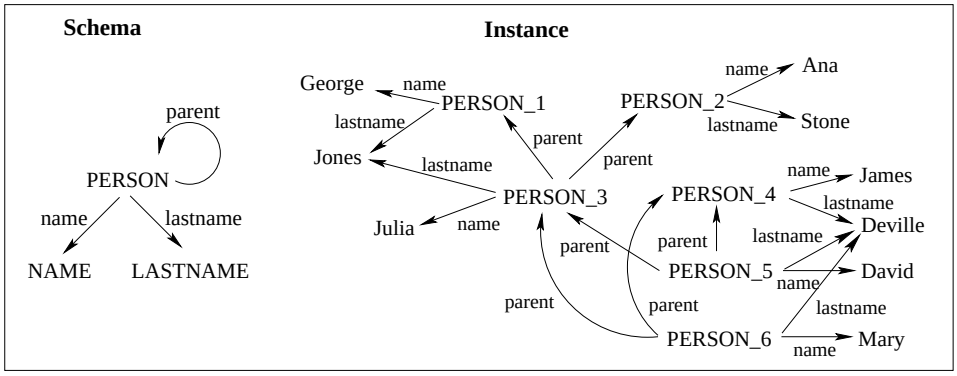


Figure A.12: Gram. At the schema level we use generalized names for definition of entities and relations. At the instance level, we create instance labels (e.g. PERSON_1) to represent entities, and use the edges (defined in the schema) to express relations between data and entities.

edge leaves from a basic-value node; (I-BVT) each basic value node has assigned a real value that is in the domain of the basic-type of the node; (I-NS) for each class-free node *n* there is a path that ends in *n* and starts in a class-labeled node; and (I-REA) composite-value nodes have either exactly one incoming edge or are labeled with exactly one class name, but not both. In addition the model considers the notion of consistency defining *extension relations* which are many-to-many relations between the nodes in the data graph and nodes in the schema graph, indicating correspondence between entities and classes.

The proposal includes a graph-based update language called GUL, that is based on pattern matching. GUL permits addition and deletion operations, plus a reduction operation that reduces well-formed data graphs to instance graphs by merging similar basic-value nodes and similar composite-value nodes.

The GDM model presents the following benefits. The independence of the definition of the notions of schema and instance permits that instances can exist without a schema, allowing representation of semi-structured data. It permits the explicit representation of complex values, inheritance (using ISA edges) and definition of n-ary symmetric relationships. The composite-value nodes allow simple definition of multi-attribute and multi-valued relations. Finally, let us remark that this model introduces notions of consistency and well-formed graphs.

A.11 Gram: A Graph Data Model and Query Language

Motivated by hypertext querying, Amann and Scholl [62] introduced Gram, a graph db-model where data are organized as a graph (see example in Figure A.12).

A schema in Gram is a directed labeled multigraph, where each node is labeled with a symbol called a *type*, which has associated a domain of values. In the same way, each edge has assigned a label representing a relation between types . A feature of Gram is the use of

regular expressions for explicit definition of paths called *walks*. An alternating sequence of nodes and edges represent a walk, which combined with other walks conforms other special objects called *hyperwalks*.

For querying the model (particularly path-like queries), an algebraic language based on regular expressions is proposed. For this purpose a hyperwalk algebra is defined, which presents unary operations (projection, selection, renaming) and binary operations (join, concatenation, set operations), all closed under the set of hyperwalks.

A.12 Related Data Models

Besides the models reviewed, there are other proposals that present graph-like features, although not explicitly designed to model the structure and connectivity of the information. In this section we will describe the most relevant of these.

A.12.1 GraphDB

Güting [64] proposed an explicit model named GraphDB, which allows simple modeling of graphs in an object-oriented environment. The model permits an *explicit representation* of graphs by defining object classes whose objects can be viewed as nodes, edges and explicitly stored paths of a graph (which is the whole database instance). A database in GraphDB is a collection of object classes partitioned into three kinds of classes: simple, link and path classes. Also there are data types, object types and tuple types. There are four types of operators to query GraphDB data: Derive statements, Rewrite operations, Union operator, and Graph operations (Shortest path search).

The idea of modeling graphs using object-oriented concepts is presented in other proposals, generically called *object-oriented graph models*. A typical example is GOQL [61], a proposal of graph query language for modeling and querying of multimedia application graphs (represented as DAGs). This proposal defines an object-oriented db-model (similar to GraphDB) that defines four types of objects: node, edge, path and graph. GOQL uses an SQL-like syntax for construction, querying and manipulation of such objects.

A.12.2 Database Graph Views

A database graph view [56] provides a functional definition of graphs over data that can be stored in either relational, object-oriented or file systems. In other words, the model proposes the definition of personalized graph views of the data with management and querying purposes, independent of its implementation. In brief the model defines underlying graphs over the database and proposes a set of primitives called *derivation operators* for definition and querying of graph views. Unary derivation operators allow selection of nodes and edges.

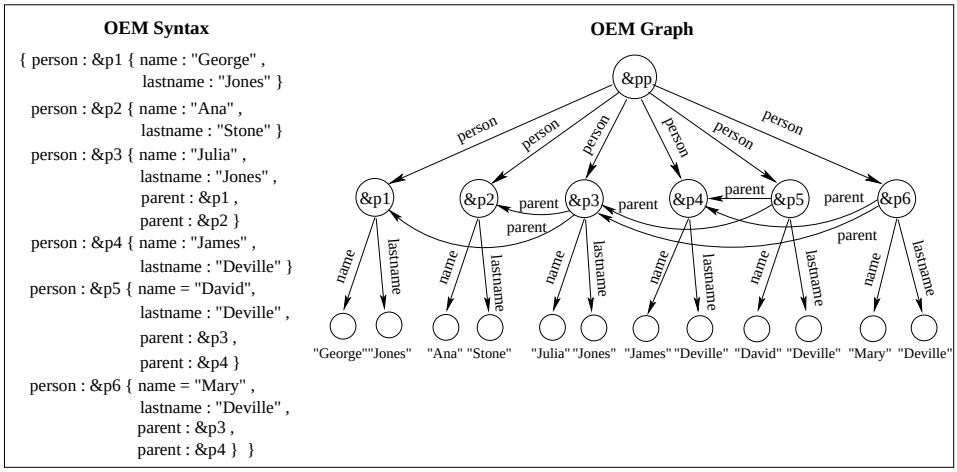


Figure A.13: Object Exchange Model (OEM). Schema and instance are mixed. The data are modeled beginning in a root node &pp, with children *person nodes*, each of them identified by an Object-ID (e.g. &p2). These nodes have children that contain data (*name* and *lastname*) or references to other nodes (*parent*). Referencing permits to establish relations between distinct hierarchical levels. Note the tree structure obtained if one forgets the pointers to OIDs, a characteristic of semistructured data.

Binary derivation operators are used to build new graph views resulting from the union, intersection or difference of two graph views.

A.12.3 Object Exchange Model (OEM)

OEM [52] is a semistructured db-model that allows simple and flexible modeling of complex features of a source using the ideas of nesting and object identity from object-oriented db-models (features such as classes, methods and inheritance are omitted). The main motivation of OEM was the information integration problem. Therefore it defines a syntax that is well suited for information exchange in heterogeneous dynamic environments. An example of OEM is presented in Figure A.13.

