Cloud-Based RDF Data Management
Synthesis Lectures on Data Management

Editor
H.V. Jagadish, University of Michigan

Founding Editor
M. Tamer Özsü, University of Waterloo

Synthesis Lectures on Data Management is edited by H.V. Jagadish of the University of Michigan. The series publishes 80–150 page publications on topics pertaining to data management. Topics include query languages, database system architectures, transaction management, data warehousing, XML and databases, data stream systems, wide scale data distribution, multimedia data management, data mining, and related subjects.

Cloud-Based RDF Data Management
Zoi Kaoudi, Ioana Manolescu, and Stamatis Zampetakis
2020

Community Search over Big Graphs
Xin Huang, Laks V.S. Lakshmanan, and Jianliang Xu
2019

On Transactional Concurrency Control
Goetz Graefe
2019

Data-Intensive Workflow Management: For Clouds and Data-Intensive and Scalable Computing Environments
Daniel C.M. de Oliveira, Ji Liu, and Esther Pacitti
2019

Answering Queries Using Views, Second Edition
Foto Afrati and Rada Chirkova
2019

Transaction Processing on Modern Hardware
Mohammad Sadoghi and Spyros Blanas
2019
**Data Management in Machine Learning Systems**  
Matthias Boehm, Arun Kumar, and Jun Yang  
2019

**Non-Volatile Memory Database Management Systems**  
Joy Arulraj and Andrew Pavlo  
2019

**Scalable Processing of Spatial-Key Word Queries**  
Ahmed R. Mahmood and Walid G. Aref  
2019

**Data Exploration Using Example-Based Methods**  
Matteo Lissandrini, Davide Mottin, Themis Palpanas, and Yannis Velegrakis  
2018

**Data Profiling**  
Ziawasch Abedjan, Lukasz Golab, Felix Naumann, and Thorsten Papenbrock  
2018

**Querying Graphs**  
Angela Bonifati, George Fletcher, Hannes Voigt, and Nikolay Yakovets  
2018

**Query Processing over Incomplete Databases**  
Yunjun Gao and Xiaoye Miao  
2018

**Natural Language Data Management and Interfaces**  
Yunyao Li and Davood Rafiei  
2018

**Human Interaction with Graphs: A Visual Querying Perspective**  
Sourav S. Bhowmick, Byron Choi, and Chengkai Li  
2018

**On Uncertain Graphs**  
Arijit Khan, Yuan Ye, and Lei Chen  
2018

**Answering Queries Using Views**  
Foto Afrati and Rada Chirkova  
2017

**Databases on Modern Hardware: How to Stop Underutilization and Love Multicores**  
Anatasia Ailamaki, Ericta Liarou, Pinar Tözün, Danica Porobic, and Iraklis Psaroudakis  
2017
<table>
<thead>
<tr>
<th>Title</th>
<th>Editors/Authors</th>
<th>Year</th>
</tr>
</thead>
<tbody>
<tr>
<td>Generating Plans from Proofs: The Interpolation-based Approach to Query Reformulation</td>
<td>Michael Benedikt, Julien Leblay, Balder ten Cate, and Efthymia Tsamoura</td>
<td>2016</td>
</tr>
<tr>
<td>Datalog and Logic Databases</td>
<td>Sergio Greco and Cristina Molinaro</td>
<td>2015</td>
</tr>
<tr>
<td>Big Data Integration</td>
<td>Xin Luna Dong and Divesh Srivastava</td>
<td>2015</td>
</tr>
<tr>
<td>Similarity Joins in Relational Database Systems</td>
<td>Nikolaus Augsten and Michael H. Böhlen</td>
<td>2013</td>
</tr>
<tr>
<td>Information and Influence Propagation in Social Networks</td>
<td>Wei Chen, Laks V.S. Lakshmanan, and Carlos Castillo</td>
<td>2013</td>
</tr>
<tr>
<td>Data Cleaning: A Practical Perspective</td>
<td>Venkatesh Ganti and Anish Das Sarma</td>
<td>2013</td>
</tr>
<tr>
<td>Data Processing on FPGAs</td>
<td>Jens Teubner and Louis Woods</td>
<td>2013</td>
</tr>
</tbody>
</table>
Perspectives on Business Intelligence
Raymond T. Ng, Patricia C. Arocena, Denilson Barbosa, Giuseppe Carenini, Luiz Gomes, Jr., Stephan Jou, Rock Anthony Leung, Evangelos Milios, Renée J. Miller, John Mylopoulos, Rachel A. Pottinger, Frank Tompa, and Eric Yu
2013

Semantics Empowered Web 3.0: Managing Enterprise, Social, Sensor, and Cloud-based Data and Services for Advanced Applications
Amit Sheth and Krishnaprasad Thirunarayan
2012

Data Management in the Cloud: Challenges and Opportunities
Divyakant Agrawal, Sudipto Das, and Amr El Abbadi
2012

Query Processing over Uncertain Databases
Lei Chen and Xiang Lian
2012

Foundations of Data Quality Management
Wenfei Fan and Floris Geerts
2012

Incomplete Data and Data Dependencies in Relational Databases
Sergio Greco, Cristian Molinaro, and Francesca Spezzano
2012

Business Processes: A Database Perspective
Daniel Deutch and Tova Milo
2012

Data Protection from Insider Threats
Elisa Bertino
2012

Deep Web Query Interface Understanding and Integration
Eduard C. Dragut, Weiyi Meng, and Clement T. Yu
2012

P2P Techniques for Decentralized Applications
Esther Pacitti, Reza Akbarinia, and Manal El-Dick
2012

