RDF Database Systems

Triples Storage and SPARQL Query Processing

Olivier Curé and Guillaume Blin

MORGAN KAUFMANN

RDF Database Systems
TRIPLES STORAGE AND SPARQL QUERY PROCESSING

Edited by

OLIVIER CURÉ
Associate Professor of Computer Science,
Université Paris Est Marne la Vallée and
Université Pierre et Marie Curie, France

GUILLAUME BLIN
Professor of Computer Science,
Université de Bordeaux, France

Amsterdam • Boston • Heidelberg • London
New York • Oxford • Paris • San Diego
San Francisco • Singapore • Sydney • Tokyo
Morgan Kaufmann is an Imprint of Elsevier

Executive Editor: Steve Elliot
Editorial Project Manager: Kaitlin Herbert
Project Manager: Anusha Sambamoorthy
Designer: Mark Rogers

Morgan Kaufmann is an imprint of Elsevier
225 Wyman Street, Waltham, MA 02451, USA

First edition 2015

Notice

No responsibility is assumed by the publisher for any injury and/or damage to persons or property as a matter of products liability, negligence or otherwise, or from any use or operation of any methods, products, instructions or ideas contained in the material herein. Because of rapid advances in the medical sciences, in particular, independent verification of diagnoses and drug dosages should be made.

Library of Congress Cataloging-in-Publication Data

Curé, Olivier.
RDF database systems : triples storage and SPARQL query processing / by Olivier Curé, Guillaume Blin. — First edition.
pages cm
Includes index.
ISBN 978-0-12-799957-9 (paperback)
1. Database management. 2. RDF (Document markup language) 3. Query languages (Computer science) 4. Querying (Computer science) I. Blin, Guillaume. II. Title.
QA76.9.D3C858 2015
005.74 — dc23

2014034632

British Library Cataloguing in Publication Data
A catalogue record for this book is available from the British Library

ISBN: 978-0-12-799957-9

CONTENTS

LIST OF FIGURES AND TABLES

PREFACE

In 1999, the *World Wide Web Consortium* (W3C) published a first recommendation on the *Resource Description Framework* (RDF) language. Since its inception, this technology is considered the cornerstone of the *Semantic Web* and *Web of Data* movements. Its goal is to enable the description of resources that are uniquely identified by *uniform resource identifiers* (URIs), for example, `http://booksite.mkp.com`. Recently, RDF has gained a lot of attention, and as a result an increasing number of data sets are now being represented with this language. With this popularity came the need to efficiently store, query and reason over large amounts of RDF data. This has obviously motivated some research work on the design of adapted and efficient data management systems. These systems, frequently referred to as *RDF stores*, *triple stores*, or *RDF data management systems*, must handle a data model that takes the form of a directed labeled graph where nodes are URIs or literals (e.g., character strings) and edges are URIs. This data model is quite different from the currently popular relational model encountered in a *relational database management system* (RDBMS). But RDF stores have other features that make their design a challenging task. In the following, we present four major features that differentiate RDF stores with other data management systems.

The first feature is related to the data model of RDF, which is composed of triples corresponding to a subject, a property, and an object, where the subject takes an object as a value for a property. A set of triples is characterized by the omnipresence of URIs at the three positions of RDF statements, although over forms can also be used, e.g., literals at the object position. The fact that a given URI can be repeated multiple times in a data set and that URIs are generally long strings of characters raises some memory footprint issues. Therefore, the use of efficient dictionaries—encoding URIs with identifiers that are used for the representation of triples—is crucial in triple stores, while in relational models a suitable database schema minimizes such data redundancy.

The second feature is that a data management system requires a language to query the data it contains. For this purpose, the Semantic Web group at the W3C has published the *SPARQL Protocol and RDF Query Language* (SPARQL) recommendation; one of its main aspects is to support a query language. Although some clauses such as `SELECT` and `WHERE` make it look like the popular *Structured Query Language* (SQL) language of RDBMS, SPARQL is based on the notion of triples and not on relations. Answering a SPARQL query implies some pattern-matching mechanisms between the triples of the query and the data set. This requires research in the field of query processing and optimization that goes beyond the state of the art of the relational model.

The third feature concerns vocabularies associated to RDF data sets. The most predominant ones are *RDF Schema* (RDFS) and *Web Ontology Language* (OWL), both of which enable the ability to reason over RDF data—that is, they enable the discovery of implicit data from explicitly represented ones.

The final feature is that RDF data is concerned with the *Big Data* phenomenon, which induces the distribution of data sets over clusters of machines. The graph nature of the RDF data model makes the distribution a challenging task that has to be understood in an inference-enabled context.

This book aims at presenting the fundamentals on all these features by introducing both the main concepts and techniques, as well as the principal research problems that emerge in the ecosystem of RDF database management systems. The presentations are based on the study of a large number of academic and commercial products. It is important to understand that the fulfillment of the Semantic Web vision largely depends on the emergence of robust, efficient, and feature-complete RDF stores.

WHO SHOULD READ THIS BOOK

The book targets a wide range of readers. We consider that technology practitioners, including developers, project leaders, and database professionals, will find a comprehensive survey of existing commercial and open-source systems that will help them to select a system in a given application implementation context. We also consider that students and teachers could use this book for lectures in Semantic Web and knowledge representation, as well as for advanced database courses. In fact, the idea of providing a textbook for our master's students at the University of Paris Est was one of the motivations for starting this work. Finally, researchers in both the Semantic Web and database fields will find in this book a comprehensive and comparative overview of an active research domain. Indeed, we were in need for such an overview when we started our work on the **roStore** and **WaterFowl** RDF stores.

ORGANIZATION OF THE BOOK

This book is organized in two parts composed of short chapters to support an efficient and easy reading. The first part defines the scope of the book but also motivates and introduces the importance of efficiently handling RDF data. To make the book self-contained, it also provides background knowledge on both database management systems and Semantic Web technologies.

With these notions understood, readers can delve into the second part of the book, which investigates an important number of existing systems based on the criteria we have defined: the definition of an RDF dictionary, backend storage of RDF triples, indexation, query processing, reasoning, distribution of workloads, and query federation. We dedicate one chapter per criteria for easier reading of the book. With this structure, for example, a reader can learn about a given aspect of RDF stores in Chapter 4 on RDF dictionaries without searching through the complete book. This approach imposes that the different components of important systems are detailed in different chapters—for

example, the *RDF-3X* system is discussed in Chapters 4–6. At the end of each chapter we provide a summary list of the main notions studied.

Finally, the book concludes and anticipates on future trends in the domain of RDF storage and the Semantic Web.

GUIDELINES FOR USING THIS BOOK

We identify three distinct readings of this book that are mainly motivated by a reader's expertise in either database management or Semantic Web technologies. Of course, readers not confident about their knowledge on any of these domains should read both background knowledge chapters—that is, Chapters 2 and 3.

In the case of a reader with database expertise, Chapter 2 can be skipped, but we warn that several different forms of database management systems are studied. For example, most readers with a database profile may already master all notions considered in the section dedicated to the relational model, but it may not be the case on the sections addressing *NoSQL* and novel extensions of the relational model.

If a reader's profile corresponds to a Semantic Web literate, then reading Chapter 3 may be optional because it spends most of its content presenting RDF, SPARQL, and ontology languages such as RDFS and OWL, as well as associated reasoning services.

The second part of the book is its cornerstone. All readers should be interested in reading its content and learn about the different approaches to address selected dimensions.

IT project managers and developers will acquire knowledge on the main characteristics of existing, commercial, or open-source systems, while researchers will learn about the latest and most influential systems with references to publications.

CONVENTIONS USED IN THIS BOOK

This book contains the following typographical conventions:

- **Bold** highlights the first occurrence of a software, system, or company name within a chapter.
- `Code font` is used for queries (e.g., expressed in SPARQL or SQL) and URIs, and can also be found within paragraphs to refer to query or program elements.

SUPPLEMENTAL MATERIALS

Additional materials for this book can be found at http://igm.mlv.fr/~ocure/rdfstores/

ACKNOWLEDGMENTS

We would like to thank our technical reviewer, Dean Allemang, special thanks go to our editor: Andrea Dierna, Kaitlin Herbert, and Anusha Sambamoorthy. We are also grateful to David Celestin Faye for participating in the design of roStore.

A very special thanks go to our families for their support: Virginie, Noé, Jeanne, Axelle, and Charlie.

CHAPTER ONE

Introduction

If you have this book in your hands, we guess you are interested in database management systems in general and more precisely those handling *Resource Description Framework* (RDF) as a data representation model. We believe it's the right time to study such systems because they are getting more and more attention in industry with communities of Web developers and *information technology* (IT) experts who are designing innovative applications, in universities and engineering schools with introductory and advanced courses, and in both academia and industry research to design and implement novel approaches to manage large RDF data sets. We can identify several reasons that are motivating this enthusiasm.

In this introductory chapter, we will concentrate on two really important aspects. An obvious one is the role played by RDF in the emergence of the *Web of Data* and the *Semantic Web*—both are extensions of the original *World Wide Web*. The second one corresponds to the impact of this data model in the *Big Data* phenomenon. Based on the presentation of these motivations, we will introduce the main characteristics of an RDF data management system, and present some of its most interesting features that support the comparison of existing systems.

1.1 BIG DATA

Big Data is much more than a buzzword. It can be considered as a concrete phenomenon that is attracting all kinds of companies facing strategic and decisional issues, as well as scientific fields interested in understanding complex observations that may lead to important discoveries. The *National Institute of Standards and Technologies* (NIST) has proposed a widely adopted definition referring to the data deluge: "a digital data volume, velocity and/or variety that enable novel approaches to frontier questions previously inaccessible or impractical using current or conventional methods; and/or exceed the capacity or capability of current or conventional methods and systems" (NIST Big Data, 2013). Most Big Data definitions integrate this aspect of the three V's: volume, velocity, and variety (which is sometimes extended with a fourth V for veracity).

Volume implies that the sizes of data being produced cannot be stored and/or processed using a single machine, but require a distribution over a cluster of machines. The challenges are, for example, to enable the loading and processing of exabytes (i.e., 10^3 petabytes or 10^6 terabytes) of data while we are currently used to data loads in the range of at most terabytes.

Velocity implies that data may be produced at a throughput that cannot be handled by current methods. Solutions, such as relaxing transaction properties, storing incoming data on several servers, or using novel storage approaches to prevent input/output latencies, are being proposed and can even be combined to address this issue. Nevertheless, they generally come with limitations and drawbacks, which are detailed in Chapter 2.

Variety concerns the format (e.g., **Microsoft**'s *Excel* [XLS], *eXtended Markup Language* [XML], *comma-separated value* [CSV], or RDF) and structure conditions of the data. Three main conditions exist: structured, semi-structured, and unstructured. Structured data implies a strict representation where data is organized in entities, and then similar entities are grouped together and are described with the same set of attributes, such as an identifier, price, brand, or color. This information is stored in an associated schema that provides a type to each attribute—for example, a price is a numerical value. The data organization of a *relational database management system* (RDBMS) is reminiscent of this approach. The notion of semi-structured data (i.e., self-described) also adopts an entity-centered organization but introduces some flexibility. For example, entities of a given group may not have the same set of attributes, attribute order is generally not important in the description of an entity, and an attribute may have different types in different entity groups. Common and popular examples are XML, *JavaScript Object Notation* (JSON), and RDF. Finally, unstructured data is characterized by providing very little information on the type of data it contains and the set of formatting rules it follows. Intuitively, text, image, sound, and video documents belong to this category.

Veracity concerns the accuracy and noise of the data being captured, processed, and stored. For example, considering data acquired, one can ask if it's relevant to a particular domain of interest, if it's accurate enough to support decision making, or if the noise associated to that data can be efficiently removed. Therefore, data quality methods may be required to identify and clean "dirty" data, a task that in the context of the other three V's may be considered one of the greatest challenges of the data deluge. Surprisingly, this dimension is the one that has attracted the least attention from Big Data actors.

The emergence of Big Data is tightly related to the increasing adoption of the Internet and the Web. In fact, the Internet proposes an infrastructure to capture and transport large volumes of data, while the Web, and more precisely its 2.0 version, has brought facilities to produce information from the general public. This happens through interactions with personal blogs (e.g., supported by **WordPress**), wikis (e.g., **Wikipedia**), online social networks (e.g., **Facebook**), and microblogging (e.g., **Twitter**), and also through logs of activities on the most frequently used search engines (e.g., **Google**, **Bing**, or **Yahoo!**) where an important amount of data is produced every day. Among the most stunning recent values, we can highlight that Facebook announced that, by the beginning of 2014, it is recording 600 terabytes of data each day in its 300 petabytes data warehouse and an average of around 6,000 tweets are stored at Twitter per second with a record of 143,199 tweets on August 3, 2013.

The *Internet of Things* (IoT) is another contributor to the Big Data ecosystem, which is just in its infancy but will certainly become a major data provider. This Internet branch is mainly concerned with *machine-to-machine* (M2M) communications that are evolving on a Web environment using Web standards such as *Uniform Resource Identifiers* (URIs), *HyperText Transfer Protocol* (HTTP), and *representational state transfer* (REST). It focuses on the devices and sensors that are present in our daily lives, and can belong to either the industrial sector or the consumer market. These active devices may correspond but are not limited to smartphones, *radio-frequency identification device* (RFID) tagged objects, wireless sensor networks, or ambient devices.

IoT enables the collection of temporospatial information—that is, regrouping temporal as well as spatial aspects. In 2009, considered an early year of IoT, Jeff Jonas in his blog (Jonas, 2009) was already announcing that 600 billion geospatially tagged transactions were generated per day in North America. This ability to produce enormous volumes of data at a high throughput is already a data management challenge that will expand in the coming years. To consider its evolution, a survey conducted by **Cisco** (Cisco, 2011) emphasized that from 1.84 connected devices per person in 2010, we will reach 6.58 in 2020, or approximately 50 billion devices. Almost all of these devices will produce massive amounts of data on a daily basis.

As a market phenomenon, Big Data is not supervised by any consortium or organism. Therefore, there is a total freedom about the format of generated, stored, queried, and manipulated data. Nevertheless, best practices of the major industrial and open-source actors bring forward some popular formats such as XLS, CSV, XML, JSON, and RDF. The main advantages of JSON are its simplicity, flexibility (it's schemaless), and native processing support for most Web applications due to a tight integration with the *JavaScript* programming language. But RDF is not without assets. For example, as a semi-structured data model, RDF data sets can be described with expressive schema languages, such as *RDF Schema* (RDFS) or *Web Ontology Language* (OWL), and can be linked to other documents present on the Web, forming the *Linked Data* movement.

With the emergence of Linked Data, a pattern for hyperlinking machine-readable data sets that extensively uses RDF, URIs, and HTTP, we can consider that more and more data will be directly produced in or transformed into RDF. In 2013, the *linked open data* (LOD), a set of RDF data produced from open data sources, is considered to contain over 50 billion triples on domains as diverse as medicine, culture, and science, just to name a few. Two other major sources of RDF data are building up with the *RDF in attributes* (RDFa) standard, where attributes are to be understood in an (X)HTML context, and the **Schema.org** initiative, which is supported by Google, Yahoo!, Bing, and **Yandex** (the largest search engine in Russia). The incentive of being well referenced in these search engines already motivates all kinds of Web contributors (i.e., companies, organizations, etc.) to annotate their web page content with descriptions that can be transformed

into RDF data. In the next section, we will present some original functionalities that can be developed with Linked Data, such as querying and reasoning at the scale of the Web.

As a conclusion on Big Data, the direct impact of the original three V's is the calling for new types of database management systems. Specifically, those that will be able to handle rapidly incoming, heterogeneous, and very large data sets. Among others, a major advantage of these systems will be to support novel, more efficient data integration mechanisms. In terms of features expected from these systems, Franklin and colleagues (2005) were the first to propose a new paradigm. Their *DataSpace Support Platforms (DSSP)* are characterized by a pay-as-you-go approach for the integration and querying of data. Basically, this paradigm is addressing most of the issues of Big Data. Later, in Dong and Halevy (2007), more details on the possible indexing methods to use in data spaces were presented. Although not mentioning the term *RDF*, the authors presented a data model based on triples that matches the kind of systems this book focuses on and that are considered in Part 2 of this book.

1.2 WEB OF DATA AND THE SEMANTIC WEB

The Web, as a global information space, is evolving from linking documents only, to linking both documents and data. This is a major (r)evolution that is already supporting the design of innovative applications. This extension of the Web is referred to as the *Web of Data* and enables the access to and sharing of information in ways that are much more efficient and open than previous solutions. This efficiency is due to the exploitation of the infrastructure of the Web by allowing links between distributed data sources. Three major technologies form the cornerstone of this emerging Web: URIs, HTTP, and RDF. The first two have been central to the development of the Web since its inception and respectively address the issues of identifying Web resources (e.g., web pages) and supporting data communication for the Web. The latter provides a data representation model, and the management of such data is the main topic of this book.

The term *semantics* is getting more and more attention in the IT industry as well as in the open-source ecosystem. It basically amounts to providing some solutions for computers to automatically interpret the information present in documents. The interpretation mechanism is usually supported by annotating this information with vocabularies the elements of which are given a well-defined meaning with a logical formalism. The logical approach enables some dedicated reasoners to perform some inferences. Of course, the Web, and in particular the Web of Data, is an important document provider; in that context, we then talk about a Semantic Web consisting of a set of technologies that are supporting this whole process. In Berners-Lee et al. (2001), the Semantic Web is defined as "an extension of the current web in which information is given well-defined meaning, better enabling computers and people to work in cooperation" (p. 1). This emphasizes that there is no rupture between a previous non-semantic Web and a semantic one.

They will both rely on concepts such as HTTP, URIs, and the stack of representational standards such as *HyperText Markup Language* (HTML), *Cascade Style Sheets* (CSS), and all accompanying programming technologies like JavaScript, *Ruby*, and *HyperText PreProcessor* (PHP).

The well-defined meaning aspect is related to RDF annotations and vocabularies expressed in RDFS and the OWL standards. These languages are enabling the description of schemata associated to RDF. Together with such vocabularies, reasoning procedures enable the deduction of novel information or knowledge from data available in the Web of Data. In fact, one of the sweet-spots of RDF and other Semantic Web technologies is data integration, which is the ability to efficiently integrate new information in a repository. This leverages on the Linked Data movement, which is producing very large volumes of RDF triples, dereferenceable URIs (i.e., a resource retrieval method that is making use of Internet protocols such as HTTP), and a flexible data model. This will support the development and maintenance of novel mashup-based applications (i.e., mixing distinct and previously not related information resources) that are going far beyond what is used today.