The data in OEM can be represented as a rooted directed connected graph with Object-IDs representing node-labels and OEM-labels representing edge-labels. Atomic objects are leaf nodes where the OEM-value is the node value. The main feature of OEM data is that it is self-describing, in the sense that it can be parsed without recurring to an external schema and uses human understandable labels that add semantic information about objects. Due to the fact that there is no notion of schema or object class (although each object defines its own schema), OEM offers the flexibility needed in heterogeneous dynamic environments.

A.12.4 eXtended Markup Language (XML)

The XML [42] model did not originate in the database community. It was introduced as standard for exchanging information between Web applications. XML allows annotating data with information about their meaning rather than just their presentation [234]. From an abstract point of view, XML data are labeled ordered trees (with labels on nodes), where internal nodes define the structure and leaves the data (schema and data are mixed).

Compared to graph data db-models, 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 semistructured 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 db-models this information is described by the schema graph in a more flexible fashion by using relations between entities. From this point of view, graph db-models use connections to explicitly represent generalization, compositions, hierarchy, classification, and any/other types of relations.

Bibliography

[1] Grigoris Antoniou and Frank van Harmelen. *A Semantic Web Primer*. The MIT Press, 2004.

[2] Tim Berners-Lee, James Hendler, and Ora Lassila. The Semantic Web. *Scientific American*, May 2001.

[3] Tim Berners-Lee. *Weaving the Web: The Original Design and Ultimate Destiny of the World Wide Web by Its Inventor*. Harper San Francisco, 1999. ISBN 1402842937.

[4] Graham Klyne and Jeremy Carroll. Resource Description Framework (RDF) Concepts and Abstract Syntax, February 2004. URL `http://www.w3.org/TR/2004/REC-rdf-concepts-20040210/`.

[5] The Friend of a Friend (FOAF) project, 2000. URL `http://www.foaf-project.org/`.

[6] Wordnet - A lexical database for the English language, 2006. URL `http://wordnet.princeton.edu/`.

[7] Wikipedia3, 2005. URL `http://labs.systemone.at/wikipedia3`.

[8] The DBpedia Knowledge Base, 2007. URL `http://wiki.dbpedia.org/About`.

[9] Semantic Interoperability of Metadata and Information in unLike Environments (SIMILE), 2003. URL `http://simile.mit.edu/`.

[10] Avi Silberschatz, Henry F. Korth, and S. Sudarshan. Data Models. *ACM Computing Surveys*, 28(1):105–108, 1996. ISSN 0360-0300.

[11] William C. McGee. On User Criteria for Data Model Evaluation. *ACM Transactions on Database Systems (TODS)*, 1(4):370–387, 1976. ISSN 0362-5915.

[12] E. F. Codd. Data Models in Database Management. In *Proceedings of the 1980 Workshop on Data abstraction, Databases and Conceptual Modeling*, pages 112–114. ACM Press, 1980. ISBN 0-89791-031-1.

[13] E. F. Codd. A Relational Model of Data for Large Shared Data Banks. *Communications of the ACM*, 26(1):64–69, 1983. ISSN 0001-0782.

[14] Patrick Hayes. RDF Semantics, February 2004. URL http://www.w3.org/TR/2004/REC-rdf-mt-20040210/.

[15] Ora Lassila and Ralph R. Swick. Resource Description Framework (RDF) Model and Syntax Specification, February 1999. URL http://www.w3.org/TR/1999/REC-rdf-syntax-19990222/.

[16] R.V.Guha, Ora Lassila, Eric Miller, and Dan Brickley. Enabling Inferencing. *The Query Languages Workshop (QL)*, Dec 1998.

[17] Eric Prud'hommeaux and Andy Seaborne. SPARQL Query Language for RDF. W3C Recommendation 15 January, 2008. URL http://www.w3.org/TR/2008/REC-rdf-sparql-query-20080115/.

[18] Claudio Gutierrez, Carlos Hurtado, and Alberto O.Mendelzon. Foundations of Semantic Web Databases. In *Proceedings of the 23rd Symposium on Principles of Database Systems (PODS)*, pages 95–106, June 14-16 2004.

[19] Peter Haase, Jeen Broekstra, Andreas Eberhart, and Raphael Volz. A Comparison of RDF Query Languages. In *Proceedings of the 3rd International Semantic Web Conference (ISWC)*, number 3298 in Lecture Notes in Computer Science, page 502. Springer-Verlag, November 7-11 2004.

[20] Tim Berners-Lee, Yuhsin Chen, Lydia Chilton, Dan Connolly, Ruth Dhanaraj, James Hollenbach, Adam Lerer, and David Sheets. Tabulator: Exploring and Analyzing linked data on the Semantic Web. In *Procedings of the 3rd International Semantic Web User Interaction Workshop (SWUI06)*, 2006.

[21] The Unicode Consortium, 1991. URL http://www.unicode.org/.

[22] Uniform Resource Identifier (URI): Generic Syntax, 2005. URL http://tools.ietf.org/html/rfc3986.

[23] Internationalized Resource Identifiers (IRIs), 2005. URL http://tools.ietf.org/html/rfc3987.

[24] Tim Bray, Jean Paoli, C. M. Sperberg-McQueen, Eve Maler, and Francois Yergeau. Extensible Markup Language (XML) 1.0 (Third Edition), W3C Recommendation, 04 February 2004. URL http://www.w3.org/TR/2004/REC-xml-20040204/.

[25] Tim Bray, Dave Hollander, Andrew Layman, and Richard Tobin. Namespaces in XML 1.1, W3C Recommendation, 4 February 2001. URL http://www.w3.org/TR/2004/REC-xml-names11-20040204/.

[26] Dave Beckett. RDF/XML Syntax Specification, Feb 2004. URL http://www.w3.org/TR/2004/REC-rdf-syntax-grammar-20040210/.

[27] Tim Berners-Lee. Notation 3 - An RDF language for the Semantic Web, 2001. URL http://www.w3.org/DesignIssues/Notation3.

[28] Dan Brickley and R.V.Guha. RDF Vocabulary Description Language 1.0: RDF Schema, February 2004. URL http://www.w3.org/TR/2004/REC-rdf-schema-20040210/.

[29] RxPath Specification Proposal, 2004. URL http://rx4rdf.liminalzone.org/RxPathSpec.

[30] W3C in 7 points. URL http://www.w3.org/Consortium/Points/.

[31] Semantically-Interlinked Online Communities (SIOC), 2004. URL http://sioc-project.org/.

[32] BioGateway, 2008. URL http://www.semantic-systems-biology.org/.

[33] D. C. Tsichritzis and F. H. Lochovsky. Hierarchical Data-Base Management: A Survey. *ACM Computing Surveys*, 8(1):105–123, 1976. ISSN 0360-0300.

[34] Robert W. Taylor and Randall L. Frank. CODASYL Data-Base Management Systems. *ACM Computing Surveys*, 8(1):67–103, 1976. ISSN 0360-0300.