Query Answer Authentication
Hwee Hwa Pang and Kian-Lee Tan
2012
<table>
<thead>
<tr>
<th>Topic</th>
<th>Author(s)</th>
<th>Year</th>
</tr>
</thead>
<tbody>
<tr>
<td>Declarative Networking</td>
<td>Boon Thau Loo and Wenchao Zhou</td>
<td>2012</td>
</tr>
<tr>
<td>Full-Text (Substring) Indexes in External Memory</td>
<td>Marina Barsky, Ulrike Stege, and Alex Thomo</td>
<td>2011</td>
</tr>
<tr>
<td>Spatial Data Management</td>
<td>Nikos Mamoulis</td>
<td>2011</td>
</tr>
<tr>
<td>Database Repairing and Consistent Query Answering</td>
<td>Leopoldo Bertossi</td>
<td>2011</td>
</tr>
<tr>
<td>Managing Event Information: Modeling, Retrieval, and Applications</td>
<td>Amarnath Gupta and Ramesh Jain</td>
<td>2011</td>
</tr>
<tr>
<td>Fundamentals of Physical Design and Query Compilation</td>
<td>David Toman and Grant Weddell</td>
<td>2011</td>
</tr>
<tr>
<td>Methods for Mining and Summarizing Text Conversations</td>
<td>Giuseppe Carenini, Gabriel Murray, and Raymond Ng</td>
<td>2011</td>
</tr>
<tr>
<td>Probabilistic Databases</td>
<td>Dan Suciu, Dan Olteanu, Christopher Ré, and Christoph Koch</td>
<td>2011</td>
</tr>
<tr>
<td>Peer-to-Peer Data Management</td>
<td>Karl Aberer</td>
<td>2011</td>
</tr>
<tr>
<td>Probabilistic Ranking Techniques in Relational Databases</td>
<td>Ihab F. Ilyas and Mohamed A. Soliman</td>
<td>2011</td>
</tr>
<tr>
<td>Uncertain Schema Matching</td>
<td>Avigdor Gal</td>
<td>2011</td>
</tr>
<tr>
<td>Fundamentals of Object Databases: Object-Oriented and Object-Relational Design</td>
<td>Suzanne W. Dietrich and Susan D. Urban</td>
<td>2010</td>
</tr>
</tbody>
</table>
Advanced Metasearch Engine Technology
Weiyi Meng and Clement T. Yu
2010

Web Page Recommendation Models: Theory and Algorithms
Sule Gündüz-Ögüdücü
2010

Multidimensional Databases and Data Warehousing
Christian S. Jensen, Torben Bach Pedersen, and Christian Thomsen
2010

Database Replication
Bettina Kemme, Ricardo Jimenez-Peris, and Marta Patino-Martinez
2010

Relational and XML Data Exchange
Marcelo Arenas, Pablo Barcelo, Leonid Libkin, and Filip Murlak
2010

User-Centered Data Management
Tiziana Catarci, Alan Dix, Stephen Kimani, and Giuseppe Santucci
2010

Data Stream Management
Lukasz Golab and M. Tamer Özsu
2010

Access Control in Data Management Systems
Elena Ferrari
2010

An Introduction to Duplicate Detection
Felix Naumann and Melanie Herschel
2010

Privacy-Preserving Data Publishing: An Overview
Raymond Chi-Wing Wong and Ada Wai-Chee Fu
2010

Keyword Search in Databases
Jeffrey Xu Yu, Lu Qin, and Lijun Chang
2009
Cloud-Based
RDF Data Management

Zoi Kaoudi
Technische Universität Berlin

Ioana Manolescu
INRIA

Stamatis Zampetakis
TIBCO Orchestra Networks

SYNTHESIS LECTURES ON DATA MANAGEMENT #62
ABSTRACT

Resource Description Framework (or RDF, in short) is set to deliver many of the original semi-structured data promises: flexible structure, optional schema, and rich, flexible Universal Resource Identifiers as a basis for information sharing. Moreover, RDF is uniquely positioned to benefit from the efforts of scientific communities studying databases, knowledge representation, and Web technologies. As a consequence, the RDF data model is used in a variety of applications today for integrating knowledge and information: in open Web or government data via the Linked Open Data initiative, in scientific domains such as bioinformatics, and more recently in search engines and personal assistants of enterprises in the form of knowledge graphs.

Managing such large volumes of RDF data is challenging due to the sheer size, heterogeneity, and complexity brought by RDF reasoning. To tackle the size challenge, distributed architectures are required. Cloud computing is an emerging paradigm massively adopted in many applications requiring distributed architectures for the scalability, fault tolerance, and elasticity features it provides. At the same time, interest in massively parallel processing has been renewed by the MapReduce model and many follow-up works, which aim at simplifying the deployment of massively parallel data management tasks in a cloud environment.

In this book, we study the state-of-the-art RDF data management in cloud environments and parallel/distributed architectures that were not necessarily intended for the cloud, but can easily be deployed therein. After providing a comprehensive background on RDF and cloud technologies, we explore four aspects that are vital in an RDF data management system: data storage, query processing, query optimization, and reasoning. We conclude the book with a discussion on open problems and future directions.

KEYWORDS

RDF, cloud computing, MapReduce, key-value stores, query optimization, reasoning
# Contents

1 Introduction ......................................................... 1

2 Preliminaries ....................................................... 5
   2.1 Resource Description Framework (RDF) ...................... 5
      2.1.1 Data Model ............................................. 5
      2.1.2 The SPARQL Query Language .......................... 10
   2.2 Distributed Storage and Computing Paradigms ............. 13
      2.2.1 Distributed File Systems .............................. 13
      2.2.2 Distributed Key-Value Stores ......................... 14
      2.2.3 Distributed Computation Frameworks: MapReduce and Beyond .......................... 17
   2.3 Summary ...................................................... 20

3 Cloud-Based RDF Storage ......................................... 21
   3.1 Partitioning Strategies ..................................... 21
      3.1.1 Storage Description Grammar .......................... 22
      3.1.2 Logical Partitioning .................................. 24
      3.1.3 Graph Partitioning .................................... 24
   3.2 Storing in Distributed File Systems ......................... 25
      3.2.1 Triple Model ............................................ 25
      3.2.2 Vertical Partitioning Model ............................ 25
      3.2.3 Partitioning Based on RDF Entities .................. 28
   3.3 Storing in Key-Value Stores ................................ 29
      3.3.1 Triple-Based ............................................ 29
      3.3.2 Graph-Based ........................................... 32
   3.4 Storing in Multiple Centralized RDF Stores ............... 34
   3.5 Storing in Main Memory Stores .............................. 38
   3.6 Storing in Multiple Back-End Stores ....................... 39
   3.7 Summary ...................................................... 39

4 Cloud-Based SPARQL Query Processing .......................... 43
   4.1 Relational-Based Query Processing ......................... 43
The Resource Description Framework (RDF) [W3C, 2004] first appeared in 2004 to realize the vision of the Semantic Web [Berners-Lee et al., 2001]. The goal of Semantic Web was to evolve the Web in order to accommodate intelligent, automatic processes that perform tasks on behalf of the users utilizing machine-readable data. This data should have well-defined semantics enabling better data integration and interoperability. RDF provided the standardized means of representing this structured and meaningful information on the Web.

Today a vast amount of data is available online accommodating many aspects of human activities, knowledge, and experiences. RDF provides a simple and abstract knowledge representation for such data on the Web which are uniquely identified by Universal Resource Identifiers (URIs). RDF Schema (RDFS) [W3C, 2014b] is the vocabulary language of RDF. It gives meaning to resources, groups them into concepts and identifies the relationships between these concepts. Web Ontology Language (OWL) [W3C OWL Working Group, 2012] can also be used for conceptualization and provides further expressiveness in stating relationships among the resources. Ontology languages, such as RDFS and OWL, allow for deriving entailed information through reasoning. For instance, one can reason that any student is also a human, or that if X worksWith Y, then X also knows Y; or that if X drives car Z, then X is a human and Z is a vehicle. Finally, to be able to explore and query structured information expressed in RDF, SPARQL [W3C, 2013] has been the official W3C recommendation language since 2008.