A first question one may ask is, how popular the Semantic Web is getting? First, the fact that the principles or even the main technologies of the Semantic Web may not be known by the general public cannot be considered a setback of the overall approach. Semantic Web technologies are expected to be present on the server side, not on the client side (i.e., web browsers). So it should not be transparent to the general public and should only be known to application designers and developers. The emergence of semantics can be found in different domains. For instance, it's spreading throughout the search process performed at Google. Rapidly after the announcement in May 2012 of using its *Knowledge Graph* to improve search by providing smarter answers, Google was claiming to answer almost 20% of its total search directly using semantics. In fact, these searches try to understand the context of a given phrase by analyzing its language and terms. This analysis is based on the Knowledge Graph, a graph database originating from the **Freebase** project, Wikipedia, and the **CIA World Factbook**. It consists of a semantic network, the kind of technology RDF and RDFS enable to design, to describe over 600 million objects and 20 billion facts about them as well as their relationships. Note that Google is not the only Web company tackling this issue; for instance, Facebook proposes a similar approach with its *Open Graph Protocol* (OGP), and Microsoft's Bing engine is using its so-called Satori Knowledge Base.

Web technologies such as HTML, CSS, and JavaScript address a wider spectrum of people involved in Web content consultation and creation. The rapid success of the Web in its early days can, in part, be attributed to the ease of designing web pages and websites with computerized tools, so-called *what you see is what you get* (WYSIWYG) editors, that are automatically generating the lines of codes from interactions with the system. This may only partially happen for the Semantic Web, which requires some computer science

knowledge for someone to apprehend the whole stack of standards. For example, one may have difficulties apprehending ontology languages (e.g., RDFS or OWL) and their reasoning services without any understanding on concepts such as entailment regimes and open/closed world assumption. Even efficient and elegant methods to annotate web pages with metadata processable by agents were not identified until recently. Now, with the emergence of the RDFa *World Wide Web Consortium* (W3C) standard, one is able to add information to a web page that will be used for both rendering and structured description. Several *content manager systems* (CMSs), such as **Drupal** or WordPress, have already integrated this aspect in their recent implementations, and therefore allow anyone to contribute to the Semantic Web project without being aware of it.

The use of semantic technologies can also be spotted at *The New York Times*, which, since 2009, publishes large *Simple Knowledge Organization System* (SKOS, a W3C Semantic Web recommendation) vocabularies about articles and topics proposed in the 150 years of the newspaper's history. In addition, **BBC** provides RDF descriptions of its TV and radio programs.

1.3 RDF DATA MANAGEMENT

So far, we have highlighted that the generation of large volumes of RDF data is increasing but we have not considered storage and processing issues. The volume and velocity of produced RDF data have a major positive impact on the Semantic Web. But it can also be a limit to its deployment if we do not consider issues such as scalability, availability, query answering, data exchange, and reasoning in an efficient way. In fact, since the first W3C's RDF recommendation in 1999, more than 50 data management systems have been proposed and developed for RDF data. Among them, some have been implemented by the three major database vendors: **Oracle**, **IBM**, and Microsoft (yet another proof of maturity of RDF technology that addresses important expectations of the IT market). Other systems are being produced by smaller companies that are more or less specialized on RDF standards: **Ontotext** with the **OWLIM** (now GraphDB) system suite, **OpenLink** with the **Virtuoso** system, **Systap LLC** and its **Bigdata** solution, **Franz Inc.** with its **Allegrograph** database, and **Clark & Parsia**'s **Stardog** system. Finally, a large number of systems frequently released under open-source licenses have been implemented in universities and research institutes, such as **RDF-3X** (Neumann and Weikum, 2008), **Hexastore** (Weiss et al., 2008), and **BitMat** (Atre et al., 2009).

These systems are designed using quite different technologies. For example, concerning backend storage and indexing, some are adopting a so-called native approach—that is, they are not relying on another management system—while others go for a non-native approach and benefit from existing systems, such as RDBMSs or NoSQL stores. Similarly, different approaches to address query processing, reasoning, data distribution, and federation have been identified and implemented. In the context of a given application,

this makes the selection of an adapted RDF data management system a nontrivial task, probably harder than for a RDBMS. Over the last few years, we have met and worked with IT project managers and developers who were facing hard decisions on which RDF stores to use. We consider that a thorough investigation of the main dimensions differentiating existing systems can be useful when deciding which systems to integrate in an application. All these aspects are investigated in Part 2 of this book.

1.4 DIMENSIONS FOR COMPARING RDF STORES

We have defined a set of dimensions on which existing RDF stores should be analyzed and compared. The identification to these dimensions is mainly motivated by the discrepancies in the approaches adopted in certain systems. For example, our first dimension concerns the dictionary aspect. In most RDF data management systems, the RDF triples, mainly composed of URIs, are not stored as is. This is mainly motivated by two properties of URIs: the number of their occurrences over a triples set can be fairly high, and they generally correspond to rather long strings of characters. Therefore, for data compression reasons, almost all systems encode, using so-called dictionaries, these URIs into integers (enabling them to gain storage space). Nevertheless, the manner in which these encodings are processed and stored can be quite different from one system to another. Moreover, some systems propose a clever approach to this encoding that supports some form of almost free inferences and enables the efficient processing of simple regular expressions.

A second dimension considers the storage of RDF data. This has been the subject of an important number of research articles, which unsurprisingly have been published for both Semantic Web and database conferences. Many approaches have been presented, some storing the triples in raw files, RDBMSs, and recently *NoSQL* stores. But even within a given storage approach, several logical approaches have been proposed—for example, there exists at least half a dozen logical solutions over the relational model.

Just like in any database management systems, indexing the data is an important solution to speed up its access. Given the schema flexibility of RDF data and the combination of possible indexes over RDF triples, many approaches have been implemented and each comes with its set of advantages and drawbacks.

The next dimension, query processing, is highly related to the previous data indexation. It contains several aspects that discriminate systems—for example, query optimization with both the definition of logical and physical query plans, as well as query execution.

The distribution dimension addresses the data deluge aspect and considers that the volumes of RDF data we are dealing with must be distributed over a set of machines. Several approaches have been envisaged, some pertaining to standard management systems (e.g., range or hash-based), while others are more settled down into graph theory with graph partitioning.

The federation dimension is an important aspect of the Web of Data and the use of Linked Data. The goal of database federation is to enable the querying of a database over several databases in a transparent way. The use of dereferenceable URIs together with the SPARQL query language and SPARQL endpoints supports an unbounded form of data federation that enables the integration of data and knowledge efficiently and in an unprecedented manner.

So far, the dimensions we have presented are anchored in database management considerations. This is the main reason for the presence of Chapter 2, which presents some important notions on both relational and NoSQL stores.

The final dimension we are considering focuses on reasoning services that enable the enrichment of result sets when querying an RDF repository. Such inference considerations are almost never addressed in other existing data management systems, except with the integration of a *datalog* engine in an RDBMS. The understanding of the peculiarities of this dimension depends on a clear comprehension of the available ontology languages and their associated forms of reasoning in the Semantic Web, which is presented in Chapter 3.

CHAPTER TWO

Database Management Systems

The objective of this chapter is to provide some background knowledge on database management systems, a software the purpose of which is to define, create, manage, query, and update a database. We present certain aspects of systems that have been the most widely used in production for the last couple of decades. We also consider some trends that have emerged during the last few years. We limit this investigation to systems that are currently being used or have been in the past as the backend of existing *Resource Description Framework* (RDF) database management systems. Due to space limitations, we cannot provide a thorough presentation of these systems, therefore we concentrate on some peculiar characteristics that motivated their adoption in RDF data management and are main differentiators among these systems.

The first category of systems we consider is the *relational database management system* (RDBMS). It is a very popular family of systems that has dominated the database market for the last 30 years. We provide a short introduction on this topic for readers coming with a (Semantic) Web background, but a complete presentation is out of the scope of this book, (Elsmari, 2010) and (Kifer, 2005) are good introductions. Nevertheless, we define some notions and terms that are going to be used throughout this book. Then, we introduce concepts that are needed for understanding the particularities of the different RDF systems that are studied in Part 2 of this book. These concepts are generally presented in books investigating the internal description of RDBMSs, and are usually not considered in classic RDBMS books that concentrate more on how to model for and use these systems.

The second kind of systems we address corresponds to *NoSQL* systems. These systems only appeared a couple of years ago but already benefit from a high adoption rate in many IT infrastructures. They are used in many different application domains and are far from being limited to the issues of Web companies. This presentation concerns the four main families of NoSQL systems and also introduces some tools that are frequently associated with these systems, such as the popular parallel *MapReduce* processing framework.

Finally, we provide trails on the evolution of RDBMS and NoSQL systems. For instance, we present an introduction to some novel approaches in developing RDBMSs. Here, we are mainly concerned with implementing systems that leverage on the evolution of the hardware environment—for example, the availability of larger main memory spaces, the emergence of *solid-state drives* (SSDs), and the emergence of cloud-based systems. We also emphasize on the appearance of novel functionalities in NoSQL database systems. All these aspects may play a role in the evolution of RDF database management systems in the near future.

2.1 TECHNOLOGIES PREVAILING IN THE RELATIONAL DOMAIN

2.1.1 Relational model

The relational model was introduced by Edgar F. Codd in the early 1970s (Codd, 1970) and is the foundation of RDBMSs. In this model, the first-class citizen is a specific structure named a *relation* that contains tuples (a.k.a. *records*). All the tuples in a relation have the same set of *fields* (a.k.a. *attributes*) where each field has a certain type, such as an integer, a date, or a string. This matches the definition of structured data presented in Chapter 1. The operations performed over this model are handled by an algebra that principally serves as a query language to retrieve data. A selection operation retrieves information from a single relation or a set of relations. In this last case, relations are generally joined over some common-type attributes. An important aspect of this algebra is that it produces an output that takes the form of a structure that is itself a relation. Thus, this approach enables the composition of relational queries one into another—that is, using the result of a query as the input of another one.

The concepts pertaining in the relational model are transposed into an RDBMS using the following term translation: *relations*, *attributes*, and *tuples* correspond respectively to *tables*, *columns*, and *rows* (but these terms can be used interchangeably to denote the same notions). In an RDBMS, the query language is named *Structured Query Language* (SQL), and it implements most of the operations available in the relational algebra but also provides additional ones. Many consider that SQL is a major reason for the success and dominance of this type of data management system. This is mainly due to its short learning curve and its expressivity which has been defined to support good computational complexity properties. In other words, a lot of practical questions can be expressed with few concepts and answered relatively rapidly even over large data sets. Moreover, the wide adoption in most existing RDBMS systems of a common, standardized subset of SQL is another major advantage. Therefore, it enables one user to easily switch from one RDBMS to another with relative ease, such as from **MySQL** to **Oracle** or the other way around. SQL also extends the relation algebra by update operations, which are the ability to delete, insert, or modify information.

For example, we consider an oversimplified blog application containing the following information. Blog entries are being written by users, themselves being characterized by an identifier, first and last names, and a gender. For each blog entry, we store the content of the entry (i.e., the text it contains), its storage date (i.e., the date at which it's being stored in the database), the user who has produced the entry, and the category of the entry. A category corresponds to a subject area, such as sports, technology, or science. Finally, a subscription feature enables end-users to follow the blog entries of other users. Many solutions are available to represent the corresponding conceptual data model (e.g., an *entity relationship* (ER) *diagram*), but we have opted for a *Unified Modeling*

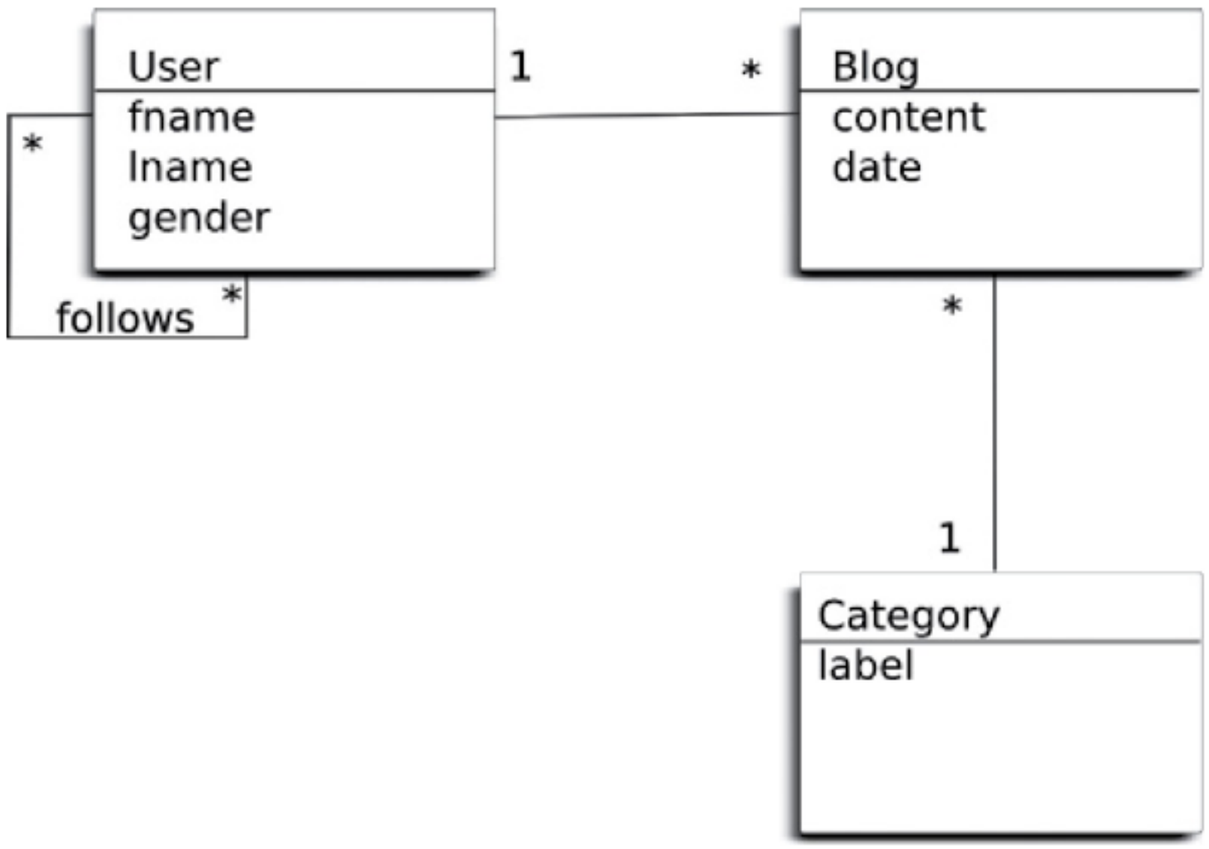

Figure 2.1 Data model for the blog use case using UML notation.

Language (UML) notation using a class diagram, see Figure 2.1. Note that in this diagram, we consider that identifiers are explicit (e.g., user identifier), and we therefore do not display them in the figure.

When the target database is an RDBMS, the conceptual model is translated into a relation schema. This is presented with some sample data in Figure 2.2. The schema contains four tables, namely `User`, `Category`, `Blog`, and `Follows`. The first three are direct translations from the classes proposed in Figure 2.1. The latter corresponds to the `follows` role the cardinalities of which are many-to-many (the two ⋆ symbols in

User

id	fname	lname	gender
1	Joe	Doe	Male
2	Mary	Smith	Female

Follows

followerId	followedId
1	2
1	3
2	1
2	4

Category

id	label
1	Sport
2	Science
3	Technology

Blog

id	content	date	userId	categoryId
1	Today..	10/13/2013	1	3
2	Science is	10/15/2013	1	2
3	My phone	10/16/2013	2	3

Figure 2.2 Sample data for the blog use case.

Figure 2.1), meaning that a user can follow as many users as he or she wants and he or she can also be followed by an unrestricted number of users. The attributes forming this relation correspond to user identifiers from both the follower and the followed users.

We can also see that some columns have been added to some relations. This corresponds to many-to-one relationships between entities (represented as `1 *` associations between boxes in Figure 2.1). This is aimed to support joins between relations—for example, the `CategoryId` of the `Blog` relation enables joins with the `id` attribute of the `Category` relation. This approach implies column redundancy, which allows the definition of queries that may be useful when designing domain application. For example, the following query retrieves all the blogs belonging to the `Science` category and that have been written by persons followed by Mary:

```
SELECT b.content FROM category AS c, blog AS b, follows AS f
WHERE c.label LIKE 'Science' AND c.id=b.categoryId AND followerId=2
AND followedId=userId;
```

The capitalized terms correspond to reserved words of SQL. This query contains three clauses—`SELECT`, `FROM`, and `WHERE`—which respectively retrieve some columns to be displayed in the result set, specify the tables needed for the query to execute (`AS` is used for the creation of aliases for table names, which induces easier reading of the query), and can define some filters and/or joins. This query requires three tables and the `WHERE` clause contains two filters (i.e., on the `label` and `followerId` columns using the `LIKE` and equality operators) and two joins (e.g., on the columns `id` of `Category` and `categoryId` of `Blog`). Executed over the data sample of Figure 2.2, the query's result set contains the blog entry starting with `Science is`, which has been written by Joe (`userId = 1`), who is followed by Mary (`userId = 2`).

Note that an SQL query does not specify how to retrieve the answer set. We qualify languages that are adopting this approach as declarative because they just declare what to retrieve and from which source and under which conditions. This can be opposed to a procedural approach that describes the procedures to execute to obtain a result. An important consequence of being a declarative query language is that it leaves a complete freedom to a processor toward obtaining the answer set. This freedom generally implies that some optimizations are being processed, mainly to ensure fast executions. We will provide more details on the query processor of an RDBMS in Section 2.1.3.

Other forms of SQL queries correspond to update operations such as `UPDATE`, `INSERT`, and `DELETE`, which respectively modify, insert, or remove one or more tuples from a table. The following are examples for each of these operations on the blog example:

- `INSERT INTO User VALUES(3, 'Susan','Doe', 'Female');` introduces a novel row in the `User` table.

