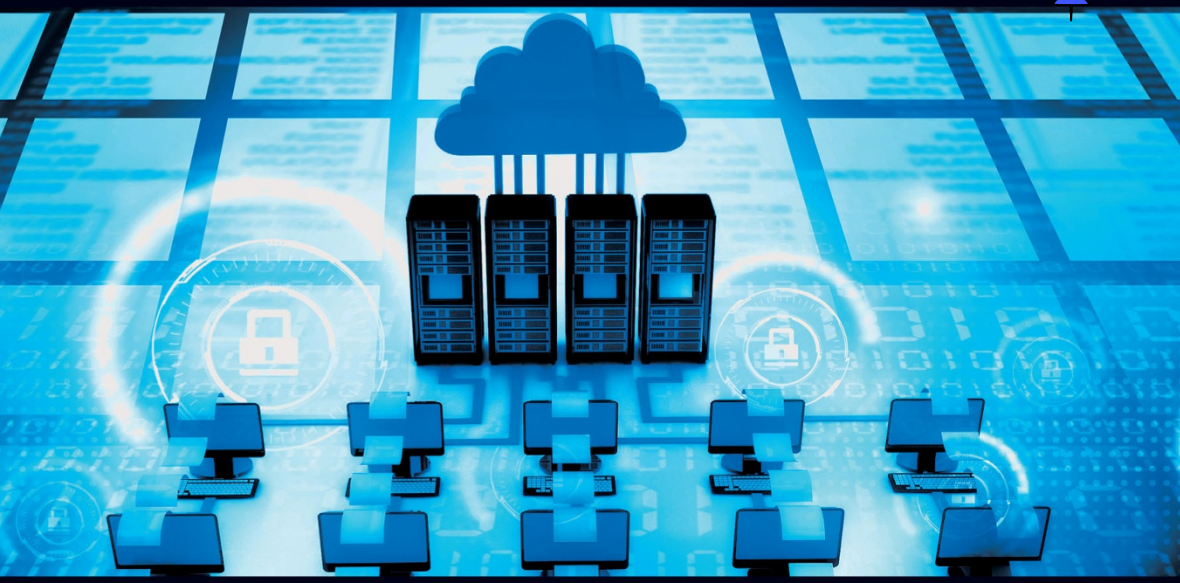
COMPUTER ENGINEERING SERIES

DATABASES AND BIG DATA SET



Volume 1

NoSQL Data Models

Trends and Challenges

Edited by
Olivier Pivert

ISTE

WILEY

NoSQL Data Models

Databases and Big Data Set

coordinated by
Dominique Laurent and Anne Laurent

Volume 1

NoSQL Data Models

Trends and Challenges

Edited by

Olivier Pivert

ISTE

WILEY

First published 2018 in Great Britain and the United States by ISTE Ltd and John Wiley & Sons, Inc.

ISTE Ltd
27-37 St George's Road
London SW19 4EU
UK

www.iste.co.uk

John Wiley & Sons, Inc.
111 River Street
Hoboken, NJ 07030
USA

www.wiley.com

Library of Congress Control Number: 2018941384

British Library Cataloguing-in-Publication Data
A CIP record for this book is available from the British Library
ISBN 978-1-78630-364-6

Contents

Foreword

This volume is part of a series entitled *Database and Big Data*, or DB & BD for short, whose content is motivated by the radical and rapid evolution (not to say *revolution*) of database systems during the last decade.

Indeed, since the 1970s, inspired by the relational database model, many research topics have emerged in the database community, such as, just to cite a few, Deductive Databases, Object-Oriented Databases, Semi-Structured Databases, Resource Description Framework (RDF), Open Data, Linked Data, Data Warehouses, Data Mining, and more recently, Cloud Computing, NoSQL and Big Data. Currently, the last three issues are becoming increasingly important and attract the most research efforts in the domain of databases.

Consequently, considering that Big Data environments are now to be handled in most current applications, the goal of this series is to address some of the latest issues in such environments. By doing so, while reporting on specific recent research results, we aim to provide readers with evidence that database technology is significantly changing, so as to face important challenges encountered in the majority of these applications.

More precisely, although relational databases are still commonly used in traditional applications, it is clear that most current Big Data applications cannot be handled by Relational DataBase Management Systems (RDBMSs), mainly because of the following reasons:

– there is a strong need to consider heterogeneous, structured, semi-structured or even unstructured data, for which no common schema exists.

RBMSs are not flexible enough to handle such variety of data, because these database systems were designed for handling tabular data;

– efficiency when facing Big Data in a distributed and replicated environment is now a key issue that RDBMSs fail to achieve, in particular when it comes to combining large tables.

New database systems have been proposed during the past few years, which are known under the generic term *NoSQL Databases*. These systems aim to solve the previous two points, and all claim to achieve their goal.

However, these systems need to be investigated further, because some important issues remain open (semantics of data, constraint satisfaction, transaction processing, privacy preservation, optimization, etc.). The volumes of this series aim to address some of these challenging issues and to present some of the most recent research results in this field.

Considering that the numerous currently available proposals are based on various concepts and data models (column-based, text-based, graph or hyper graph-based), this volume addresses issues related to trends and challenges related to NoSQL data models.

Anne LAURENT
Dominique LAURENT

Preface

As is well known, a major event in the field of data management was the introduction of the relational model by Codd in the early 1970s, which laid the foundations for a genuine theory of databases. After a somewhat slow start, due to the important Research and Development effort necessary to define efficient systems, relational database management systems reigned supreme for several decades.

However, around the end of the 20th Century, several phenomena modified the data management landscape. First, new types of applications in several domains were introduced to handle data for which the relational model appeared inadequate or inefficient. Typical examples are *semi-structured data* on the one hand, and *graphs* on the other (social networks, bibliographic databases, cartographic databases, genomic data, etc.) for which specific models and systems had to be designed. Second, a major event was the rise of the Semantic Web whose aim is, according to the W3C, to "provide a common framework that allows data to be shared and reused across application, enterprise and community boundaries". The Semantic Web uses models and languages specifically designed for linked data, which facilitate automated reasoning on such data. Besides, the amount of useful data in some application domains has become so huge that it cannot be stored or processed by traditional database solutions. This latter phenomenon is commonly referred to as *Big Data*. In terms of database technology, as a response to these new needs, we have seen the appearance of what have come to be called *NoSQL databases*.

The term NoSQL was coined by Carlo Strozzi in 1998, who designed a relational database system without SQL implementation and named it Strozzi NoSQL. However, this system is distinct from the circa-2009 general concept of NoSQL databases, which are typically non-relational. Many data models have been proposed: key-value stores, document stores (key-value stores that restrict values to semi-structured formats such as JSON), wide column stores, RDF, graph databases, XML, etc[1].

While the management of large volumes of data has always been subject to many research efforts, recent results in both the distributed systems and database communities have led to an important renewal of interest in this topic. Large scale distributed file systems such as Google File System[2] and parallel processing paradigm/environments such as MapReduce[3] have been the foundation of a new ecosystem with data management contributions in major database conferences and journals. Different (often open-source) systems have been released, such as Pig[4], Hive[5] or, more recently, Spark[6] and Flink[7], making it easier to use data center resources to manage Big Data. However, many research challenges remain, related, for instance, to system efficiency, and query language expressiveness and flexibility.

This book presents a sample of recent works by French research teams active in this domain. As the reader will see, it covers various aspects of NoSQL research, from semantic data management to graph databases, as well

1 GESSERT F., WOLFRAM W. FRIEDRICH S., "NoSQL database systems: a survey and decision guidance", *Computer Science - R&D*, vol. 32, nos 3–4, pp. 353–365, 2017.

2 GHEMAWAT S., GOBIOFF H., LEUNG S.-T., *Proceedings of the Symposium on Operating Systems Principles (SOSP)*, Bolton Landing, USA, pp. 29–43, 2003.

3 DEAN J., GHEMAWAT S., "MapReduce: simplified data processing on large clusters", *Communications of the ACM*, vol. 51, no. 1, pp. 107–113, 2008.

4 OLSTON C., REED B., SRIVASTAVA U. *et al.*, "Pig latin: a not-so-foreign language for data processing", *Proceedings of the SIGMOD International Conference on Management of Data*, Vancouver, Canada, pp. 1099–1110, 2008.

5 THUSOO A., SARMA J.S., JAIN N. *et al.*, "Hive – a petabyte scale data warehouse using Hadoop", *Proceedings of the International Conference on Data Engineering (ICDE)*, Long Beach, USA, pp. 996–1005, 2010.

6 ZAHARIA M., CHOWDHURY M., DAS T. *et al.*, "Resilient distributed datasets: a fault-tolerant abstraction for in-memory cluster computing", *Proceedings of the USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, San Jose, USA, pp. 15–28, 2012.

7 CARBONE P., KATSIFODIMOS A., EWEN S. *et al.*, "Apache Flink: Stream and Batch Processing in a Single Engine", *IEEE Data Engineering Bulletin*, vol. 4, pp. 28–38, 2015.

as Big Data management in cloud environments, dealing with data models, query languages and implementation issues. The book is organized as follows:

Chapter 1, by Kim Nguyễn, from LRI and the University of Paris-Sud, presents an overview of NoSQL languages and systems. The author highlights some of the technical aspects of NoSQL systems (in particular, distributed computation with MapReduce) before discussing current research trends: join implementations on top of MapReduce, models for NoSQL languages and systems, and the perspective that consists of defining a formal model of NoSQL databases and queries.

Chapter 2, entitled "Distributed SPARQL Query Processing: A Case Study with Apache SPARK", by Bernd Amann, Olivier Curé and Hubert Naacke, from the LIP6 laboratory in Paris, is devoted to the issue of evaluating SPARQL queries over large RDF datasets. The authors present a solution that consists of using the MapReduce framework to process SPARQL graph patterns and show how the general purpose cluster computing platform Apache Spark can be used to this end. They emphasize the importance of the physical data layer for query evaluation efficiency and show that hybrid query plans combining partitioned and broadcast joins improve query performances in almost all cases.

Chapter 3, authored by Manel Achichi, Mohamed Ben Ellefi, Zohra Bellahsene and Konstantin Todorov, from the LIRMM laboratory in Montpellier, is entitled "Doing Web Data: From Dataset Recommendation to Data Linking". It deals with the production of web data and focuses on the data linking stage, seen as an operation which generates a set of links between two different datasets. The authors first study the prior task which consists of discovering relevant datasets leading to the identification of similar resources to support the data linking issue. They provide an overview of recommendation approaches for candidate datasets, then present and classify the different techniques that are applied by the currently available data linking tools. The main challenge faced by all of these techniques is to overcome different heterogeneity problems that may occur between the considered datasets, such as differences in descriptions at different levels (value, ontological or logical) in order to compare the resources efficiently, and the authors show that further research efforts are still needed to better cope with these heterogeneity issues.

Chapter 4, entitled "Big Data Integration in Cloud Environments: Requirements, Solutions and Challenges", by Rami Sellami and Bruno Defude, from CETIC Charleroi and Telecom SudParis respectively, presents and discusses the requirements of Big Data integration in cloud environments. In such a context, applications may need to interact with several heterogeneous data stores, depending on the types of data they have to manage (traditional data, documents, graph data from social networks, simple key-value data, etc.). A first constraint is that, to make these interactions possible, programmers have to be familiar with different APIs. A second difficulty is that the execution of complex queries over heterogeneous data models cannot currently be achieved in a declarative way and therefore requires extra implementation efforts. Moreover, cloud discovery as well as application deployment and execution are generally performed manually by programmers. The authors analyze and discuss the current state-of-the-art regarding four requirements (automatic data stores selection and discovery, unique access for all data stores, transparent access for all data stores, global query processing and optimization), provide a global synthesis according to three groups of criteria, and highlight important challenges that remain to be tackled.

Chapter 5 is authored by Vijay Ingalalli, Dino Ienco and Pascal Poncelet, from the LIRMM laboratory in Montpellier, and is entitled "Querying RDF Data: A Multigraph-based Approach". In this chapter, the authors cope with two challenges faced by the RDF data management community: first, automatically generated queries cannot be bounded in their structural complexity and size; second, the queries generated by retrieval systems (or any other application) need to be efficiently answered in a reasonable amount of time. In order to address these challenges, the authors advocate an approach to RDF query processing that involves two steps: an offline step where the RDF database is transformed into a multigraph and indexed, and an online step where the SPARQL query is transformed into a multigraph too, which makes query processing boil down to a subgraph homomorphism problem. An RDF query engine based on this strategy is presented, named AMbER, which exploits structural properties of the multigraph query as well as the indices previously built on the multigraph structure.

Chapter 6 is entitled "Fuzzy Preference Queries to NoSQL Graph Databases" and is authored by Arnaud Castelltort, Anne Laurent, Olivier Pivert, Olfa Slama and Virginie Thion; the first two authors being affiliated to

the LIRMM laboratory in Montpellier, and the last three authors to the IRISA laboratory in Lannion. This chapter deals with flexible querying of graph databases that may involve gradual relationships. The authors first introduce an extension of attributed graphs where edges may represent a fuzzy concept (such as *friend* in the case of a social network, or *co-author* in the case of a bibliographic database). Then, they describe an extension of the query language Cypher that makes it possible to express fuzzy requirements, both on attribute values and on structural aspects of the graph (such as the length or the strength of a path). Finally, they deal with implementation issues and outline a query processing strategy based on the derivation of a regular Cypher query from the fuzzy query to be evaluated, through an add-on built on top of a classical graph database management system.

Finally, Chapter 7, by Cédric du Mouza and Nicolas Travers, from CNAM Paris, is entitled "Relevant Filtering in a Distributed Content-based Publish/Subscribe System", and deals with textual data management. More precisely, it considers a crucial challenge faced by Publish/Subscribe systems, which is to efficiently filter feeds' information in real time. Publish/Subscribe systems make it possible to subscribe to flows of items coming from diverse sources and notify the users according to their interests, but the existing systems hardly address the issue of *document relevance*. However, numerous sources may provide similar information, or a new piece of information may be "hidden" in a large flow. The authors introduce a real-time filtering process based on relevance that notably integrates the notions of *novelty* and *diversity*, and they show how this filtering process can be efficiently implemented in a NoSQL environment.

Olivier PIVERT
May 2018

1

NoSQL Languages and Systems

1.1. Introduction

1.1.1. *The rise of NoSQL systems and languages*

Managing, querying and making sense of data have become major aspects of our society. In the past 40 years, advances in technology have allowed computer systems to store vast amounts of data. For the better part of this period, relational database management systems (RDBMS) have reigned supreme, almost unchallenged, in their role of sole keepers of our data sets. RDBMS owe their success to several key factors. First, they stand on very solid theoretical foundations, namely the relational algebra introduced by Edgar F. Codd [COD 70], which gave a clear framework to express, in rigorous terms, the limit of systems, their soundness and even their efficiency. Second, RDBMS used one of the most natural representations to model data: tables. Indeed, tables of various sorts have been used since antiquity to represent scales, account ledgers, and so on. Third, a domain-specific language, SQL, was introduced almost immediately to relieve the database *user* from the burden of low-level programming. Its syntax was designed to be close to natural language, already highlighting an important aspect of data manipulation: people who can best make sense of data are not necessarily computer experts, and vice versa. Finally, in sharp contrast to the high level of data presentation and programming interface, RDBMS have always thrived to

Chapter written by Kim NGUYỄN.

NoSQL Data Models: Trends and Challenges, First Edition. Edited by Olivier Pivert.

offer the best possible performances for a given piece of hardware, while at the same time ensuring consistency of the stored data *at all times*.