[35] Larry Kerschberg, Anthony C. Klug, and Dennis Tsichritzis. A Taxonomy of Data Models. In *Proceedings of Systems for Large Data Bases (VLDB)*, pages 43–64. North Holland and IFIP, Sept 1976. ISBN 0-7204-0546-7.

[36] E. F. Codd. A Relational Model of Data for Large Shared Data Banks. *Communications of the ACM*, 13(6):377–387, 1970. ISSN 0001-0782.

[37] Joan Peckham and Fred J. Maryanski. Semantic Data Models. *ACM Computing Surveys*, 20(3):153–189, 1988.

[38] Peter Pin-Shan Chen. The Entity-Relationship Model - Toward a Unified View of Data. *ACM Transactions on Database Systems (TODS)*, 1(1):9–36, 1976. ISSN 0362-5915.

[39] W. Kim. Object-Oriented Databases: Definition and Research Directions. *IEEE Transactions on Knowledge and Data Engineering (TKDE)*, 2(3):327–341, 1990. ISSN 1041-4347.

[40] Catriel Beeri. Data Models and Languages for Databases. In *Proceedings of the 2nd International Conference on Database Theory (ICDT)*, volume 326 of *LNCS*, pages 19–40. Springer, Aug - Sept 1988. ISBN 3-540-50171-1.

[41] Peter Buneman. Semistructured Data. In *Proceedings of the 16th Symposium on Principles of Database Systems (PODS)*, pages 117–121. ACM Press, May 1997. ISBN 0-89791-910-6.

[42] Tim Bray, Jean Paoli, and C. M. Sperberg-McQueen. Extensible Markup Language (XML) 1.0, W3C Recommendation, 10 February 1998. URL `http://www.w3.org/TR/1998/REC-xml-19980210`.

[43] Shamkant B. Navathe. Evolution of Data Modeling for Databases. *Communications of the ACM*, 35(9):112–123, 1992. ISSN 0001-0782.

[44] Jonathan Hayes and Claudio Gutierrez. Bipartite Graphs as Intermediate Model for RDF. In *Proceedings of the 3th International Semantic Web Conference (ISWC)*, number 3298 in LNCS, pages 47–61. Springer-Verlag, Nov 2004.

[45] Georg Lausen. Relational Databases in RDF: Keys and Foreign Keys. In *Semantic Web, Ontologies and Databases, VLDB Workshop (SWDB-ODBIS)*, volume 5005 of *LNCS*, pages 43–56. Springer, 2007.

[46] Dave Beckett and Jan Grant. Semantic Web Scalability and Storage: Mapping Semantic Web Data with RDBMSes, 2003. URL `http://www.w3.org/2001/sw/Europe/reports/scalable\_rdbms\_mapping\_report/`.

[47] Mark Levene and Alexandra Poulovassilis. An Object-Oriented Data Model Formalised Through Hypergraphs. *Data & Knowledge Engineering (DKE)*, 6(3):205–224, 1991. ISSN 0169-023X.

[48] 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 (EDBT)*, volume 580 of *LNCS*, pages 21–38. Springer, March 1992. ISBN 3-540-55270-7.

[49] Marc Gyssens, Jan Paredaens, Jan Van den Bussche, and Dirk Van Gucht. A Graph-Oriented Object Database Model. In *Proceedings of the 9th Symposium on Principles of Database Systems (PODS)*, pages 417–424. ACM Press, 1990.

[50] Christophe Lécluse, Philippe Richard, and Fernando Vélez. O2, an Object-Oriented Data Model. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 424–433. ACM Press, June 1988.

[51] Valerie Bönström, Annika Hinze, and Heinz Schweppe. Storing RDF as a graph. In *Proceedings of the First Latin American Web Congress (LA-WEB'03)*. IEEE Computer Society, 2003.

[52] Yannis Papakonstantinou, Hector Garcia-Molina, and Jennifer Widom. Object Exchange across Heterogeneous Information Sources. In *Proceedings of the 11th International Conference on Data Engineering (ICDE)*, pages 251–260. IEEE Computer Society, 1995.

[53] Yolanda Gil and Varun Ratnakar. A Comparison of (Semantic) Markup Languages. In *Proceedings of the 15th International FLAIRS Conference*, May 14-16 2002.

[54] Sinuhe Arroyo, Ying Ding, Ruben Lara, Michael Stollberg, and Dieter Fensel. Semantic Web Languages. Strengths and Weakness. In *International Conference in Applied computing (IADIS04)*, March 23-26 2004.

[55] 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)*, pages 592–603. ACM Press, 2002. ISBN 1-58113-449-5.

[56] Alejandro Gutiérrez, Philippe Pucheral, Hermann Steffen, and Jean-Marc Thévenin. Database Graph Views: A Practical Model to Manage Persistent Graphs. In *Proceedings of the 20th International Conference on Very Large Data Bases (VLDB)*, pages 391–402. Morgan Kaufmann, September 1994. ISBN 1-55860-153-8.

[57] Jan Hidders. Typing Graph-Manipulation Operations. In *Proceedings of the 9th International Conference on Database Theory (ICDT)*, pages 394–409. Springer-Verlag, 2002. ISBN 3-540-00323-1.

[58] M. Graves, E. R. Bergeman, and C. B. Lawrence. A Graph-Theoretic Data Model for Genome Mapping Databases. In *Proceedings of the 28th Hawaii International Conference on System Sciences (HICSS)*, page 32. IEEE Computer Society, 1995.

[59] Jan Hidders and Jan Paredaens. GOAL, A Graph-Based Object and Association Language. *Advances in Database Systems: Implementations and Applications, CISM*, pages 247–265, Sept. 1993.

[60] Marc Gyssens, Jan Paredaens, Jan Van den Bussche, and Dirk Van Gucht. A Graph-Oriented Object Database Model. Technical Report 91-27, University of Antwerp (UIA), Belgium, March 1991.

[61] Lei Sheng, Z. Meral Ozsoyoglu, and Gultekin Ozsoyoglu. A Graph Query Language and Its Query Processing. In *Proceedings of the 15th International Conference on Data Engineering (ICDE)*, pages 572–581. IEEE Computer Society, March 1999.

[62] Bernd Amann and Michel Scholl. Gram: A Graph Data Model and Query Language. In *European Conference on Hypertext Technology (ECHT)*, pages 201–211. ACM, Nov. - Dec. 1992. ISBN 0-89791-547-X.

[63] Norbert Kiesel, Andy Schurr, and Bernhard Westfechtel. GRAS: A Graph-Oriented Software Engineering Database System. In *IPSEN Book*, pages 397–425, 1996.

[64] Ralf Hartmut Güting. GraphDB: Modeling and Querying Graphs in Databases. In *Proceedings of 20th International Conference on Very Large Data Bases (VLDB)*, pages 297–308. Morgan Kaufmann, Sept. 1994. ISBN 1-55860-153-8.

[65] H. S. Kunii. DBMS with Graph Data Model for Knowledge Handling. In *Proceedings of the 1987 Fall Joint Computer Conference on Exploring technology: today and tomorrow*, pages 138–142. IEEE Computer Society Press, 1987. ISBN 0-8186-0811-0.