- `DELETE FROM Category WHERE id = 1;` removes the row corresponding to Sport from the `Category` table.
- `UPDATE Blog SET userId = 2 WHERE id = 1;` modifies the first row of the `Blog` table by stating that this blog entry has been written by the user whose `id` value is `2`—that is, Mary Smith.

In the next section, we discuss the mechanisms developed to efficiently query large data sets.

2.1.2 Indexes

Agents (i.e., end-users or programs) interacting with a database management system expect that the execution of queries is optimal and will be performed as fast as possible. In cases of large data sets, a naive approach, such as reading all tuples of a given relation to find a particular one, to query execution is far from guaranteeing optimal performance. Because data persistence is taken care of by secondary storage, usually *hard-disk drives* (HDDs) even if SSDs are more and more common, the cost of transferring disk blocks (i.e., the basic disk storage unit for an RDBMS) to the main memory generally exceeds the other query processing operations.

For example, let's assume that the users relation of the blog example contains 100,000 tuples, each record being about 160 bytes long, and that disk blocks are 16 kilobytes per block. This means that 100 records fit into a disk block and that the relation occupies 1,000 blocks. Moreover, only 10 users have Smith as their last name. Consider the following simple query: `SELECT fname, gender FROM users WHERE lname LIKE 'Smith'`. To retrieve all users having Smith as their last name, our naive approach induces the transfer of all 1,000 blocks to the main memory. It's obvious that most of these disk transfers are unnecessary because, at most, 10 of them may contain all the Smith users. A clever approach will consist in identifying these disk blocks and proceeding to at most 10 disk blocks transferred to obtain the correct answer set.

This clever method of identifying the tuples of a given attribute value limits to its minimum the number of *input/output* (I/O) operations, and is enabled by the notion of *indexes*. Intuitively, an index is a data structure the goal of which is to facilitate the discovery of relevant disk blocks needed for the query to execute efficiently. Note that this is the same notion of searching for a given term in a large book. No one would read all the book pages to find all term occurrences. If it's available, one would go to the index section and efficiently find in an alphabetically sorted entry a line for that term. That line would contain all page numbers where the term appears. Most RDBMSs propose, through SQL operators, the creation of different types of indexes that are used for speeding up queries, for example, to efficiently access some values of a given attribute or to improve joins involving a set of attributes.

The definition of indexes comes at a certain price corresponding to their maintenance cost. That is, whenever an update is performed on an indexed attribute (a.k.a. search key), both the relation and the index have to be properly modified. Within the book metaphor,

a change of the content involving an indexed term would imply an index maintenance, precisely to add or remove the page number of the content modification. Therefore, the selection of attributes to index has an important impact on the overall performance of the database. This selection requires an estimation of the importance of the queries to be executed over the database. Intuitively, one only needs the definition of indexes over attributes that are used and filtered in the most frequent queries executed over the database.

There are several indexing organization types. For example, a so-called *clustered index* ensures that the order of tuples in the file storing the data matches the order of the entries in the index. Therefore, only one clustered index is possible on a given file. This is opposed to an *unclustered index* where the physical order of the tuples is not the same as the indexed ones. Thus, several unclustered indexes are possible for a given relation. An index can also be qualified as *dense*, if there is a least one data entry per search key, otherwise it's called *sparse*.

Finally, several implementations are possible, the most frequent ones being balanced trees, B-trees or one of its variations (e.g, B+trees), and *hashes*. But the research has been prolific on this topic and some other indices are particularly adapted to certain domains, such as an *R-index* and other *generalized search trees* (GiSTs). To select one instead of another, the *database administrator* (DBA) generally needs to be acknowledged on the kind of queries that will frequently be executed. For example, if one popular query requires the access to a range of attribute values, then a balanced tree index is preferable to a hash-based one. This is due to the fact that hash indexes only support equality comparison. In Part 2 of this book, we will see that many RDF stores are using B+tree to index triples, therefore we only present this approach.

A B+tree is a structure that organizes its blocks in a tree like manner. The structure is composed by three components: a root node, internal nodes, and leaves. In a nutshell, the structure of each of these components is common, and composed of n search keys and `n + 1` pointers. The pointers of the root or of any internal node point to other internal nodes or leaves. The pointers of leaves lead to the tuples. The main reason for the popularity of this structure in a database management system lies in the fact that it's *balanced*. This property means that all paths from the root to any leaf have the same length and ensures a worst-case complexity of `O(log(n))` for the search, insert, and delete operations, and `O(n)` for space occupied by the structure.

Finally, note that the disk block approach is adopted in both HDDs and SSDs. That is, the manner in which data is retrieved from an HDD resembles the one of SSDs but with an average of two orders of magnitude more I/O per second for the latter. Nevertheless, even if your financial situation allows for SSDs, it would be a big mistake not to use indexes. After all, the objective is the transfer of data from a secondary storage to the main memory, and *random access memory* (RAM) is still three orders of magnitude faster than SSD. So in our all SSD scenario, your performance bottleneck will still be I/O transfer and indexes are an important helper in that case.

2.1.3 Query processing

To fully appreciate the architecture of RDF database systems, it's important to present the common components found in most RDBMSs' query processing (Figure 2.3). The two main components of query processing are *query compilation* and *execution*. Two steps are distinguished by the query compiler: *parsing* and *optimizing*.

Parsing involves scanning a query to make sure that it's well formed—that is, it complies to a set of syntactical rules and makes sure that all entities, such as tables, views, attributes, and functions, referred to are known by the system.

In the case where a query passes the parsing stage, the query is transformed into an internal format that generally takes the form of an algebra operator tree. This internal representation, which uses most of the operators of the relational algebra as well as some additional ones (e.g., for handling aggregation operations), aims at facilitating the optimization step. Several tree representations are possible for a given query and a component is responsible for selecting the most cost-effective logical query plan. This is performed using a set of rules that are defined over the relational algebra and its operations.

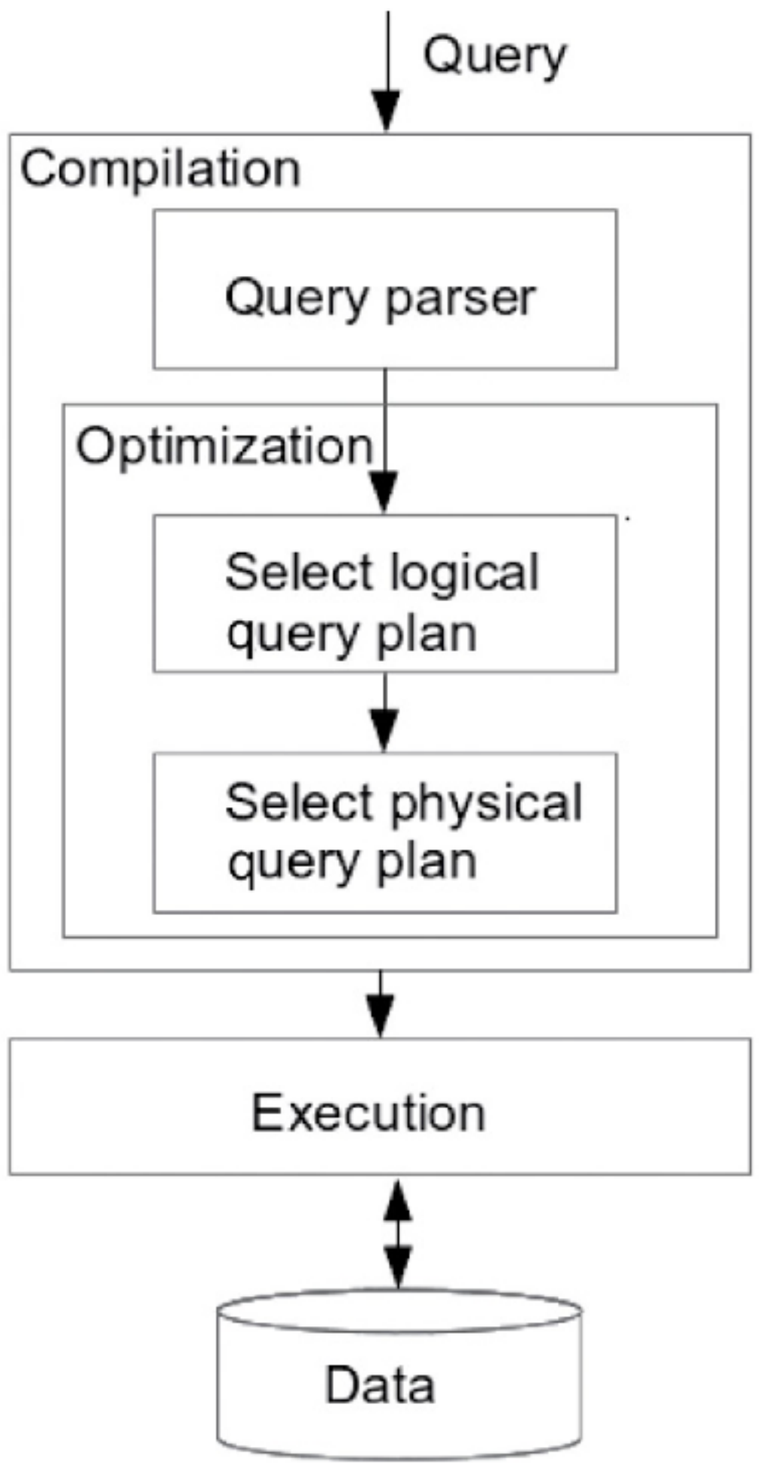

Figure 2.3 Overview of query processing.

The query plan is qualified as "logical" since it only defines the order on which the different operations will be performed and does not specify anything about which methods will be used for accessing stored data. To address this issue, another optimization form is required by the system and amounts to the translation of logical query plans into operations performed on the files containing the data, what is referred to as the *physical query plan*. Again, several physical query plans can be envisaged from a single logical query plan, and the selection of the most efficient one is based on an estimation of their execution duration. This implies a cost-estimation approach that takes the form of a set of heuristics that are using different parameters, such as the relation sizes, index availability, or join types. For example, histograms corresponding to statistics on the distribution of standard values of a relation are extensively used. They use information on indexes available for the involved relations. The order of joins is also the responsibility of this component, and it results in identifying the join algorithms that are being selected. The goal of this selection is not limited to the definition of operations required for the query to execute but also specifies the order on which they are performed.

Once a physical query plan is selected, it's sent to the query execution component, which communicates with the stored data to produce an answer set. This component executes the algorithms specified in the physical plan. There exists an important number of different algorithms that can be distinguished by whether one of the relations to be joined fits or not into the main memory, or whether the join requires a sort of one of the involved relations. This motivates whether a one-, two-, or multiple-pass algorithm can be used. For some of these algorithms, sorting or hashing the content of relations may be required for speeding up query executions.

2.1.4 ACID transactions and OLTP

In this section, we are not directly interested in the internal aspect of RDBMSs, but rather consider their usages. We can consider that they can be partitioned into two big categories: the category that is mainly handling transactions, called *online transactional processing* (OLTP), and the one concerned with the analysis of important workloads, called *online analytical processing* (OLAP).

An objective of OLTP systems is to properly maintain the different states of a real-world model in the context of possibly concurrent transactions. In this setting, the notion of a transaction corresponds to a non empty set of operations performed in SQL over the database. The set of operations is usually less important than the number encountered in an OLAP system, but the number of transactions per second can be very important, meaning that the state of the database is constantly evolving. To maintain the database in a coherent state, a set of properties are expected from such systems. This is characterized by the *atomicity*, *consistency*, *isolation*, and *duration* (ACID) properties.

Atomicity is an all-or-nothing approach on the set of operations constituting the transaction. This means that if one of the transaction operations cannot be properly

executed, then every operation before that one has to be canceled and the transaction is not performed at all. Atomicity is tightly related to the consistency property, which states that the execution of a transaction must lead the database to a consistent state. We will see in the next section that this consistency is a very important aspect of the NoSQL ecosystem. Intuitively, a database state is consistent if it does not violate integrity constraints that can be defined in SQL or in some programming language. The isolation property is related to interferences that can occur between transactions that are executed in the same time window. It states that until a transaction is properly terminated (committed in the database jargon), all concurrent access on the same data is hidden—that is, other users will see the older data values, not the one being modified by the transaction. The duration property induces that all committed transactions are persisted on a secondary storage, most frequently an HDD. In general, OLTP applications are best described by relatively frequent updates, short and simple transactions accessing usually a small portion of the database. A ubiquitous example using OLTP databases is e-commerce applications.

The goal of OLAP, together with data mining, is to analyze the data contained in a database such that decisions and predictions can be taken by end-users or computer agents. These systems are implemented in so-called *data warehouses* and are frequently used in fields such as *business intelligence* (BI). In general, transactions are rare in OLAP (e.g., write operations are not frequent), but when some are performed they usually involve a very large number of operations. The execution of a million rows on a weekly or monthly basis is not rare. Therefore, the state of a database is more stable because it changes less frequently than in the OLTP context. This reduces the effort to maintain data consistency.

Because OLAP systems are read-intensive, they can be highly denormalized. This means that the conceptual schema of the database is modified in such a way that some queries require less joins (which is a bottleneck of query processing). This comes at the price of data redundancy, but it's usually not a problem for servers supporting data warehouses. Therefore, OLAP solutions usually do not need the whole ACID machinery because their write and update transactions are performed adopting a scheduled batch-processing approach—that is, every indicated time period a large chunk of data is inserted in the data warehouse. Retail stores make intensive use of OLAP data warehouses, for example, to identify product consumptions.

Both OLTP and OLAP are being queried with SQL, although more procedural approaches with a programming language can be used. The standard set of SQL operators is extended for OLAP systems because the queries can be more complex than in OLTP and can access a significant fraction of the database. These queries frequently handle multidimensional operations where a dimension corresponds to an attribute in a table. For instance, in a retail database, product sales would have the store location, product and time identifiers as dimensions. This could be represented as a cube where the quantity

sold of a product at a location at a certain time would be stored in cells of that matrix. Typical operations performed on such cubes are *consolidation, drilling-down, rolling-up, slicing, and dicing*. Consolidation aggregates data that can be computed in one or several dimensions. The drill-down and rolling-up techniques allow a navigation along those dimensions, e.g., along the time dimension (weeks, months or quarters). Finally, slicing and dicing respectively retrieves and views slices from different points of view. The efficiency of performing these analytical operations is supported by efficient implementations of new SQL operations, e.g., GROUPY BY CUBE and GROUP BY ROLLUP, denormalized schemata and materialized views. These are the main reasons why OLTP systems are highly inefficient at performing similar analysis operations.

2.1.5 Row versus column stores

To summarize what we have already expressed in this chapter, an RDBMS persists its data in secondary memory (principally in an HDD, but SSDs are more and more frequent), and transferring data to the main memory is required whenever we want to do something useful with the data. So far we have indicated that the unit of transfer is a disk block (a.k.a. a page), and that a page contains a set of tuples. That approach corresponds to a row store—that is, a page stores some tuples. Another approach consists in storing columns of a relation separately so that they can be accessed independently—that is a page stores some columns.

The row store approach has been the most dominant in RDBMS history, but things are evolving toward the adoption of its column counterpart, mainly due to the need to analyze very large data sets as encountered in Big Data. Some systems are also proposing hybrid RDBMSs where the DBA can specify which of the two approaches makes more sense. For example, a table `T1` may physically be stored as rows because some important queries frequently executed by an application must retrieve all its columns at a time. At the same time, another table `T2` may adopt a column-oriented approach because many queries are retrieving a subset of its columns. The fact that `T1` and `T2` are respectively frequently or rarely updated would confirm this physical layout.

The context of adopting a row store is related to the use cases of OLTP—that is, domains requiring both fast reads and writes. In contrast, column stores are optimized in the context of OLAP—that is, they are characterized by mostly read queries with infrequent writes. These systems propose reasonably fast load times, which work efficiently for batch jobs but do not possess the capacity to handle high update rates. The kind of queries generally executed over a data warehouse involve big aggregations and summarizations over an identified number of columns. This matches cases where a column store is particularly efficient compared to a row store.

The storage layer is an important differentiator of row and column stores. Consider the tuples of the `User` relation in Figure 2.2. A row store would organize the tuples as displayed in that figure—that is values of the User table attributes are stored together

in the same structure. Although different column store approaches exist among existing systems, a standard representation would be the following:

```
(userid) 1, 2, …
(fname) Joe, Mary, …
(lname) Doe, Smith, …
(gender) Male, Female, …
```

Note that the tuples are stored in several (one for each attribute) structures, denoted as *files* in this section, which correspond to a set of disk blocks. Moreover, the values in each of these files are ordered, which enables the reconstruction of the tuples if required. For example, the second values of each file support the retrieval of the information of the second tuple of the original table. To match values based on their position in the corresponding files, this approach imposes that columns that are undefined for a particular row must explicitly store a `NULL` value in the corresponding file. This storage organization clearly presents some advantages and drawbacks.

As advantages, queries that only retrieve a small subset of the attributes of a relation will transfer to the main memory a higher rate of useful information than a row store that transfers all attributes of some tuples. Another advantage is the possibility to compress the data of each column because all values have the same domain. For example, consider the compression possibilities on the `gender` column in Figure 2.2 where only two values are possible. Note that in certain use cases, query answering can be performed without decompressing the data. A typical example consists in counting the number of entries in data clusters—for example, counting the number of blog entries in each category. As a major disadvantage, tuple updates are slow because they may require the reorganization of several files.

Comparatively, row stores present the assets of more efficient update operations over a relation's tuple because a tuple is stored over a small number of disk blocks. An important drawback is related to the fact that most useful queries rarely retrieve all the information from a given tuple, but rather retrieve only a subset of it. That implies that a large portion of the tuple's data is unnecessarily transferred into the main memory. This has an impact on the I/O efficiency of row stores.

In Abadi (2007) the author states that column stores are good candidates for extremely wide tables and for databases handling sparse data. The paper demonstrates the potential of column stores for the Semantic Web through the storage of RDF. Based on these remarks, it's not a surprise that the current trend with database vendors emphasizes that column stores are getting more popular and can, in fact, compete with row stores in many use cases. Many recent systems have emerged in this direction, such as **Sybase IQ**, **Vertica**, **VectorWise**, **MonetDB**, **ParAccel**, and **Infobright**, together with older systems, such as **Teradata**.