At the turn of the year 2000 with the advances in high speed and mobile networks, and the increase in storage and computing capacity, the amount of data produced by humans became massive, and new usages were discovered that were impractical previously. This increase in both data volumes and computing power gave rise to two distinct but related concepts: "Cloud Computing" and "Big Data". Broadly speaking, the Cloud Computing paradigm consists of having data *processing* performed remotely in data centers (which collectively form the so-called cloud) and having end-user devices serve as terminals for information display and input. Data is accessed on demand and continuously updated. The umbrella term "Big Data" characterizes data sets with the so-called three "V"s [LAN 01]: Volume, Variety and Velocity. More precisely, "Big Data" data sets must be large (at least several terabytes), heterogeneous (containing both structured and unstructured textual data, as well as media files), and produced and processed at high speed. The concepts of both Cloud Computing and Big Data intermingle. The sheer size of the data sets requires some form of distribution (at least at the architecture if not at the logical level), preventing it from being stored close to the end-user. Having data stored remotely in a distributed fashion means the only realistic way of extracting information from it is to execute computation close to the data (i.e. remotely) to only retrieve the fraction that is relevant to the end-user. Finally, the ubiquity of literally billions of connected end-points that continuously capture various inputs feed the ever growing data sets.

In this setting, RDBMS, which were the be-all and end-all of data management, could not cope with these new usages. In particular, the so-called ACID (Atomicity, Consistency, Isolation and Durability) properties enjoyed by RDBMS transactions since their inception (IBM Information Management System already supported ACID transactions in 1973) proved too great a burden in the context of massively distributed and frequently updated data sets, and therefore more and more data started to be stored outside of RDBMS, in massively distributed systems. In order to scale, these systems traded the ACID properties for performance. A milestone in this area was the MapReduce paradigm introduced by Google engineers in 2004 [DEA 04]. This programming model consists of decomposing a high-level data operation into two phases, namely the *map* phase where the data is transformed locally on each node of the distributed system where it resides,

and the *reduce* phase where the outputs of the map phase are exchanged and migrated between nodes according to a partition key – all groups with the same key being migrated to the same (set of) nodes – and where an aggregation of the group is performed. Interestingly, such low-level operations where known both from the functional programming language community (usually under the name *map* and *fold*) and from the database community where the map phase can be used to implement *selection* and *projection*, and the *reduce* phase roughly corresponds to *aggregation*, *grouping* and *ordering*.

At the same time as the Big Data systems became prevalent, the so-called CAP theorem was conjectured [BRE 00] and proved [GIL 02]. In a nutshell, this formal result states that no distributed data store can ensure, at the same time, optimal Consistency, Availability and Partition tolerance. In the context of distributed data stores, *consistency* is the guarantee that a read operation will return the result of the most recent *global* write to the system (or an error). *Availability* is the property that every request receives a response that is not an error (however, the answer can be outdated). Finally, *partition tolerance* is the ability for the system to remain responsive when part of its components are isolated (due to network failures, for instance). In the context of the CAP theorem, the ACID properties enjoyed by RDBMS consist of favoring consistency over availability. With the rise of Big Data and associated applications, new systems emerged that favored availability over consistency. Such systems follow the BASE principles (Basically Available, Soft state and Eventual consistency). The basic tenets of the approach is that operations on the systems (queries as well as updates) must be as fast as possible and therefore no global synchronization between nodes of the system should occur at the time of operation. This, in turn, implies that after an operation, the system may be in an inconsistent state (where several nodes have different views of the global data set). The system is only required to *eventually* correct this inconsistency (the resolution method is part of the system design and varies from system to system). The wide design space in that central aspect of implementation gave rise to a large number of systems, each having its own programming interface. Such systems are often referred to with the umbrella term *Not only SQL* (NoSQL). While, generally speaking, NoSQL can also characterize XML databases and Graph databases, these define their own field of research. We therefore focus our study on various kinds of lower-level data stores.

1.1.2. *Overview of NoSQL concepts*

Before discussing the current trends in research on NoSQL languages and systems, it is important to highlight some of the technical concepts of such systems. In fact, it is their departure from well understood relational traits that fostered new research and development in this area. The aspects which we focus on are mainly centered around computational paradigms and data models.

1.1.2.1. *Distributed computations with MapReduce*

As explained previously, the *MapReduce* paradigm consists of decomposing a generic, high-level computation into a sequence of lower-level, distributed, *map* and *reduce* operations. Assuming some data elements are distributed over several nodes, the *map* operation is applied to each element individually, locally on the node where the element resides. When applied to such an element e, the *map* function may decide to either discard it (by not returning any result) or transform it into a new element e', to which is associated a *grouping key*, k, thus returning the pair (k, e'). More generally, given some input, the *map* operation may output any number of key-value pairs. At the end of the map phase, output pairs are exchanged between nodes so that pairs with the same key are grouped on the same node of the distributed cluster. This phase is commonly referred to as the *shuffle phase*. Finally, the *reduce* function is called once for each distinct key value, and takes as input a pair $(k, [e'_1, \ldots, e'_n])$ of a key and all the outputs of the map function that were associated with that key. The *reduce* function can then either discard its input or perform an operation on the set of values to compute a partial result of the transformation (e.g. by aggregating the elements in its input). The result of the *reduce* function is a pair (k', r) of an output key k' and a result r. The results are then returned to the user, sorted according to the k' key. The user may choose to feed such a result to a new pair of *map/reduce* functions to perform further computations. The whole MapReduce process is shown in Figure 1.1.

This basic processing can be optimized if the operation computed by the reduce phase is associative and commutative. Indeed, in such a case, it is possible to start the *reduce* operations on subsets of values present locally on nodes after the *map* phase, before running the shuffle phase. Such an operation is usually called a *combine* operation. In some cases, it can drastically improve performance since it reduces the amount of data moved

around during the *shuffle phase*. This optimization works particularly well in practice since the *reduce* operations are often aggregates which enjoy the commutativity and associativity properties (e.g. sum and average).

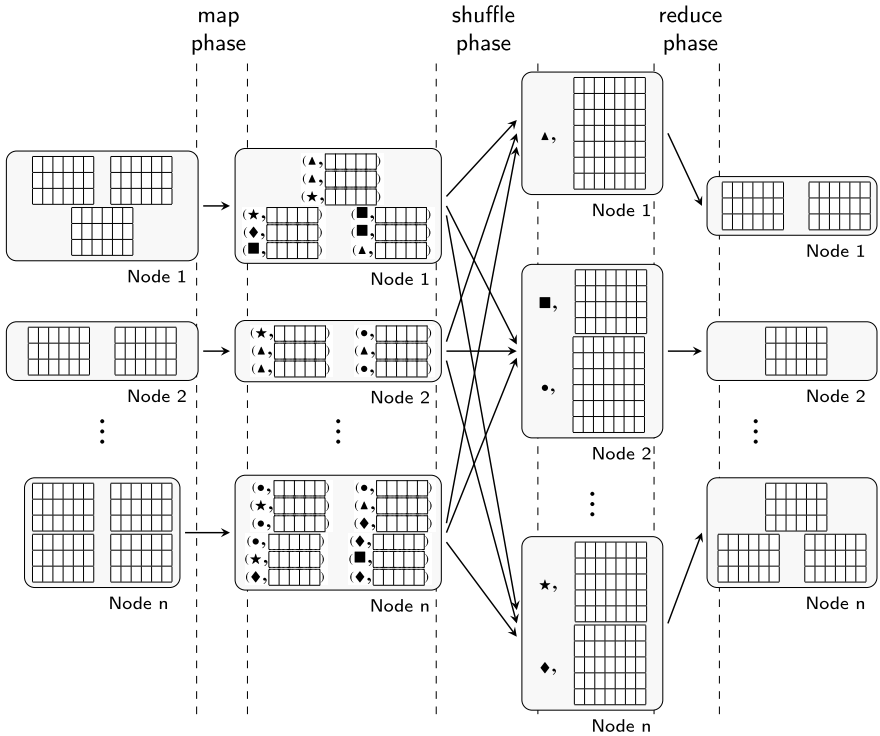


Figure 1.1. *MapReduce*

The most commonly used MapReduce implementation is certainly the Apache Hadoop framework [WHI 15]. This framework provides a Java API to the programmer, allowing us to express map and reduce transformations as Java methods. The framework heavily relies on the Hadoop Distributed File System (HDFS) as an abstraction for data exchange. The map and reduce transformations just read their input and write their output to the file system, which handle the lower-level aspects of distributing chunks of the files to the components of the clusters, and handle failures of nodes and replication of data.

1.1.2.2. *NoSQL databases*

A common trait of the most popular NoSQL databases in use is their nature as key-value stores. A key-value store is a database where collections are inherently *dictionaries* in which each entry is associated with a key that is unique to the collection. While it seems similar to a relational database where tables may have primary keys, key-value stores differ in a fundamental way from relational tables in that they do not rely on – nor enforce – a fixed *schema* for a given collection. Another striking aspect of all these databases is the relatively small set of operations that is supported natively. Updates are usually performed one element at a time (using the key to denote the element to be added, modified or deleted). Data processing operations generally consist of filtering, aggregation and grouping and exposing a MapReduce-like interface. Interestingly, most NoSQL databases do not support join operations, rather they rely on data *denormalization* (or materialized joins) to achieve similar results, at the cost of more storage usage and more maintenance effort. Finally, some databases expose a high-level, user-friendly query language (sometimes using an SQL compatible syntax) where queries are translated into combinations of lower-level operations.

1.1.3. *Current trends of French research in NoSQL languages*

NoSQL database research covers several domains of computer science, from system programming, networking and distributed algorithms, to databases and programming languages. We focus our study on the *language* aspect of NoSQL systems, and highlight two main trends in French research that pertains to NoSQL languages and systems.

The first trend aims to add support for well-known, relational operations to NoSQL databases. In particular, we survey the extensive body of work that has been done to add support for join operations between collections stored in NoSQL databases. We first describe how join operations are implemented in NoSQL systems (and in particular how joins can be decomposed into sequences of MapReduce operations).

The second trend of research is aimed at unifying NoSQL systems, in particular their query languages. Indeed, current applications routinely interact with several data stores (both relational and NoSQL), using high-level programming languages (PHP, Java, Ruby or JavaScript for Web applications,

Python and R for data analytics, etc.). We will survey some of the most advanced work in the area, particularly the definition of common, intermediate query language that can map to various data stores.

1.2. Join implementations on top of MapReduce

While NoSQL databases generally support a more flexible data model than relational ones, many users of NoSQL databases still use a somewhat flat and homogeneous encoding of data (i.e. what is stored in NoSQL databases is still mostly relational tables). In this respect, the join operation is still of paramount importance. Indeed, although denormalization is possible, it increases the cost of writing to the database (since the join must be maintained) and furthermore, such writes may leave the system inconsistent for a while (since, in general, no notion of transaction exists in NoSQL databases). As a result, a large body of work has been done recently to compute joins effectively on top of MapReduce primitives.

Before exploring some of the most prevalent work in this area, we recall the definition of the join operator. Let R and S be two collections[1], the join operation between R and S is defined as:

$$R \bowtie_\theta S = \{(r, s) \mid r \in R, s \in S \wedge \theta(r, s)\},$$

where θ is a Boolean predicate over r and s called the join condition. When θ is an equality condition between (parts of) r and s, the join is called an equijoin. Joins can be generalized to an arbitrary number of collections (n-way joins) and several variations of the basic join operator exist.

A straightforward way to implement joins is the so-called nested loop: which iterates over all r elements in R, and for each r, performs an iteration over all s elements in S, and tests whether $\theta(r, s)$ holds (for instance, see [RAM 03], Chapter 14). While this technique is often used by relational databases to evaluate joins, it cannot be used in the context of a MapReduce evaluation, since it is impossible to iterate over the whole collection (which is distributed over several nodes). In the context of equijoins, however, a distributed solution can be devised easily and is given in Algorithm 1.1.

1 Since the system at hand is not one of *relational* databases, we use the broad term "collection" rather than relation.

Algorithm 1.1. Reduce-side join

```
1:
2:  function MAP(Element elem, String origin)
3:      OUTPUT(Hash(elem), (elem, origin))
4:
5:
6:  function REDUCE(Integer key, List(Element,String) elems)
7:      l_1 ← {e | (e, "R") ∈ elems}
8:      l_2 ← {e | (e, "S") ∈ elems}
9:      for all r ∈ l_1 do
10:         for all s ∈ l_2 do
11:             if θ(r, s) then
12:                 OUTPUT(r, s)
```

To perform the join, we assume that the MAP function is applied to each element of either collection, together with a tag indicating its origin (a simple string with the name of the collection, for instance). The MAP function outputs a pair of a *key* and the original element and its origin. The key must be the result of a hashing (or partition) function that is compatible with the θ condition of the join, that is:

$$\theta(r, s) \Rightarrow \text{Hash}(r) = \text{Hash}(s).$$

During the shuffle phase of the MapReduce process, the elements are exchanged between nodes and all elements yielding the same hash value end up on the same node. The REDUCE function is then called on the key (the hash value) and the sequence of all elements that have this key. It then separates this input sequence with respect to the origin of the elements and can perform, on these two sequences, a nested loop to compute the join. This straightforward scheme is reminiscent of the Hash Join (e.g. see [RAM 03], Chapter 14) used in RDBMS. It suffers however from two drawbacks. The first is that it requires a hashing function that is compatible with the θ condition, which may prove difficult for conditions other than equality.

Second, and more importantly, the cost of data exchange in the shuffle phase may be prohibitive. These two drawbacks have given rise to a lot of research in recent years. A first area of research is to reduce the data exchange by filtering bad join candidates early, during the map phase. The second area is to develop *ad-hoc* MapReduce implementations for particular joins (where the particular semantics of the θ condition is used).

In [PHA 16] and [PHA 14], Phan *et al.* reviewed and extended the state of the art on filter-based joins. Filter-based joins discard non-joinable tuples early by using Bloom filters (named after their inventor, Burton H. Bloom [BLO 70]). A Bloom filter is a *compact* data structure that soundly approximates a set interface. Given a set S of elements and a Bloom filter F constructed from S, the Bloom filter can tell whether an element e is *not* part of the set or if it is present with a high probability, that is, the Bloom filter F is sound (it will never answer that an element not in S belongs to F) but not complete (an element present in F may be absent from S). The advantage of Bloom filters is their great compactness and small query time (which is a fixed parameter k that only depends on the precision of the filter, and not on the number of elements stored in it). The work of Phan *et al.* extends existing approaches by introducing *intersection filter-based joins* in which Bloom filters are used to compute equijoins (as well as other related operators such as semi-joins). Their technique consists of two phases. Given two collections R and S that must be joined on a common attribute x, a first pre-processing phase projects each collection on attribute x, collects both results in two Bloom filters F_{R_x} and F_{S_x} and computes the intersection filter $F_x = F_{R_x} \cap F_{S_x}$ which is very quick and easy. In practice, this filter is small enough to be distributed to all nodes. In a second phase, computing the distributed join, we may test during the map phase if the x attribute of the given tuple is in F_x, and, if not, discard it from the join candidates early. Phan *et al.* further extend their approach for multi-way joins and even recursive joins (which compute the transitive closure of the joined relations). Finally, they provide a complete cost analysis of their techniques, as well as others that can be used as a foundation for a MapReduce-based optimizer (they take particular care evaluating the cost of the initial pre-processing).