[66] Jan Paredaens, Peter Peelman, and Letizia Tanca. G-Log: A Graph-Based Query Language. *IEEE Transactions on Knowledge and Data Engineering (TKDE)*, 7(3):436–453, 1995. ISSN 1041-4347.

[67] Mark Levene and Alexandra Poulovassilis. The Hypernode Model and its Associated Query Language. In *Proceedings of the 5th Jerusalem Conference on Information technology*, pages 520–530. IEEE Computer Society Press, 1990. ISBN 0-8186-2078-1.

[68] Alexandra Poulovassilis and Mark Levene. A Nested-Graph Model for the Representation and Manipulation of Complex Objects. *ACM Transactions on Information Systems (TOIS)*, 12(1):35–68, 1994. ISSN 1046-8188.

[69] Mark Levene and George Loizou. A Graph-Based Data Model and its Ramifications. *IEEE Transactions on Knowledge and Data Engineering (TKDE)*, 7(5):809–823, 1995.

[70] Mariano Consens and Alberto Mendelzon. Hy+: a Hygraph-based Query and Visualization System. *SIGMOD Record*, 22(2):511–516, 1993.

[71] Gabriel M. Kuper and Moshe Y. Vardi. A New Approach to Database Logic. In *Proceedings of the 3th Symposium on Principles of Database Systems (PODS)*, pages 86–96. ACM Press, April 1984. ISBN 0-89791-128-8.

[72] Gabriel M. Kuper and Moshe Y. Vardi. The Logical Data Model. *ACM Transactions on Database Systems (TODS)*, 18(3):379–413, 1993.

[73] Marc Gemis and Jan Paredaens. An Object-Oriented Pattern Matching Language. In *Proceedings of the First JSSST International Symposium on Object Technologies for Advanced Software*, pages 339–355. Springer-Verlag, 1993. ISBN 3-540-57342-9.

[74] Nick Roussopoulos and John Mylopoulos. Using Semantic Networks for Database Management. In *Proceedings of the International Conference on Very Large Data Bases (VLDB)*, pages 144–172. ACM, Sept 1975.

[75] Michel Mainguenaud. Simatic XT: A Data Model to Deal with Multi-scaled Networks. *Computer, Environment and Urban Systems*, 16:281–288, 1992.

[76] Frank WM. Tompa. A Data Model for Flexible Hypertext Database Systems. *ACM Transactions on Information Systems (TOIS)*, 7(1):85–100, 1989. ISSN 1046-8188.

[77] Carolyn Watters and Michael A. Shepherd. A Transient Hypergraph-Based Model for Data Access. *ACM Transactions on Information Systems (TOIS)*, 8(2):77–102, 1990. ISSN 1046-8188.

[78] David W. Shipman. The Functional Data Model and the Data Language DAPLEX. *ACM Transactions on Database Systems (TODS)*, 6(1):140–173, 1981. ISSN 0362-5915.

[79] Deborah L. McGuinness and Frank van Harmelen. OWL Web Ontology Language Overview, W3C Recommendation, 10 February 2004. URL `http://www.w3.org/TR/2004/REC-owl-features-20040210/`.

[80] Mark Graves, Ellen R. Bergeman, and Charles B. Lawrence. Graph Database Systems for Genomics. *IEEE Engineering in Medicine and Biology. Special issue on Managing Data for the Human Genome Project*, 11(6), 1995.

[81] Serge Abiteboul, Dallan Quass, Jason McHugh, Jennifer Widom, and Janet L. Wiener. The Lorel query language for semistructured data. *International Journal on Digital Libraries (JODL)*, 1(1):68–88, 1997.

[82] Peter Buneman, Susan Davidson, Gerd Hillebrand, and Dan Suciu. A Query Language and Optimization Techniques for Unstructured Data. *SIGMOD Record*, 25(2):505–516, 1996. ISSN 0163-5808.

[83] Richard Hull and Roger King. Semantic Database Modeling: Survey, Applications, and Research Issues. *ACM Computing Surveys*, 19(3):201–260, 1987. ISSN 0360-0300.

[84] Serge Abiteboul and Richard Hull. IFO: A Formal Semantic Database Model. In *Proceedings of the 3th Symposium on Principles of Database Systems (PODS)*, pages 119–132. ACM Press, 1984. ISBN 0-89791-128-8.

[85] Michael Hammer and Dennis McLeod. The Semantic Data Model: A Modelling Mechanism for Data Base Applications. In *Proceedings of the 1978 ACM SIGMOD International Conference on Management of Data*, pages 26–36. ACM, May 1978.

[86] Serge Abiteboul. Querying Semi-Structured Data. In *Proceedings of the 6th International Conference on Database Theory (ICDT)*, volume 1186 of *LNCS*, pages 1–18. Springer, Jan. 1997. ISBN 3-540-62222-5.

[87] Lincoln D. Stein and Jean Tierry-Mieg. AceDB: A genome Database Management System. *Computing in Science & Engineering (CiSE)*, 1(3):44–52, 1999.

[88] Mary Fernández, Daniela Florescu, Jaewoo Kang, Alon Levy, and Dan Suciu. Catching the boat with Strudel: Experiences with a Web-site Management System. In *Proceedings of the 1998 ACM SIGMOD International Conference on Management of Data*, pages 414–425. ACM Press, June 1998.

[89] Michel Mainguenaud. Modelling the Network Component of Geographical Information Systems. *International Journal of Geographical Information Systems (IJGIS)*, 9(6): 575–593, 1995.

[90] 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*, pages 24–33. ACM Press, 1990. ISBN 0-89791-365-5.

[91] M. E. J. Newman. The Structure and Function of Complex Networks. *SIAM Review*, 45 (2):167–256, 2003.

[92] Reka Albert and Albert-Laszlo Barabási. Statistical mechanics of complex networks. *Reviews of Modern Physics*, 74:47, Jan 2002.

[93] S. N. Dorogovtsev and J. F. F. Mendes. *Evolution of Networks - From Biological Nets to the Internet and WWW*. Oxford University Press, 2003.

[94] Frank Olken. Tutorial on Graph Data Management for Biology. *IEEE Computer Society Bioinformatics Conference (CSB)*, Aug 2003.

[95] H. V. Jagadish and Frank Olken. Data Management for the Biosciences: Report of the NLM Workshop on Data Management for Molecular and Cell Biology. Technical Report LBNL-52767, National Library of Medicine, 2003.

[96] M. Tsvetovat, J. Diesner, and K.M. Carley. NetIntel: A Database for Manipulation of Rich Social Network Data. Technical Report CMU-ISRI-04-135, Carnegie Mellon

University, School of Computer Science, Institute for Software Research International, 2004.

[97] Robert A. Hanneman. Introduction to Social Network Methods. Technical report, Department of Sociology, University of California, Riverside, 2001.

[98] Amit Sheth, Boanerges Aleman-Meza, I. Budak Arpinar, Christian Halaschek-Wiener, Cartic Ramakrishnan, Clemens Bertram, Yashodhan Warke, David Avant, F. Sena Arpinar, Kemafor Anyanwu, and Krys Kochut. Semantic Association Identification and Knowledge Discovery for National Security Applications. *Journal of Database Management (JDM)*, 16(1):33–53, Jan. – March 2005.