RDF is used today in a variety of applications. A particularly interesting one comes from the Open Data concept that “certain data should be freely available to everyone to use and republish as they wish, without restrictions from copyright, patents or other mechanisms of control.” Open Data federates players of many roles, from organizations such as business and government aiming at demonstrate transparency and good (corporate) governance, to end users interested in consuming and producing data to share with others, to aggregators that may build business models around warehousing, curating, and sharing this data [Raschia et al., 2012]. Sample governmental Open Data portals are the ones from the U.S., UK, and France. At the same time, if Open Data designates a general philosophy, Linked Data refers to the “recommended best practice for exposing, sharing, and connecting pieces of data, information, and knowledge on the Semantic Web using URIs and RDF” [Berners-Lee, 2006]. In practice, Open and Linked data are frequently

---

2 [www.data.gov](http://www.data.gov)
3 [www.data.gov.uk](http://www.data.gov.uk)
4 [www.etalab.fr](http://www.etalab.fr)
1. INTRODUCTION

combined to facilitate data sharing, interpretation, and exploitation LOD. Sample applications of Linked Open Data are DBPedia (the Linked Data version of Wikipedia), BBC’s platform for the World Cup 2010 and the 2012 Olympic games [Kiryakov et al., 2010].

In addition, RDF is the main data model behind private or public knowledge graphs (e.g., DBPedia [Auer et al., 2007], YAGO [Suchanek et al., 2007], Wikidata5 (previously Freebase)). Knowledge graphs have been increasingly used recently by enterprises to facilitate and enhance the functionality of their products. Knowledge graphs are simply graphs that connect entities via their relationships. For example, Google uses a knowledge graph in its search engine and personal assistant, Microsoft built a knowledge graph [Gao et al., 2018] used in its products (e.g., Bing and Cortana), and Walmart [Deshpande et al., 2013] and Amazon [Dong, 2018] use knowledge graphs in a variety of applications such as product search and advertising.

To exploit large volumes of RDF data, one could try to build a centralized warehouse. Some of the very first systems that appeared in the Semantic Web community include Jena [Wilkinson et al., 2003] and Sesame [Broekstra and Kampman, 2002]. Later on, RDF-based stores had gained interest in the database community as well, as illustrated by the works of Abadi et al. [2009], Neumann and Weikum [2010b] and Weiss et al. [2008b]. Moreover, commercial database management systems also had started providing support for RDF, such as Oracle 11g [Chong et al., 2005] or IBM DB2 10.1 [Bornea et al., 2013]. These works mostly focused on RDF viewed as a relational database on which to evaluate conjunctive queries and do not consider RDF-specific features such as those related to reasoning. A different line of research focused on viewing RDF as a graph and exploited graph models for the indexing and storage and subgraph matching for querying [Udeea et al., 2007, Zou et al., 2014].

Large and increasing data volumes have also raised the need for distributed storage architectures. Past works on distributed RDF query processing and reasoning have relied on peer-to-peer platforms [Kaoudi and Koubarakis, 2013, Kaoudi et al., 2010] or clustered architectures [Erling and Mikhailov, 2009, Harris et al., 2009, Owens et al., 2008]. Most of these approaches have been proved inadequate to scale to the large amounts of RDF data that we encounter nowadays. Peer-to-peer architectures suffer more on long latency during query evaluation because of the many communication steps required when exchanging large amounts of data and the geo-distribution of peers. Clustered architectures, on the other hand, require very fine-grained tuning of the cluster and have long loading times.

Cloud computing is an emerging paradigm massively adopted in many applications for the scalability, fault tolerance and elasticity features it offers, which also allows for effortless deployment of distributed and parallel architectures. At the same time, interest in massively parallel processing has been renewed by the MapReduce model [Dean and Ghemawat, 2004] and many follow-up works, which aim at simplifying the deployment of massively parallel data management tasks in a cloud environment. For these reasons, cloud-based stores are an interesting avenue to explore for handling very large volumes of RDF data.

5https://www.wikidata.org
1. INTRODUCTION

The main goal of this book is to study state-of-the-art RDF data management in a cloud environment. It also investigates the most recent advances of RDF data management in parallel/distributed architectures that were not necessarily intended for the cloud, but can easily be deployed therein. We provide a description of existing systems and proposals which can handle large volumes of RDF data while classifying them along different dimensions and highlighting their limitations and opportunities. We start by identifying four dimensions according to the way in which systems implement four fundamental functionalities: data storage, query processing, query optimization, and reasoning. Then, within each dimension we classify each system according to their basic characteristics.

The remainder of this book is organized as follows. We start with Chapter 2 by introducing the main features of RDF and its accompanying schema language RDFS. The same chapter gives an overview of the cloud-based frameworks and tools used up through today for RDF data management. In Chapter 3 we present current approaches on RDF data storage and in Chapter 4 we describe different query processing paradigms for evaluating RDF queries. Chapter 5 describes the state-of-the-art in query optimization used for cloud-based query evaluation. In Chapter 6 we lay out the state-of-the-art in RDFS reasoning on top of cloud platforms. Finally, we conclude in Chapter 7 and give insights into open problems and directions.
CHAPTER 2

Preliminaries

This chapter introduces the main concepts of RDF and its accompanying schema language RDFS. It additionally describes the main characteristics of the distributed paradigms and frameworks used in the cloud that have been used in building RDF data management systems.

2.1 RESOURCE DESCRIPTION FRAMEWORK (RDF)

We start by introducing the background for our topic of interest, notably the RDF data model (Section 2.1.1) and SPARQL query language (Section 2.1.2). We then outline the major current architectures for storing and processing data, upon which large-scale RDF data management systems are built (Section 2.2).

2.1.1 DATA MODEL

RDF data is organized in triples of the form \( (s \ p \ o) \), stating that the subject \( s \) has the property (a.k.a. predicate) \( p \) whose value is the object \( o \). Unique Resource Identifiers (URIs) are central in RDF: One can use URIs in any position of a triple to uniquely refer to some entity or concept. Notice that literals (constants) are also allowed in the \( o \) position.

RDF allows some form of incomplete information through blank nodes, standing for unknown constants or URIs. One may think of blank nodes as labeled nulls from the database literature [Abiteboul et al., 1995].