One reason why join algorithms may perform poorly is the presence of *data skew*. Such bias may be due to the original data, e.g. when the join attribute is not uniformly distributed, but may also be due to a bad distribution of the tuples among the existing nodes of the cluster. This observation was made early on in

the context of MapReduce-based joins by Hassan in his PhD thesis [ALH 09]. Hassan introduced special algorithms for a particular form of queries, namely `GroupBy-Join` queries, or SQL queries of the form:

SELECT $R.x, R.y, S.z, f(S.u)$
FROM R, S
WHERE $R.x = S.x$
GROUP BY $R.y, R.z$

Hassan gives variants of the algorithm for the case where the joined on attribute x is also part of the *GROUP BY* clause. While Hassan's work initially targeted distributed architectures, it was later adapted and specialized to the MapReduce paradigm (e.g. see [ALH 15]). While a detailed description of Hassan *et al.*'s algorithm (dubbed MRFAG-Join in their work) is outside of the scope of this survey, we give a high-level overview of the concepts involved. First, as for the reduce side join of Algorithm 1.1, the collections to be joined are distributed over all the nodes, and each tuple is tagged with its relation name. The algorithm then proceeds in three phases. The first phase uses one MapReduce operation to compute local histograms for the x values of R and S (recall that x is the joined on attribute). The histograms are local in the sense that they only take into account the tuples that are present on a given node. In the second phase, another MapReduce iteration is used to circulate the local histograms among the nodes and compute a global histogram of the frequencies of the pairs $(R.x, S.x)$. While this step incurs some data movements, histograms are merely a summary of the original data and are therefore much smaller in size. Finally, based on the global distribution that is known to all the nodes at the end of the second step, the third step performs the join as usual. However, the information about the global distribution is used cleverly in two ways: first, it makes it possible to filter out join candidates that never occur (in this regard, the global histogram plays the same role as the Bloom filter of Phan *et al.*); but second, the distribution is also used to counteract any data skew that would be present and distribute the set of sub-relations to be joined evenly among the nodes. In practice, Hassan *et al.* showed that early filtering coupled with good load balancing properties allowed their MRFAG-join algorithm to outperform the default evaluation strategies of a state-of-the-art system by a factor of 10.

Improving joins over MapReduce is not limited to equijoins. Indeed, in many cases, domain-specific information can be used to prune non-joinable candidates early in the MapReduce process. For instance, in [PIL 16],

Pilourdault *et al.* considered the problem of computing top-k temporal joins. In the context of their work, relations R and S are joined over an attribute x denoting a time interval. Furthermore, the join conditions involve high-level time functions such as *meet*$(R.x, S.x)$ (the $R.x$ interval finishes exactly when the $S.x$ interval starts), *overlaps*$(R.x, S.x)$ (the two interval intersects), and so on. Finally, the time functions are not interpreted as Boolean predicates, but rather as scoring functions, and the joins must return the top k scoring pairs of intervals for a given time function. The solution which they adopt consists of an initial offline, query-independent pre-processing step, followed by two MapReduce phases that answer the query. The offline pre-processing phase partitions the time into consecutive *granules* (time intervals), and collects statistics over the distribution of the time intervals to be joined among each granule. At query time, a first MapReduce process distributes the data using the statistics computed in the pre-processing phase. In a nutshell, granules are used as reducers' input keys, which allows a reducer to process all intervals that occurred during the same granule together. Second, bounds for the scoring time function used in the query are computed and intervals that have no chance of being in the top-k results are discarded early. Finally, a final MapReduce step is used to compute the actual join among the reduced set of candidates.

In the same spirit, Fang *et al.* considered the problem of nearest-neighbor joins in [FAN 16]. The objects considered in this work are *trajectories*, that is, sequences of triple (x, y, t) where (x, y) are coordinates in the Euclidean plane and t a time stamp (indicating that a moving object was at position (x, y) at a time t). The authors focus on k nearest-neighbor joins, that is, given two sets of trajectories R and S, and find for each element of R, the set of k closest trajectories of S. The solution proposed by the authors is similar in spirit to the work of Pilourdault *et al.* on temporal joins. An initial pre-processing step first partitions the time in discrete consecutive intervals and then the space in rectangles. Trajectories are discretized and each is associated with the list of time interval and rectangles it intersects. At query time, the pair of an interval and rectangle is used as a partition key to assign pieces of trajectories to reducers. Four MapReduce stages are used, the first three collect statistics, perform early pruning of non-joinable objects and distribution over nodes, and the last step is – as in previously presented work – used to perform the join properly.

Apart from the previously presented works which focus on binary joins (and sometimes provide algorithms for ternary joins), Graux *et al.* studied the problem of n-ary equijoins [GRA 16] in the context of SPARQL [PRU 08] queries. SPARQL is the standard query language for the so-called semantic Web. More precisely, SPARQL is a W3C standardized query language that can query data expressed in the Resource Description Framework (RDF) [W3C 14]. Informally, the RDF data are structured into the so-called triples of a *subject*, a *predicate* and an *object*. These triples allow *facts* about the World to be described. For instance, a triple could be (`"John"`, `"lives in"`, `"Paris"`) and another one could be (`"Paris"`, `"is in"`, `"France"`). Graux *et al.* focused on the query part of SPARQL (which also allows new triple sets to be reconstructed). SPARQL queries rely heavily on joins of triples. For instance, the query:

```
SELECT ?name ?town
WHERE {
 ?name "lives in" ?town .
 ?town "is in" "France"
}
```

returns the pair of the name of a person and the city they live in, for all cities located in France (the "`.`" operator in the query acts as a conjunction). As we can see, the more triples with free variables in the query, the more joins there are to process. Graux *et al.* showed in their work how to efficiently store such triple sets on a distributed file system and how to translate a subset of SPARQL into Apache Spark code. While they use Spark's built-in join operator (which implements roughly the reduce side join of Algorithm 1.1), Graux *et al*. made a clever use of statistics to find an optimal order for join evaluations. This results in an implementation that outperforms both state-of-the-art native SPARQL evaluators as well as other NoSQL-based SPARQL implementations on popular SPARQL benchmarks. Finally, they show how to extend their fragment of SPARQL with other operators such as union.

1.3. Models for NoSQL languages and systems

A striking aspect of the NoSQL ecosystem is its diversity. Concerning data stores, we find at least a dozen heavily used solutions (MongoDB, Apache Cassandra, Apache HBase, Apache CouchDB, Redis, Microsoft CosmosDB to name a few). Each of these solutions comes with its integrated query

interface (some with high-level query languages, others with a low-level operator API). But besides data store, we also find processing engines such as Apache Hadoop (providing a MapReduce interface), Apache Spark (general cluster computing) or Apache Flink (stream-oriented framework), and each of these frameworks can target several of the aforementioned stores. This diversity of solutions translates to ever more complex application code, requiring careful and often brittle or inefficient abstractions to shield application business logic from the specificities of every data store. Reasoning about such programs, and in particular about their properties with respect to data access has become much more complex. This state of affairs has prompted a need for unifying approaches allowing us to target multiple data stores uniformly.

A first solution is to consider SQL as the unifying query language. Indeed, SQL is a well-known and established query language, and being able to query NoSQL data stores with the SQL language seems natural. This is the solution proposed by Curé *et al.* [CUR 11]. In this work, the database user queries a "virtual relational database" which can be seen as a *relational view* of different NoSQL stores. The approach consists of two complementary components. The first one is a *data mapping* which describes which part of a NoSQL data store is used to populate a virtual relation. The second is the *Bridge Query Language* (BQL), an intermediate query representation that bridges the gap between the high-level, declarative SQL, and the low-level programming API exposed by various data stores. The BQL makes some operations, such as iteration or sorting, explicit. In particular, BQL exposes a *foreach* construct that is used to implement the SQL join operator (using nested loops). A BQL program is then translated into the appropriate dialect. In [CUR 11], the authors give two translations: one targeting MongoDB and the other targeting Apache Cassandra, using their respective Java API.

While satisfactory from a design perspective, the solution of Curé *et al.* may lead to sub-optimal query evaluation, in particular in the case of joins. Indeed, in most frameworks (with the notable exception of Apache Spark), performing a double nested loop to implement a join implies that the join is actually performed on the *client* side of the application, that is, both collections to be joined are retrieved from the data store and joined in main memory. The most advanced contribution to date that provides not only a unified query language and data model, but also a robust query planner is the work of Kolev *et al.* on CloudMdsQL [KOL 16b]. At its heart, CloudMdsQL is a query language based on SQL, which is extended in two ways. First, a

CloudMdsQL program may reference a table from a NoSQL store using the store's native query language. Second, a CloudMdsQL program may contain blocks of Python code that can either produce synthetic tables or be used as user-defined functions (UDFs) to perform application logic. A programmer may query several data stores using *SELECT* statements (the full range of SQL's *SELECT* syntax is supported, including joins, grouping, ordering and windowing constructs) that can be arbitrarily nested. One of the main contributions of Kolev *et al.* is a modular query planner that takes each data store's capability into account and furthermore provides some cross data store optimizations. For instance, the planner may decide to use *bind joins* (see [HAA 97]) to efficiently compute a join between two collections stored in different data stores. With a bind join, rather than retrieving both collections on the client side and performing the join in main memory, one of the collections is migrated to the other data store where the join computation takes place. Another example of optimization performed by the CloudMdsQL planner is the rewriting of Python *for each* loops into plain database queries. The CloudMdsQL approach is validated by a prototype and an extensive benchmark [KOL 16a].

One aspect of CloudMdsQL that may still be improved is that even in such a framework, reasoning about programs is still difficult. In particular, CloudMdsQL is not so much a unified query language than the juxtaposition of SQL's *SELECT* statement, Python code and a myriad of *ad-hoc* foreign expressions (since every data manipulation language can be used inside quotations). A more unifying solution, from the point of view of the intermediate query representation, is the Hop.js framework of Serrano *et al.* [SER 16]. Hop.js is a multi-tier programming environment for Web application. From a single JavaScript source file, the framework deduces both the view (HTML code), the client code (client-side JavaScript code) and the server code (server-side JavaScript code with database calls), as well as automatically generating server/client communications in the form of asynchronous HTTP requests. More precisely, in [COU 15], Serrano *et al.* applied the work of Cheney *et al.* [CHE 13b, CHE 13a, CHE 14] on language-integrated queries to Hop.js. In [COU 15], *list comprehension* is used as a common query language to denote queries to different data stores.

More precisely, borrowing the Array comprehension syntax[2] of the Ecmascript 2017 proposal, the author can write queries as:

```
[ for ( x of table )
      if ( x.age >= 18 ) { name: x.name, age: x.age } ]
```

which mimics the mathematical notation of set comprehension:

$$\{(x.\mathit{name}, x.\mathit{age}) \mid x \in \mathit{Table} \wedge x.\mathit{age} \geq 18\}$$

In this framework, joins may be written as nested for loops. However, unlike the work of Curé *et al.*, array comprehension is compiled into more efficient operations if the target back-end supports them.

Despite their variety of data models, execution strategies and query languages, NoSQL systems seem to agree on one point: their lack of support for schema! As is well-known, the lack of schema is detrimental to both readability and performance (for instance, see the experimental study of the performance and readability impact of data modeling in MongoDB by Gómez *et al.* [GÓM 16]). This might come as a surprise, database systems have a long tradition of taking types seriously (from the schema and constraints of RDBMS to the various schema standards for XML documents and the large body of work on type-checking XML programs). To tackle this problem, Benzaken *et al.* [BEN 13] proposed a core calculus of operators, dubbed *filters*. Filters reuse previous work on semantic sub-typing (developed in the context of static type checking of XML transformations [FRI 08]) and make it possible to: (i) model NoSQL databases using regular types and extensible records, (ii) give a formal semantics to NoSQL query languages and (iii) perform type checking of queries and programs accessing data. In particular, the work in [BEN 13] gives a formal semantics of the JaQL query language (originally introduced in [BEY 11] and now part of IBM BigInsights) as well as a precise type-checking algorithm for JaQL programs. In essence, JaQL programs are expressed as sets of mutually recursive functions and such functions are symbolically executed over the *schema* of the data to compute an output type of the query. The filter calculus was generic enough to encode not only JaQL but also MongoDB's query language (see [HUS 14]). One of the downsides of the filter approach, however, is that

2 Array comprehensions have been retired from EcmaScript2017 (ES8) standard.

to be generic enough to express any kind of operator, filters rely on low-level building blocks (such as recursive functions and pattern matching) which are not well-suited for efficient evaluation.

1.4. New challenges for database research

Since their appearance at the turn of the year 2000, NoSQL databases have become ubiquitous and collectively store a large amount of data. In contrast with XML databases, which rose in popularity in the mid-1990s to settle on specific, document-centric applications, it seems safe to assume that NoSQL databases are here to stay, alongside relational ones. After a first decade of fruitful research in several directions, it seems that it is now time to unify all these research efforts.

First and foremost, in our sense, a formal model of NoSQL databases and queries is yet to be defined. The model should play the same role that relational algebra played as a foundation for SQL. This model should in particular allow us to:

– describe the data-model precisely;

– express complex queries;

– reason about queries and their semantics. In particular, it should allow us to reason about query equivalence;

– describe the cost model of queries;

– reason about meta-properties of queries (type soundness, security properties, for instance, non-interference, access control or data provenance);

– characterize high-level optimization.

Finding such a model is challenging in many ways. First, it must allow us to model data as it exists in current – and future – NoSQL systems, from the simple key-value store, to the more complex document store, while at the same time retaining compatibility with the relational model. While at first sight the *nested relational algebra* seems to be an ideal candidate (see, for instance, [ABI 84, FIS 85, PAR 92]), it does not allow us to easily model heterogeneous collections which are common in NoSQL data stores. Perhaps an algebra based on nested data types with extensible records similar to [BEN 13] could be of

use. In particular, it has already been used successfully to model collections of (nested) heterogeneous JSON objects.

Second, if a realistic cost model is to be devised, the model might have to make the *distributed* nature of data explicit. This distribution happens at several levels: first, collections are stored in a distributed fashion, and second, computations may also be performed in a distributed fashion. While process calculi have existed for a long time (for instance, the one introduced by Milner *et al.* [MIL 92]), they do not seem to tackle the data aspect of the problem at hand.

Another challenge to be overcome is the interaction with high-level programming languages. Indeed, for database-oriented applications (such as Web applications), programmers still favor directly using a query language (such as SQL) with a language API (such as Java's JDBC) or higher-level abstractions such as Object Relational Mappings, for instance. However, data analytic oriented applications favor idiomatic R or Python code [GRE 15, VAN 17, BES 17]. This leads to inefficient idioms (such as retrieving the bulk of data on the client side to filter it with R or Python code). Defining *efficient*, truly language-integrated queries remains an unsolved problem. One critical aspect is the server-side evaluation of user-defined functions, written in Python or R, close to the data *and* in a distributed fashion. Frameworks such as Apache Spark [ZAH 10], which enable data scientists to write efficient idiomatic R or Python code, do not allow us to easily reason about security, provenance or performance (in other words, they lack formal foundations). A first step toward a unifying solution may be the work of Benzaken *et al.* [BEN 18]. In this work, following the tradition of compiler design, an intermediate representation for queries is formally defined. This representation is an extension of the λ-calculus, or equivalently of a small, pure functional programming language, extended with data operators (e.g. joins and grouping). This intermediate representation is used as a common compilation target for high-level languages (such as Python and R). Intermediate terms are then translated into various back-ends ranging from SQL to MapReduce-based databases. This preliminary work seems to provide a good framework to explore the design space and address the problems mentioned in this conclusion.

Finally, while some progress has been made in implementing high-level operators on top of distributed primitives such as MapReduce, and while all

these approaches seem to fit a similar template (in the case of join: prune non-joinable items early and regroup likely candidates while avoiding duplication as much as possible), it seems that some avenues must be explored to unify and formally describe such low-level algorithms, and to express their cost in a way that can be reused by high-level optimizers.

In conclusion, while relational databases started both from a formal foundation and solid implementations, NoSQL databases have developed rapidly as implementation artifacts. This situation highlights its limits and, as such, database and programming language research aims to 'correct' it in this respect.

1.5. Bibliography

[ABI 84] ABITEBOUL S., BIDOIT N., "Non first normal form relations to represent hierarchically organized data", *Proceedings of the 3rd ACM SIGACT-SIGMOD Symposium on Principles of Database Systems*, PODS'84, New York, USA, pp. 191–200, 1984.