[99] Barry Wellman, Janet Salaff, Dimitrina Dimitrova, Laura Garton, Milena Gulia, and Caroline Haythornthwaite. Computer Networks as Social Networks: Collaborative Work,Telework, and Virtual Community. *Annual Review of Sociology*, 22:213–238, 1996.

[100] Ulrik Brandes. *Network Analysis*. Number 3418 in LNCS. Springer-Verlag, 2005.

[101] D. J. de S. Price. Networks of Scientific papers. *Science*, 149:510–515, 1965.

[102] Daniela Florescu, Alon Levy, and Alberto O .Mendelzon. Database Techniques for the World-Wide Web: A Survey. *SIGMOD Record*, 27(3):59–74, 1998. ISSN 0163-5808.

[103] Ravi Kumar, Prabhakar Raghavan, Sridhar Rajagopalan, D. Sivakumar, Andrew Tomkins, and Eli Upfal. The Web as a Graph. In *Proceedings of the 19th Symposium on Principles of Database Systems (PODS)*, pages 1–10. ACM Press, May 2000. ISBN 1-58113-214-X.

[104] Andrei Broder, Ravi Kumar, Farzin Maghoul, Prabhakar Raghavan, Sridhar Rajagopalan, Raymie Stata, Andrew Tomkins, and Janet Wiener. Graph structure in the Web. In *Proceedings of the 9th International World Wide Web conference on Computer networks : the international journal of computer and telecommunications networking*, pages 309–320. North-Holland Publishing Co., 2000.

[105] Wolfgang Nejdl, Wolf Siberski, and Michael Sintek. Design Issues and Challenges for RDF- and Schema-Based Peer-to-Peer Systems. *SIGMOD Record*, 32(3):41–46, 2003.

[106] W3C. WordNet RDF and OWL Files, January 2007. URL `http://www.w3.org/2006/03/wn/wn20/`.

[107] Shashi Shekhar, Mark Coyle, Brajesh Goyal, Duen-Ren Liu, and Shyamsundar Sarkar. Data Models in Geographic Information Systems. *Communications of the ACM*, 40(4): 103–111, 1997. ISSN 0001-0782.

[108] Claudia Bauzer Medeiros and Fatima Pires. Databases for GIS. *SIGMOD Record*, 23 (1):107–115, March 1994.

[109] Mark Graves. Graph Data Models for Genomics. *Submitted to ACM Transactions on Database Systems (TODS)*.

[110] Joachim Hammer and Markus Schneider. The GenAlg Project: Developing a New Integrating Data Model, Language, and Tool for Managing and Querying Genomic Information. *SIGMOD Record*, 33(2):45–50, 2004. ISSN 0163-5808.

[111] Yves Deville, David Gilbert, Jacques van Helden, and Shoshana J. Wodak. An Overview of Data Models for the Analysis of Biochemical Pathways. In *Proceedings of the First International Workshop on Computational Methods in Systems Biology*, page 174. Springer-Verlag, 2003. ISBN 3-540-00605-2.

[112] Gil Benkö, Christoph Flamm, and Peter F. Stadler. A Graph-Based Toy Model of Chemistry. *Journal of Chemical Information and Computer Sciences (JCISD)*, 43(1):1085–1093, Jan. 2003.

[113] Renzo Angles and Claudio Gutierrez. Querying RDF Data from a Graph Database Perspective. In *Proceedings of the 2nd European Semantic Web Conference (ESWC)*, number 3532 in LNCS, pages 346–360, 2005.

[114] James P. Fry and Edgar H. Sibley. Evolution of Data-Base Management Systems. *ACM Computing Surveys*, 8(1), 1976.

[115] C. J. Date. Referential Integrity. In *Proceedings of the 7th International Conference on Very Large Data Bases (VLDB)*, pages 2–12. IEEE Computer Society, Sept 1981.

[116] Bernhard Thalheim. *Dependencies in Relational Databases*. Leipzig, Teubner Verlag, 1991.

[117] Grant E. Weddell. Reasoning about Functional Dependencies Generalized for Semantic Data Models. *ACM Transactions on Database Systems (TODS)*, 17(1):32–64, 1992. ISSN 0362-5915.

[118] BernHard Thalheim. An Overview on Semantical Constraints for Database Models. In *Proceedings of the 6th International Conference Intellectual Systems and Computer Science*, Dec 1996.

[119] Klaus-Dieter Schewe, Bernhard Thalheim, Joachim W. Schmidt, and Ingrid Wetzel. Integrity Enforcement in Object-Oriented Databases . In *Proceedings of the 4th International Workshop on Foundations of Models and Languages for Data and Objects*, Oct. 1993.

[120] Peter Buneman, Wenfei Fan, and Scott Weinstein. Path Constraints in Semistructured and Structured Databases. In *Proceedings of the 17th Symposium on Principles of Database Systems (PODS)*, pages 129–138. ACM Press, June 1998. ISBN 0-89791-996-3.

[121] Natasha Alechina, Stéphane Demri, and Maarten de Rijke. A Modal Perspective on Path Constraints. *Journal of Logic and Computation*, 13(6):939–956, 2003.

[122] Roberto Zicari. A Framework for Schema Updates In An Object-Oriented Database System. In *Proceedings of the 7th International Conference on Data Engineering (ICDE)*, pages 2–13. IEEE Computer Society, 1991. ISBN 0-8186-2138-9.

[123] Moshe Y. Vardi. The Complexity of Relational Query Languages (Extended Abstract). In *Proceedings of the 14th ACM Symposium on Theory of Computing (STOC)*, pages 137–146. ACM Press, 1982. ISBN 0-89791-070-2.

[124] Raghu Ramakrishnan and Jeffrey D. Ullman. A Survey of Research on Deductive Database Systems. *Journal of Logic Programming (JLP)*, 23(2):125–149, 1993.

[125] Andreas Heuer and Marc H. Scholl. Principles of Object-Oriented Query Languages. In *Datenbanksysteme in Büro, Technik und Wissenschaft (BTW)*, volume 270 of *Informatik-Fachberichte*, pages 178–197. Springer, March 1991. ISBN 3-540-53861-5.

[126] Serge Abiteboul and Victor Vianu. Queries and Computation on the Web. In *Proceedings of the 6th International Conference on Database Theory (ICDT)*, volume 1186 of *LNCS*, pages 262–275. Springer, Jan. 1997. ISBN 3-540-62222-5.

[127] Ashok K. Chandra. Theory of Database Queries. In *Proceedings of the 7th Symposium on Principles of Database Systems (PODS)*, pages 1–9. ACM Press, March 1988. ISBN 0-89791-263-2.

[128] Manoochehr Azmoodeh and Hongbo Du. GQL, A Graphical Query Language for Semantic Databases. In *Proceedings of the 4th International Conference on Scientific and Statistical Database Management (SSDBM)*, volume 339 of *LNCS*, pages 259–277. Springer, June 1988. ISBN 3-540-50575-X.

[129] Marc Andries and Gregor Engels. A Hybrid Query Language for an Extended Entity-Relationship Model. Technical Report TR 93-15, Institute of Advanced Computer Science, Universiteit Leiden, May 1993.