**Definition 2.1 RDF Triple.** Let \( U \) be a set of URIs, \( L \) be a set of literals, and \( B \) be a set of blank nodes. A well-formed RDF triple is a tuple \( (s \ p \ o) \) from \( (U \cup B) \times U \times (U \cup L \cup B) \).

The syntactic conventions for representing valid URIs, literals, and blank nodes can be found in RDF Concepts. In this book, literals are shown as strings enclosed by quotation marks, while URIs are shown as simple strings (see also discussion on namespaces below).

RDF admits a natural graph representation, with each \( (s \ p \ o) \) triple seen as an \( p \)-labeled directed edge from the node identified by \( s \) to the node identified by \( o \).

**Definition 2.2 RDF Graph.** An RDF graph is a set of RDF triples.

We use \( \text{val}(G) \) to refer to the values (URIs, literals, and blank nodes) of an RDF graph \( G \).
Figure 2.1: RDF graph in N-Triples syntax.

For instance, Figure 2.1 depicts an RDF graph in the so-called N-Triples syntax, while Figure 2.2 shows a graphical representation of the same graph.

In some cases, we need to work with several RDF graphs while still being able to distinguish the graph each triple originates from. We achieve this by considering named RDF graphs where each graph is associated with a name that can be a URI or a blank node. The notion of an RDF triple is extended as follows to capture these needs.

**Definition 2.3 RDF quad.** Let $U$ be a set of URIs, $L$ be a set of literals, and $B$ be a set of blank nodes. A well-formed RDF quad is a tuple $(s \ p \ o \ g)$ from $(U \cup B) \times U \times (U \cup L \cup B) \times (U \cup B)$.

We are now able to capture multiple RDF graphs using the notion of an RDF dataset.

**Definition 2.4 RDF dataset** is a set of RDF quads.

An RDF dataset may contain only a single graph, in which case all the quads of the form $(s \ p \ o \ g)$ have the same value for $g$. In such cases, we may use the term RDF graph and RDF dataset interchangeably.

Namespaces are supported in RDF as a means to support flexible choices of URIs as well as interoperability between different datasets. A namespace typically serves to identify a certain application domain. Concretely, a namespace is identified by a URI, which is used as a prefix of all URIs defined within the respective application domain. Thus, for instance, the URI 

http://www.w3.org/1999/02/rdf-syntax-ns

is chosen by the W3C to represent the...
domain of a small set of predefined URIs which are part of the RDF specification itself; or, for instance, http://swat.cse.lehigh.edu/onto/univ-bench.owl is used by the University of Lehigh to identify its domain of representation. To denote that the URI of a resource $r$ is part of the application domain identified by a namespace URI $u$, the URI of $u$ is a prefix of the URI of $r$. The suffix of $r$'s URI is typically called local name; it uniquely identifies $r$ among all the resources of the namespace $u$. This enables other application domains to use the same local name in conjunction with their respective namespace URIs without causing confusions between the two. On the other hand, when one wishes to refer in a dataset to a specific resource from a specific namespace, the full URI (including the namespace URI prefix) must be used.

While the above mechanism is flexible, it leads to rather lengthy URIs, which increase the space occupancy of a dataset. To solve this problem, within an RDF graph, a local namespace prefix is associated with a namespace URI and serves as a shorthand to represent the latter. Thus, URIs are typically of the form $nsp:ln$, where $nsp$ stands for the local namespace prefix while $ln$ represents the local name.

Resource descriptions can be enhanced by specifying to which class(es) a given resource belongs by means of the pre-defined rdf:type property which is part of the RDF specification.

For instance, the RDF Graph in Figure 2.1 features the classes :artist, :painter, :cubist, etc., and the resource :picasso is stated to be of type :cubist.

Further, the RDF Schema [W3C, 2014b] specification allows relating classes and properties used in a graph through ontological (i.e., deductive) constraints expressed as triples using built-in properties:
8 2. PRELIMINARIES

Table 2.1: Deductive constraints expressible in an RDF Schema

<table>
<thead>
<tr>
<th>RDFS Constraint</th>
<th>Relational Modeling</th>
</tr>
</thead>
<tbody>
<tr>
<td>s rdfs:subClassOf o</td>
<td>∀x [s(x) → o(x)]</td>
</tr>
<tr>
<td>s rdfs:subPropertyOf o</td>
<td>∀x, y [s(x, y) → o(x, y)]</td>
</tr>
<tr>
<td>p rdfs:domain c</td>
<td>∀x, y [p(x, y) → c(x)]</td>
</tr>
<tr>
<td>p rdfs:range c</td>
<td>∀x, y [p(x, y) → c(y)]</td>
</tr>
</tbody>
</table>

- sub-class constraints expressed using the property rdfs:subClassOf;
- sub-property constraints expressed using the property rdfs:subPropertyOf;
- property domain (first attribute) typing using the property rdfs:domain;
- property range (second attribute) typing using the property rdfs:range.

RDFS constraints and their corresponding relational (first-order logic) modeling are given in Table 2.1.

For instance, our sample RDF graph features the subclass constraint :sculptor rdfs:subClassOf :artist, and the subproperty constraint :paints rdfs:subClassOf :creates. Moreover, it types the first and second attribute of the property :creates by stating that someone who creates is an :artist, while the creation results in an :artifact.

The previous example reveal an important feature of RDF, which is implicit information: triples that hold in the RDF graph even though they may not be part of it explicitly. The process of inferring new triples based on existing ones (some of them may be RDF Schema constraints, while the others are simple triples, a.k.a. facts) is known as RDF entailment or inference and is guided by a set of entailment rules. Table 2.2 presents the most common RDFS entailment rules. The full set of entailment rules is defined in the RDF Semantics [W3C, 2014a].

The first two rules, $s_1$ and $s_2$, start from premises that are schema triples and lead to another schema triple; they can be seen as reasoning on the RDF Schema itself. For instance, if we know that :cubist rdfs:subClassOf :painter and :painter rdfs:subClassOf :artist, it is easy to see that :cubist rdfs:subClassOf :artist.

The remaining rules $i_1$ to $i_4$ apply on one data and one schema triple, and lead to a new data triple. Rule $i_1$ infers that resource $s$ is of type $c_2$, if $s$ is of a type $c_1$ that is more specific than $c_2$. For instance, in our sample graph, if :picasso rdf:type :cubist and :cubist rdfs:subClassOf :painter, then :picasso rdf:type :painter. Rule $i_2$ states that the property $p_2$ of a resource $s$ is $o$ as soon as $p_1$ is a subproperty of $p_2$ and $o$ is the value of the property $p_2$ of $s$. For instance, if :picasso :paints :guernica, it is also the case that :picasso :creates :guernica. Rule $i_3$ allows us to infer that a resource $s$ is of type $c$ if $s$ has a property $p$ whose domain is $c$. For instance, given that :creates has the domain :artist, and knowing that :rodin :creates :thethinker, it...
follows that :rodin is of type :artist. Finally, rule \( i_4 \) allows us to infer that a resource \( o \) is of type \( c \) if \( o \) is a value of a property \( p \) whose range is \( c \). For example, given that :thethinker is created by someone, and knowing that the range of :creates is :artifact, we can infer that :thethinker is an :artifact.