[ALH 09] AL HAJJ HASSAN M., Parallelism and load balancing in the treatment of the join on distributed architectures, PhD thesis, University of Orléans, December 2009.

[ALH 15] AL HAJJ HASSAN M., BAMHA M., "Towards scalability and data skew handling in group by-joins using map reduce model", *International Conference On Computational Science - ICCS 2015, 51*, Procedia Computer Science, Reykjavik, Iceland, pp. 70–79, June 2015.

[APA 17a] APACHE SOFTWARE FOUNDATION, Apache Hadoop 2.8, 2017.

[APA 17b] APACHE SOFTWARE FOUNDATION, Apache Spark, 2017.

[BEN 13] BENZAKEN V., CASTAGNA G., NGUYỄN K. *et al.*, "Static and dynamic semantics of NoSQL languages", *SIGPLAN Not.*, ACM, vol. 48, no. 1, pp. 101–114, January 2013.

[BEN 18] BENZAKEN V., CASTAGNA G., DAYNÈS L. *et al.*, "Language-integrated queries: a BOLDR approach", *The Web Conference 2018*, Lyon, France, April 2018.

[BES 17] BESSE P., GUILLOUET B., LOUBES J.-M., "Big data analytics. Three use cases with R, Python and Spark", in MAUMY-BERTRAND M., SAPORTA G., THOMAS-AGNAN C. (eds), *Apprentissage Statistique et Données Massives*, Journées d'Etudes en Statistisque, Technip, 2017.

[BEY 11] BEYER K.S., ERCEGOVAC V., GEMULLA R. *et al.*, "Jaql: a scripting language for large scale semistructured data analysis", *PVLDB*, vol. 4, no. 12, pp. 1272–1283, 2011.

[BLO 70] BLOOM B.H., "Space/time trade-offs in hash coding with allowable errors", *Communication ACM*, ACM, vol. 13, no. 7, pp. 422–426, July 1970.

[BRE 00] BREWER E.A., "Towards robust distributed systems (abstract)", *Proceedings of the Nineteenth Annual ACM Symposium on Principles of Distributed Computing*, ACM, New York, USA, p. 7, 2000.

[CHE 13a] CHENEY J., LINDLEY S., RADANNE G. *et al.*, "Effective quotation", *CoRR*, vol. abs/1310.4780, 2013.

[CHE 13b] CHENEY J., LINDLEY S., WADLER P., "A practical theory of language-integrated query", *SIGPLAN Not.*, ACM, vol. 48, no. 9, pp. 403–416, September 2013.

[CHE 14] CHENEY J., LINDLEY S., WADLER P., "Query shredding: Efficient relational evaluation of queries over nested multisets (extended version)", *CoRR*, vol. abs/1404.7078, 2014.

[COD 70] CODD E.F., "A relational model of data for large shared data banks", *Communication ACM*, ACM, vol. 13, no. 6, pp. 377–387, June 1970.

[COU 15] COUILLEC Y., SERRANO M., "Requesting heterogeneous data sources with array comprehensions in Hop.js", *Proceedings of the 15th Symposium on Database Programming Languages*, ACM, Pittsburgh, United States, p. 4, October 2015.

[CUR 11] CURÉ O., HECHT R., LE DUC C. *et al.*, "Data integration over NoSQL stores using access path based mappings", *DEXA 2012, 6860 Lecture Notes in Computer Science*, Toulouse, France, pp. 481–495, August 2011.

[DEA 04] DEAN J., GHEMAWAT S., "MapReduce: simplified data processing on large clusters", *Proceedings of the 6th Conference on Symposium on Opearting Systems Design & Implementation - Volume 6*, USENIX Association, Berkeley, USA, p. 10, 2004.

[FAN 16] FANG Y., CHENG R., TANG W. *et al.*, "Scalable algorithms for nearest-neighbor joins on big trajectory data", *IEEE Transactions on Knowledge and Data Engineering*, Institute of Electrical and Electronics Engineers, vol. 28, no. 3, 2016.

[FIS 85] FISCHER P.C., SAXTON L.V., THOMAS S.J. *et al.*, "Interactions between dependencies and nested relational structures", *Journal of Computer and System Sciences*, vol. 31, no. 3, pp. 343–354, 1985.

[FRI 08] FRISCH A., CASTAGNA G., BENZAKEN V., "Semantic subtyping: dealing set-theoretically with function, union, intersection, and negation types", *Journal ACM*, vol. 55, no. 4, pp. 19:1–19:64, September 2008.

[GIL 02] GILBERT S., LYNCH N., "Brewer's conjecture and the feasibility of consistent, available, partition-tolerant web services", *SIGACT News*, vol. 33, no. 2, pp. 51–59, June 2002.

[GÓM 16] GÓMEZ P., CASALLAS R., RONCANCIO C., "Data schema does matter, even in NoSQL systems!", *2016 IEEE Tenth International Conference on Research Challenges in Information Science (RCIS)*, Grenoble, France, June 2016.

[GRA 16] GRAUX D., JACHIET L., GENEVÈS P. *et al.*, "SPARQLGX: efficient distributed evaluation of SPARQL with apache spark", *The 15th International Semantic Web Conference*, Kobe, Japan, October 2016.

[GRE 15] GREENFIELD P., Keynote speech at PyData 2015: How Python Found its way into Astronomy, New York, USA, 2015.

[HAA 97] HAAS L.M., KOSSMANN D., WIMMERS E.L. *et al.*, "Optimizing queries across diverse data sources", *Proceedings of the 23rd International Conference on Very Large Data Bases*, VLDB'97, San Francisco, USA, pp. 276–285, 1997.

[HUS 14] HUSSON A., "Une sémantique statique pour MongoDB", *Journées francophones des langages applicatifs 25, Fréjus*, France, pp. 77–92, January 8–11 2014.

[KOL 16a] KOLEV B., PAU R., LEVCHENKO O. *et al.*, "Benchmarking polystores: the CloudMdsQL experience", in GADEPALLY V. (ed.), *IEEE BigData 2016: Workshop on Methods to Manage Heterogeneous Big Data and Polystore Databases*, IEEE Computing Society, Washington D.C., United States, December 2016.

[KOL 16b] KOLEV B., VALDURIEZ P., BONDIOMBOUY C. *et al.*, "CloudMdsQL: querying heterogeneous cloud data stores with a common language", *Distributed and Parallel Databases*, vol. 34, no. 4, pp. 463–503, December 2016.

[LAN 01] LANEY D., 3D Data Management: Controlling Data Volume, Velocity, and Variety, Report, META Group, February 2001.

[MIL 92] MILNER R., PARROW J., WALKER D., "A calculus of mobile processes, I", *Information and Computation*, vol. 100, no. 1, pp. 1–40, 1992.

[PAR 92] PAREDAENS J., VAN GUCHT D., "Converting nested algebra expressions into flat algebra expressions", *ACM Transaction Database Systems*, vol. 17, no. 1, pp. 65–93, March 1992.

[PHA 14] PHAN T.-C., Optimization for big joins and recursive query evaluation using intersection and difference filters in MapReduce, Thesis, Blaise Pascal University, July 2014.

[PHA 16] PHAN T.-C., D'ORAZIO L., RIGAUX P., "A theoretical and experimental comparison of filter-based equijoins in MapReduce", *Transactions on Large-Scale Data- and Knowledge-Centered Systems XXV, 9620 Lecture Notes in Computer Science*, pp. 33–70, 2016.

[PIL 16] PILOURDAULT J., LEROY V., AMER-YAHIA S., "Distributed evaluation of top-k temporal joins", *Proceedings of the 2016 International Conference on Management of Data, SIGMOD'16*, New York, USA, pp. 1027–1039, 2016.

[RAM 03] RAMAKRISHNAN R., GEHRKE J., *Database Management Systems*, McGraw-Hill, New York, 3rd ed., 2003.

[SER 16] SERRANO M., PRUNET V., "A Glimpse of Hopjs", *International Conference on Functional Programming (ICFP)*, ACM, Nara, Japan, p. 12, September 2016.

[VAN 17] VANDERPLAS J., Keynote speech at PyCon 2017, 2017.

[W3C 13] W3C, SPARQL 1.1 overview, 2013.

[W3C 14] W3C, RDF 1.1 Concepts and Abstract Syntax, 2014.

2

Distributed SPARQL Query Processing: a Case Study with Apache Spark

2.1. Introduction

The Semantic Web is rapidly growing, generating large volumes of Resource Description Framework (RDF) data [W3C 14] stored in the Linked Open Data (LOD) cloud. With data sets ranging from hundreds of millions to billions of triples, RDF triple stores are expected to meet properties such as scalability, high availability, automatic work distribution and fault tolerance. This chapter is dedicated to the problem of evaluating SPARQL queries over large RDF datasets. Section 2.2 introduces the RDF data model and the SPARQL query language. The challenges and solutions for efficiently processing SPARQL queries and in particular basic graph pattern (BGP) expressions are presented in section 2.3. The specific solution using the MapReduce framework for processing SPARQL graph patterns [DEA 04] is introduced in section 2.4. The chapter concludes with section 2.5, describing the use of Apache Spark and explaining the importance of the physical data layers for the query performance.

Chapter written by Bernd AMANN, Olivier CURÉ and Hubert NAACKE.

NoSQL Data Models: Trends and Challenges, First Edition. Edited by Olivier Pivert.

2.2. RDF and SPARQL

2.2.1. *RDF framework and data model*

The Resource Description Framework (RDF) is a family of W3C specifications for linking and sharing data on the Web. RDF was originally designed as a metadata model for describing and linking any kind of web resources, for example web pages, images, audio and video files, and services. It includes a graph-oriented abstract data model with optional schema definitions and a high-level declarative query language called SPARQL. RDF shares many features of semi-structured data models [ABI 00] for modeling and structuring heterogeneous information in open environments such as the Web. In particular, RDF data graphs can be produced without a predefined schema and SPARQL allows querying both schema and instance information simultaneously.

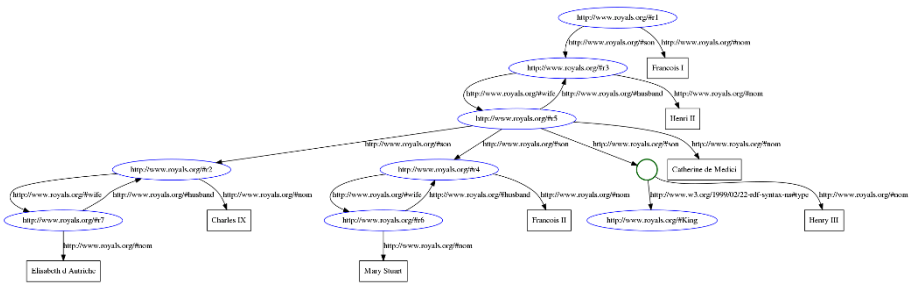


Figure 2.1. *RDF graph. For a color version of this figure, see www.iste.co.uk/pivert/nosql.zip*

EXAMPLE 2.1.– *Figure 2.1 shows an RDF graph of French kings and queens. Following the principle of other web standards, nodes (resources) and edges (properties) are identified by URIs, and information sharing is simplified by the possibility of using different serializations (XML[1] , Turtle[2] and N-Triples[3]) for encoding the graph structure. For example, the following Turtle document encodes the graph in Figure 2.1:*

1 https://www.w3.org/XML/

2 https://www.w3.org/TR/turtle/

3 https://www.w3.org/2001/sw/RDFCore/ntriples/

```
@prefix :     <http://www.royals.org/#> .
@prefix rdf:  <http://www.w3.org/1999/02/22−rdf−syntax−ns#> .

:r1 rdf:type :King; :son  :r3 ; :name "Francois␣I" .
:r2 rdf:type :King; :wife :r7 ; :name "Charles␣IX" .
:r3 rdf:type :King; :wife :r5 ; :name "Henri␣II" .
:r4 rdf:type :King; :wife :r6 ; :name "Francois␣II" .
:r5 rdf:type :Queen; :husband :r3; :name "Catherine␣de␣Medici";
    :son :r4 ,:r2, [ rdf:type :King; :name "Henry␣III" ] .
:r6 rdf:type :Queen; :husband :r4; :name "Mary␣Stuart" .
:r7 rdf:type :Queen; :husband :r2; :name "Elisabeth␣d␣Autriche" .
```

The first two lines define to XML namespace prefix `:` *and* `rdf:` *for identifying the collections of resources and properties. The third line defines three properties for resource* `:r1` *in namespace* `http://www.royals.org/#`. *Property* `rdf:type` *defines resource* `:r1` *as an RDF instance of class* `:King`, *resource* `:r3` *is the* `:son` *of* `:r1` *and property* `:name` *denotes the name of* `:r1` *by a literal value. Formally, this line can be translated into three Subject-Property-Object (SPO) triples sharing the same subject* `http://www.royals.org/#r1`:

```
<http://www.royals.org/#r1> <http://www.w3.org/1999/02/22−rdf−syntax−ns#type>
                            <http://www.royals.org/#King> .
<http://www.royals.org/#r1> <http://www.royals.org/#:son>
                            <http://www.royals.org/#r3> .
<http://www.royals.org/#r1> <http://www.royals.org/#name> "Francois␣I" .
```

2.2.1.1. *RDF schema*

RDF schemas (RDF/S) are used for *validating* data and for *inferring* new data (triples). This inference is formally defined by logical RDF/RDF/S entailment rules. This simple schema inference makes RDF a fundamental part of the Semantic Web and linked open data initiatives since it enables incremental data and knowledge integration, resource linking and semantic resource annotation.

EXAMPLE 2.2.– *The following document defines an RDF schema for data sets about royalties (namespace definitions are omitted).*

```
:King  rdfs:subClassOf :RoyalPerson , :Man .
:Queen rdfs:subClassOf :RoyalPerson , :Woman .

:child rdfs:domain :RoyalPerson ; rdfs:range :RoyalPerson .
:son   rdfs:domain :RoyalPerson ; rdfs:range :Man .
       rdfs:subPropertyOf :child ;
:name  rdfs:domain :RoyalPerson ; rdfs:range rdfs:Literal .
:husband rdfs:domain :Woman     ; rdfs:range :Man .
:wife  rdfs:domain :Man         ; rdfs:range :Woman .
```

2.2.1.2. *RDF/S entailment*

RDF/S entailment rules allow us to infer that `:RoyalPerson`, `:Man` and `:Woman` are classes (property `rdfs:subClassOf` only exists between RDF schema classes), and the son of a `:RoyalPerson` is an instance of two classes, `:Man` (range of `:son`) and `:RoyalPerson` (range of `:child`). We can recursively infer that `:r4` is a `:RoyalPerson` (`:King` is a subclass of `:RoyalPerson`) and `:Man` (range of property `:husband`). An extract of the saturated data set obtained after this entailment process is shown in Figure 2.2. Schema resources are in gray and `rdf:type` property edges are in red. The son of Catherine de Medici is a blank node (the green circle), i.e. an unidentified resource.