[130] Michael Kifer, Won Kim, and Yehoshua Sagiv. Querying Object-Oriented Databases. In *Proceedings of the 1992 ACM SIGMOD International Conference on Management of data*, pages 393–402. ACM Press, 1992. ISBN 0-89791-521-6.

[131] Isabel F. Cruz, Alberto O. Mendelzon, and Peter T. Wood. A Graphical Query Language Supporting Recursion. In *Proceedings of the Association for Computing Machinery Special Interest Group on Management of Data*, pages 323–330. ACM Press, May 1987.

[132] I. F. Cruz, A. O. Mendelzon, and P. T. Wood. G+: Recursive Queries without Recursion. In *Proceedings of the 2th International Conference on Expert Database Systems (EDS)*, pages 645–666. Addison-Wesley, April 1989.

[133] M. P. Consens and A. O. Mendelzon. Expressing Structural Hypertext Queries in Graphlog. In *Proceedings of the 2th Conference on Hypertext*, pages 269–292. ACM Press, 1989. ISBN 0-89791-339-6.

[134] A. O. Mendelzon and P. T. Wood. Finding Regular Simple Paths in Graph Databases. In *Proceedings of the 15th International Conference on Very Large Data Bases (VLDB)*, pages 185–193. Morgan Kaufmann Publishers Inc., 1989. ISBN 1-55860-101-5.

[135] Alexandra Poulovassilis and Stefan G. Hild. Hyperlog: A Graph-Based System for Database Browsing, Querying, and Update. *IEEE Transactions on Knowledge and Data Engineering (TKDE)*, 13(2):316–333, 2001.

[136] R. Giugno and D. Shasha. GraphGrep: A Fast and Universal Method for Querying Graphs. In *Proceedings of the IEEE International Conference in Pattern Recognition (ICPR)*, Aug. 2002.

[137] Sergio Flesca and Sergio Greco. Partially Ordered Regular Languages for Graph Queries. In *Proceedings of the 26th International Colloquium on Automata, Languages and Programming (ICALP)*, volume 1644 of *LNCS*, pages 321–330. Springer, July 1999. ISBN 3-540-66224-3.

[138] Sergio Flesca and Sergio Greco. Querying Graph Databases. In *Proceedings of the 7th International Conference on Extending Database Technology - Advances in Database Technology (EDBT)*, volume 1777 of *LNCS*, pages 510–524. Springer, March 2000. ISBN 3-540-67227-3.

[139] Luca Cardelli, Philippa Gardner, and Giorgio Ghelli. A Spatial Logic for Querying Graphs. In *Proceedings of the 29th International Colloquium on Automata, Languages, and Programming (ICALP)*, LNCS, pages 597–610. Springer, July 2002. ISBN 3-540-43864-5.

[140] B. Langou and M. Mainguenaud. Graph Data Model Operations for Network Facilities in a Geographical Information System. In *Proceedings of the 6th International Symposium on Spatial Data Handling*, volume 2, pages 1002–1019, 1994.

[141] Mark Graves. *Theories and Tools for Designing Application-Specific Knowledge Base Data Models*. PhD thesis, University of Michigan, 1993.

[142] Mark Graves, Ellen R. Bergeman, and Charles B. Lawrence. Querying a Genome Database using Graphs. In *In Proceedings of the 3th International Conference on Bioinformatics and Genome Research*, 1994.

[143] R. Agrawal and H. V. Jagadish. Algorithms for Searching Massive Graphs. *IEEE Transactions on Knowledge and Data Engineering (TKDE)*, 6(2):225–238, 1994. ISSN 1041-4347.

[144] Rakesh Agrawal and H. V. Jagadish. Materialization and Incremental Update of Path Information. In *Proceedings of the 5th International Conference on Data Engineering (ICDE)*, pages 374–383. IEEE Computer Society, Feb 1989. ISBN 0-8186-1915-5.

[145] Rakesh Agrawal and H. V. Jagadish. Efficient Search in Very Large Databases. In *Proceedings of the 14th International Conference on Very Large Data Bases (VLDB)*, pages 407–418. Morgan Kaufmann, Aug - Sept 1988. ISBN 0-934613-75-3.

[146] Dennis Shasha, Jason T. L. Wang, and Rosalba Giugno. Algorithmics and Applications of Tree and Graph Searching. In *Proceedings of the 21th Symposium on Principles of Database Systems (PODS)*, pages 39–52. ACM Press, 2002. ISBN 1-58113-507-6.

[147] Mihalis Yannakakis. Graph-Theoretic Methods in Database Theory. In *Proceedings of the 9th Symposium on Principles of Database Systems (PODS)*, pages 230–242. ACM Press, 1990. ISBN 0-89791-352-3.

[148] M. V. Mannino and L. D. Shapiro. Extensions to Query Languages for Graph Traversal Problems. *IEEE Transactions on Knowledge and Data Engineering (TKDE)*, 2(3):353–363, 1990. ISSN 1041-4347.

[149] A. Tansel, J. Clifford, S. Gadia, S. Jajodia, A. Segev, and R. T. Snodgrass, editors. *Temporal Databases: Theory, Design, and Implementation*. Benjamin-Cummings, 1993.

[150] Jan Chomicki. Temporal Query Languages: A Survey. In *Proceedings of the First International Conference on Temporal Logic (ICTL)*, pages 506–534. Springer-Verlag, 1994.

[151] Jan Paredaens and Bart Kuijpers. Data Models and Query Languages for Spatial Databases. *Data & Knowledge Engineering (DKE)*, 25(1-2):29–53, 1998.

[152] Hanan Samet and Walid G. Aref. Spatial Data Models and Query Processing. In *Modern Database Systems*, pages 338–360. 1995.

[153] Panos Vassiliadis and Timos Sellis. A Survey of Logical Models for OLAP Databases. *SIGMOD Record*, 28(4):64–69, 1999.

[154] Marie-Aude Aufaure-Portier and Claude Trépied. A Survey of Query Languages for Geographic Information Systems. In *Proceedings of the 3rd International Workshop on Interfaces to Databases*, pages 431–438, July 1976. ISBN 3-540-76066-0.

[155] John F. Sowa. *Principles of Semantic Networks: Explorations in the Representation of Knowledge*. Morgan Kaufmann Publishers, 1991.

[156] Robert L. Griffith. Three Principles of Representation for Semantic Networks. *ACM Transactions on Database Systems (TODS)*, 7(3):417–442, 1982. ISSN 0362-5915.

[157] John F. Sowa. Conceptual Graphs for a Database Interface. *IBM Journal of Research and Development*, 20(4):336–357, 1976.

[158] J. F. Sowa. *Conceptual Structures: Information Processing in Mind and Machine*. Reading, MA, Addison-Wesley, 1984.

[159] Yi Deng and Shi-Kuo Chang. A G-Net Model for Knowledge Representation and Reasoning. *IEEE Transactions on Knowledge and Data Engineering (TKDE)*, 2(3):295–310, Dec 1990.

[160] Steve Pepper and Graham Moore. XML Topic Maps (XTM) 1.0 - TopicMaps.Org Specification, Feb 2001. URL `http://www.topicmaps.org/xtm/1.0/xtm1-20010806.html`.