Note that in this example, there were two independent ways of inferring that :rodin rdf:type :artist, one based on rule \( i_3 \) as explained above, and another one based on the rule \( i_1 \), knowing that :rodin is a :sculptor and that every sculptor is an artist. More generally, although in our example this is not the case, the RDF graph may even have contained the explicit fact :rodin rdf:type :artist. Thus, in an arbitrary RDF graph, the same fact may be present only explicitly, or only implicitly, and there may be several ways to infer the same fact from those explicitly present in the graph.

Observe that unlike the traditional setting of relational databases, RDF Schema constraints are expressed with RDF triples themselves and are part of the RDF graph (as opposed to relational schemas being separated from the relational database instances). Within an RDF dataset, the term fact is commonly used to denote a triple whose property is not one of the predefined RDF Schema properties.

**Definition 2.5 RDFS Closure.** The RDFS closure of an RDF graph \( G \), denoted \( G^\infty \), is obtained by adding to \( G \) all the implicit triples that derive from consecutive applications of the entailment rules on \( G \) until a fixpoint is reached.

It has been shown that under RDF Schema constraints, the closure of an RDF graph is finite and unique (up to blank node renaming) [Muñoz et al., 2009, ter Horst, 2005b]. For instance, Figure 2.3 depicts the RDFS closure of the RDF graph shown in Figure 2.1. In Figure 2.3, the inferred triples that were not already present in the initial RDF graph are shown in italic font.
2. PRELIMINARIES

`sculptor :subClassOf :artist .
painter :subClassOf :artist .
cubist :subClassOf :painter .
cubist :subClassOf :painter .
paints :subPropertyOf :creates .
creates :domain :artist .
creates :range :artifact .
picasso :type :cubist .
picasso :type :painter .
picasso :type :artist .
picasso :name "Pablo" .
picasso :paints :guernica .
picasso :creates :guernica .
guernica :type :artifact .
guernica :exhibited :reinasofia .
reinasofia :located :madrid .
rodin :type :sculptor .
rodin :type :artist .
rodin :name "Auguste" .
rodin :creates :thethinker .
thethinker :type :artifact .
thethinker :exhibited :museerodin .
museerodin :located :paris .

Figure 2.3: RDFS closure of the RDF graph in Figure 2.1.

2.1.2 THE SPARQL QUERY LANGUAGE

The advent of the Semantic Web along with the arrival of the RDF data model entailed the need for a suitable declarative query language. SPARQL [W3C, 2008], which became a W3C standard in 2008, has evolved since its proposal; the current version (SPARQL 1.1 [W3C, 2013]) resembles a complex relational query language such as SQL.

SPARQL has a variety of features. The simplest ones are: conjunctive graph pattern matching, selections, projections, and joins, while the more advanced allow arithmetic and alphanumeric comparisons, aggregations, nested sub-queries, and graph construction. In this work, we consider the most common fragment of SPARQL, named basic graph pattern (BGP) queries; from a database perspective, these correspond to conjunctive select-project-join (SPJ) queries.

A central role in composing BGPs is played by triple patterns.

Definition 2.6  **Triple Pattern.** Let $U$ be a set of URIs, $L$ be a set of literals and $V$ be a set of variables, a *triple pattern* is a tuple $(s, p, o)$ from $(U \cup V) \times (U \cup V) \times (U \cup L \cup V)$. 
Triple patterns are used to specify queries against a single RDF graph. Going forward, when no confusion arises, we refer to BPG patterns as triple patterns or even simply atoms/triples.

Based on triple (or quad) patterns, one can express SPARQL BGP queries as below.

**Definition 2.7 BGP Query.** A BGP query is an expression of the form

\[
\text{SELECT } ?x_1, \ldots, ?x_m \text{ WHERE } \{ t_1, \ldots, t_n \}
\]

where \( t_1, \ldots, t_n \) are triple patterns and \( ?x_1, \ldots, x_m \) are distinguished variables appearing in \( t_1, \ldots, t_n \). We also define the size of the query as its number of distinct triple patterns.

Alternatively, for ease of presentation, BGP queries can be represented using the equivalent conjunctive query notation, e.g., the query appearing in Definition 2.7 could be denoted as

\[ q(x_1, \ldots, x_m) = t_1 \land \ldots \land t_n. \]

In the remainder we use the terms RDF query, SPARQL query, and BGP query, interchangeably referring to the SPARQL fragment described by Definition 2.7. Furthermore, we use \( \text{var}(q) \) (resp. \( \text{var}(t_i) \)) to refer to the variables of a query \( q \) (resp. an atom \( t_i \)). Additionally, we refer to the set of head variables of \( q \) as \( \text{headvar}(q) \).

An example BGP query, asking for the resources exhibited in Paris and their creators, is shown in Figure 2.4.

**Definition 2.8 Valid Assignment.** Let \( q \) be a BGP query, and \( G \) be an RDF graph, \( \mu: \text{var}(q) \to \text{val}(G) \) is a valid assignment iff \( \forall t_i \in q, t_i^\mu \in G \) where we denote by \( t_i^\mu \) the result of replacing every occurrence of a variable \( e \in \text{var}(q) \) in the triple pattern \( t_i \) by the value \( \mu(e) \in \text{val}(G) \).

**Definition 2.9 Result Tuple.** Let \( q \) be a BGP query, \( G \) be an RDF graph, \( \mu \) be a valid assignment, and \( \bar{x} = \text{headvar} \), the result tuple of \( q \) based on \( \mu \), denoted as \( \text{res}(q, \mu) \), is the tuple:

\[ \text{res}(q, \mu) = \{ \mu(x_1), \ldots, \mu(x_m) \mid x_1, \ldots, x_m \in \bar{x} \} \]

**Definition 2.10 Query Evaluation.** Let \( q \) be a BGP query, and \( G \) be an RDF graph, the evaluation of \( q \) against \( G \) is:

\[ q(G) = \{ \text{res}(q, \mu) \mid \mu: \text{var}(q) \to \text{val}(G) \text{ is a valid assignment} \} \]
where \( res(q, \mu) \) is the result tuple of \( q \) based on \( \mu \).