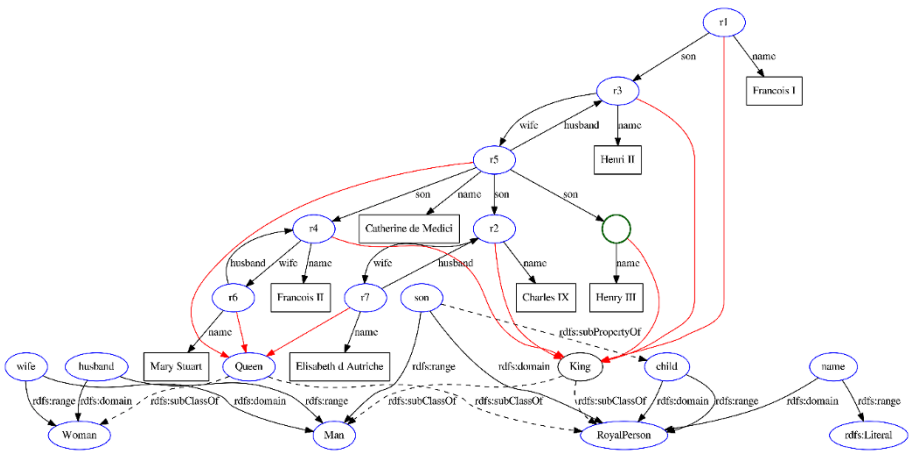


Figure 2.2. *Saturated data set `kingsandqueens.ttl`. For a color version of this figure, see www.iste.co.uk/pivert/nosql.zip*

2.2.2. *SPARQL query language*

2.2.2.1. *Graph patterns*

SPARQL queries [HAR 13] are composed of a graph pattern expression (GPE) and a solution modifier. The graph pattern expression is a logical combination (UNION, MINUS, OPTIONAL) of basic graph patterns (BGP) combined with content filtering functions (FILTER). The WHERE clause is followed by a Graph Pattern which corresponds to a conjunction of triple patterns where subjects, properties and objects can also be variables (prefixed with a "?" symbol). SPARQL 1.1 also includes *property paths* which are regular expressions over property types. Basic graph patterns are defined by RDF Turtle expressions where constants can be replaced by variables. The result of a graph pattern expression is a table of *variable bindings* which can be modified by applying classical relational operators such as projection, distinct, order, limit and group by. Solution modifiers define the query result format, which can be a simple table (`select`), an RDF graph (`construct`), a Boolean variable which is true if the result is not empty (`ask`) and a general description (`describe`) of the matching RDF resources.

EXAMPLE 2.3.– *The following query returns the father-in-law of Catherine de Medici. The pattern expression is defined in the* `where` *clause, whereas the solution modifier is a simple projection on the bindings of variable* `?nr`*:*

```
prefix : <http://www.royals.org/#>
select ?nr from <kingsandqueens.ttl>
 where { ?r :name ?nr ; :son ?s . ?s :wife ?w .
       ?w :name ''Catherine de Medici'' }
```

Listing 2.1. *Query Q1*

```
----------------
| nr           |
================
| "Francois I" |
----------------
```

Listing 2.2. *Result of* $Q1$

2.2.2.2. *Mapping semantics*

The formal semantics of a basic graph pattern (BGP) P consist of finding all mappings m from the variables in P to the nodes in the RDF data set D such that $m(P)$ is a sub-graph of D. Formally, there exists at least one such mapping m (that is, an answer) if P is isomorphic to a sub-graph of D, which is known to be an NP-complete decision problem.

EXAMPLE 2.4.– *Figure 2.3 shows the SPARQL query graph for query Q1 and an extract of the input data graph as well as a mapping from the query graph to the data graph (red lines). Since Catherine de Medici has been married exactly once, query Q1 generates exactly one mapping as shown in Figure 2.3.*

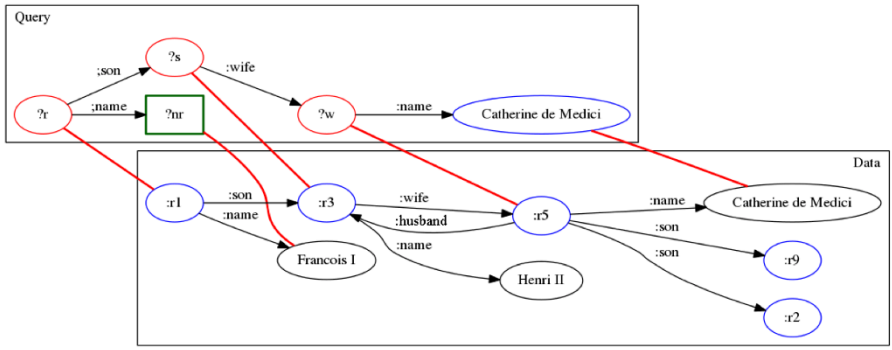


Figure 2.3. *Pattern matching semantics. For a color version of this figure, see www.iste.co.uk/pivert/nosql.zip*

2.2.2.3. *SPARQL algebra*

The core fragment of SPARQL can be formalized by an algebra over *SPO* triple set encodings of RDF graphs [PÉR 06].

EXAMPLE 2.5.– *The following query Q2 combines three BGPs and returns the names of all kings and queens whose name starts with the letter "C", the names of their wives/husbands, and, where they exist, the name of their sons. We observe that kings and queens without a spouse do not appear in the result.*

```
prefix : <http://www.royals.org/#>
select ?n1 ?n2 ?ns
 from <kingsandqueens.ttl>
 where { { ?x :name ?n1 ; :wife    [ :name ?n2 ] }
         UNION
        { ?x :name ?n1 ; :husband [ :name ?n2 ] }
       OPTIONAL { ?x :son [ :name ?ns ] }
       FILTER (regex(?n1,"^C")) }
```

Listing 2.3. *Query Q2*

The result of query $Q2$ is shown in Listing 2.4:

```
-------------------------------------------------------------------
| n1                     | n2                      | ns           |
===================================================================
| "Charles IX"           | "Elisabeth d Autriche" |              |
| "Catherine de Medici" | "Henri II"             | "Francois II" |
| "Catherine de Medici" | "Henri II"             | "Henry III"   |
| "Catherine de Medici" | "Henri II"             | "Henry III"   |
| "Catherine de Medici" | "Henri II"             | "Charles IX"  |
-------------------------------------------------------------------
```

Listing 2.4. *Result of* $Q2$

Query $Q2$ can be translated into the following SPARQL algebra expression [HAR 13] generated by JENA-ARQ[4]:

```
(project (?n1 ?n2 ?ns)
  (conditional
    (union
      (sequence
        (filter (regex ?n1 "^C")
          (bgp (triple ?x <http://www.royals.org/#name> ?n1)))
        (bgp
          (triple ?x <http://www.royals.org/#wife> ??0)
          (triple ??0 <http://www.royals.org/#name> ?n2)
        ))
      (sequence
        (filter (regex ?n1 "^C")
          (bgp (triple ?x <http://www.royals.org/#name> ?n1)))
        (bgp
          (triple ?x <http://www.royals.org/#husband> ??1)
          (triple ??1 <http://www.royals.org/#name> ?n2)
        )))
    (bgp
      (triple ?x <http://www.royals.org/#son> ??2)
      (triple ??2 <http://www.royals.org/#name> ?ns)
    )))
```

4 https://jena.apache.org/documentation/query/arq-query-eval.html

The *triple* operator computes all variable binding tuples of simple triple patterns. The *bgp* operator *joins* the binding tuples with the same values for all shared variables. The *filter* operator filters bindings according to the filtering predicate. The *union* operator computes the union of all binding tuples, whereas *conditional* adds all optional bindings obtained by the second sub-expression. The final set of variable bindings is defined by the operator *project*.

In the rest of this chapter, we are focusing on the evaluation of basic graph patterns (BGP) without filters, alternatives and union. Efficiently evaluating such patterns is essential for all SPARQL query engines and an important challenge in SPARQL query optimization.

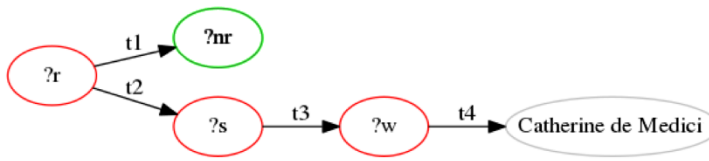


Figure 2.4. *Basic graph pattern. For a color version of this figure, see www.iste.co.uk/pivert/nosql.zip*

EXAMPLE 2.6.– *The following pattern corresponds to query Q1 in Listing 2.1 with four triple patterns t1 to t4 (see also Figure 2.4):*

```
 (prefix ((: <http://www.royals.org/#>))
   (project (?nr)
     (bgp
t1     (triple ?r :name ?nr)
t2     (triple ?r :son ?s)
t3     (triple ?s :wife ?w)
t4     (triple ?w :name "Catherine␣de␣Medici")
    )))
```

Listing 2.5. *Basic graph pattern of Q1*

Each triple pattern $t_1, ..., t_4$ *implicitly defines a* triple selection *which computes all triples respecting this pattern. Pattern* t_4 *filters all triples with property* : $name$ *and value "Catherine␣de␣Medici" and* binds *variable* $?w$ *to*

the subjects of these triples. Variables $?r$, $?s$ *and* $?w$ *are called* join variables*, since* $?r$ *joins* t_1 *and* t_2 *on their subject,* $?s$ *joins the object of* t_2 *with the subject of* t_3 *and* $?w$ *joins the object of* t_3 *with the subject of* t_4*. These joins can be combined to generate the so-called join plans. For example, the above basic graph might first evaluate the filters* t_1 *bind variables* $?r$ *and* $?nr$ *to all resources and their names, before joining these resources with the bindings for* $?r$ *and* $?s$ *obtained by filter* t_2*:* $join_r(t_1, t_2)$*. The result can then be joined with* t_3 *and* t_4 *to obtain a final join plan* $Q_{11} = join_w(t4, join_s(t_3, join_r(t_1, t_2)))$.

We also observe that the join operations are commutative and distributive and that it is possible to build several equivalent plans with different join orderings. On the contrary, as we will show in section 2.3, the processing cost can change drastically between two equivalent plans with different join orderings.

EXAMPLE 2.7.– *Since* t_4 *produces only one binding for variable* $?w$*, it might be more interesting to join* t_4 *with* t_3 *first:* $Q_{12} = join_r(t1, join_s(t_2, join_w(t_3, t_4)))$*. Other triple pattern orderings might become completely inefficient, because they generate cross products like the following plan:* $Q_{12} = join_{s,w}(t_3, join_r(t_2, cross(t_1, t_4)))$.

2.3. SPARQL query processing

RDF has become a de facto standard for publishing information and knowledge on the Web. Compared to other standards such as XML or JSON, RDF facilitates in particular the integration of information by linking RDF resources of separate data sets through `owl:sameAs`, `rdf:type`, and `rdfs:subClassOf` properties. The resulting *Semantic Web* of *linked open data* (LOD) sets is composed of billions of triples, and building systems for efficiently storing and querying the Semantic Web is a technological and scientific challenge reminiscent of other Big Data applications.

2.3.1. *SPARQL with and without RDF/S entailment*

The first challenge concerns the interaction between SPARQL's graph pattern matching semantics and RDF's data model semantics defined by the RDF/S entailment rules. There mainly exist two solutions to evaluating a

query Q over a data set D and a set of entailment rules R. The first solution is to first saturate D by recursively applying all entailment rules R to generate a saturated data set D'[5] and to then apply Q on $sat(D, R)$. The second solution is to rewrite Q into a new query $rew(Q, R)$ such that $rew(Q, R)(D) = Q(sat(D, R))$. The data saturation method has the advantage that it is simple to implement, at least for RDFS entailment rules, by using standard forward chaining, and is independent of the query language. However, it also might generate a huge amount of data and become inefficient if D changes often (preprocessing cost). The query rewriting method is more complex to implement (similar to the "query rewriting using views" problem [HAL 01]) and might take more time during query execution. The current trend in RDF stores is to apply saturation combined with efficient and scalable solutions for reducing data storage and query processing cost. The rest of this section will present an overview of these solutions.

2.3.2. *Query optimization*

Query optimization is a fundamental database problem and a central part of all data processing platforms that provide query capabilities. The main goal is to identify the optimal query execution plan for a given query. Optimality is defined with respect to the estimated costs of different query plans. These cost estimations depend on the implementation(s) of the different (algebraic) operators (*triple* filter, *bgp* join and *project*) and the underlying system architecture. In traditional centralized database systems, where data is stored on a disk, the cost is dominated by the cost of reading data into the main memory. Scalable query engines that are based on distributed data storage should also consider the data transfer cost. On the contrary, in main memory-based computation, the costs are strongly dominated by the data read and transfer costs [ÖZS 11] and are generally ignored in both settings.

The main challenge in processing complex graph pattern queries is to optimize the join operations (*bgp*) which dominate the cost of all other operators.

5 There exists exactly one such saturated set (a fixpoint) D'.

EXAMPLE 2.8.– *Consider the following basic graph pattern expression:*

```
   (bgp
t1   (triple ?r <http://www.royals.org/#name> "Francois␣I")
t2   (triple ?r <http://www.royals.org/#son> ?f)
t3   (triple ?f <http://www.royals.org/#name> ?nf)
t4   (triple ?f <http://www.royals.org/#wife> ?e)
t5   (triple ?e <http://www.royals.org/#name> ?ne)
    )
```

Listing 2.6. *Basic graph pattern* $P1$

If we consider only binary join operations, there are $5! = 120$ *possible equivalent left-linear join plans* $P = (...(t_{\pi(1)}, t_{\pi(2)}), ...t_{\pi(5)})$ *for* $P1$*, where each pattern corresponding to a permutation* π *of the sequence* $1, 2, 3, 4, 5$.

2.3.2.1. *Logical and physical join plans*

For each *logical* join plan, there exists a number of possible *physical* join plans using different join operations based on different join algorithms (nested loop, merge join, hash join, etc.) [RAM 00]. Depending on the algorithm used, each of these physical plans has different processing costs depending on the data size and distribution. In particular, there does not exist a single join implementation that dominates all other implementations. This issue will be discussed in section 2.4 in the context of *distributed join algorithms*.

2.3.2.2. *Cost estimation model*

In order to choose a query execution plan in general and a join plan in particular, the system must be able to estimate the cost of each plan without executing it. Logical cost estimation models do not take account of physical index structures and data distribution, but they mainly try to estimate the size (the number of triples) of each join argument (the triple pattern) and the join result. This estimation can be based on simple binding pattern-based heuristics using the number and position of the free variables to order the joins [TSI 12]. For instance, in the previous example, $t1$ is the only pattern with a single free variable and will be evaluated before all other patterns. This evaluation binds (join) variable $?r$ for joining with triple $t2$, etc. These simple estimations can be extended by adding knowledge about the used join algorithms and statistical information about the distribution of values, properties and/or node identifiers.

2.3.2.3. *Join operation re-ordering*

In all cases, the join ordering plays a crucial role in the generated cost. For example, when using a simple nested loop implementation, plan $((((t_1, t_2), t_3), t_4), t_5))$ is more efficient than, for example, plan $((((t_5, t_4), t_3), t_2), t_1))$: it is more efficient to bind first variable $?r$ to the node corresponding to François I (triple $t1$) and find his son (?f) and their name (?nf) and spouse (?f) than finding all resources with a name (?e) and keep only those which are the spouse of a node (?f) which is the son of François I (?r).

2.3.2.4. *Indexing*

Indexing data is a standard way to achieve better performance. The main goal is to prune the search space for filtering triples. Many index structures, like B+-trees, naturally order the indexed data sets by the index key. This is particularly useful for implementing efficient merge join algorithms. Certain RDF stores make an intensive use of such indexes [NEU 10, WEI 08] for precomputing all possible orders on triples (order by subject/property, subject/object, object/property, object/subject, property/subject and property/object).

EXAMPLE 2.9.– *For example, for processing graph pattern P1, we might then use the object/property index to rapidly find the binding (subject)* $?r$ *of triple pattern t1, the subject/property index to prune all triples t2, etc.*

The query performance gain has to be put into perspective with the additional index processing and storage cost.

2.3.2.5. *Distribution and parallelization*

Finally, an important trend used to achieve scalability is in exploiting parallel/distributed data processing infrastructures (clusters) like Hadoop and Spark for processing SPARQL queries. The main goal and challenge is to decompose global join plans over a large data set D into a composition of joins on partitions of D distributed over the cluster nodes. Within this context, the cost model includes data exchange cost between nodes. Finding the optimal data distribution and join plan that minimizes data transfer by optimizing data-to-query locality is a major issue, which we will discuss in section 2.4.