[161] ISO. International Standard ISO/IEC 13250 Topic Maps, December 1999.

[162] Jeff Conklin. Hypertext: An Introduction and Survey. *IEEE Computer*, 20(9):17–41, 1987. ISSN 0018-9162.

[163] James Clark and Steve DeRose. XML Path Language (XPath), Version 1.0, W3C Recommendation, 16 November 1999. URL `http://www.w3.org/TR/1999/REC-xpath-19991116.html`.

[164] Aimilia Magkanaraki, Grigoris Karvounarakis, Ta Tuan Anh, Vassilis Christophides, and Dimitris Plexousakis. Ontology Storage and Querying. Technical Report 308, Institute of Computer Science of the Foundation for Research and Technology - Hellas (ICS-FORTH), April 2002.

[165] Asunción Gómez Pérez. OntoWeb - A survey on ontology tools. Technical Report Deliverable 1.3, OntoWeb, Ontology-based information exchange for knowledge management and electronic commerce, May 2002.

[166] Ian Horrocks and Sergio Tessaris. Querying the Semantic Web: a Formal Approach. In *Proceedings of the 13th International Semantic Web Conference (ISWC)*, number 2342 in Lecture Notes in Computer Science, pages 177–191. Springer-Verlag, 2002. ISBN 3-540-43760-6.

[167] Bjarne Stroustrup. What Is Object-Oriented Programming? *IEEE Softw.*, 5(3):10–20, 1988. ISSN 0740-7459.

[168] Libby Miller. Public report comparing existing RDF query language functionality, documenting different scenarios and users for RDF query languages. Technical Report Deliverable 7.2, Semantic Web Advanced Development for Europe (SWAD-Europe), April 2003. URL http://www.w3.org/2001/sw/Europe/reports/rdf_ql_comparison_report/.

[169] Eric Prud'hommeaux and Benjamin Grosof. RDF Query Survey, April 2004. URL http://www.w3.org/2001/11/13-RDF-Query-Rules/.

[170] Alberto Reggiori and Andy Seaborne. Query and Rule languages Use Cases and Examples, June 2004. URL http://rdfstore.sourceforge.net/2002/06/24/rdf-query/query-use-cases.html.

[171] Scientific Literature Digital Library (CiteSeer), 2005. URL http://citeseer.ist.psu.edu.

[172] Digital Bibliography and Library Project (DBLP), 1980. URL http://dblp.uni-trier.de/.

[173] Kendall Grant Clark. RDF Data Access Use Cases and Requirements, W3C Working Draft, March 2005. URL http://www.w3.org/TR/2005/WD-rdf-dawg-uc-20050325/.

[174] Andy Seaborne. RDQL - A query Language for RDF (W3C Member Submission), January 2004. URL http://www.w3.org/Submission/2004/SUBM-RDQL-20040109/.

[175] Libby Miller. RDF Squish query language and Java implementation, 2001. URL http://ilrt.org/discovery/2001/02/squish/.

[176] RDFQL, 2000. URL http://www.intellidimension.com/.

[177] Michael Sintek and Stefan Decker. TRIPLE - A Query, Inference, and Transformation Language for the Semantic Web. In *Proceedings of the 1st International Semantic Web Conference (ISWC)*, number 2342 in LNCS, pages 364–378. Springer, June 2002.

[178] Versa: Path-Based RDF Query Language, 2005. URL http://www.xml.com/lpt/a/2005/07/20/versa.html.

[179] The SeRQL query language, 2004. URL `http://www.openrdf.org/doc/sesame/users/ch06.html`.

[180] Tao Jiang and Ah-Hwee Tan. Mining RDF metadata for generalized association rules: knowledge discovery in the semantic Web era. In *Proceedings of the 15th international conference on World Wide Web (WWW)*, pages 951–952. ACM, 2006. ISBN 1-59593-323-9.

[181] Claudio Gutierrez, Carlos Hurtado, and Alejandro Vaisman. Temporal RDF. In *Proceedings of the 2nd European Semantic Web Conference (ESWC)*, number 3532 in LNCS, pages 93–107, 2005.

[182] Sergio Muñoz, Jorge Pérez, and Claudio Gutiérrez. Minimal Deductive Systems for RDF. In Springer, editor, *Proceedings of the 4th European Semantic Web Conference (ESWC)*, volume 4519 of *LNCS*, pages 53–67, 2007.

[183] Craig Sayers. Node-centric RDF Graph Visualization. Technical Report HPL-2004-60, Mobile and Media Systems Laboratory HP Laboratories Palo Alto, April 2004.

[184] R. Guha, Rob McCool, and Eric Miller. Semantic Search. In *Proceedings of the 12th International Conference on World Wide Web (WWW)*, pages 700–709. ACM Press, 2003. ISBN 1-58113-680-3.

[185] Jason McHugh, Serge Abiteboul, Roy Goldman, Dallan Quass, and Jennifer Widom. Lore: A Database Management System for Semistructured Data. *SIGMOD Record*, 26 (3):54–66, 1997.

[186] Kemafor Anyanwu and Amit Sheth. The ρ-operator: Enabling Querying for Semantic Associations on the Semantic Web. 20-24 May 2003.

[187] Takashi Washio and Hiroshi Motoda. State of the Art of Graph-based Data Mining. *SIGKDD Explorer Newsletter*, 5(1):59–68, 2003.

[188] Jorge Pérez, Marcelo Arenas, and Claudio Gutierrez. Semantics and Complexity of SPARQL. In *Proceedings of the 5th International Semantic Web Conference (ISWC)*, number 4273 in LNCS, pages 30–43. Springer-Verlag, 2006.

[189] Serge Abiteboul, Richard Hull, and Victor Vianu. *Foundations of Databases*. Addison-Wesley, 1995. ISBN 0-201-53771-0.

[190] Mark Levene and George Loizou. *A Guided Tour of Relational Databases and Beyond*. Springer-Verlag, 1999.

[191] Tim Furche, Benedikt Linse, François Bry, Dimitris Plexousakis, and Georg Gottlob. RDF Querying: Language Constructs and Evaluation Methods Compared. In *Reasoning Web*, number 4126 in LNCS, pages 1–52, 2006.

[192] Richard Cyganiak. A relational algebra for SPARQL. Technical Report HPL-2005-170, HP Labs, 2005.

[193] A. Polleres. From SPARQL to Rules (and back). In *Proceedings of the 16th International World Wide Web Conference (WWW)*, pages 787–796. ACM, 2007.

[194] Simon Schenk. A SPARQL Semantics Based on Datalog. In *30th Annual German Conference on Advances in Artificial Intelligence (KI)*, volume 4667 of *LNCS*, pages 160–174. Springer, 2007.

[195] Jeremy J. Carroll, Christian, Pat Hayes, and Patrick Stickler. Named graphs. *Journal of Web Semantics*, 3(4), 2005.

[196] Leonidas Deligiannidis, Krys J. Kochut, and Amit P. Sheth. RDF data exploration and visualization. In *Proceedings of the ACM first workshop on CyberInfrastructure: information management in eScience (CIMS)*, pages 39–46. ACM, 2007. ISBN 978-1-59593-831-2.