The evaluation of query QA (shown in Figure 2.4) against the graph of Figure 2.1 is the tuple \( \text{:rodin :thethinker} \).

Query evaluation only accounts for triples explicitly present in the graph. If entailed triples are not explicitly in the graph, evaluating the query may miss some results which would have been obtained otherwise. For instance, consider the query:

```sql
SELECT ?a
WHERE { ?a rdf:type ?artifact . }
```

Evaluating the query on the graph in Figure 2.1 returns \( \text{:guernica} \) but not \( \text{:thethinker} \) because the latter is not explicitly of type \( \text{:artifact} \). Instead, query answering on the RDFS closure of the graph depicted in Figure 2.3 leads to two answers \( \text{:guernica} \) and \( \text{:thethinker} \).

The following definition allows capturing results due to both the explicit and implicit triples in an RDF graph:

**Definition 2.11 Query Answering.** The *answer of a BGP query* \( q \) *over an RDF graph* \( G \) *is the evaluation of* \( q \) *over* \( G^\infty \).

It is worth noting that while the relational SPJ queries are most often used with set semantics, SPARQL, just like SQL, has bag (multiset) semantics.

An alternative strategy to RDFS closure for enabling complete query answering w.r.t. implicit information is known as query reformulation. Query reformulation expands the original query into a reformulated one whose answer over the initial graph is complete.

**Definition 2.12 Query Reformulation.** Given a query \( q \) and an RDF graph \( G \), a query \( q^{ref} \) is a reformulation of \( q \) w.r.t. the RDFS constraints of \( G \), iff \( q(G^\infty) = q^{ref}(G) \).

Different query reformulation algorithms have been proposed in literature. Reformulating into a union of conjunctive queries (UCQ) applies to various fragments of RDF ranging from Description Logics (DL) to Database [Goasdoué et al., 2013], [Urbani et al., 2013], [Kaoudi et al., 2008], [Urbani et al., 2011b], [De Giacomo et al., 2012], [Adjiman et al., 2007], [Goasdoué et al., 2011], [Calvanese et al., 2007], [Gottlob et al., 2011], [König et al., 2015]. Another notable reformulation technique computes semi-conjunctive queries (SCQ) reformulations [Thomazo, 2013] and can be applied to the DL fragment of RDF. Join of unions of conjunctive queries (JUCQ) reformulation, a generalization of both UCQ and SCQ, has been proposed for RDF [Bursztyn et al., 2015b] and its DL fragment [Bursztyn et al., 2015a], and has shown to be able to improve the execution performance of reformulated queries.

For illustration, the UCQ reformulation of the query asking for artifacts is:
In the UCQ, all expansions of the original term \( ?a \text{ rdf:type:artifact} \) are evaluated, and a union of their results is returned. For this very simple one-atom query, the UCQ, SCQ and JUCQ reformulations coincide.

It is easy to see that for more complex queries, each atom may have a large expansion, and the possible combinations among these expansions lead to a potentially large reformulation. The interest of SCQs and JUCQs, which are clearly equivalent to the UCQ reformulation, is to propose different orders among the query operators, which oftentimes lead to more efficient evaluation through a standard RDBMS.

### 2.2 DISTRIBUTED STORAGE AND COMPUTING PARADIGMS

We now outline the main features of distributed storage and computing paradigms including distributed file systems, key-value stores and computation frameworks.

#### 2.2.1 DISTRIBUTED FILE SYSTEMS

Distributed file systems have their roots way back in the 1980s [Sandberg et al., 1985]. However, the topic has seen renewed interest with the emergence of the cloud computing.

In an attempt to cover growing data processing needs, Google introduced the Google File System [Ghemawat et al., 2003] (GFS), a distributed file system which aims to provide performance, scalability, reliability and availability. However, it makes some radical design choices to support more effectively the following scenarios. First, because it is meant to be deployed on very large sets of standard hardware, node failures are considered the rule rather than the exception. Second, it is assumed that typical files are extremely large (in GBs) while the number of files is rather moderate, thus performance-wise, choices are made to support the large ones. Third, updates are typically handled by appending to the file rather than overwriting.

GFS follows a master/slave architecture. Files are split into fixed-size chunks. Each chunk is given a unique 64-bit Id by the master and the slaves store file chunks on local disk as normal files. For reliability, each file chunk is replicated on multiple servers (3 times by default). The master node keeps all required metadata for placement, replication, reading and writing of file chunks.

In the same spirit, and closely following the GFS design principles, other distributed file systems were developed like Apache’s Hadoop Distributed File System [Hadoop] (HDFS), and Amazon’s S3 [S3]. HDFS became popular due to the open source implementation that Apache provided.
Distributed file systems, however, do not provide fine-grained data access, and thus, selective access to a piece of data can only be achieved at the granularity of a file. There have been works like [Dittrich et al., 2010, 2012] which extend Hadoop and improve its data access efficiency with indexing functionality, but the proposed techniques are yet to be adopted by Hadoop’s development community.

### 2.2.2 DISTRIBUTED KEY-VALUE STORES

A key-value store (key-value database) is a system for storing, retrieving, and managing associative arrays. An associative array is a data structure that can hold a set of \( \{ \text{key} : \text{value} \} \) pairs such that each possible key appears just once. Other common names for associative arrays include map, dictionary, and symbol table. Key-value stores have been around as long as relational databases. In contrast with relational databases, key-value stores have a very limited application programming interface (API) and the vast majority do not support join operations between different arrays. The most basic operations supported by all key-value stores are \( \text{Get}(\text{k}, v) \) and \( \text{Put}(\text{k}, v) \).

Recently, key-value stores gained a lot of popularity due to their simple design, horizontal scaling, and finer control over availability. Google’s Bigtable [Chang et al., 2006] inspired many of the key-value stores that are used nowadays. Bigtable takes the idea of associative arrays one step further, defining each array as a sparse, distributed, persistent multidimensional sorted map. The Bigtable’s map indexes a value using a triple composed of the row key, column key, and a timestamp. Each map implies a nested structure of the form \( \{ \text{rowkey} : \{ \text{columnkey} : \{ \text{time} : \text{value} \} \} \} \). By considering the map (array) as part of the nested structure, the complete BigTable’s architecture can be described as \( \{ \text{tablename} : \{ \text{rowkey} : \{ \text{columnkey} : \{ \text{time} : \text{value} \} \} \} \} \). Using the ER diagram formalism, key-value stores similar to BigTable adopt the schema shown in Figure 2.5.