2.3.3. *Triple store systems*

This section focuses on graph database systems based on the RDF abstract data model. Hence, we will not consider systems based on the property graph model, which can be considered as direct competitors of triple stores. A triple store is a system for storing and querying RDF data sets. It provides the so-called SPARQL endpoints [CUR 15a] and can be organized into triple store federations which collaborate through their SPARQL endpoints to answer global SPARQL queries. This federation capacity plays a crucial role in building the linked open data cloud.

In this section, we give an overview of existing triple store systems and classify them by the underlying general technologies they use. More specific challenges and solutions related to SPARQL query processing will be discussed in section 2.3, followed by section 2.4, which will focus on distributed map and reduce-based implementations.

The survey [ÖZS 16] distinguishes between the following six approaches[6]:

2.3.3.1. *Relational DB stores with SPARQL to SQL mapping*

These build on standard relational DBMS systems for encoding RDF data and executing SPARQL queries through SQL. The data set is translated into a single table $Triples(S, P, O)$, that stores triples with some auxiliary tables for encoding URIs, storing values, etc. SPARQL queries are translated into SQL and processed using a standard SQL processor. The main advantage is the reuse of standard technology. However, SPARQL graph patterns generate a specific class of complex SQL queries which are difficult to optimize using a standard SQL query optimizer.

2.3.3.2. *Single-table triple stores with extensive indexing*

These approaches also apply a direct relational mapping into a single triple table, but use a native RDF storage system (i.e. the system does use an external database management system to handle RDF triples) and an optimized SPARQL query processor. This processor mainly implements an efficient merge join operator by generating an index for each possible permutation SPO, SOP, PSO, POS, OSP and OPS. This avoids the costly sorting

6 In the following, S, P and O denote the triple subject, property and object respectively.

phase, since each index defines a different lexicographic ordering of the triples and enables the selection of an optimal merge join algorithm for any join variable position. In particular, star queries can be efficiently evaluated as simple range queries over the corresponding index.

2.3.3.3. *Property table-based triple stores*

These are designed for exploiting structural RDF patterns appearing in RDF data sets. *Clustered property tables* group together sets of properties which are frequently shared by the same subjects, whereas property class tables regroup resource instances of the same *rdfs:Class*. The main challenge is to define efficient data structures for taking into account multi-valued properties and irregularities (NULL values).

2.3.3.4. *Binary table-based triple stores*

These follow a column-oriented schema organization defining a binary table $p(S, O)$ sorted by the subject S for each property p. This vertical partitioning reduces the I/O cost, and star queries are efficiently implemented through merge joins.

2.3.3.5. *Graph-based triple stores*

These directly attempt to implement the mapping semantics of SPARQL (section 2.2.2.2). The corresponding subgraph isomorphism decision problem is NP-complete [GAR 79] and the main idea consists of reducing the search space before the matching step using a false-positive pruning step to find a safe set of candidate subgraphs. This pruning can be achieved through particular encoding and indexing methods as described in [ZOU 14]. TrinityRDF [ZEN 13] is a distributed in-memory RDF store proposed on the Microsoft Azure cloud. It is based on the Trinity key-value store and adopts a hash-based partitioning.

2.3.3.6. *Cloud-based distributed triple stores*

These achieve scalability by exploiting the data parallelism of modern cloud infrastructures such as Hadoop and Spark. See section 2.4.2.1 and [KAO 15] for a detailed overview on cloud-based triple stores.

2.4. SPARQL and MapReduce

The features expected from modern RDF triple stores are reminiscent of the Big Data trend in which solutions implementing specialized data stores

from scratch are rare due to the enormous development effort they require. Instead, many RDF triple stores prefer to rely on existing infrastructures based on MapReduce [DEA 04] and clusters of distributed data and computation nodes for achieving efficient parallel processing over massively distributed data sets (see section 2.4.2.1). However, these cluster infrastructures are not designed as fully-fledged data management systems [STO 10] and integrating an efficient query processor on top of them is a challenging task. In particular, data storage and communication costs generated by the evaluation of joins (including data preprocessing and indexing) over distributed data need to be addressed cautiously. This section mainly reflects the work published in [NAA 17, NAA 16].

2.4.1. *MapReduce-based SPARQL processing*

Given a triple data set D, a query expression Q and a cluster of computation nodes C, we assume the following global query evaluation process: (i) the data set D is partitioned and distributed over the cluster C following a predefined query-independent hash-based partitioning strategy; (ii) each node can evaluate any triple selection locally over its own triple set; (iii) the join plans are executed following a distributed physical join plan using different physical join implementations. Next, we provide details on each of these steps.

2.4.1.1. *Data partitioning*

Due to its high efficiency, hash-based data partitioning is the foundation of MapReduce-based parallel data processing infrastructures. Consider a cluster $C = (node_1, \cdots node_m)$ of m nodes and some query q with variables V over an input data set D. Any subset V' defines a *partitioning scheme* for q, denoted $q^{V'}$, which describes the partitioning of the triples matched by q with respect to the bindings of a variable subset $V' \subseteq V$.

EXAMPLE 2.10.– *For example,* $(?x\ prop\ ?y)^{?x}$ *denotes that all triples with the property* $prop$ *are partitioned by their subject,* $(?x\ ?p\ ?y)^{?p}$ *denotes a vertical partitioning by property type and* $(?x\ ?p\ ?y)^{?x}$ *denotes a horizontal partitioning by subject. It is also possible to partition by subject and object,* $(?x\ ?p\ ?y)^{?x\ ?y}$. *By definition,* $(?x\ ?p\ ?y)^{?x\ ?p\ ?y}$ *is equivalent to* $(?x\ ?p\ ?y)^{\emptyset}$ *and denotes a random partitioning.*

In the following, we suppose that all triples of the input data are partitioned by their subject.

2.4.1.2. *Triple selection (FILTER)*

Given a triple pattern t, the triple selection algorithm consists of computing all bindings for the variables in t. All triple selections are evaluated locally on each cluster node and generate no data transfer. In our experiments (section 2.5.3), we rely on a semantic encoding [CUR 15b] to accelerate the filtering process. A second optimization consists of *merging* all triple patterns into a "disjunctive" pattern as a first pruning step. Consider a query $Q = \{t_1, \cdots, t_n\}$ composed of n triple patterns. Since all t_i are expressed over the same data set D, there are opportunities to save on access cost for evaluating Q. The basic idea is to replace n scans over the whole data set D by a single scan over the whole data set and k scans over a much smaller subset. For this, we first rewrite the selections in q into a single selection $S = \sigma_{c_1 \vee \cdots \vee c_n}(D)$ where c_i is the select condition of t_i which returns all triples $\bigcup_{i=1}^{n} t_i$ necessary for evaluating Q. This approach (known as a shared-scan) tends to reduce the access cost for selective queries that only access a small subset of D.

Triple selection preserves the partitioning schemes of their input, i.e. the result of a triple selection has the same partitioning as the input data set. For instance, considering query Q_8 over a triple set D partitioned by subject, we obtain the following partitioning schemes for each triple selection query: t_1^x, t_2^y, t_3^x, t_4^y, t_5^x.

2.4.1.3. *Partitioned join: $Pjoin$*

Let $Q = join_V(q_1^{p_1}, q_2^{p_2})$ be a join query, with q_i a triple pattern or a subquery. The *partitioned join* operator, henceforth denoted as $Pjoin_V(q_1^{p_1}, q_2^{p_2})$, repartitions and distributes, when necessary, the input data over the bindings of all variables in V (i.e. it shuffles on V) and then computes the join result for each partition in parallel as detailed in algorithm 2.1.

We distinguish three cases depending on p_i values:

i) $p_1 = V \wedge p_2 = V$; the join is local since every q_i is already partitioned on the join key V. This case generates no data transfer.

ii) $p_1 = V \wedge p_2 \neq V$; the result of q_2 is shuffled on V before evaluating the join;

iii) $p_1 \neq V \wedge p_2 \neq V$. Every q_i's result is shuffled on V before evaluating the join. The result of Q is partitioned on V, which is denoted as Q^V. The corresponding transfer cost is:

$$\sum_{1 \leq i \leq 2 \wedge p_i \neq V} Tr(q_i) \; with \; Tr(q_i) = \theta_{comm} * \Gamma(q_i),$$

where $\Gamma(q)$ is the result size of a given subquery q and θ_{comm} is the unit transfer cost.

Algorithm 2.1. Partitioned join

Input: $\{q_1^{p_1}, q_2^{p_1}\}$, join variables V
Output: result fragment $Result_j$ on each node $node_j$
▷ Evaluate and shuffle sub-query results
for all q_i **do**
  for all $node_j$ **do**
    $d_{ij} \leftarrow$ evaluate q_i on node $node_j$
    if $p_i \neq V$ **then**
      repartition d_{ij} on V into $\{d_{ij1}, \cdots, d_{ijm}\}$
      for all $node_k \neq node_j$ **do**
        transfer d_{ijk} from $node_j$ to $node_k$

▷ Compute join locally on each node
for all $node_j$ **do**
  $Result_j^V \leftarrow (\bigcup_{x=1}^{m} d_{1xj}) \bowtie (\bigcup_{x=1}^{m} d_{2xj})$

For example, in Figure 2.5, consider the subquery joining t_2, t_3, t_4 on y. An evaluation of this subquery is $Pjoin_y(t_3^x, Pjoin_y(t_2^y, t_4^y)^y)^y$. Since D is partitioned by subject, t_2 and t_4 are adequately partitioned on y and join locally without any transfer. This plan only partitions and shuffles (i.e. distributes on its object y) the result of t_3 before computing the last join.

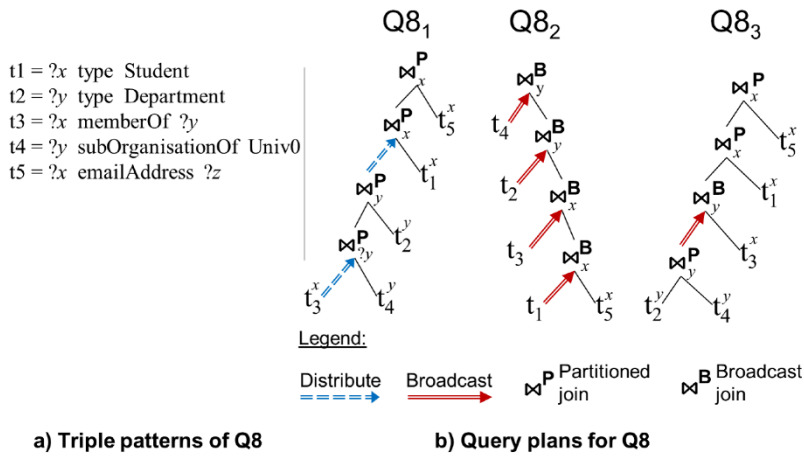


Figure 2.5. *Evaluation plans for query $Q8$ on the LeHigh University Benchmark (LUBM). For a color version of this figure, see www.iste.co.uk/pivert/nosql.zip*

2.4.1.4. *Broadcast join:* $Brjoin$

The *broadcast join*, denoted as $Brjoin_V(q_1^{p_1}, q_2^{p_2})$, consists of sending the query result of q_1 to *all* compute nodes, as detailed in algorithm 2.2. Without loss of generality, we assume that q_2 is the target subquery, excluded from the broadcast step, and has a larger size than q_1. The broadcast join does not consider the partitioning of its arguments and preserves the partitioning of the target query, i.e. the result of the broadcast join has the same partitioning as q^{p_2}. The corresponding transfer cost is:

$$(m-1) * Tr(q_1),$$

where m is the number of nodes and $Tr(q_1)$ is defined as before. A broadcast join does not require any specific data partitioning and preserves the partitioning of the target query, i.e. the result of the broadcast join has the same partitioning as the target query q^{p_n}. For example, consider the subquery t_2, t_3, t_4 of Q_8. An evaluation of $join_y(t_2, t_3, t_4)$ is $Brjoin_y(t_4^y, Brjoin_y(t_2^y, t_3^x)^x)^x$ which broadcasts the bindings of t_2 and t_4 to all nodes and evaluates the join locally at each node, whatever the partitioning of t_3. The partitioning of the query result is the same as t_3, i.e. a partitioning on x.

Algorithm 2.2. Broadcast join

Input: $\{q_1^{p_1}, q_2^{p_2}\}$, join variables V
Output: result fragment $Result_j$ on each node $node_j$
▷ Evaluate and broadcast q_1 result
for all $node_j$ **do**
    $d_{1j} \leftarrow$ evaluate q_1 on node $node_j$
    for all $node_k \neq node_j$ **do**
        transfer d_{1j} from $node_j$ to $node_k$
▷ Compute join locally on each node
for all $node_j$ **do**
    $d_{2j} \leftarrow$ evaluate q_2 on node $node_j$
    $Result_j^{p_2} \leftarrow \bigcup_{y=1}^{m} d_{1y} \bowtie d_{2j}$

2.4.2. *Related work*

2.4.2.1. *Cloud-based triple stores*

Edutella [NEJ 02] and RDFPeers [CAI 04], the first systems considering distributed storage and query answering for RDF data, were based on a P2P infrastructure. They are mainly based on hash-based indexing techniques and used mainly in tackling partitioning issues. More recently, YARS2 [HAR 07] and Virtuoso [ERL 12] were based on hashing one of the RDF triple components, most frequently the subject. SHARD [ROH 10] is the first triple store built on top of Apache Hadoop distributed data storage (HDFS) and MapReduce. Triple sets are stored in HDFS as flat files where each line represents all the triples associated with a given subject (a subject-based partitioning). The SHARD query processing engine iterates over the set of triples for each triple pattern in the SPARQL query, and incrementally attempts to bind query variables to literals in the triple data, while satisfying all of the query constraints. The SPARQL query clauses are processed in several MapReduce steps, forcing high latency due to a large number of I/O operations over HDFS. Huang *et al.* [HUA 11] used a graph partitioner to distribute (with replication) triples over a set of RDF database instances.

Hadoop is used for the execution of certain queries, and to supervise query processing of queries where the answer set spans multiple partitions. However, the system suffers from Hadoop's start-up overhead and inherent I/O latencies. SemStore [WU 14] divides the RDF graph into a set of paths which cover the original graph nodes. These paths are denoted as Rooted SubGraphs (RSG) since they are generated starting from root nodes (with a null in-degree) to all their possible leaves. RSG are clustered into partitions using k-means regrouping RSG paths with shared segments. The main limitations of this approach are the inefficiency of k-means for highly dimensional vectors and the difficulty in achieving an efficient load balancing of the triples across the partitions. The design of the SHAPE system [LEE 13] is motivated by the limited scalability of graph partitioning-based approaches and applies simple hash partitioning for distributing RDF triples. Like in [HUA 11], SHAPE replicates data for achieving n-hop locality guarantees and takes the risk of costly inter-partition communication for query chains which are longer than their n-hop guarantee. Sempala [SCH 14] executes SPARQL queries using the Hadoop-based Impala [KOR 15] database for the parallelization and fragmentation of SQL queries.

Sempala is responsible for translating SPARQL queries into Impala's SQL dialect. The data layout corresponds to a triples table (justified by triple patterns with unbound properties) along a so-called unified property table, i.e. a unique relation composed of one column containing subjects and as many columns as there are properties. The overhead of this data layout is mitigated by the efficient representation of NULL values in Parquet. The unified property table layout is efficient for star-shaped queries, but it is not adapted to other query shapes. The on-disk storage approach of Sempala motivated its authors to propose a new system called S2RDF [SCH 16] which is built on Spark and uses its SQL interface to execute SPARQL queries. Its main goal is to address all SPARQL query shapes efficiently. Its data layout corresponds to the vertical partitioning approach presented in [ABA 07], i.e. triples are distributed in binary SO relations for each RDF property. Additional relations are precomputed at the data load time to limit the number of comparisons when joining triple patterns. Considering query processing, each triple pattern of a query is translated into a single SQL query and the query performance is optimized using the set of statistics and additional data structures computed during this preprocessing step. The data preprocessing step generates an important data loading overhead.