Note that a new breed of RDBMS is trying to enjoy both worlds of row and column data storage. Systems such as **Greenplum** propose a so-called *polymorphic data storage* approach where a DBA can specify the storage, execution, and compression associated to each table.

2.1.6 Distributed and parallel DBMS

So far we have assumed that an RDBMS is running on a single server. In this context, issues related to the support of a concurrent access are handled using transactions and their ACID properties. Nevertheless, due to scalability, security, or location aspects, it may be necessary to distribute the data over a set of machines. In this section, we consider such situations and present some special kind of database systems that integrate distribution and related processes at their core.

A distributed database corresponds to an integrated set of databases that is physically distributed over different sites of a computer network. The software that manages such databases is denoted as a distributed management system, and its main characteristic is to make the different interactions with end-users and applications totally transparent from the distribution aspect—that is, the end-user does not have to know about the organization of the data over this network. This approach is obvious for certain organizations where the data is naturally (geographically) distributed over a network. In some cases, this is so profoundly anchored in the business model that it may not even be possible to deploy a centralized system. The advantage of this approach is to reinforce the autonomy of each site (e.g., by administering its own data), which reduces the communication overhead because the data is most of the time retrieved from a user's own local site. Finally, the availability of the data is increased, because in case a site falls down, the other functioning sites can potentially ensure data retrieval and query processing on replicated data.

Distributed systems typically involve replication of data across a number of physical (as opposed to virtual) machines. There exists several reasons that motivate the adoption of a replication approach. Some, as we have already seen, aim to design a more robust system, one that is more tolerant to parts of the system failing, or aim to build a system with higher availability properties by ensuring close location of machines needed for some processing. Another motivation is to support a form of parallelism that could be valuable when different processes are simultaneously operating on the same data by providing each of them with their own copy.

We can distinguish two types of data replications that depend on whether the copy of data is performed synchronously or asynchronously. A *synchronous replication* is performed via atomic write operations and therefore assures that no data is lost in the process. To guarantee completeness of the write operations, both the original and replica machines need for emission a form of acknowledgment (referred to as a *handshake*). This approach frequently implies that applications wait for completeness before accepting other operations. Thus, synchronous replication is frequently associated with latencies and poor

performances. In *asynchronous replication,* the goal is to prevent latencies at the cost of possibly losing some data. This is performed with a different completeness strategy. Intuitively, a write is complete as soon as the original data is updated. Then the replica is updated, but one does not need to wait for any form of acknowledgment. The first systems designed with the concepts of a distributed database were **SSD-1** and **Distributed INGRES** in the 1970s. But the first mature systems can be considered as appearing in the mid-1980s with **INGRES/Star** and **Oracle version 7**.

A parallel database system manages data stored on a multiprocessor computer. It generally implements all the functionalities of a standard database management system. The main principle of such a system is to partition the data over the different multiprocessor nodes. Advantages of this approach consist in improving the performance and availability through parallelism and replication. By being able to address these operations efficiently, a parallel database management system is able to support very large volumes of data, a prerequisite in the current data deluge.

In DeWitt and Gray (1992), the authors stress ideal properties expected from a parallel database: linear speed-up and linear scale-up. The former means that "twice as much hardware can perform the task in half the elapsed time," (p. 3) while the latter implies that "twice as much hardware can perform twice as large a task in the same elapsed time" (p. 3) Ensuring such properties usually comes at the cost of providing an adapted hardware architecture. Stonebraker (1986) proposes a taxonomy that, although defined more than 25 years ago, has never been as relevant as now. It defines the following three architectures. The *shared-nothing* architecture (Figure 2.4a) consists of a set of connected, via an interconnection network (e.g., a gigabit ethernet), processors that can access their own memory and disk. Alternatives are the *shared-memory* architecture (Figure 2.4b) where a set of processors share, through an interconnection network, a common main memory and all disks. This facility of global memory and disks eases the design and implementation of the database software, but it comes at several scaling limitations. Two of these limitations are the interference of shared resources (e.g., lock tables and buffer manager) and the fact that the memory system can rapidly become a bottleneck. Finally, in a *shared-disk* architecture (Figure 2.4c), each processor has its own private memory but they can all access all disks via an interconnection network. Such an architecture induces a coordination of accesses to shared data that is implemented with a complex distributed lock manager. Oracle's *Real Application Clusters* (RAC) is one implementation of the shared-disk approach. Among the three possible architectures, the shared-nothing approach is, by far, the most adapted to parallel databases, and we will soon see that it's widely used in the NoSQL ecosystem and in cloud computing architectures.

The main advantages of the shared-nothing approach are that it minimizes interference between each machine—that is, no exchanges of information stored in main memories and disks are needed—and it can be scaled out to thousands of machines, by adding new commodity machines to the cluster, without impacting the global infrastructure's

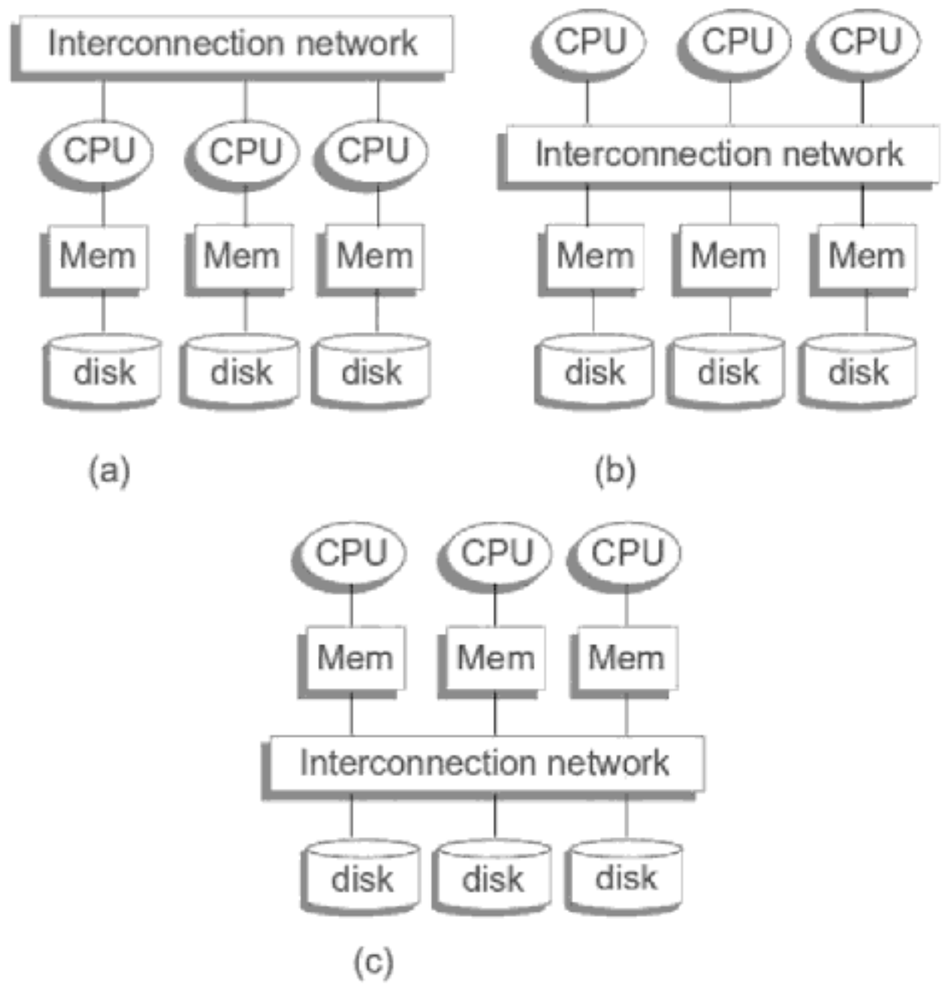

Figure 2.4 Taxonomy of parallel architectures: (a) shared nothing, (b) shared memory, and (c) shared disk.

performance. For example, it's known to be really difficult to scale shared-disk and shared-memory systems to more than respectively 10–20 and 100 nodes. Nevertheless, increasing the number of machines also increases the probabilities of failover of the architecture—that is, the more machines you get in your cluster, the more likely one will break down. Guaranteeing a quality of service for applications developed over such clusters comes at the cost of defining clever partitioning (e.g., an efficient data skew) and replication strategies.

Data partitioning is an operation performed at the relation level involving the distribution of its set of tuples over a set of disks. The main partitioning strategies are *round-robin*, *hash*, and *range* based. They respectively map the *i*th tuple of a relation to the (*i modulo n*) disk, each tuple to a disk identified by a hash function, and contiguous attribute ranges to distinct disks. Other partitioning approaches can be defined but are less frequently implemented. In the case of graph-oriented data, several strategies can be applied using graph operations (e.g., graph centrality), which are denoted as *graph partitioning*. In Part 2 of this book, we will present some uses of these techniques over RDF graphs.

Few open-source systems exist in this category. Most systems are commercial and are generally very expensive. This forces many Web actors to develop their own parallel storage and processing infrastructure, mostly on a shared-nothing architecture. The first systems, such as Teradata, date back to the end of the 1970s. Recently several systems emerged in this parallel database environment: Vertica, Greenplum, and **Xtreme Data**.

2.2 TECHNOLOGIES PREVAILING IN THE NOSQL ECOSYSTEM

2.2.1 Introduction

The term *NoSQL* was coined in 2009 to advertise a meeting in the Silicon Valley on novel data stores that are not based on the relational model. At the time, the acronym was standing for "NO SQL" and was reflecting a situation where programmers and software engineers were complaining about RDBMS and object-relational mapping (ORM) tools that prevent them to concentrate on their original mission, i.e., programming or designing systems. Among others, this fatigue could be justified by the omnipresence of SQL code in programming code lines, the impedance mismatch between data stored in the RDBMS and the programs, the hardness to distribute relations and their processing over a cluster of machines, and the inadequacy of the relational data model to some application domains (e.g. graph data). Therefore, the NO SQL movement was interpreted as a rejection of RDBMSs with their SQL and ACID machinery.

Since then, the term has evolved slightly toward a less controversial definition. It's now interpreted as "Not only SQL" (hence the small "o" in our spelling). This recognizes that RDBMSs have many qualities that justify their presence in the IT ecosystem, but that other forms of data management systems are needed to reply to expectations motivated by the Big Data phenomenon. This clearly indicates that there is room for more than one database management system type and that their coexistence will be required in many IT architectures. This corresponds to the notion of polyglot persistence on which we will come back to several times in Part 2 of this book.

Even if the NoSQL/NOSQL term appeared in 2009, the first database systems (we will use the term *store* interchangeably) considered to be part of this ecosystem were presented in papers published in research conferences in 2006 and 2007. These papers were respectively describing the **BigTable** database designed at **Google** (Chang et al., 2006) and the **Dynamo** system implemented at **Amazon** (DeCandia et al., 2007). It's not a surprise to see that these systems were pioneered by Web companies with Big Data issues.

It's important to understand what motivated the design of these first systems. In the previous section, we emphasized that parallel database systems are required for companies managing very large data sets. But available parallel database management systems are very expensive and are not open source. While many (Web) companies are able to run their business from off-the-shelf database systems, the likes of Google and Amazon are in need of special tuning, data, and processing distributions, and modification and extension capabilities of existing systems. Therefore, they need complete control over the frameworks running in their data infrastructure. This requisite motivates them to use open-source solutions for security and extensibility reasons—that is, they have the ability to study, modify, optimize, maintain, and distribute source codes, as well as to build contributing communities around them. Moreover, they are facing a data deluge that

forces them to consider storing and processing very large volumes of data and arriving at high throughput, with novel approaches. The need to distribute the workload on distant data centers could not be handled efficiently by only relying on RDBMSs, although they are widely used by these companies (e.g., MySQL or **MariaDB**). Therefore, they engaged into contributions to this field up to a point where it now represents an essential part of their technology stack. This approach allows an increase in their productivity in terms of application development and scalability. We will see in this chapter that other large Web companies have contributed to this ecosystem, such as **Facebook** with **Cassandra** and **Twitter** with **FlockDB**.

NoSQL does not correspond to a single form of database system and does not follow any standard. Rather, it covers a wide range of technologies and data architectures for managing Web-scale data while having the following common features: persistent data, nonrelational data model, flexible schema approaches, individual (usually procedural) query solutions rather than using a standard declarative query language, avoiding join operations, distribution, massive horizontal scaling, replication support, consistency within a node of the cluster and eventually across the cluster, and simple transactions. We will now clarify each of these notions.

The concept of *persistent data* is the same as the one presented in the context of RDBMS. It mainly states that data aims to be stored in a nonvolatile memory, such as secondary memory, which used to be exclusively HDDs but are now sometimes replaced with SSDs. For some NoSQL systems, such as the **Aerospike** store (`http://www.aerospike.com/`), the use of SSDs is an important feature and motivates design decisions that are announced as the main reason for high performances.

None of the NoSQL systems follows the relational data model. Rather, they are influenced by key-value structures (e.g., hash tables) and are sometimes referred as *distributed hash tables* (DHTs). At least three out of four NoSQL system families adopt this approach, and they are mainly differentiated by the type of information they can store at the value position. The other quarter of NoSQL systems rely on the graph model and have a set of totally different properties and use cases.

An important aspect of the application domains motivating the emergence of NoSQL stores was the need for flexibility in terms of database schema. In fact, the designers of these systems pushed this characteristic to its logical limit by not providing facilities to define a schema. This schemaless property facilitates an important feature needed in many data management approaches: the integration of novel data into the database. For example, consider a relation where one can add as many attributes as needed. In an RDBMS, this would imply a high rate of `NULL` values for tuples where some attributes do not apply or are not known. In many NoSQL stores, one simply does not assign values for the attributes he or she does not want or know.

This flexibility imposes the design of novel query methods. In general, early NoSQL systems were supported by the application programmer community (as opposed

to the database community) who were eager to access databases using their favorite programming language rather than a query language like SQL or an ORM solution, such as **TopLink** and **Hibernate**. In fact, until recently, most NoSQL stores were not proposing a dedicated query language, and the only method supported to retrieve and update data was through a set of *application programming interfaces* (APIs) and code written in some programming language. Because, in this context, queries are procedurally defined, as opposed to declaratively with a language like SQL, it's not possible anymore to automatically optimize them. This means that the optimization responsibility totally rests on the shoulders of the programmers! We consider this one aspect has to be taken care of very seriously when adopting such a database system.

For performance reasons, NoSQL database instances are frequently accessed by a single application. This enables users to fine-tune the model to handle, as previously identified, a set of queries required by the application. The goal is thus to model, as early as possible, the database to efficiently perform these peculiar queries. This usually comes at the cost of having poor performances for other queries. This approach is reminiscent to the denormalization step (see section 2.1.4), which is usually encountered at a tuning stage of an RDBMS. In this chapter, we already emphasized that denormalization aims at limiting the number of joins present in the most frequently executed queries. For example, consider that a frequent query in the blog application retrieves, for a given user, blog entries with the associated label category. For Joe Doe, our query would be: `SELECT b.content, b.date, c.label FROM category AS c, blog AS b WHERE b.userId = 1 AND c.id = b.categoryId.` In this case, storing the label category in the blog relation would enable us to write the more efficient following query: `SELECT content, date, label FROM blog WHERE userId = 1.` NoSQL systems go one step further by not supporting joins. Practically, join operations are programmed into the applications, giving again a complete freedom in terms of data structures and algorithms to the programmers. The impact of denormalization is not limited to reducing the number of joins; it also implies to increase data redundancy and support atomic transactions more efficiently—that is, transactions that occur at the atomic level of the entities supported by the NoSQL store (e.g., a document in a document store).

Except for graph databases, NoSQL systems integrate data distribution as a main feature through the adoption of the shared-nothing architecture. The scalability issue can be addressed in two different ways. The first way, known as *vertical scaling* or *scaling up*, upgrades a given machine by providing more *central processing unit* (CPU) power, more memory, or more disks. This solution has an intrinsic limit because one can reach the limit of upgrading a given machine. This approach is also considered to be very expensive and establishes a strong connection with a machine provider. The second approach, known as *horizontal scaling* or *scaling out*, implies buying more (commodity) machines when scalability issues are emerging. This solution has almost no limit in the number of machines one can add, and is less expensive due to low prices of commodity hardware.

In terms of data distribution, two main approaches are identified: *functional scaling* and *sharding*. With functional scaling, groups of data are created by functions and spread across database instances. In our running example, functions could be users, categories, and blogs. Each of them would be stored on different machines. The sharding approach, which can be used together with functional scaling, aims at splitting data within functions across multiple database instances. For example, suppose that if our user function is too large to fit into one machine, we can create shards out of its data set and distribute them on several machines. The attribute on which we create the shards is named the sharding key. In our running example, this could be the last name of the users. A second factor of this sharding-based distribution is the number of machines available. Let's consider that we have two machines. A naive approach could be to store all users with a last name starting with letters from A to M included on the first machine and with letters from N to Z on the second machine. This approach may not be very efficient if users' last names are not evenly distributed over the two machines. That is, there may be 80% of the names in the A to M range and only 20% in the N to Z range, causing some load-balancing issues—that is, the first machine stores and certainly processes a lot more data than the second machine.

In addition to these two distribution approaches, replication strategies can be used to store more than one copy of the data. Suppose we are replicating the A to M user's data set on three machines that are situated on different data centers. Then, if one machine (or data center) breaks down, we can still access the data set from the two remaining machines. Note that it's also possible to balance the read operations on these three machines and therefore provide better performances. In our running example, consider that the blogs of Joe Doe are stored on a server `S1` (in Europe) and that Mary Smith, a follower of Joe Doe, is accessing Joe's blog entries using another server `S2` (in the United States). At a certain time, Joe is entering a new blog entry, and before any replication can be performed between `S1` and `S2`, there is a network failure—that is, communications between the nodes of `S1` and `S2` are not possible anymore. In a case where Joe's blogs are replicated on some nodes of `S2`, then Mary will be able to access any of Joe's blogs but the last one. Once the partition is solved and the data transfer between `S1` and `S2` is completed, Mary will be able to access Joe's last blog entry.