Popular key-value stores that have been used by RDF systems include: Apache’s Cassandra [Cassandra], Apache’s Accumulo [Accumulo], Apache’s HBase [HBase], Amazon’s SimpleDB [SimpleDB], and Amazon’s DynamoDB [Dynamo]. Although they share the basic elements of their interfaces, these systems differ with respect to their internal architecture, access control policies, authentication, consistency, etc. Below we briefly present Accumulo, Cassandra, and HBase, while we provide a slightly more elaborated overview for SimpleDB, and Dy-
namoDB to provide more insight regarding the general API of key-value stores found in the cloud and expose various limitations also present in commercial systems.

**HBase** is an open-source, distributed, versioned, non-relational database modeled after Google’s Bigtable and implemented on top of HDFS. A data row in HBase is composed of a sortable row key and an arbitrary number of columns, which are further grouped into column families. A data cell can hold multiple versions of data which are distinguished by timestamps. Data stored in the same column family are stored together in the file system, while data in different column families might be distributed. HBase provides a B+ tree-like index on the row key by default. HBase supports ACID-level semantics on a per-row basis (row-level consistency). In addition, the notion of coprocessors is introduced, which allow the execution of user code in the context of the HBase processes. The result is roughly comparable to the relational database world’s triggers and stored procedures.

**Accumulo** is very similar to HBase since it also follows the Bigtable design pattern and is implemented on top of HDFS. In contrast with HBase and Bigtable, it also provides a server-side programming mechanism, called iterator, that helps increase performance by performing large computing tasks directly on the servers and not on the client machine. By doing this, it avoids sending large amounts of data across the network. Furthermore, it extends the Bigtable data model, adding a new element to the key called “Column Visibility.” This element stores a logical combination of security labels that must be satisfied at query time in order for the key and value to be returned as part of a user request. This allows data with different security requirements to be stored in the same table. As a consequence, users can see only those keys and values for which they are authorized.

**Cassandra** is also inspired by Bigtable and implemented on top of HDFS, thus sharing a lot of similarities with Accumulo and HBase. Nevertheless, it has some distinctive features. It extends the Bigtable data model by introducing supercolumns. A storage model with supercolumns looks like: `{rowkey:{superkey:{columnkey:value}}}`. Supercolumns can be either stored based on the hash value of the supercolumn key or in sorted order. In addition, supercolumns can be further nested. Cassandra natively supports secondary indices, which can improve data access performance in columns whose values have a high level of repetition. Furthermore, it has configurable consistency. Both read and write consistency can be tuned, not only by level, but in extent. Finally, Cassandra provides an SQL-like language, CQL, for interacting with the store.

**SimpleDB** is a non-relational data store provided by Amazon which focuses on high availability (ensured through replication), flexibility, and scalability. SimpleDB supports a set of APIs to query and store items in the database. A SimpleDB data store is organized in domains. Each domain is a collection of items identified by their name. Each item contains one or more attributes; an attribute has a name and a set of associated values. There is a one-to-one mapping from SimpleDB’s data model to the one proposed by Bigtable shown in Figure 2.5.
correspond to tables, items to rows, attributes to columns, and values to cells. The main operations of SimpleDB API are the following (the respective delete/update operations are also available).

- **ListDomains()** retrieves all the domains associated with one Amazon Web Services (AWS) account.

- **CreateDomain(D)** and **DeleteDomain(D)** respectively creates a new domain D and deletes an existing one.

- **PutAttributes(D, k, (a,v)+)** inserts or replaces attributes \((a,v)\)+ into an item with name \(k\) of a domain \(D\). If the item specified does not exist, SimpleDB will create a new item.

- **BatchPutAttributes** performs up to 25 **PutAttributes** operations in a single API call, which allows for obtaining a better throughput performance.

- **GetAttributes(D, k)** returns the set of attributes associated with item \(k\) in domain \(D\).

It is not possible to execute an API operation across different domains as it is not possible to combine results from many tables in Bigtable. Therefore, if required, the aggregation of results from API operations executed over different domains has to be done in the application layer. AWS ensures that operations over different domains run in parallel. Hence, it is beneficial to split the data in several domains in order to obtain maximum performance. As with most non-relational databases, SimpleDB does not follow a strict transactional model based on locks or timestamps. It only provides the simple model of conditional puts. It is possible to update fields on the basis of the values of other fields. It allows for the implementation of elementary transactional models such as some entry level versions of optimistic concurrency control.

AWS imposes some size and cardinality limitations on SimpleDB. These limitations include:

- **Number of domains:** The default settings of an AWS account allow for at most 250 domains. While it is possible to negotiate more, this has some overhead (one must discuss with a sale representative etc.—it is not as easy as reserving more resources through an online form).

- **Domain size:** The maximum size of a domain cannot exceed 10 GB and the 109 attributes.

- **Item name length:** The name of an item should not occupy more than 1024 bytes.

- **Number of (attribute, value) pairs in an item:** This cannot exceed 256. As a consequence, if an item has only one attribute, that attribute cannot have more than 256 associated values.

- **Length of an attribute name or value:** This cannot exceed 1024 bytes.
2.2. DISTRIBUTED STORAGE AND COMPUTING PARADIGMS

DynamoDB is the successor of SimpleDB that resulted from combining the best parts of the original Dynamo [DeCandia et al., 2007] design (incremental scalability, predictable high performance) with the best parts of SimpleDB (ease of administration of a cloud service, consistency, and a table-based data model that is richer than a pure key-value store).

The main operations of the DynamoDB API are the following (the respective delete/update operations are also available):

- `ListTables()` retrieves all the tables associated with one AWS account in a specific AWS Region.
- `createTable(T, Key(pk, rk?))` creates a new table T having a primary key pk and a range key rk.
- `PutItem(T, Key(hk, [rk]), (a,v)+)` creates a new item in the table T containing a set of attributes (a,v)+ and having a key composed by a hash key hk and range key rk, or replaces it if it already existed. Specifying the range key is optional.
- `BatchWriteItem(item+)` puts and/or deletes up to 25 items in a single request, thus obtaining better performance.
- `GetItem(T, Key(hk, [rk]), (a)*)` returns the item having the key `Key(hk, [rk])` in table T. Again, specifying the range key is optional. It is possible to retrieve only a subset of the attributes associated with an item by specifying their names (a)* in the request.

DynamoDB was designed to provide seamless scalability and fast, predictable performance. It runs on solid state disks (SSDs) for low-latency response times, and there is no limit on the request capacity or storage size for a given table. This is because Amazon DynamoDB automatically partitions the input data and workload over a sufficient number of servers to meet the provided requirements. In contrast with its predecessor (SimpleDB), DynamoDB does not automatically build indexes on item attributes leading to more efficient insert, delete, and update operations, as well as improving the scalability of the system. Indexes can still be created if requested.