2.4.2.2. *Distributed multi-way join processing*

Distributed multi-way join processing has been the topic of many research efforts over the decades [LU 91]. We will cite only some of the most recent representative contributions in parallel distributed multi-way joins over partitioned data. In [AFR 10], a solution is presented for the computation of multi-join queries in a single communication round. The algorithm was originally designed for the MapReduce approach, thus justifying the importance of limiting communication costs which are associated with high I/O costs. The authors of [BEA 14] have generalized this single-communication n-ary join problem over a fixed number of servers and designed a new algorithm, named HyperCube, by providing lower and upper communication bounds. HyperCube is also used in [CHU 15] which is a promising approach for evaluating SPARQL queries in a MapReduce setting where the number of rounds has to be restricted. CliqueSquare [GOA 15] also tries to reduce the number of rounds by producing flat multi-way join plans.

2.5. SPARQL on Apache Spark

2.5.1. *Apache Spark*

Apache Spark [ZAH 10] is a cluster computing engine which can be understood as a main memory extension of the MapReduce model, enabling parallel computations on unreliable machines and automatic locality-aware scheduling, fault tolerance and load balancing. While both Spark and Hadoop are based on a data flow computation model, Spark is more efficient than Hadoop for applications requiring the frequent reuse of working data sets across multiple parallel operations. This efficiency is mainly due to two complementary distributed main memory data abstractions, as shown in Figure 2.6: (i) Resilient Distributed Data sets (RDD) [ZAH 12], a distributed, lineage-supported, fault-tolerant memory data abstraction for in-memory computations (when Hadoop is mainly disk-based) and (ii) Data Frames (DF), a compressed and schema-enabled data abstraction. Both data abstractions ease the programming task by natively supporting a subset of relational operators such as $project$, $join$ and $filter$.

On top of RDD and DF, Spark proposes two higher-level data access models, GraphX and Spark SQL. Spark GraphX [GON 14] is a library enabling the manipulation of graphs through an extension of Spark's RDD and

follows a vertex-centric computation model which is dedicated to evaluating iterative graph algorithms in parallel, for example, PageRank. This processing model is not adapted to set-oriented graph pattern matching and is not considered in our evaluation. Spark SQL [ARM 15] allows for querying structured data stored in DFs. It translates a SQL query into an algebraic expression composed of DF operators such as *selection*, *projection* and *join*. Its query optimizer, Catalyst [ARM 15] reorders the operations to obtain a more efficient execution plan, e.g. either process selections first or change the order of successive joins.

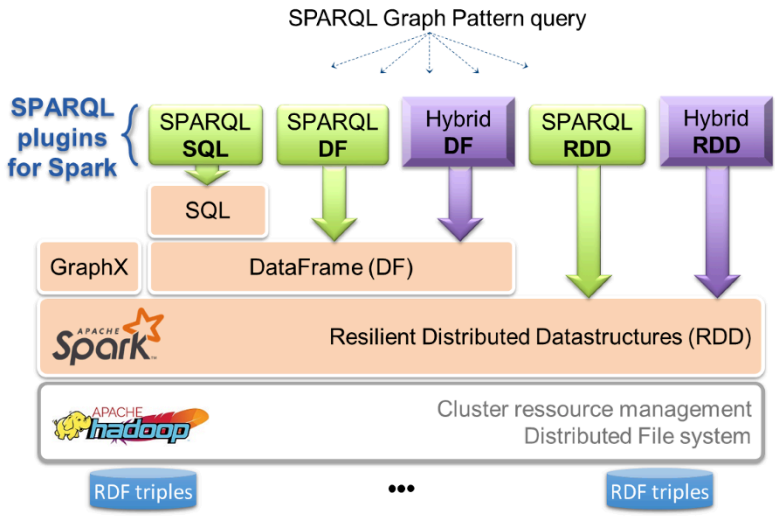


Figure 2.6. *SPARQL on Spark architecture*

2.5.2. *SPARQL on Spark*

Spark, being a general purpose cluster computing platform, does not support SPARQL query processing in particular. This raises the problem of leveraging the data manipulation operators provided by Spark to process SPARQL queries efficiently. The main challenge is to evaluate parallel and distributed join plans with Spark such that data transfers are reduced in favor of more local computation. We investigate to what extent each Spark layer (i.e. SQL, RDD and DF) is suitable for SPARQL processing and what efficiency

could be achieved: how far each does method support the algorithms presented in section 2.2.2 that are to evaluate joins? With this question in mind, we propose five "plugin" approaches to enable SPARQL query processing on top of Spark's SQL, RDD and DF layers. The first three approaches (listed in the top row of Figure 2.6) are worth mentioning because they are used in many state-of-the art distributed SPARQL processing solutions. We consider them as baselines. The last two approaches (the remaining boxes named as 'Hybrid'), which we proposed in [NAA 17], demonstrate the importance of combining several join algorithms within a query plan.

2.5.2.1. *SPARQL SQL*

The SPARQL SQL method consists of rewriting a given SPARQL query Q into a SQL query Q' which is submitted to the Spark SQL layer. The execution plan of Q' is determined by the embedded Catalyst optimizer using the Spark DF data abstraction, which applies the broadcast join method. In our experiments with Spark SQL version 1.5.2, we observed that when a query contains a chain of more than two triple patterns, a Cartesian product is used rather than joins. We consider three triple patterns $t_1 = (a, p1, x)$, $t_2 = (x, p2, y)$ and $t_3 = (y, p3, b)$, and the query $join_y(join_x(t1, t2), t3)$. Then, for the corresponding SQL expression, Catalyst generates the physical plan $P = Brjoin_x(Brjoin\_(t_1, t_3), t_2)$ which computes a cross product between t_1 and t_3 before joining with t_2. This is obviously less efficient than, for example, a plan $P' = Brjoin_y(Brjoin_x(t_1, t_2), t_3))$.

2.5.2.2. *SPARQL RDD*

The SPARQL RDD approach consists of using the Spark RDD data abstraction and specifically the $filter$ and $join$ methods of the RDD class for evaluating SPARQL queries over large triple sets. Every logical join translates into a call to the $join$ method which implements the $Pjoin$ algorithm introduced in section 2.4.1.3. This strategy translates each join into a $Pjoin$ operator, following the order specified by the input logical query, and recursively merges successive joins on the same variable into one n-ary $Pjoin$. This ends up with a sequence of (possibly n-ary) joins on different variables. The result of the first n-ary join on a variable, say v_1, is distributed before processing the next join on a variable, say v_2, and so on. Figure 2.5 shows the *Pjoin plan* of Q_8: $Q_{8_1} = Pjoin_x(Pjoin_x(Pjoin_y(Pjoin_y(t_3^x, t_4^y)^y, t_2^y)^y, t_1^x)^x, t_5^x)^x$. It distributes t_3 triples based on y, then joins them with t_4 and t_2 on y. The result is shuffled

on x to be joined with t_1 and t_5 on x. The overall transfer cost of Q_{8_1} is $\theta_{comm} \cdot (\Gamma(t_3) + \Gamma(join_y(t_4, t_2, t_3)))$.

It evaluates star pattern (sub-) queries locally (i.e. no shuffle) when the input data is partitioned by the join variable, which is obviously efficient. However, it lacks efficiency when a broadcast join is cheaper, e.g. joining a small with a large data set. We observe that SPARQL RDD always reads the entire data set for each triple pattern evaluation. This is remedied by merging multiple triple selections (section 2.4.1.2).

2.5.2.3. *SPARQL DF*

Spark Data Frame (DF) provides an abstraction for manipulating tabular data through specific relational operators. Translating a SPARQL query using the DF DSL is straightforward: triple selections translate into DF $where$ operators, whereas SPARQL n-ary join expressions are transformed into trees of binary DF $join$ operators. The main benefit of using this approach comes from the columnar, which is a compressed in-memory representation of DF. The advantages are twofold. First, it allows for managing larger data sets (i.e. up to 10 times larger compared with RDD) for a given memory space, and second, DF compression saves on data transfer cost.

DF uses a cost-based join optimization approach by preferring a single broadcast join to a sequence of partitioned join (i.e. a $Pjoin$ plan) if the size of the data set is less than a given threshold. This achieves efficient query processing when joining several small data sets with a large one. However, we observe two important drawbacks in applying the SPARQL DF approach. The first drawback comes from the fact that DF only takes into account the size of the input data set for choosing $Brjoin$. Therefore, DF does not efficiently handle very frequent join expressions $join(s, t)$ where s is a highly selective filtering expression over a large data set. In this case, $Brjoin$ would be more efficient since it would avoid the data transfer for pattern t (cost comparison for partitioned and broadcast joins in the following section on *Hybrid joins*). Example Q_{8_2} illustrates the processing of Q_8 through the DF layer. The second drawback is that SPARQL DF (up to version 1.5) does not consider data partitioning and there is no way to declare that an attribute among (S, P, or O) is the partitioning key. Consequently, partitioned joins always distribute data and cause costly data transfers. This penalizes star pattern queries where the

result of each triple pattern is already distributed adequately, since the query could have been answered without any transfer.

2.5.2.4. *SPARQL Hybrid*

The goal of this method is to overcome the limitations found in the SPARQL SQL, RDD and DF solutions in order to provide a more efficient SPARQL processing solution on top of Spark. In particular, SPARQL Hybrid aims to: (i) take into account current data partitioning to avoid useless data transfers, (ii) enable data compression provided by the DF layer to save data transfers and manage larger data sets and (iii) reduce the data access cost of self join operations.

As emphasized in the evaluation (see section 2.5.3), this SPARQL Hybrid strategy allows us to combine $Pjoin$ and $Brjoin$. First, this allows the query optimizer to exploit knowledge about the existing data partitioning in order to combine local partitioned joins with broadcast joins. For example, if a subject-based partitioning scheme has been applied to the data set, an optimal join plan for a "snowflake" query pattern like Q_8 is to join the result of a set of local partitioned joins ("star" subqueries) through a sequence of broadcast joins. Plan Q_{8_3} in Figure 2.5 first joins t_4 with t_2 on y without any transfer, because t_4 and t_2 are adequately partitioned on their subject y. Then, it broadcasts the result and joins it on y with t_3 preserving the partitioning of t_3 on x. Finally, it locally joins the result with the remaining patterns t_1 and t_5 which are also adequately partitioned on their subject x. Plan Q_{8_3} has a lower transfer cost than the plans generated by the other planning strategies. Second, we rely on our cost model to demonstrate that combining the $Brjoin$ and *distributed* $Pjoin$ algorithms might also yield more efficient plans than all other plans using only one distributed join algorithm. We highlight an example where a plan combining both $Pjoin$ and $Brjoin$ algorithms is beneficial. Figure 2.7 shows three join plans for query Q_9 on the LeHigh University Benchmark (LUBM), (using the same legend as in Figure 2.5):

$$Q_{9_1} = Pjoin_y(t_1^x, Pjoin_z(t_2^y, t_3^z)^z)^y \quad [2.1]$$

$$Q_{9_2} = Brjoin_z(t_3^z, Brjoin_y(t_2^y, t_1^x)^x)^x \quad [2.2]$$

$$Q_{9_3} = Pjoin_y(t_1^x, Brjoin_z(t_3^z, t_2^y)^y)^y \quad [2.3]$$

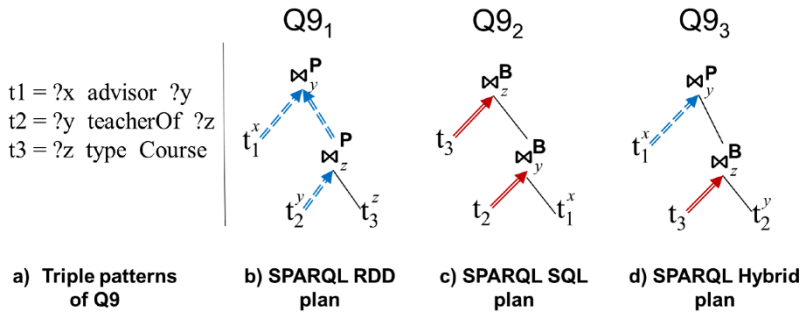


Figure 2.7. *LUBM query Q_9. For a color version of this figure, see www.iste.co.uk/pivert/nosql.zip*

Plan Q_{9_1} is composed of two distributed partitioned joins, plan Q_{9_2} is composed of two broadcast joins, whereas Q_{9_3} is a hybrid plan combining a broadcast join on z and a distributed partitioned join on y. Suppose the following order on the size of the patterns $\Gamma(t_1) > \Gamma(t_2) > \Gamma(t_3)$ and $\Gamma(join_y(t_1, t_2)) > \Gamma(join_z(t_2, t_3))$. Then, it is easy to see that Q_{9_1} is the optimal partitioned join plan and Q_{9_2} the optimal broadcast join plan. The cost of these plans is:

$$cost(Q9_1) = \theta_{comm} * (\Gamma(t_1) + \Gamma(t_2) + \Gamma(join_z(t_2, t_3))) \quad [2.4]$$

$$cost(Q9_2) = \theta_{comm} * (m-1) * (\Gamma(t_2) + \Gamma(t_3)) \quad [2.5]$$

$$cost(Q9_3) = \theta_{comm} * (\Gamma(t_1) + (m-1) * \Gamma(t_3)) \quad [2.6]$$

Based on this cost model, the best plan depends on the number of machines. For small m, Q_{9_2} wins because it broadcasts small-sized triple patterns. For large m, Q_{9_1} wins because it does not broadcast any data. In between, we infer the following two inequalities specifying the range of values for which the Q_{9_3} hybrid plan is most effective:

$$\Gamma(t_1) < (m-1) * \Gamma(t_2) \text{ and } (m-1) * \Gamma(t_3) < \Gamma(t_2) + \Gamma(join_z(t_2, t_3))$$

If m is "high enough", then distributing the large-sized t_1 is cheaper than broadcasting the medium-sized t_2. If m is not "too high", then broadcasting the small-sized t_3 is cheaper than distributing both the medium-sized t_2 and the result of $join(t_2, t_3)$.

We have implemented a simple dynamic greedy SPARQL optimization strategy using this cost model which introduces a fine-grained control of the query evaluation plan at the operator level. The initial input plan is a set of triple patterns with the size estimation for each pattern (necessary statistics are generated during the data loading phase). An evaluation step then consists of: (1) choosing the pair of subqueries and the join operator which generates the minimal cost using our cost model, (2) executing the obtained join expression, and (3) replacing the join arguments by the join expression and the exact result size estimation. This step is iteratively executed until there remains a single join expression in the input plan. This strategy is implemented in both Spark data abstraction layers, and in RDD and DF. For the RDD abstraction (which does not support $Brjoin$ natively), we decompose the $Brjoin$ operator into two functions: one for broadcasting the data and the other for computing the join result based on the broadcast data using the *mapPartition* RDD transformation method[7]. For the DF abstraction, to ensure that the $Brjoin$ operator runs consistently according to the hybrid choice, we switch off the less efficient threshold-based join selection of the Catalyst optimizer.

2.5.2.5. *Analytical comparison*

SPARQL Hybrid is an enhanced query execution method which reduces data transfer overhead between the cluster nodes and scans the whole data set just once for any query. To better highlight the advantages of this method, Table 2.1 presents a synthetic view of the query processing properties of all methods presented in this section:

– *Co-partitioning*. Triples partitioned on the join key can be joined locally without any data transfer. DF does not support co-partitioning until Apache Spark version 1.5.