High replication also comes with some limitations, the main one being latencies related to the data replication. One kind of approach is to have a single machine accepting the write operations (denoted the master) and several slaves that are only accepting read operations from their clients. A simple running principle is the following: each time the slave receives a write, it will communicate with all the slaves to provide an update, therefore enabling them to stay up-to-date. The replication strategy can be synchronous or asynchronous, as presented in section 2.1.6, implying different latencies that may be more or less acceptable given the frequency of updates at the master. Of course, multiple

replication solutions are possible, which more or less prevent some problems, such as architecture with multiple masters.

The final term of our description refers to the notion of consistency, which corresponds to the C in the ACID transaction properties. In the context of the CAP theorem (introduced in the next section), most NoSQL systems address consistency in a different manner than in RDBMS—that is, not using the machinery associated with the ACID properties.

According to their data model, distribution and replication strategies, we distinguish four NoSQL categories. Each one having its own peculiarities and facilitating the management of some particular kind of data: view of a database as a storage solution for persisting a value (*key-value stores*), allowing more flexibility about stored data (*document stores*), managing use cases like relationships (*graph databases*), or aggregating data (*column databases*). These systems have frequently been used to store various system logs, some components of social networks, or shopping carts in e-commerce web applications. But recently, new domains such as science and finance have started to be interested in these systems. NoSQL systems are becoming so popular that even database giants like Oracle, **IBM**, and **Microsoft** are enriching their offers to match some of the NoSQL features.

2.2.2 CAP and BASE

The *CAP* conjecture was proposed by Eric Brewer in 2000 (Brewer, 2000). This conjecture became a theorem after Gilbert and Lynch provided a demonstration in 2002 (Gilbert and Lynch, 2002). The CAP acronym corresponds to the three properties expected from a distributed system: *consistency*, *availability*, and *partition tolerance*. Intuitively, the theorem states that one cannot embrace the full potential of all three properties at the same time in a distributed system. To understand its impact on the design of distributed systems, we provide the definitions proposed in Gilbert and Lynch's 2002 paper. *Consistency* is defined as the fact that a service operates fully or not at all. This matches the notion of *atomicity* rather than *consistency* of ACID, but CAP sounds better than AAP. *Availability* is described as "every request received by a nonfailing node in the system must reply a response result" (p. 3). Note that this definition does not state anything about the amount of time needed to return the response. For many, the real constraint on availability is the notion of *latency*—that is, how is it going to take the nonfailing node to return the response? Finally, partition tolerance is specified as the fact that "no set of failures less than total network failure is allowed to cause the system to respond incorrectly" (p. 4). Here, a total network failure means that none of the nodes of the network can respond to a request. This theorem clearly had and still has a big influence on current database management systems, most notably in NoSQL stores.

The consequence of this theorem is that one has to identify which properties are the most relevant in a given system implementation. That is, if we cannot get all three properties at the same time, which one can be abandoned to still obtain a valuable

system? In fact, there is not a single reply to this question, and the following configurations correspond to practical system requirements:

- We retain the consistency and availability properties (so our system is qualified as CA), but we renounce tolerance to network partitions. These systems typically correspond to standard RDBMS, such as MySQL and **Postgresql** to name open-source systems.
- We retain consistency and tolerance to network partitions (CP), but we accept that the system may not be available. Obviously, the idea here is that, in cases of network partitions, the system gives up availability and prefers consistency. Systems in this category are BigTable, **MongoDB**, **HBase**, and **Redis**.
- We retain the availability and tolerance to network partitions (AP), but we are ready to neglect consistency. Again, this defines another trade-off between availability and consistency where one prefers to be available rather than consistent. Systems adopting this approach are Dynamo from Amazon, Cassandra, **CouchDB**, and **Riak**.

In practice, the way consistency is handled is more subtle. In Vogels (2009), the author presents several forms of consistency that can be implemented in distributed services or data stores. Here, we focus on the two main ones. *Strong consistency* is the notion that we have considered in RDBMS with the ACID properties. This approach increases latencies in favor of data consistency. *Eventual consistency*, probably the most adopted in NoSQL stores, states that if no further updates are performed on the same data object, eventually all accesses are going to retrieve the last updated value. Said differently, all machines will be consistent after a given amount of time, denoted as the *inconsistency window*. In our running example, consider that Joe Doe has just added a new science blog entry that is stored in Europe on the S1 server. Before the replication to S2 (in the United States) is performed, Mary Smith is refreshing her science page. Because we are in the inconsistency window—that is, the time taken by the replication process to guarantee that Joe Doe's science entries are the same in all replica—Mary's science page will not display the last entry from Joe. At another time, outside of the inconsistency window, the most recent blog on Mary's science page will contain the last entry from Joe. These various consistency definitions are generally defined over three parameters:

- `N`, the number of nodes storing a replica of the data.
- `W`, the number of replicas needing to submit an update acknowledgment before the update is considered complete.
- `R`, the number of replicas that must be contacted for a read operation.

The different consistency strategies can be expressed in terms of relations among `N`, `W`, and `R`.

The term *basically available*, *soft state*, *eventually consistent* (BASE) has been presented by Pritchett (2008) and obviously aims to be opposed to ACID. Except for availability, the concepts composing BASE have not been precisely defined. Pritchett clearly presents BASE as a method to develop distributed systems using some dedicated design patterns. This interesting approach enables us to define services that can reject synchronization

in favor of fast response times and therefore accepts some form of inconsistencies. These services usually possess their own methods to correct these inconsistencies—for example, by merging contradicting customer orders in an e-commerce situation. The main thing to retain is that a whole spectrum of architecture can be defined between the ACID and BASE extrema by implementing these design patterns into the application code.

2.2.3 NoSQL systems

In this section, we present the principal characteristics of the four main NoSQL families. The first three—key-value store, document store, and column family—can be considered as one family that is sometimes referred to as DHT or aggregate-oriented databases (Sadalage and Fowler, 2012), because they can all be viewed as key-value stores differing on what can be recorded and accessed at the value position. We will compare these systems on the following dimensions: *data structure*, *consistency*, *scalability*, *transaction support*, *indexing methods*, and *query facilities*. Because there does not exist any standard for any of these families, we will, for each family, consider the approach generally assumed for each dimension. Moreover, for each family, we will exhibit typical use cases and concisely present the most popular existing system.

Key-value stores

The *key-value store* family is the simplest category of NoSQL databases in terms of model and query facilities. This family corresponds to a hash table that is mainly used to access data stored using a primary key. This key is, in general, the only available index created for this kind of database. It can be represented in an RDBMS using a single table with two columns: one for the key and another one for the value. Like any hash table available in most programming languages, it's very efficient to retrieve a value given a certain key but is not adapted if one needs to obtain the key from a given value or to access values of a range of keys. Therefore, the main operations performed over this model are `get`, `put`, and `delete`, which respectively correspond to retrieve a value from a key; add a key-value pair in the case where the key does not exist, otherwise replace the value for that key; and to delete a key-value pair.

Many key-value stores exist, popular ones are: **Memcached**, Redis, **BerkeleyDB** (Oracle), **DynamoDB** (Amazon), **Voldemort** (Project), and Riak. Existing systems support different types for a value. Most of these systems can be of any type, ranging from simple types like a string of characters or a numerical value, to more complex objects like JSON and XML, and can possibly use lists or sets. In general, there is no native method to access a specific field in a complex value—that is, one cannot directly retrieve a section of a JSON document or access the *i*th element of a list. Given the three available operations, consistency can only be addressed on a single key. Some systems, like Riak, adopt the eventual consistency approach. In that case, conflict resolution is handled by either a newest-writes-win strategy or by retrieving all possible values and letting the end-user resolve the conflict.

Sharding the data is the principal solution to address scalability. A classic approach is to distribute given key-value pairs to determine on which machine the key-value pair is stored. Although this can improve performance by adding more machines, this method also comes with several problems. For example, if a node goes down, then its data is not accessible anymore and no data can be further stored on it. Another problem is related to data skew, which is the fact that the data as well as the workload are not evenly distributed over the nodes. In certain systems, a configuration based on the `N`, `W`, and `R` properties of the cluster can be defined to ensure performances for read and write operations. Usually transactions on write operations are not guaranteed in key-value stores. Nevertheless, some systems implement special approaches to overcome this limitation. For instance, Riak uses a write quorum approach. It amounts to setting certain values to `N`, `W`, and `R` such that not all replicas have to acknowledge an update, but rather half of it is sufficient. This can be stated as `W > N/2`. Typical use cases for key-value stores are domains with a low-granularity model such as a Web session, shopping-cart information, and user profiles. For models in need of more fine-grained modeling, the next two systems—document store and column family store—may be preferable.

We can now try to model the blog running example with a key-value store. It's first important to note that these kinds of database systems are considered special-purpose and are not designed to replace an RDBMS. Nevertheless, to bring to light the assets and drawbacks of key-value stores, we present a complete model of the blog example with an abstraction of a key-value store that possesses some of the characteristics of the Redis system. Our representation takes the form of a unique, very large DHT where all keys are being stored. To differentiate among key entries that are going to store users, categories, as well as blogs, we prefix the keys with respectively `uid`, `cat`, and `blog`. The right table in Figure 2.5 displays the different entries associated to a given key. The value entries follow the pattern label/value, where the label corresponds to an attribute in an RDBMS table. The left table in Figure 2.5 contains seven keys and respectively identifies two users, a username, two categories, and two blog entries.

The `username:msmith` needs some clarification. Recall that a key-value store only allows us to access information from a key. Therefore, if the system only provides a user-related key based on its identifier (e.g., `uid:1`), there will be no way to access information given its username. This is the main motivation behind the creation of a username-based key for each of the users. For instance, consider that Mary logs in the system by providing her `msmith` username and her password. The system needs to check if the username and password match an entry in the store. This is performed by searching for a key with `username:msmith`, which has a match, and checks if the retrieved value contains a password corresponding to the user-submitted password. Of course, with this mechanism, both the absence of an account for a user and the detection of login with an incorrect password are easily detected. Once the login step for Mary is performed, we also retrieved her user identifier. This enables us to search the key-value store for all

Keys	Values					
uid:2	username/ msmith	fname/ Mary	lname/ Smith	gender/ female	following/ [uid:1, uid:4]	blogs/ [blog:102, ..]
uid:1	username/ jdoe	fname/ Joe	lname/ Doe	gender/ male	following/ [uid:2, uid:3]	blogs/ [blog:100, blog:101,...]
...						
username:msmith	uid/ uid:2	password/ *******				
...						
cat:1	label/ Sport					
cat:2	label/ Science					
...						
blog:100	date/ 10/13/2013	content/ Today...	uid/ uid:1	cat/ cat:3		
blog:101	date/ 10/15/2013	content/ Science is...	uid/ uid:1	cat/ cat:2		

Figure 2.5 Key-value store of the blog example.

her information—that is, accessing the value of the `uid:1` key. Thus, we are able to personalized her homepage with her first and last names, list of followers, etc. Finally, to display all the blog entries that Mary follows, we retrieve the list of `uids` that she follows and which are stored under the following attribute. The application then accesses all these users, retrieve the list of their blogs, and search all of their blogs in the store. All these operations have to be programmed because no join solutions are available. Note that our solution does not enable us to know by whom someone is followed. If this feature is important in the application, we can add an extract "column" in the value that will contain a list of followers.

This simple example has emphasized some modeling limits of key-value stores. For instance, these systems are not adapted to schema requiring one-to-many and many-to-many relationships, like those shown in Figure 2.1. One should limit the use of these stores to the efficient retrieval of atomic values from given keys. Such patterns are already quite frequent on the Web and by themselves justify the presence of Memcached-based key-value stores on a very large portion of the most frequented websites.

Document stores

Document databases focus on storage and access methods optimized for documents as opposed to rows or records in an RDBMS. The data model is a set of collections of documents that contain key-value collections. In a *document store* the values can be

nested documents or lists, as well as scalar values. The nesting aspect is one important differentiator with the advanced key-value stores we just presented. The attribute names are not predefined in a global schema, but rather are dynamically defined for each document at runtime. Moreover, unlike RDBMS tuples, a wide range of values are authorized. A document stores data in tree-like structures and requires the data to be stored in a format understood by the database. In theory, this storage format can be XML, JSON, *Binary JSON* (BSON), or just about anything, as long as the database can understand the document's internal structure.

Several systems are available in that category with MongoDB certainly being the most popular and used in productions. Nevertheless, solutions such as CouchDB (Apache), **MarkLogic** (an enterprise-ready system that possesses all the features of a RDF store), and **RavenDB** are also active. MongoDB is an open-source, schema-free, document-oriented database using a collection-oriented storage. Collections are analogous to tables in a relational database. Each collection contains documents that can be nested in complex hierarchies and still support efficient query and index implementations. A document is a set of fields, each one being a key-value pair. A key is a string, and the value associated can be a basic type, a document, or an array of values. In addition, it allows efficient storage of binary data including large objects (e.g., photos and videos).

MongoDB provides support for indexes and queries for fetching data. Indexing techniques rely on B+trees and support multikey and secondary indexes. Dynamic queries are also supported with automatic use of indices, like in most RDBMSs. Each query goes through an optimization phase before being executed. MongoDB also supports MapReduce techniques for complex aggregations across documents. It provides access in many languages such as *C*, *C + +*, *C#*, *Ruby*, *Java*, etc.

Other systems such as CouchDB propose a totally different query formalism through writing views in JavaScript using a MapReduce approach. Some systems support secondary indexes. MongoDB scales reads by using replica sets and it scales writes by using sharding. Here we present a possible design for the blog example using the JSON approach of MongoDB. Although the use case could be designed in many different ways, we propose one where all the information related to an end-user is stored in a single document. Note that our solution contains a mix of subdocuments and lists. In what follows, we only present the document of the Joe Doe end-user.

```
{
"userId" : "1", "fname" : "Joe", "lname" : "Doe", "gender" : "male",
"follows" : [2,3],
"isFollowedBy" : [2],
"writes" : [ { "content" : "Today ..", "category" : "Tech"},
       { "content" : "Science is ..", "category" : "Science"} ]
}
```

We have decided to store the label characterizing a blog entry in the value of the "writes" key. This enables us not to require any join when we are storing a new blog entry for an end-user. On the other hand, this approach is far from ideal if we want to retrieve all blogs of a given category, say, all Science blogs. This is a reason why some document stores, such as MongoDB, allow us to define secondary indexes and thus to bypass this access problem. Another aspect of our proposed solution is the storage of user identifiers in the "follows" and "isFollowedBy" keys. This clearly states that some forms of joins will be needed whenever one wants to retrieve the first and last names of an end-user following another one. As said earlier, most NoSQL databases do not propose query languages that natively support joins. Therefore, such joins will be handled in the application programs and the efficiency, in terms of algorithms used, will be the responsibility of the programmer. Another approach would have been to store the last and first names in the "follows" and "isFollowedBy" keys, but that approach would not have been efficient for other joins.

To generate Mary's science page, we are providing an extract of her document:

```
{
"userId" : "2", "fname" : "Mary", "lname" : "Smith", "gender" : "fname"
"follows", : [1,4],
"isFollowedBy" : [1],
...
}
```

Once, this document is accessed, the system will scan the entries of the "follows" list, and for each of those users, access their documents and filter the Science entries of the "writes" list.

Document stores present a particular interest in use cases such as blog applications, content management systems, Web and real-time analytics, and event logging.

Column family

Column family stores or BigTable-like databases (Chang et al., 2006) are very similar on the surface to relational databases, but they are, in fact, quite different because they are oriented differently to maximize disk performance. Note that these systems are not to be confused with the column stores described in Section 2.1.5. Therefore, we make a distinction between column stores and column family stores. Recall that a column store is a kind of RDBMS with a special storage layer, one where each column is stored in a distinct file. In a column family store, the term *store* is associated to a particular data structure that is called a "column" and that is regrouped into so-called column families. This approach is motivated by the fact that generally a query does not return every column of a record.

These systems store their data such that it can be rapidly aggregated with less I/O activity. A BigTable-like database consists of multiple tables, each one containing a set of addressable rows. Each row consists of a set of values that are considered columns. Among the systems other than BigTable adopting this approach, we can identify HBase (Apache), Accumulo (Apache), early versions of Cassandra (Apache and **DataStax**), **Hypertable**, and **SimpleDB** (Amazon). In this section, we concentrate on Cassandra and HBase. It's a column family stores having a data model that is dynamic and column-oriented. Unlike a relational database, there is no need to model all of the columns required by an application upfront, as each row is not required to have the same set of columns and columns can be added with no application downtime.

A table in HBase is a distributed multidimensional map indexed by a key. The value is an object that is highly structured and the row key in a table is a string. Columns are grouped together into sets called *column families*. Column families contain multiple columns, each of which has a name, a value, and a timestamp, and is referenced by row keys. While Cassandra was, originally, proposing two kinds of column families, denoted *simple* and *super* (intuitively a column family within a column family), the latest version of Cassandra is limited to simple column families, thus matching HBase and BigTable approaches. The column families are fixed when a HBase database is created, but columns can be added to a family at any time. The index of the row keys of a given column family serves as a primary index. It's the responsibility of each participating node to maintain this index for the subset of data it manages. Additionally, because each node is aware of ranges of keys managed by the others nodes, requesting rows can be more efficient. Cassandra supports secondary indexes—that is, indexes on column values.

Cassandra allows fast lookups, and supports ordered range queries. Cassandra is recognized to be really fast for writes in a write-heavy environment. However, reads are slower than writes. This may be caused by not using B+trees and in-place updates on disk unlike all major relational databases and some NoSQL systems. In terms of data access, Cassandra used to only propose a very low-level API that is accessed through its *remote procedure call* (RPC) serialization mechanism, such as Thrift. Recently, *Cassandra Query Language* (CQL) has appeared as an alternative to the existing API. Since its release, DataStax has promoted CQL (currently in version 3 in 2014) as the preferred query solution.

Figure 2.6 proposes a possible model for the blog example. It emphasizes the creation of four different column families. The gray boxes correspond to row keys (some of which are generated automatically by the system, for example, `1234xyz` in the `Blog` column family, while others are provided by the end-user) and the white ones represent columns. For readability reasons, we do not detail all three components of each column. We again consider the creation of Mary's science page. We assume that Mary logged in correctly to the website with her username msmith. This enables us to retrieve all the users she is following via the `Follows` column family—that is, `jdoe` and `mdavis`.