2.2.3 DISTRIBUTED COMPUTATION FRAMEWORKS: MAPREDUCE AND BEYOND

MapReduce is probably the best-known framework for massively parallel computation. Its roots in some sense can be traced back to the LISP [McCarthy, 1960] functional programming language; core concepts in functional programming are `map` (to distribute the computation of a function over a list of inputs) and `reduce` (to aggregate such function results). Inspired by the simplicity of this model, Google proposed the MapReduce framework [Dean and Ghemawat, 2004] for processing and generating large data sets. MapReduce resulted from
a need to easily parallelize tasks in a large cluster of commodity computers without deep knowledge of parallel and distributed systems. Users write MapReduce programs using the map and reduce primitive operations, and the framework is responsible for parallelizing the program, saving the user from tasks like resource allocation, synchronization, fault tolerance, etc.

A MapReduce program is defined by jobs, each of which consists of three main phases:

- A map phase, where the input is divided into sub-inputs, each handled by a different map task. The map task takes as input key/value pairs, processes them (by applying the operations defined by the user), and again outputs key/value pairs.

- A shuffle phase, where the key/value pairs emitted by the mappers are grouped and sorted by key, and are then assigned to reducers.

- A reduce phase, where each reduce task receives key/value pairs (sharing the same key) and applies further user-defined operations, writing the results into the file system.

To store inputs and outputs of MapReduce tasks, a distributed file system (e.g., GFS, HDFS, S3) is typically used.

Many recent massively parallel data management systems leverage MapReduce in order to build scalable query processors for both relational [Li et al., 2014] and RDF [Kaoudi and Manolescu, 2015] data. The most popular open-source implementation of MapReduce is provided by the Apache's Hadoop [Hadoop] framework, used by many RDF data management platforms [Goasdoué et al., 2015, Huang et al., 2011, Husain et al., 2011, Kim et al., 2011, Lee and Liu, 2013, Papailiou et al., 2012, 2013, 2014, Ravindra et al., 2011, Rohloff and Schantz, 2010, Schätzle et al., 2011, Wu et al., 2015].

Following the success of MapReduce proposal, other systems and models have emerged, which extend its expressive power and eliminate some of its shortcomings. Among the most well-known frameworks are the Apache projects Flink (previously known as Stratosphere) [Alexandrov et al., 2014] and Spark [Zaharia et al., 2010].

Spark
Spark achieves orders of magnitude better performance than Hadoop thanks to its main-memory resilient distributed dataset (RDD) [Zaharia et al., 2012]. RDDs are parallel data structures that let users persist intermediate results in memory and manipulate them using a rich set of operators. They provide fault tolerance via keeping lineage information. Spark’s programming model extends the MapReduce model by also including traditional relational operators (e.g., groupby, filter, join). Operations are either transformations or actions. Transformations (e.g., map, reduce, join) are performed in a lazy execution mode, i.e., they are not executed until an action operator (e.g., collect, count) is called. Spark has gained a lot of popularity for supporting interactive ad hoc batch analytics.

Flink
At the core of the Flink platform lies the PACT (Parallelization Contracts) parallel computing model [Battré et al., 2010], which can be seen as a generalization of MapReduce.
2.2. DISTRIBUTED STORAGE AND COMPUTING PARADIGMS

PACT plans are built of *implicitly parallel data processing operators* that are optimized and translated into *explicitly parallel data flows* by the Flink platform. PACT operators manipulate *records* made of several fields; each field can be either an atomic value or a list of records. Optionally, the records in a given record multiset\(^1\) may have an associated *key*, consisting of a subset of the atomic fields in those records.

The data input to/output by a PACT operator is stored in a (distributed) file. A *PACT plan* is a directed acyclic graph (DAG, in short) of operators, where each operator may have one or multiple inputs; this contrast with the “linear” pattern of MapReduce programs, where a single Reduce consumes the output of each Map. As Figure 2.6 shows, a PACT consists of a *parallelization contract*, a *user function* (UF in short), and possibly some *annotations* and *compiler hints* characterizing the UF behavior. The PACT parallelization contract describes how input records are organized into *groups* (prior to the actual processing performed by the operator). A simple possibility is to group the input records by the value(s) of some attribute(s), as customary in MapReduce, but in PACT other choices are also possible. The user function is executed independently over groups of records created by the parallelization contract; therefore these executions can take place in parallel. Finally, annotations and/or compiler hints may be used to enable optimizations (with no impact on the semantics), thus we do not discuss them further.

Although the PACT model allows creating custom parallelization contracts, a set of them for the most common cases is built in:

- The *Map* contract has a single input and builds a singleton for each input record.
- The *Reduce* contract also has a single input; it groups together all records that share the same key.
- The *Cross* contract builds the cartesian product of two inputs, that is: for each pair of records, one from each input, it produces a group containing these two records.
- The *Match* contract builds all pairs of records from its two inputs, having the same key value.
- The *CoGroup* contract can be seen as a “Reduce on two inputs;” it groups the records from both inputs, sharing the same key value.

\(^1\)Similarly to SQL and SPARL but differently from the classical relational algebra, PACT operates on bags (multisets) of records rather than sets.
2. PRELIMINARIES

Observe that PACT operators provide a level of abstraction above MapReduce by manipulating fine-granularity records, whereas MapReduce only distinguishes keys and values (but does not model the structure which may exist within them). Further, separating the input contract (which is only concerned with the grouping of input records) from the actual processing applied in parallel by the operators enables on one hand flexible adaptation to parallel processing, and on the other hand, smooth integration of user functions; this gives significant generality to the PACT model.

2.3 SUMMARY

We have presented all the necessary preliminaries for RDF and the cloud. Given these foundations, the reader should be able to follow the next chapters.

In summary, RDF data is organized in *triples* of the form \((s \ p \ o)\), stating that the subject \(s\) has the property (a.k.a. predicate) \(p\) whose value is the object \(o\). RDF data can also be seen as graphs with \(s\) and \(o\) being nodes connected with a directed edge from \(s\) to \(o\) labeled with \(p\). RDF Schema defines classes as groups of entities and properties as relations among classes. Its deductive rules lead to new information being inferred through a reasoning process. SPARQL is a declarative query language for exploring and querying RDF data which mainly consists of a set of triple patterns, i.e., triples which can contain variables.

To set the background for processing in the cloud we have introduced: (i) existing distributed file systems, such as HDFS and S3, (ii) the concept of distributed key-value stores and some stores commonly found in the cloud, such as HBase and DynamoDB, and (iii) distributed computation frameworks, such as MapReduce and Spark. All these components are used by the state-of-the-art distributed RDF data management systems that we will detail in the following.