[197] Eyal Oren, Renaud Delbru, and Stefan Decker. Extending faceted navigation for RDF data. In *Proceedings of the 5th International Semantic Web Conference (ISWC)*, LNCS, 2006.

[198] Emmanuel Pietriga. IsaViz: A Visual Authoring Tool for RDF, 2001. URL `http://www.w3.org/2001/11/IsaViz/`.

[199] Charles D. Hansen and Chris R. Johnson, editors. *The Visualization Handbook*. Elsevier Inc., 2005.

[200] The Tabulator, 2005. URL `http://www.w3.org/2005/ajar/tab`.

[201] Renzo Angles and Claudio Gutierrez. Survey of graph database models. *ACM Computing Surveys (CSUR)*, 40(1):1–39, 2008. ISSN 0360-0300.

[202] University of Maryland UMBC eBiquity Research Group. Swoogle, 2004. URL `http://swoogle.umbc.edu/`.

[203] Disco - Hyperdata Browser, 2007. URL `http://www4.wiwiss.fu-berlin.de/bizer/ng4j/disco/`.

[204] Marbles Linked Data Browser, 2008. URL `http://beckr.org/marbles`.

[205] Franz Inc. Gruff: A Grapher-Based Triple-Store Browser, June 2008. URL http://agraph.franz.com/gruff/.

[206] Damian Steer. BrownSauce RDF Browser, October 2002. URL http://brownsauce.sourceforge.net/.

[207] Mike Dean and Kelly Barber. DAML Viewer, March 2001. URL http://www.daml.org/viewer/.

[208] Zitgist DataViewer, 2007. URL http://zitgist.com/products/dataviewer/dataviewer.html.

[209] Ka-Ping Yee, Kirsten Swearingen, Kevin Li, and Marti Hearst. Faceted metadata for image search and browsing. In *Proceedings of the Conference on Human factors in Computing Systems (CHI)*, pages 401–408. ACM, 2003. ISBN 1-58113-630-7.

[210] Michiel Hildebrand, Jacco van Ossenbruggen, and Lynda Hardman. /facet: A Browser for Heterogeneous Semantic Web Repositories. In *Proc. of the 5th International Semantic Web Conference (ISWC)*, number 4273 in LNCS, pages 272–285, 2006.

[211] UC Berkeley School of Information. The Flamenco Search Interface Project, 2000. URL http://flamenco.berkeley.edu/.

[212] Eyal Oren and Renaud Delbru. A prototype for faceted browsing of RDF data (BrowseRDF), 2006. URL http://browserdf.org/.

[213] SIMILE Project MIT. Longwell, February 2002. URL http://simile.mit.edu/wiki/Longwell.

[214] Michiel Hildebrand. Slash facet, 2006. URL http://slashfacet.semanticweb.org/.

[215] Chris Bizer, Richard Cyganiak, Oliver Maresch, and Tobias Gauss. The WIQA Browser, 2006. URL http://www4.wiwiss.fu-berlin.de/bizer/WIQA/browser/index.htm.

[216] Christian Bizer, Richard Cyganiak, and Tobias Gauss1and Oliver Maresch. The TriQL.P Browser: Filtering Information using Context-, Content- and Rating-Based Trust Policies. In *Semantic Web and Policy Workshop at the 4th International Semantic Web Conference*, November 2005.

[217] Tim Berners-Lee. Linked Data, 2006. URL http://www.w3.org/DesignIssues/LinkedData.html.

[218] Chris Bizer, Richard Cyganiak, and Tom Heath. How to Publish Linked Data on the Web, 2007. URL http://www4.wiwiss.fu-berlin.de/bizer/pub/LinkedDataTutorial/20070727/.

[219] Serge Abiteboul and Richard Hull. Restructuring Hierarchical Database Objects. *Theoretical Computer Science*, 62(1-2):3–38, 1988.

[220] Dieter Jungnickel. *Graphs, Network and Algorithms (Third Edition)*. Springer-Verlag, 2008.

[221] Klara Weiand, Francois Bry, and Tim Furche. Reasoning & Querying - State of the Art: Keyword-based querying for XML and RDF. Technical Report ICT211932, EU 7th Framework Programme (FP7/2007-2013), 2008.

[222] F. Alkhateeb, J.-F. Baget, and J. Euzenat. Complex path queries for RDF graphs. In *International Semantic Web Conference (ISWC), Poster Paper*, 2005.

[223] F. Alkhateeb, J.-F. Baget, and J. Euzenat. RDF with Regular Expressions. Technical Report 6191, Institut National de Recherche en Informatique et en Automatique (INRIA), May 2007.

[224] F. Alkhateeb, J.-F. Baget, and J. Euzenat. Constrained Regular Expressions in SPARQL. Technical Report 6360, Institut National de Recherche en Informatique et en Automatique (INRIA), Oct 2007.

[225] Krys J. Kochut and Maciej Janik. SPARQLeR: Extended Sparql for Semantic Association Discovery. In *Proceedings of the 4th European Semantic Web Conference (ESWC)*, volume 4519 of *LNCS*, pages 145–159, 2007.

[226] Kemafor Anyanwu, Angela Maduko, and Amit Sheth. SPARQ2L: Towards support for subgraph extraction queries in RDF databases. In *Proceedings of the 16th International Conference on World Wide Web (WWW)*, pages 797–806. ACM, 2007. ISBN 978-1-59593-654-7.

[227] L. T. Detwiler, D. Suciu, and J. F. Brinkley. Regular Paths in SparQL: Querying the NCI Thesaurus. In *Proceedings of the American Medical Informatics Association Fall Symposium*, pages 161–165, 2008.

[228] Dawit Seid and Sharad Mehrotra. Grouping and Aggregate queries Over Semantic Web Databases. In *Proceedings of the International Conference on Semantic Computing (ICSC)*, pages 775–782. IEEE Computer Society, 2007. ISBN 0-7695-2997-6.

[229] Jorge Pérez, Marcelo Arenas, and Claudio Gutierrez. nSPARQL: A Navigational language for RDF. In *Proceedings of the 7th International Semantic Web Conference (ISWC)*, 2008.

[230] Eran Tuv, Alexandra Poulovassilis, and Mark Levene. A Storage Manager for the Hypernode Model. In *Proceedings of the 10th British National Conference on Databases*, number 618 in LNCS, pages 59–77. Springer-Verlag, 1992.

[231] C. Berge. *Graphs and Hypergraphs*. North-Holland, Amsterdam, 1973.

[232] Marc Gemis, Jan Paredaens, Inge Thyssens, and Jan Van den Bussche. GOOD: A Graph-Oriented Object Database System. In *Proceedings of the 1993 ACM SIGMOD International Conference on Management of Data*, pages 505–510. ACM Press, 1993. ISBN 0-89791-592-5.

[233] Jan Hidders. *A Graph-based Update Language for Object-Oriented Data Models*. PhD thesis, Technische Universiteit Eindhoven,, Dec. 2001.

[234] Victor Vianu. A Web Odyssey: from Codd to XML. *SIGMOD Record*, 32(2):68–77, 2003. ISSN 0163-5808.