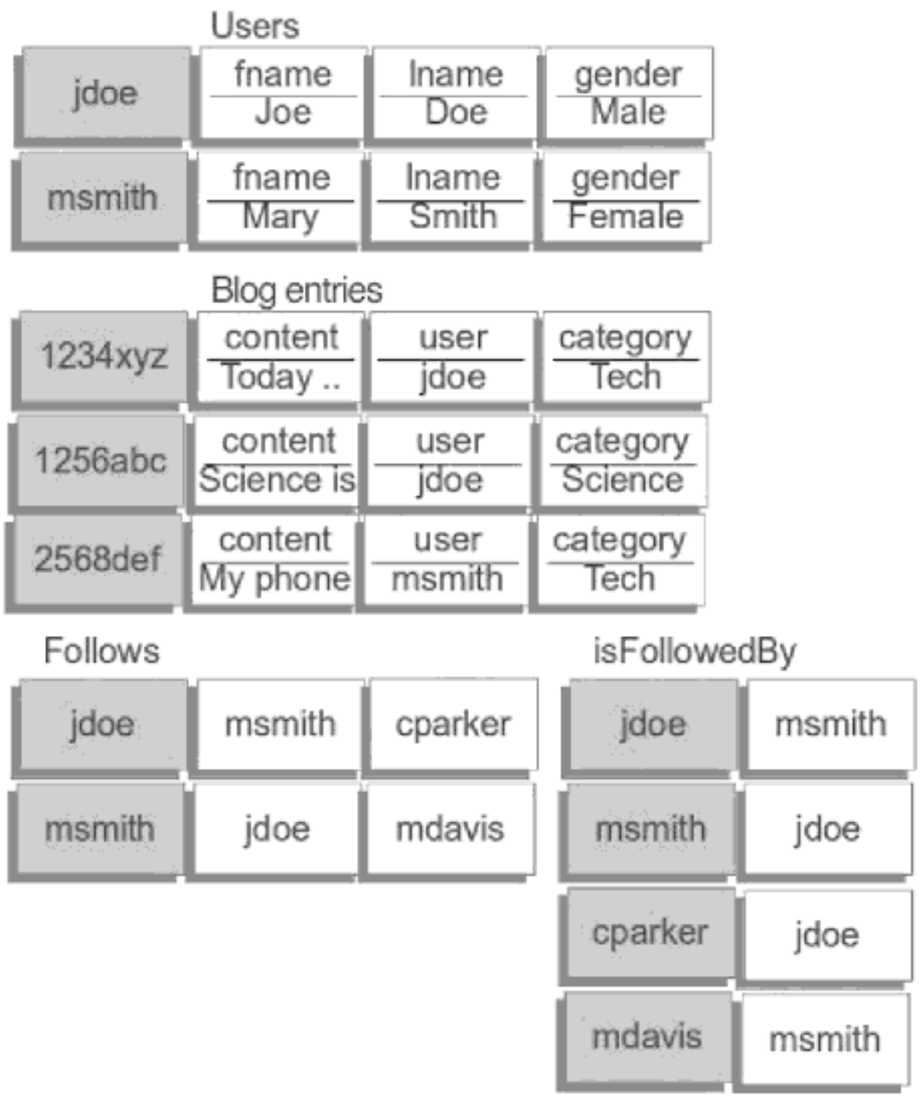

Figure 2.6 Column family model for the blog example.

Using the secondary indexing feature of HBase, we will be able to find all the blog entries with user `jdoe` (and `mdavis`) efficiently. For each of these blogs, we will only retain those with a `Science` category.

Column family stores have common ideal use cases with document stores: content management systems, blogging applications, and event logging. Their integration of a timestamp in columns suits them for handling counters and time-related information efficiently.

Graph database stores

A graph database stores a graph in the mathematical sense—that is, it deals with a set of nodes and relationships holding between these nodes. There are many available graph database store systems: **Infinite Graph**, **Titan**, **OrientDB**, **FlockDB** (Apache, originally developed at Twitter), and **Neo4J**, which is considered a leader in this category. Most systems addressing such representation use a traversing query approach by navigating along the nodes following the edges. Because edges are materialized in the graph, no computation, such as joins in RDBMS, are needed. This makes the traversing approach an efficient one. Nevertheless, to traverse a graph, one needs to find, among millions or billions of nodes, the one to start the search from, therefore indexes are required. In most systems, properties of a graph can be indexed to support fast lookups. The *Gremlin* query language is dedicated to traversing graphs and is used in many databases that implement the *Blueprints* property graph, which is a collection of interfaces, implementations, and

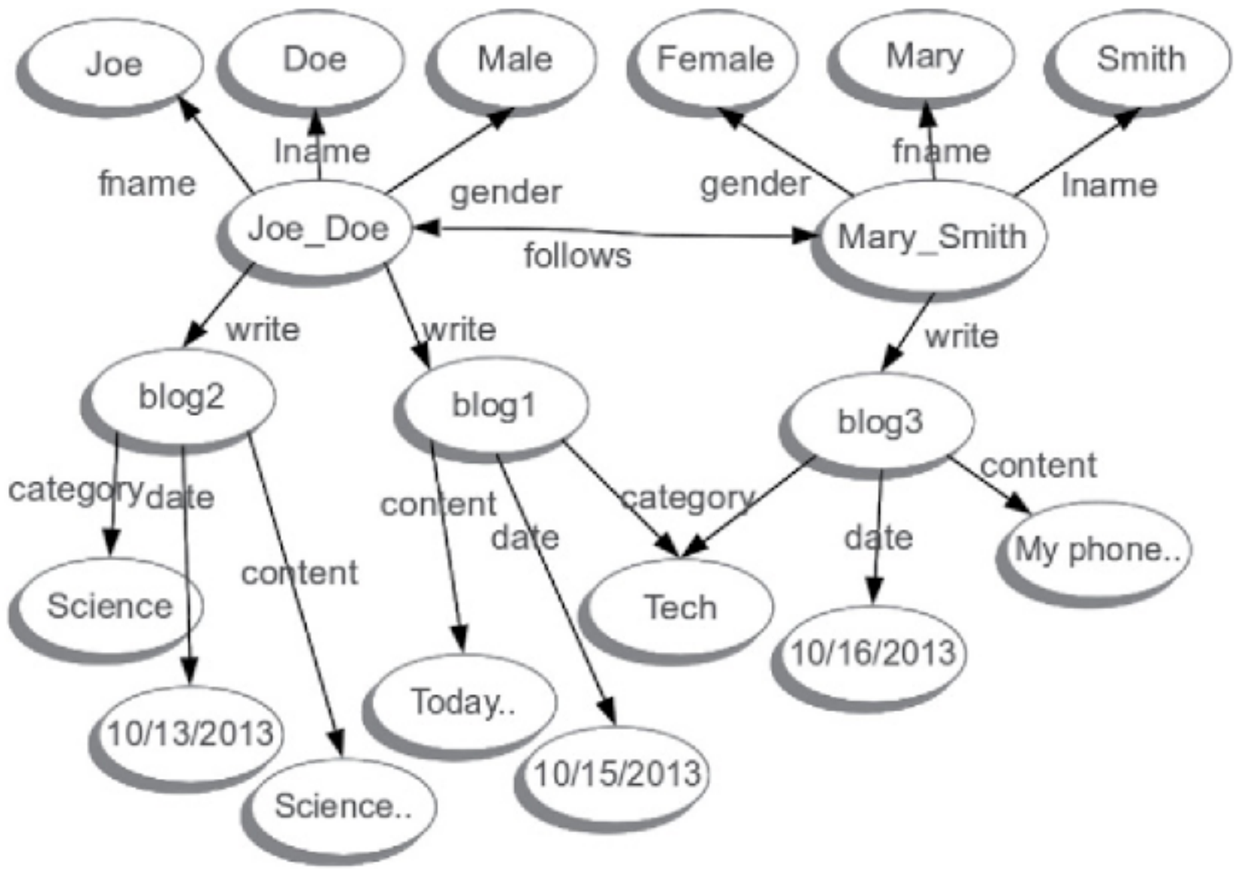

Figure 2.7 Graph for the blog example.

test suites for the property graph data model. Neo4J possesses its own query language called *Cypher*.

The transaction adopted by most graph databases conforms to the ACID approach. This is motivated by the fact that most of these systems do not distribute nodes among a cluster. Storing the whole graph over a single server enables us to address strong consistency. However, systems like Infinite Graph do propose distribution. In that situation, sharding and replication may be used as the main choice for scalability. Methods to split nodes on several machines are known but are very complex to maintain efficiently. Replication may be efficient with a master–slave approach if read operations dominate the write ones.

Figure 2.7 presents data for the blog example. All information corresponds to nodes with links between them represented as edges. To generate Mary's science page, we are using a navigation approach. That is, we will start from the `Mary_Smith` node and pursue all outgoing edges with the `follows` label. That leads the system to the `Joe_Doe` node from which the system follows the `write` outgoing edges. This points to two distinct nodes corresponding to blog entries. For each of them, the system will navigate through the `category` edge and will only retain those with a `Science` value—that is, in the figure, only `blog2` matches to our search.

Typical use cases of graph databases are social and e-commerce domains, as well as recommendation systems.

2.2.4 MapReduce

In the previous section, we emphasized on solutions that enable us to store data on cluster commodity machines. To apprehend the full potential of this approach, this also has

to come with methods to process this data efficiently—that is, to perform the processing on the servers and to limit the transfer of data between machines to its minimum. MapReduce, a programming model that has been proposed by engineers at Google in 2004 (Dean and Ghemawat, 2004), is such a framework. It's based on two operations that have existed for decades in functional programming: the `map` and `reduce` functions.

In a MapReduce framework, the user programs the `map` and `reduce` functions in a given programming language, such as Java or C++. But abstractions to program these two functions are available using an SQL-like query language, such as *PIG LATIN*. When writing these programs, one does not need to take care about the data distribution and parallelism aspects. In fact, the main contribution of MapReduce-based systems is to orchestrate the distribution and execution of these `map` and `reduce` operations on a cluster of machines over very large data sets. It also fault-tolerant, meaning that if a machine of the cluster fails during the execution of a process, its job will be given to another machine automatically. Therefore, most of the hard tasks from an end-user point of view are automatized and taken care of by the system: data partitioning, execution scheduling, handling machine failure, and managing intermachine communication.

In this framework, the `map` function processes key-value pairs and outputs an intermediate set of key-value pairs. The `reduce` function processes the key-value pairs generated by the `map` function by operating over the values of the same associated keys. The framework partitions the input data over a cluster of machines and sends the map function to each machine. This supports a parallel execution of the `map` function. After all `map` function operations have been completed, a shuffle phase is performed to transfer to all `map` outputs to the `reduce` nodes—that is, all `map` outputs with the same key are sent to the same `reduce` node. Then the `reduce` jobs are executed and produce the final output.

Over the last several years, this framework has gained popularity and resulted in several implementations. *Hadoop* is currently the most popular of them and is a product of the Apache Software Foundation and is thus open-source. Given the features of MapReduce, some may consider it a database system. For instance, one may consider that the `map` and `reduce` functions respectively correspond to a `GROUP BY` clause and an aggregation function of SQL, such as a sum or average. Stonebraker and colleagues (2010) provide a complete comparison of Hadoop and existing parallel DBMSs. The paper emphasizes that they are not competitors but rather are complementary. In fact, MapReduce can be considered as an *extract transform load* (ETL) tool rather than as a complete DBMS. The framework is considered faster at loading data than parallel databases. Nevertheless, once loaded, the data is more efficiently operated on in a parallel database system.

At the time of writing this book, MapReduce is not a single framework anymore and comes with a full stack of technologies, libraries, and tools. We have already mentioned PIG LATIN as a solution to ease writing MapReduce tasks. But systems like Hadoop also depend on other components such as a distributed file system that is especially

designed for very large data volumes, such as the *Hadoop Distributed File System* (HDFS), *Zookeeper* a centralized service providing coordination to distributed applications, i.e., maintaining naming, configuration information, and handling distributed synchronization. *Hive* as a data warehouse, and *Mahout* as a library to perform machine learning.

2.3 EVOLUTIONS OF RDBMS AND NOSQL SYSTEMS

Due to the Big Data phenomenon, data management systems are almost obliged to evolve to cope with new needs. In this section, we focus on evolutions that impact the two kinds of systems presented in this chapter and that will probably influence future solutions in the management of RDF data. Nevertheless, due to space limitations, we do not consider specific fields such as stream processing and scientific database systems.

During their long history, RDBMSs have faced several contenders, such as object databases in the 1990s and XML databases in the 2000s. Each time, they have adapted to the situation by introducing novel functionalities and retained their market dominance. These adaptations never involved deep architectural modifications and most of the main components of RDBMSs still rely on the design choices of the 1970s and 1980s.

NoSQL can be considered the latest threat for RDBMS dominance. To cope with the goals of NoSQL—that is, storing and processing large data sets on machine clusters—RDBMSs may have to rearchitecture at least some of their main components. In Stonebraker et al. (2007), the authors argue that the one-size-fits-all property of RDBMSs is over. In fact, the paper states that if they are not adapting rapidly, they could even lose their leading position in their OLTP niche market. The needed adaptations have to consider the evolution of hardware that has happened during the last few years—for example, the cost of main memory is decreasing so rapidly that servers with hundreds of gigabytes is not uncommon; SSDs are getting less expensive and are starting to replace disks in some situations; faster CPUs and networks are arising; computing with *graphics processing units* (GPUs) is easier through APIs and programming languages; and dominance of shared-nothing architecture is being confirmed. The main components responsible for the performance bottleneck of current RDBMS systems have been identified in Harizopoulos et al. (2008) and are related to ACID transactions (i.e., logging, locking, and latching), as well as buffer management operations. In the new hardware era, all these components could be implemented to reside in the main memory. Such systems recently started to appear and are sometimes denoted as *NewSQL*. They share the high-performance and scalability characteristics with NoSQL and at the same time retain full ACID properties and the SQL language. Available systems are **VoltDB, Clustrix, NuoDB, MemSQL, NimbusDB, Akkiban,** and **SQLFire**.

With the accession to new markets, NoSQL systems are also facing the needs of new clients. Some of their requisites concern the integration of new features: declarative

query languages, solutions for defining schemata, the ability to select different consistency characteristics (e.g., strong or eventual), and integrating integrity constraints to enhance data quality and business intelligence processing. The most successful NoSQL stores are all going this way. For instance, an important work has been conducted by the team at DataStax (the main contributor on the Cassandra database) on designing a declarative, SQL-influenced query language, namely *CQL*. Note that this language does not just provide a *Data Manipulation Language* (DML) but also a *Data Definition Language* (DDL) that enables us to create/drop keyspaces (i.e., databases), tables, and indexes. Other popular systems such as CouchDB are also proposing an SQL-like solution, denoted *UnQL*. MongoDB and Neo4J, potential leaders in document and graph stores, have proposed query languages for quite a while now. Considering the consistency aspect, systems like Cassandra and MongoDB already propose configuration tools that enable us to select a particular approach for a given database. In fact, most of these desired features are already present in RDBMSs and one can ask what NoSQL stores will look like if they are all added.

As a final direction on the evolution of database management systems, it's always interesting to look for innovations provided by major Web companies. During the last couple of years, many consider that the most innovative systems have been designed at Google. Thus, it's amusing to witness that after leading the NoSQL movement, Google, through its **Spanner** system, is going back a more conventional relational model. The Spanner system (Corbett et al., 2013) has been presented at the 2012 OSDI conference. It's an attempt to implement an ACID- and SQL-compliant relational database over a global scale and geographically distributed cluster of machines. An important contribution of the paper is to present the TrueTime API: the system's solution to support externally consistent distributed transactions at a global scale. Therefore, all data are versioned using the timestamp of its commit. The design of this database mainly responds to requests from Google employees who needed a solution that enables easier schema evolution and a strong consistency in the presence of wide-area replication than available solutions, such as BigTable and Megastore.

2.4 SUMMARY

- The database management system market is very active due to the emergence of Big Data.
- For almost 30 years, this market has been dominated by systems based on the relational model. Due to their long history, they are considered the most mature and robust approaches. They are constituted of many features, such as complete query language (SQL) and optimization facilities, indexes for speed-up queries, processing of transactions, and concurrency support.

- NoSQL is an emerging family of data storage solutions that is gaining traction in many organizations and companies dealing with a data deluge. Existing systems are quite diverse and do not rely on standards, but leaders are active and very responsive to the expectations of their clients.
- Innovative system approaches are appearing and some of their objectives are to address high throughput with data availability and distribution. Some of these systems are based on the relational model and tend to go through a rearchitecturing process to gain some of these properties.
- These systems serve as the storage backend of several RDF systems we will study in Part 2 of this book.

CHAPTER THREE

RDF and the Semantic Web Stack

In Chapter 2, we presented the main characteristics of database management systems available on the market. Recall that such a system corresponds to a complete software used to define, create, manage, query, and update some set of data. Of course, an RDF database management system (we use the term *RDF store* interchangeably) handles RDF data. To motivate and understand the details of these systems, we begin this chapter with a presentation of an important part of the Semantic Web stack. This stack regroups some general notions that have emerged in the context of Artificial Intelligence (AI), such as Knowledge Representation (KR), and that support the description of reasoning services over schemata and facts expressed in this extension of the current Web. Note that this inference aspect is a main peculiarity of the systems we are covering in this book—that is, standard database management systems do not natively support inference features.

3.1 SEMANTIC WEB

From its inception at the beginning of the 1990s, the Web has evolved. From a static version, mainly supported by HTML and CSS, it evolved into a more dynamic and write-intensive Web, qualified as 2.0. Nevertheless, both of these versions can be qualified as syntactic because the languages they are built upon only convey information toward rendering in web browsers. The Web evolution is far from being finished, and some obviously make reference to the next extension as Web 3.0, to include the addition of semantics-supported features. In this book, we prefer the term *Semantic Web* because it refers to a computer science field that emerged in the early 2000s (Berners-Lee et al., 2001). In contrast, the Semantic Web provides meaning to the information contained in Web documents.

This description and its interpretation are supported by a stack of technologies that have been designed and recommended by the World Wide Web Consortium (W3C) since 1999. This stack is frequently referred to as the *Semantic Web cake*, and one of its generally adopted current versions is presented in Figure 3.1. The layer organization of this technology stack implies that the elements described at a given layer are compliant with the standards defined at the lower layers. In this book, we are only concerned with the five bottom layers.

The lowest layer of this stack provides a global identification solution for the resources found on the Web. In Figure 3.1 they are referred to as *uniform/internationalized*

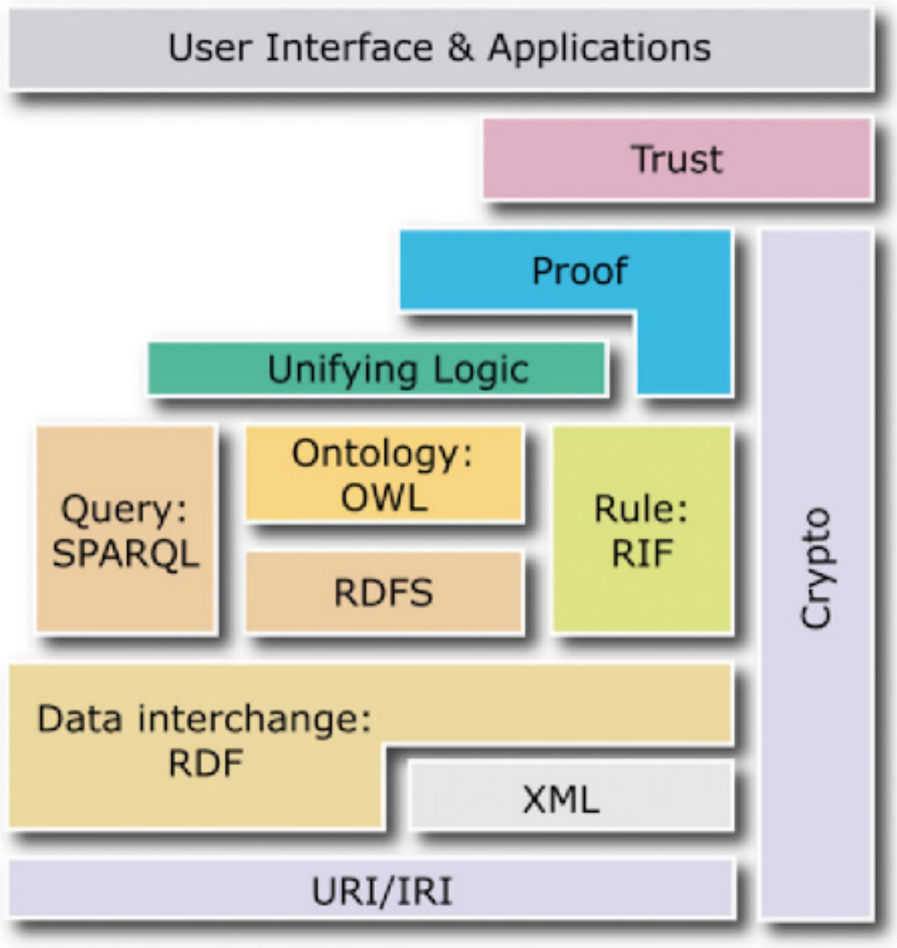

Figure 3.1 Semantic Web cake.

resource identifiers (URIs/IRIs). URIs and IRIs are now used interchangeably, so we will note in this book when a distinction is needed.

The second layer supports the definition of a syntax that is based on XML, a meta-language based on the notion of tags. Note that XML comes with some associated technologies that enable the definition of schemata for document instances, such as *document type definition* (DTD) or XML Schema, and a naming convention (i.e., namespaces) is supported to disambiguate the use of overlapping tags coming from different languages.

In the third layer, we really begin our journey in the Semantic Web with the RDF language. In Figure 3.1, it's qualified as data interchange because its objective is to enable the exchange of facts among agents. Note that RDF relies on both XML and the URI layer. This implies that several syntaxes are available for RDF: one based on XML, denoted RDF/XML, and other ones that we will present in this chapter.

The fourth layer provides a reply to the limitation that RDF represents facts only. With *RDF Schema* (RDFS), a first solution to define metadata on some elements of an RDF document instance is proposed. The features of this language do not support the specification of very expressive vocabularies but this aspect is taken care of in the next layer.

The fifth layer is composed of the *Web Ontology Language* (OWL). OWL enables us to define more expressive ontologies than RDFS does, but it comes at an increased computational complexity of reasoning.

Finally, spreading across layers four and five are SPARQL and the *Resource Interchange Format* (RIF). SPARQL is most widely used as a query language over RDF data. Therefore, it can be considered as the SQL for RDF stores. SPARQL is also a protocol that we will describe in Section 7 in the context of query federation. RIF supports an inference

form that is based on the processing of rules (i.e., *prolog* and *datalog* like), and is usually not considered in RDF database systems. Therefore, we are not covering it in this book.

The remaining layers are not considered in RDF stores and are out of the scope of this book.

One term we need to clarify before delving into the standards of the Semantic Web is *ontology*. Ontology refers to the definition of a domain in terms of its concepts and their relationships. For instance, we can define an ontology for a bedroom where we would find concepts such as `bed`, `mattress`, and `pillow`. Some relationships among these concepts are `on` (to state that a pillow is on a mattress) or `composedOf` (to specify that a bed is composed of a mattress). Of course, ontologies can be defined for domains far more complex than a bedroom, such as medicine or biology. And some of the languages used to define these ontologies have to be more or less expressive. Expressivity characterizes the precision with which the concepts and relationships can be defined. For example, an RDBMS schema, expressed in an entity relationship (ER) diagram, can be considered as an ontology with a very low degree of expressiveness—that is, it's unable to represent constructors such as negation, disjunction, and quantifications. In our Semantic Web stack, RDFS is considered the least expressive language and, as will we see later on, OWL proposes several variants (called profiles) that present different degrees of expressiveness. An interesting aspect of the Semantic Web is that many quality ontologies are already available in these formats in fields as diverse as science, culture and economics to name a few.

We will highlight that it's possible to make inferences—that is, deduce information not explicitly stored in the source—over an ontology alone. But in general, it's more interesting to reason over data that is associated to a given ontology. This data is represented in RDF documents. In an AI terminology, the association of a schema and some data is referred as a Knowledge Base (KB). Additionally, in the context of *Description Logic* (DL), which is the logic underlying some of the OWL languages, one makes reference to an ontology as a terminological box (Tbox) and to a set of facts as an assertional box (Abox).

3.2 RDF

The first aim of the Resource Description Framework (RDF) was to provide a metadata data model to the Web. Nowadays, it emerges as a data model for the Web of Data and the Semantic Web providing a logical organization defined in terms of some data structures to support the representation, access, constraints, and relationships of objects of interest in a given application domain.

The underlying data model is quite simple, but, as we will see afterwards, nevertheless expressive. The basic unit of information is given as a triple `(s,p,o)` composed of a *subject* (s), a *predicate* (p), and an *object* (o). Each triple represents a fact on a thing being described

Table 3.1 RDF Triples Corresponding to `User` Table in Chapter 2

Subject	Predicate	Object
JD	firstName	Joe
JD	lastName	Doe
JD	gender	Male
MS	firstName	Mary
MS	lastName	Smith
MS	gender	Female

(i.e., the subject, which is also referred to as the resource), on a specific property (i.e., the predicate), and with a given value (i.e., the object). Table 3.1 provides a simplified overview of the RDF triples corresponding to the `User` table presented in Chapter 2.

A more compact way of representing a set of triples is using a directed, labeled graph representation. In such an RDF graph, a triple is represented by an edge between two nodes. The source node corresponds to the subject, the destination node to the object, and the edge to the predicate. As illustrated in Figure 3.2, any subject or object is represented by a single node and all its related information corresponds to a subgraph.

As previously mentioned, one of the main characteristic of the Web of Data movement is data distribution over multiple sources. This distribution induces a merging task whenever one wants to retrieve all information on a given set of resources. This merging process is facilitated by the representation of the unit of information in RDF. Indeed, in RDF, because any information is provided as a triple, merging data sources corresponds to merging sets of triples. The main problem that can arise is the identification of common resources among multiple sources. This problem is solved in RDF by the use of URIs or its generalization IRIs, which support encoding in Unicode rather than in ASCII.

URIs represent common global identifiers for resources across the Web. The syntax and format of URIs are very similar to the well-known *uniform resource locators* (URLs; e.g., `http://example.com/Blog#JD`). In fact, URLs are just special cases of URIs. Another form of URIs is a *uniform resource name* (URN), which identifies something that is not associated to a Web resource but on which people on the Web want to write about, such as a book or a tree.

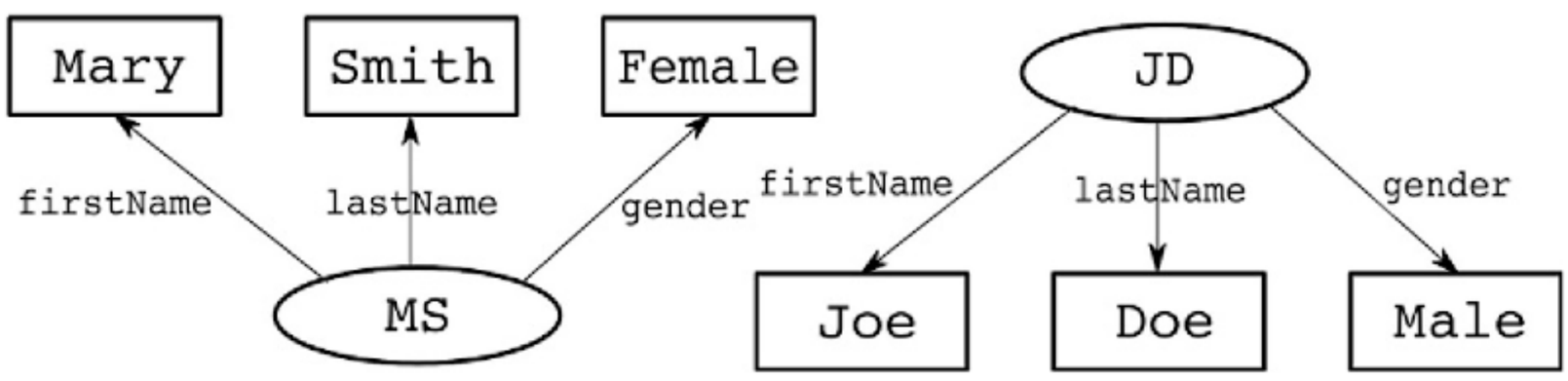

Figure 3.2 RDF graph representation of Table 3.1 triples.

Any two data sources on the Web can refer to the same resource by using the same URI. One of the benefits of the common form of URIs and URLs is that URIs may be *dereferenced*—that is, each part of the URI is a locator for the resource (e.g., server name, directories, filenames, etc.) where potential extra information can be found. For example, the URI `http://www.w3.org/standards/techs/rdf#w3c_all` specifies a protocol (i.e., http), a server name (i.e., www.w3.org), some directories on that server (i.e., standards/techs), a document (i.e., rdf), and a location in that document (i.e., w3c_all). The de-referencable quality of URIs is quite important in the context of the Semantic Web because it supports both the identification in a global Web infrastructure, and as an effect enables data integration at scale—that is, the ability to consider identifiers as links to other resources. Nevertheless, in RDF, the main objective of URIs is to provide a unique name for a resource or a property.

URIs will naturally support the related problem of distinguishing different senses of a word like "apple." The well-known computer brand and the delicious fruit will be differentiated by referring to different URIs, such as `<http://www.apple.com/public/RDF/#Apple>` or `<http://www.fruits.com/fall/#Apple>`. In these URI references, we can observe a fragment identifier that is preceded by the # symbol. Whatever appears before that symbol identifies a resource and the fragment identifier identifies a part of that resource. For ease of notation, in RDF, one may define a *prefix* to represent a namespace, such as `Fruits:Apple` where `Fruits` represents the namespace `<http://www.fruits.com/fall/#>`. Prefixes are not global identifiers and should be declared with their corresponding namespace (as similarly done with qualified names in XML).

While the prior aim of the Web was to provide access to information for humans via web pages, the aim of the Web of Data movement is to provide knowledge processing by nonhuman agents like software. For this purpose, one cannot rely only on strings and should be able to add structure and knowledge to the resource of interest. Thus, most of the information in RDF data sets will consist of URIs, for subjects, predicates, and objects. More precisely, non-URI values, namely *literals*, will only be supported for objects. A literal value may be accompanied with a *data type* (e.g., `xsd:integer`, `xsd:decimal`, `xsd:boolean`, `xsd:float`[1]) and will be referred to as a *typed literal*, or *plain literal* otherwise. Finally, while most of the resources are given an identifier (i.e., URI), RDF provides the possibility for resources to be anonymous. Such resources are represented as *blank nodes* (also called *bnodes*), which do not have a permanent identity. Blank nodes are defined with a specific namespace denoted by a _. The main purpose of such nodes is either to state the existence of some resources that we cannot name, or to group together some knowledge. For example, one would like to state that an anonymous follower added a comment to Joe Doe's blog about science.

[1]The `xsd` prefix corresponds here to the XML Schema namespace `http://www.w3.org/2001/XMLSchema#`.

Table 3.2 Prefixes `ex:` and `blog:` Respectively Correspond to `http://example.com/terms#` and `http://example.com/Blog#`

Subject	Predicate	Object
blog:JD	ex:firstName	"Joe"
blog:JD	ex:lastName	"Doe"
blog:JD	**rdf:type**	**ex:Man**
blog:MS	ex:firstName	"Mary"
blog:MS	ex:lastName	"Smith"
blog:MS	**rdf:type**	**ex:Woman**

The RDF standard provides a specific namespace often referred to with the `rdf` prefix name, such as <`http://www.w3.org/1999/02/22-rdf-syntax-ns#`>. Among the standard identifiers defined in this namespace, `rdf:type` is a predicate allowing elementary typing ability in RDF. That mechanism is quite useful to state the gender of Joe Doe and Mary Smith with respectively the `Man` and `Woman` concepts, as shown in Table 3.2.

The object of such triples will be understood as a type. As we will see, this mechanism will help us infer nonexplicit knowledge, such as Joe Doe and Mary Smith are persons, without the need to state it explicitly in the data but relying on the knowledge that `ex:Man` and `ex:Woman` are subtypes of `ex:Person`.

The identifier `rdf:Property` can be combined with `rdf:type` to indicate that a given resource may be used as a predicate rather than a subject or an object. In other words, it allows to introduce new predicates. In the blog running example context, it would be suitable to state that Joe Doe is following Mary Smith directly using a predicate named `blog:isFollowing`. The corresponding RDF declaration is given in Table 3.3.

Let's assume now that we want to express that Joe Doe is following Mary Smith since last January. Our main issue is that this extra information is specific to the previous statement. One would be tempted to add a `started` property to the `isFollowing` edge between the `blog:MS` and `blog:JD` nodes of our graph but this is not allowed in RDF graphs. One solution is to treat a statement as a resource. For this purpose, RDF provides

Table 3.3 Prefixes `ex:` and `blog:` Respectively Correspond to `http://example.com/terms#` and `http://example.com/Blog#`

Subject	Predicate	Object
blog:JD	ex:firstName	"Joe"
blog:JD	ex:lastName	"Doe"
blog:JD	rdf:type	ex:Man
blog:JD	**blog:isFollowing**	**blog:MS**
blog:MS	ex:firstName	"Mary"
blog:MS	ex:lastName	"Smith"
blog:MS	rdf:type	ex:Woman
blog:isFollowing	**rdf:type**	**rdf:Property**

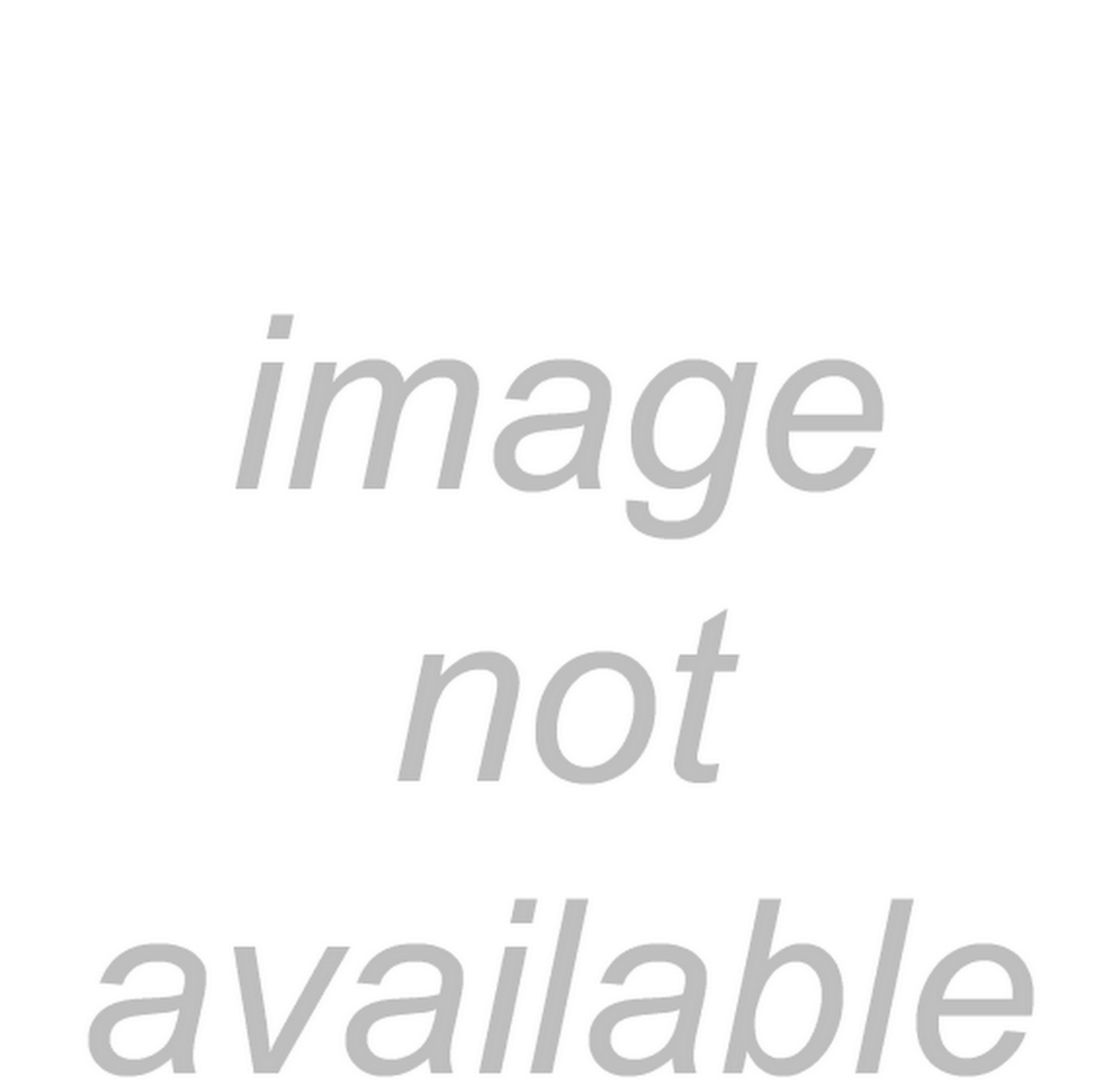
image
not
available

for publishing data in RDF on the Web. In the following, we will stick to the running blog context and consider the data presented in Chapter 2 for each format.

3.2.1 RDF/XML

The W3C recommendation for textual representation is using an XML serialization of RDF, a so-called *RDF/XML* (`http://www.w3.org/TR/rdf-syntax-grammar/`). Roughly, any resource can be defined as an `rdf:Description` XML element with an `rdf:about` attribute that states its URI. Multiple characteristics concerning a given subject are given as child elements of the corresponding XML element. To refer to a given URI, one should use the `rdf:resource` attribute. In the example shown in the following code, four statements about Joe Doe (<`http://example.com/Blog#JD`>) are given.

RDF/XML allows a more concise expression of the `rdf:type` predicate by replacing the `rdf:Description` element by an element directly named with the namespaced element corresponding to the RDF URI of the `rdf:resource` attribute.

```
<?xml version="1.0"?>
<rdf:RDF xmlns:rdf="http://www.w3.org/1999/02/22 -rdf-syntax-ns#"
         xmlns:blog="http://example.com/Blog#"
         xmlns:ex="http://example.com/terms#">
  <rdf:Description rdf:about="htt p://example.com/Blog#JD">
      <rdf:type rdf:resource="http://example.com/Blog#User"/>
      <blog:hasGender rdf:resource="http://example.com/terms#Male"/>
      <ex:firstName>Joe</ex:firstName>
      <ex:lastName>Doe</ex:lastName>
  </rdf:Description>
</rdf:RDF>
```

Moreover, any predicate of which the value is a literal can be expressed as an XML attribute of the corresponding subject element. The previous example code can thus be rephrased as follows.

```
<?xml version="1.0"?>
<rdf:RDF xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
         xmlns:blog="http://example.com/Blog#"
         xmlns:ex="http://example.com/terms#">
  <blog:User rdf:about="http://example.com/Blog#JD"
ex:firstName="Joe" ex:lastName="Doe">
      <blog:gender rdf:resource="http://example.com/terms#Male"/>
  </blog:User>
</rdf:RDF>
```

Finally, RDF/XML provides a vocabulary to manage groups of resources or literals. Groups are either defined with RDF *containers* or RDF *collections*. The main difference is

that RDF collections are groups containing only the specified members. Three kinds of RDF containers are defined: an `rdf:Bag` corresponds to an unordered group allowing duplicates, an `rdf:Seq` corresponds to an ordered `rdf:Bag`, and an `rdf:Alt` corresponds to alternative choices. Each member of a group is defined by an `rdf:li` element. A full illustration based on the data presented in Chapter 2 is provided in the following code block. In this example, note the presence the `rdf:ID` attribute, which enables us to define a named node, therefore it's quite similar to `rdf:about`. Nevertheless, the latter is usually preferred when one refers to a resource with a globally well-known identifier or location.

```
<?xml version="1.0"?>
<rdf:RDF xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
xmlns:xsd="http://www.w3.org/2001/XMLSchema#"
xmlns:blog="http://example.com/Blog#"
xmlns:cat="http://example.com/Cat#"
xmlns:ex="http://example.com/terms#"
xml:base="http://blogs.com/">
  <blog:User rdf:about="JD" ex:firstName="Joe" ex:lastName="Doe">
    <blog:gender rdf:resource="ex:Male"/>
    <blog:isFollowing>
      <rdf:Bag>
        <rdf:li rdf:resource="MS"/>
        <rdf:li rdf:resource="#blogger3"/>
      </rdf:Bag>
    </blog:isFollowing>
  </blog:User>
  <blog:User rdf:about="MS" ex:firstName="Mary" ex:lastName="Smith">
    <blog:gender rdf:resource="ex:Female"/>
    <blog:isFollowing>
      <rdf:Bag>
        <rdf:li rdf:resource="JD"/>
        <rdf:li rdf:resource="#blogger4"/>
      </rdf:Bag>
    </blog:isFollowing>
  </blog:User>
  <blog:User rdf:ID="blogger3"/>
  <blog:User rdf:ID="blogger4"/>
  <cat:Category rdf:about="cat:Sport" cat:title="Sport"/>
  <cat:Category rdf:about="cat:Science" cat:title="Science"/>
  <cat:Category rdf:about="cat:Tech" cat:title="Technology"/>
  <blog:Blog rdf:about="blog1" blog:content="Today…">
    <ex:writtenOn rdf:datatype="xsd:date">10/13/2013</ex:writtenOn>
    <blog:owner rdf:resource="JD"/>
    <blog:category rdf:resource="cat:Tech"/>
  </blog:Blog>
  <blog:Blog rdf:about="blog2" blog:content="Science is">
    <ex:writtenOn rdf:datatype="xsd:date">10/15/2013</ex:writtenOn>
    <blog:owner rdf:resource="JD"/>
    <blog:category rdf:resource="cat:Science"/>
  </blog:Blog>
  <blog:Blog rdf:about="blog3" blog:content="My phone">
    <ex:writtenOn rdf:datatype="xsd:date">10/16/2013</ex:writtenOn>
    <blog:owner rdf:resource="MS"/>
  <blog:category rdf:resource="cat:Tech"/>
 </blog:Blog>
</rdf:RDF>
```

In the following sections, we present three additional serializations for RDF documents that do not suffer the verbosity of the XML syntax. They are frequently used for readability reasons but are also handled by most of the RDF tools we will see later in this chapter e.g., parser, serializer.

3.2.2 N-triples

N-triples is the simplest form of textual representation of RDF data but it's also the most difficult to use in a print version because it does not allow URI abbreviation. The triples are given in subject, predicate, and object order as three complete URIs separated by spaces and encompassed by angle brackets (< and >). Each statement is given on a single line ended by a period (.). The following code illustrates the statements regarding the category `Science`.

```
<http://example.com/Cat#Science    http://www.w3.org/1999/02/22-rdf-
syntax-ns#type http://example.com/Cat#Category>.
<http://example.com/Cat#Science   http://example.com/Cat#title   "Science">.
```

3.2.3 N3

Notation 3, or *N3* for short, was proposed by Tim Berners-Lee as a compromise between the simplicity of N-triples with the expressiveness of RDF/XML. The notation is very similar to N-triples. The main syntactic differences are:

- The surrounding brackets have been removed.
- URIs can be abbreviated by prefix names.
- There is no restriction on the number of separating spaces.
- Shortcut syntax for statements sharing a predicate and/or an object have been introduced:
  - `"subject stuff ; morestuff ."` stands for `"subject stuff. subject morestuff ."`
  - `"subject predicate stuff, morestuff ."` stands for `"subject predicate stuff. subject predicate morestuff ."`
  - Blank nodes can be replaced by declared existential variables (see `http://www.w3.org/DesignIssues/Notation3` for more details).

N3 has other features unrelated to the serialization of RDF (e.g., RDF-based rules) that will not be presented in this book because they have been superseded by Turtle and SPARQL. A full illustration is given in the following code.

```
@prefix blog: <http://example.com/Blog#> .
@prefix cat: <http://example.com/Cat#> .
@prefix ex: <http://example.com/terms#> .
@prefix rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#> .
@prefix rdfs: <http://www.w3.org/2000/01/rdf-schema#> .
@prefix xml: <http://www.w3.org/XML/1998/namespace> .
@prefix xsd: <http://www.w3.org/2001/XMLSchema#> .
<cat:Sport> a cat:Category ;
    cat:title "Sport" .
<cat:Science> a cat:Category ;
    cat:title "Science" .
<cat:Tech> a cat:Category ;
    cat:title "Technology" .
<http://blogs.com/blog1> a blog:Blog ;
    blog:category <cat:Tech> ;
    blog:content "Today…" ;
    blog:owner <http://blogs.com/JD> ;
    ex:writtenOn "10/13/2013"^^<xsd:date> .
<http://blogs.com/blog2> a blog:Blog ;
    blog:category <cat:Science> ;
    blog:content "Science is" ;
    blog:owner <http://blogs.com/JD> ;
    ex:writtenOn "10/15/2013"^^<xsd:date> .
<http://blogs.com/blog3> a blog:Blog ;
    blog:category <cat:Tech> ;
    blog:content "My phone" ;
    blog:owner <http://blogs.com/MS> ;
    ex:writtenOn "10/16/2013"^^<xsd:date> .
<http://blogs.com/#blogger3> a blog:User .
<http://blogs.com/#blogger4> a blog:User .
<http://blogs.com/MS> a blog:User ;
    blog:gender <ex:Female> ;
    blog:isFollowing [ a rdf:Bag ;
            rdf:_1 <http://blogs.com/JD> ;
            rdf:_2 <http://blogs.com/#blogger4> ] ;
    ex:firstName "Mary" ;
    ex:lastName "Smith" .
<http://blogs.com/JD> a blog:User ;
    blog:gender <ex:Male> ;
    blog:isFollowing [ a rdf:Bag ;
            rdf:_1 <http://blogs.com/MS> ;
            rdf:_2 <http://blogs.com/#blogger3> ] ;
    ex:firstName "Joe" ;
    ex:lastName "Doe" .
```

3.2.4 Turtle

Turtle (the Terse RDF Triple Language) is a simplified version of N3 that is becoming increasingly popular and is now a W3C recommendation. In Turtle, as a predicate, `a` is equivalent to the complete URI corresponding to `rdf:type`. RDF blank nodes can either be explicitly expressed as `_:id`, where `id` is the identifier of the blank node, or anonymously by putting all the triples that are subjects between square brackets (`[` and `]`).

Finally, Turtle provides a collection structure for lists as a sequence of elements separated by spaces and encompassed by parenthesis ((and)). The following code illustrates the statements regarding Mary Smith.

```
@prefix rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#> .
@prefix cat: <http://example.com/Cat#> .
@prefix blog: <http://example.com/Blog#> .
@prefix ex: <http://example.com/terms#> .
…
blog:MS     a                  blog:User ;
            ex:firstName         "Mary" ;
            ex:lastName          "Smith" ;
            blog:hasGender     ex:Female ;
            blog:isFollowing   ( blog:JD  _:blogger4 ) .
_:blogger4  a                  blog:User .
…
```

3.2.5 Other serializations

Because a huge amount of data is already reachable on the Web, through HTML pages, some web page authors are interested in embedding information for agents (not necessarily human) to read and use. This is the aim of *microformats* (http://microformats.org/), which do not have effect on how the information is displayed through the browser but allow us to embed RDF data in a web page. This is done by the use of specific attributes of HTML/XHTML elements. A similar way to write RDF data in a web page is to use *RDF in attributes* (RDFa; http://www.w3.org/TR/rdfa-core/), which was proposed by W3C. This approach follows the style of microformats by minimizing repetition in a document—that is, the information is written once and both support extraction of metadata by agents and rendering in the web browser. RDFa defines existing and new HTML attributes where subjects, predicates, and objects should be stored and retrieved. RDFa has been adopted by search engines such as **Google**, **Bing**, **Yandex**, and **Yahoo!** (through the http://schema.org/ initiative) to enable data extraction by agents. Microformats and schema.org provide vocabularies that cover a limited amount of domains (events, contact information, review, product). Moreover, RDFa is intrinsically not limited to any domain and can related to any well-formed compliant vocabulary.

3.3 SPARQL

We just presented RDF as the data model of the Semantic Web. In general, a data model comes equipped with a language to support queries over some data set. RDF follows that approach and *SPARQL*, which stands for SPARQL Protocol and RDF Query Language, corresponds to a group of specifications providing languages and protocols to

query and manipulate RDF graphs. In the context of this chapter, we will concentrate on the query language aspect. To increase its adoption rate, the language borrowed part of its syntax from the popular and widely adopted Structured Query Language (SQL) of RDBMS. The standard `SELECT <result template> FROM <data set definition> WHERE <query pattern>` pattern of SQL is adapted in the context of SPARQL.

The peculiarity of SPARQL is to handle RDF graphs and to support a navigation-based approach to data retrieval. The idea is to start from a given node and move from node to node following certain edges until the goal node is reached. This approach is quite different from the SQL execution, which is based on joins between tables. To perform such operations, variables must be supported in the RDF triples patterns allowed in the query language, in particular in the `WHERE` clause. A *variable* is identified by a question mark (`?`) followed by the name of the variable. A variable will, on one hand, allow us to store any information resulting from the graph matching problem and, on the other hand, could be combined afterwards in other triples. As in Turtle, the use of prefixes declaration allows us to lighten the overall textual representation of the data. In fact, the triples pattern of the `WHERE` clause is based on Turtle. Following is a first simple query that retrieves the first names of users with a `Doe` last name. In this query, we have two variables, `?user` and `?name`, and only the latter is proposed in the result set since it is the only one presenting the SELECT clause (such a variable is qualified as distinguished).

```
PREFIX blog: <http://example.com/Blog#>
PREFIX ex:  <http://example.com/terms#>
SELECT ?name
WHERE {
  ?user      ex:lastName       "Doe" ;
             ex:firstName      ?name ;
}
```

While the `WHERE` part (i.e., the query pattern) of the query allows us to restrict the RDF subgraph of interest that should match the conditions, the `SELECT` part (i.e., the result template) allows us to moreover select the part of the result we want to exhibit. The clauses in the `WHERE` part are defined as the standard RDF triples pattern including variables. Variables not present in the `SELECT` part are used to connect up different triples patterns (similar to SQL's `JOIN`) and are qualified as non-distinguished. The following code presents a query requiring a join. Intuitively, it asks for the first names of persons with a `Doe` last name and who are following someone with a `Mary` first name. The join is supported by the co-occurrence of the `?user2` variable in both the third and fourth line of the query, respectively, at the object and subject positions.

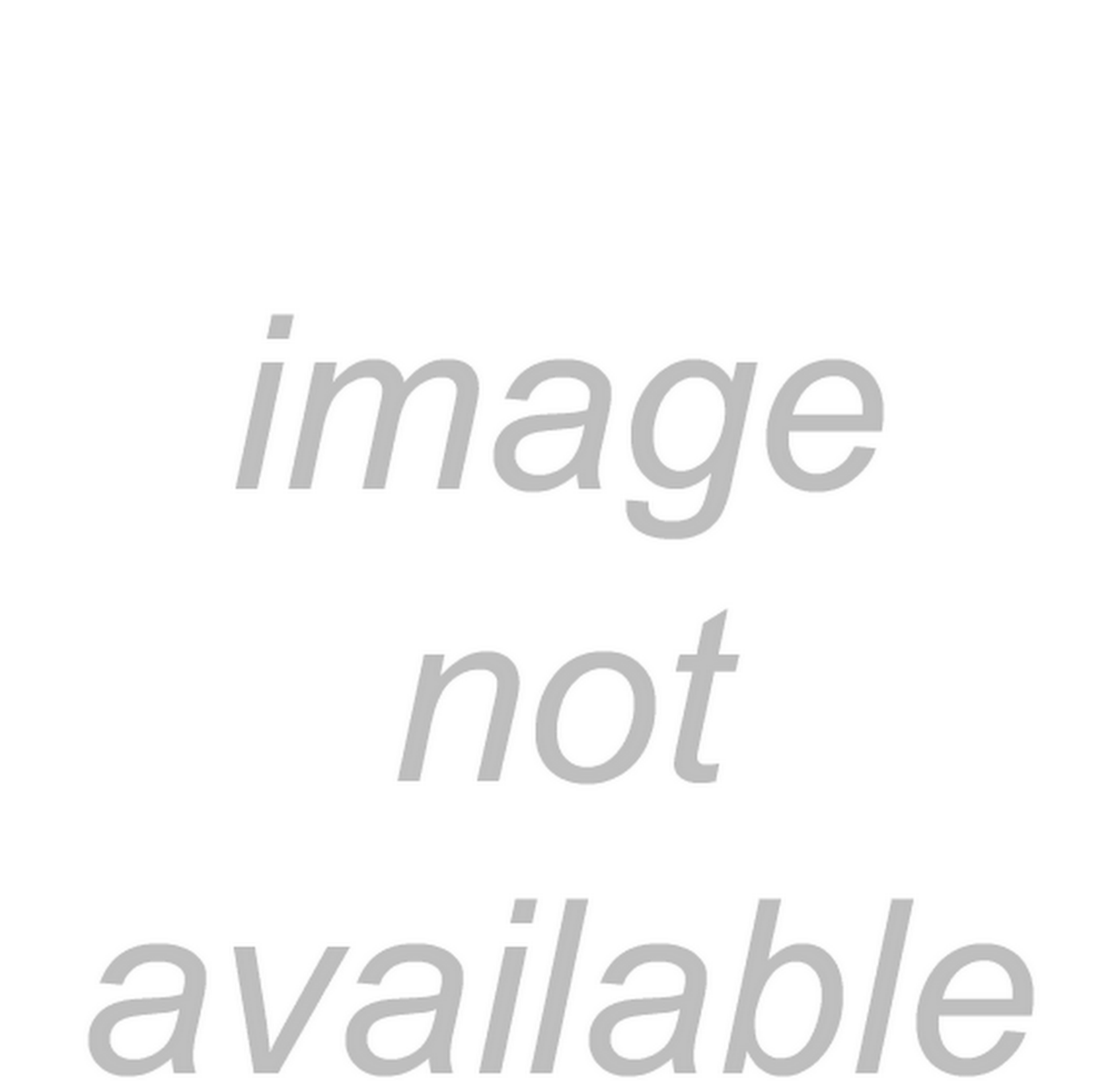
image
not
available