– *Join algorithm*. $Brjoin1$ (respectively $Brjoin+$) implies the support of a single (respectively several) $Brjoin$ per query. $Pjoin$ is an abbreviation for partioned join.

– *Merged access*. The ability to evaluate several triple selections with a single scan of the data set.

– *Optimization*. The ability to choose among several join algorithms. We qualify the choice made by Spark DF as "poor" because it ignores broadcast joins for large data sets with highly selective triple patterns.

7 http://spark.apache.org/docs/latest/programming-guide.html#transformations

– *Data compression*. DF abstraction uses data compression and can manage data sets ten times larger than RDD, for equal memory capacity.

Method	Co-partitioning	Join algorithm	Merged access	Query optimization	Data compression
RDD	✓	$Pjoin$	✗	✗	✗
DF	✗($\leq$ v1.5)	$Pjoin, Brjoin1$	✗	poor	✓
SQL	✗	$Pjoin, Brjoin1$	✗	cross product	✓
Hybrid RDD	✓	$Pjoin, Brjoin+$	✓	cost-based	✗
Hybrid DF	✓	$Pjoin, Brjoin+$	✓	cost-based	✓

Table 2.1. *Qualitative analysis of five query processing methods*

The table shows that the SPARQL Hybrid method offers equal or higher support for all the considered properties. Interestingly, SPARQL Hybrid suits both data abstractions, RDD and DF, because we strive to design it in a generic way, decoupling the join optimization logic from the lower level Spark data representations. We therefore are also confident that SPARQL Hybrid could easily be extended to support forthcoming Spark data abstractions such as DataSet or GraphFrame.

2.5.3. *Experimental evaluation*

We validated the query processing methods discussed in section 2.5 over three common SPARQL query shapes, star, chain and snowflake, which can be combined to describe other query shapes, e.g. triangles and trees. The evaluation was conducted on an 18 DELL PowerEdge R410 cluster interconnected by 1 GB/s Ethernet. Each machine runs a Debian Linux distribution and has two Intel Xeon E5645 processors. Each processor consists of six cores running at 2.4 GHz and running two threads in parallel (hyper-threading). We used Spark version 1.6.2 and implemented all experiments in Scala with a configuration of 300 cores and 50 GB of RAM per machine.

We selected two synthetic and three real-world knowledge bases (ranging from half a million to close to 1.4 billion triples). The synthetic data sets we used are the LeHigh University Benchmark (LUBM) [GUO 05] and the Waterloo SPARQL Diversity Test Suite (WatDiv) [ALU 14]. They both provide a data generator and a set of queries.

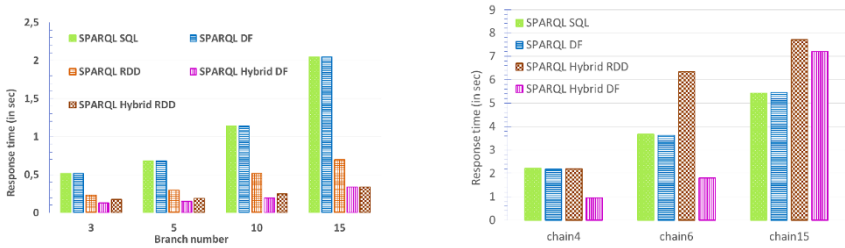
The real-world data sets correspond to open source DBPedia, Wikidata and DrugBank RDF dumps. We compared the performance of five query processing methods over oriented star, chain and snowflake queries. We adopted the partitioning strategy of section 2.4.1.1 (i.e. by subject) and compared our solution with the S2RDF system [SCH 16]. More experimentation details are available on the companion web site[8].

2.5.3.1. *Star queries*

This experiment was conducted over the DrugBank knowledge base, which contains high out-degree nodes describing drugs. A first practical use case is to search for a drug satisfying multi-dimensional criteria; we defined four star queries with a number of branches ranging from 3 to 15. We processed each query using our five SPARQL query processing approaches and reported query response times in Figure 2.8(a). SPARQL SQL decides to reorder the joins only if it reduces the number of join operations which is obviously not possible for a star query that contains only one join variable. Thus, SPARQL SQL generates the same evaluation plan (and cost) as SPARQL DF. Both methods ignore the actual data partitioning and broadcast the result of every triple pattern across the machines. On the contrary, SPARQL RDD, SPARQL Hybrid RDD, and SPARQL Hybrid DF are aware that the data are partitioned on the subject (i.e. the join variable) and thus are able to process the query without any data transfer. We observe that the total costs are dominated by the transfer cost, which explains why SPARQL DF is at least 2.2 times slower than the transfer-free methods.

When comparing the transfer-free methods, we observe that both SPARQL Hybrid methods are 1.4 to 2 times faster than SPARQL RDD. In fact, SPARQL Hybrid reads the data set only once, whereas the data access cost of SPARQL RDD is proportional to the number of branches (triple selection patterns). Finally, SPARQL Hybrid DF slightly outperforms SPARQL Hybrid RDD for star queries with up to 10 branches. This is due to the way SPARQL Hybrid implements partitioning of intermediate results: SPARQL Hybrid RDD relies on a built-in `groupBy` operator whose implementation is slightly more efficient than the user-defined `groupBy` operator of SPARQL Hybrid DF.

8 https://sites.google.com/site/sparqlspark/home



(a) Star queries (b) Chain queries

Figure 2.8. *Query response time with respect to evaluation strategies on real-world data sets. For a color version of this figure, see www.iste.co.uk/pivert/nosql.zip*

2.5.3.2. *Property chain queries*

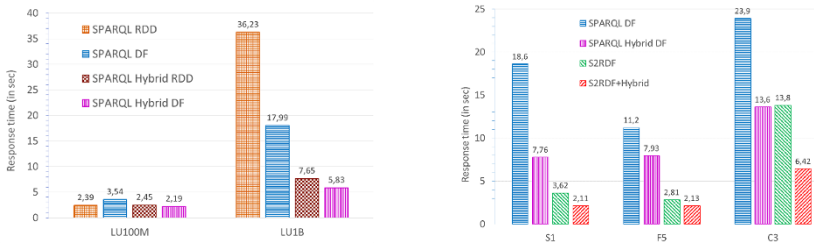
This experiment is done over the DBPedia knowledge base and a set of queries with a path length ranging from 4 to 15. We report the query response times in Figure 2.8(b).

Chain queries $chain4$ and $chain6$ contain large (not selective) triple patterns followed by small (selective) ones. These "large.small" sub-chains should be evaluated by broadcasting the smaller pattern instead of shuffling (the act of exchanging data between machines of the cluster) the larger one. The strength of SPARQL Hybrid DF is here in estimating the patterns' selectivity at run-time and more accurately than SPARQL DF. This allows SPARQL Hybrid DF to choose broadcast joins for this case, whereas SPARQL DF inaccurately estimated the pattern selectivities and favored partitioned joins, which caused large transfer costs.

SPARQL Hybrid DF recursively chooses the lowest cost join based on the size estimations of the intermediate results and the remaining triple patterns. This might lead to a suboptimal plan as shown for chain query $chain15$ (SPARQL DF only uses a partitioned join which is more efficient). In this specific query, the first triple pattern (say t_1) and the following one (say t_2) are large compared to the other ones, but joining t_1 with t_2 yields a very small intermediate result. However, this knowledge is not available before evaluating the join and cannot be exploited by SPARQL Hybrid DF.

2.5.3.3. *Snowflake queries*

First, we focus on the most complex snowflake query of the LUBM ($Q8$). The evaluation plans for $Q8$ have been introduced in section 2.2.2 and we report the response times in Figure 2.9(a). $Q8$ does not run to completion with SPARQL SQL. The evaluation plan contains a Cartesian product that is prohibitively expensive. This emphasizes that the Spark's Catalyst optimizer strategy to replace two joins by one Cartesian product should be applied more adequately by taking into account the actual transfer cost. SPARQL DF and SPARQL RDD confirm that working with compressed data is beneficial as soon as the data set is large enough. Although SPARQL DF ignores data partitioning, thus distributing more triples (320M instead of 104M triples for the partitioning-aware approach), its transfer time is lower than SPARQL RDD, thanks to compression.



(a) LUBM query Q8 (b) WatDiv queries on 1B triples

Figure 2.9. *Performance of benchmark queries and comparison with S2RDF. For a color version of this figure, see www.iste.co.uk/pivert/nosql.zip*

The major experimental result is that SPARQL Hybrid outperforms existing methods by a factor of 2.3 for compressed (DF) and 6.2 for uncompressed (RDD) data. This is mostly due to reduced transfers (only few hundred triples instead of over one hundred million triples for the best existing approach). SPARQL Hybrid also saves on the number of data accesses: two, against three and five for SPARQL RDD and SPARQL DF respectively.

2.5.3.4. *Comparison with S2RDF*

Finally, we compare our Hybrid approach with the state-of-the-art S2RDF [SCH 16] solution which outperforms many existing distributed SPARQL processing solutions. We conducted the S2RDF comparison experiments over the same WatDiv one billion triple data sets on a cluster with approximately similar computing power than that used in the S2RDF evaluation (we used 48 cores in our experiment against 50 cores used in

the S2RDF experiments). Our main goal was to show that our solution is complementary and can be combined with the S2RDF approach. For this, as a baseline, we first selected three representative queries from the WatDiv query set[9], one for each category: S_1 is a star query, F_5 a snowflake and C_3 a complex query. We executed S_1, F_5 and C_3 over one large data set containing all the triples (i.e. without S2RDF VP fragmentation), using SPARQL SQL and SPARQL Hybrid strategies. Then, we split the data set according to the S2RDF VP approach (i.e. one data set per property) and ran the queries using SPARQL SQL along with the S2RDF ordering method and SPARQL Hybrid strategies (the response times are shown in Figure 2.9(b)). The SPARQL S2RDF+Hybrid solution outperforms the baseline SPARQL SQL by a factor ranging between $[1.76, 2.4]$ and the S2RDF solution by a factor ranging between $[1.72, 2.16]$ which is encouraging. The benefit mainly comes from reduced data transfers: our approach saved 483 MB for S_1, 284 MB for F_5 and 1.7 GB for C_3. Note that in reproducing the S2RDF experiments, we obtained response times more than twice as fast as those reported in [SCH 16] (e.g. 3.6 s compared to 8.8 s for query S_1) and our 1.72 minimal improvement ratio is a fair comparison. This highlights that our approach easily combines with S2RDF to provide additional benefit. We did not compare our approach with the concept of ExtVP relations of S2RDF's solution, because it comes at high preprocessing overhead (17 hours for preprocessing the one billion triple data sets) which does not comply with our objective of reducing data preprocessing and loading cost.

In this chapter, we introduced the problem of querying large RDF data sets with SPARQL and presented an exhaustive study comparing SPARQL query processing strategies over an in-memory-based cluster computing engine (Apache Spark). The results emphasize that hybrid query plans combining partitioned and broadcast joins improve query performance in almost all cases. Although SPARQL Hybrid RDD is slightly more efficient than the hybrid DF solution, due to the absence of a data compression/decompression overload, it becomes interesting to switch to the DF representation when the size of RDDs almost saturates the main memory of the cluster. In this case, we can store almost 10 times more data on the same cluster size with only a small loss in performance.

9 http://dsg.uwaterloo.ca/watdiv/basic-testing.shtml

2.6. Bibliography

[ABA 07] ABADI D.J., MARCUS A., MADDEN S. *et al.*, "Scalable semantic web data management using vertical partitioning", *VLDB Conference*, pp. 411–422, 2007.

[ABI 00] ABITEBOUL S., BUNEMAN P., SUCIU D., *Data on the Web: From Relations to Semistructured Data and XML*, Morgan Kaufmann, 2000.

[AFR 10] AFRATI F.N., ULLMAN J.D., "Optimizing joins in a map-reduce environment", *International Conference on Extending Database Technology (EDBT)*, pp. 99–110, 2010.

[ALU 14] ALUÇ G., HARTIG O., ÖZSU M.T. *et al.*, "Diversified stress testing of RDF data management systems", *International Semantic Web Conference (ISWC)*, pp. 197–212, 2014.

[ARM 15] ARMBRUST M., XIN R.S., LIAN C. *et al.*, "Spark SQL: relational data processing in spark", *Int'l Conference on Management of Data (SIGMOD)*, pp. 1383–1394, 2015.

[BEA 14] BEAME P., KOUTRIS P., SUCIU D., "Skew in parallel query processing", *ACM PODS*, pp. 212–223, 2014.

[CAI 04] CAI M., FRANK M., "RDFPeers: A scalable distributed RDF repository based on a structured peer-to-peer network", *Proceedings of the 13th International World Wide Web Conference*, pp. 650–657, 2004.

[CHU 15] CHU S., BALAZINSKA M., SUCIU D., "From theory to practice: efficient join query evaluation in a parallel database system", *SIGMOD*, pp. 63–78, 2015.

[CUR 15a] CURÉ O., GUILLAUME B. (eds), *RDF Database Systems: Triples Storage and SPARQL Query Processing*, Morgan Kaufmann, 2015.

[CUR 15b] CURÉ O., NAACKE H., RANDRIAMALALA T. *et al.*, "LiteMat: a scalable, cost-efficient inference encoding scheme for large RDF graphs", *2015 IEEE International Conference on Big Data*, pp. 1823–1830, Santa Clara, October–November 2015.

[DEA 04] DEAN J., GHEMAWAT S., "MapReduce: simplified data processing on large clusters", *Symposium on Operating System Design and Implementation (OSDI)*, pp. 137–150, 2004.

[ERL 12] ERLING O., "Virtuoso, a hybrid RDBMS/graph column store", *IEEE Data Engineering Bulletin*, vol. 35, no. 1, pp. 3–8, 2012.

[GAR 79] GAREY M.R., JOHNSON D.S., *Computers and Intractability: A Guide to the Theory of NP-completeness*, W. H. Freeman, 1979.

[GOA 15] GOASDOUÉ F., KAOUDI Z., MANOLESCU I. *et al.*, "CliqueSquare: flat plans for massively parallel RDF queries", *IEEE ICDE*, pp. 771–782, 2015.

[GON 14] GONZALEZ J.E., XIN R.S., DAVE A. *et al.*, "GraphX: graph processing in a distributed dataflow framework", *11th USENIX Symposium on Operating Systems Design and Implementation*, pp. 599–613, Broomfield, October 2014.

[GUO 05] GUO Y., PAN Z., HEFLIN J., "LUBM: A benchmark for OWL knowledge base systems", *Journal of Web Semantics*, vol. 3, nos 2–3, pp. 158–182, 2005.

[HAL 01] HALEVY A.Y., "Answering queries using views: a survey", *The VLDB Journal*, vol. 10, no. 4, pp. 270–294, 2001.

[HAR 07] HARTH A., UMBRICH J., HOGAN A. *et al.*, "YARS2: a federated repository for querying graph structured data from the web", *The Semantic Web, 6th International Semantic Web Conference, 2nd Asian Semantic Web Conference, ISWC 2007 + ASWC 2007*, pp. 211–224, Busan, Korea, November 2007.

[HAR 13] HARRIS S., SEABORNE A., "SPARQL 1.1 query language W3C recommendation", http://www.w3.org/TR/sparql11-query/, 2013.

[HUA 11] HUANG J., ABADI D.J., REN K., "Scalable SPARQL querying of large RDF graphs", *Proceedings of the VLDB Endowment*, vol. 4, no. 11, pp. 1123–1134, 2011.

[KAO 15] KAOUDI Z., MANOLESCU I., "RDF in the clouds: a survey", *The VLDB Journal*, vol. 24, no. 1, pp. 67–91, 2015.

[KOR 15] KORNACKER M., BEHM A., BITTORF V. *et al.*, "Impala: a modern, open-source SQL engine for Hadoop", *7th Biennial Conference on Innovative Systems Research, CIDR*, Asilomar, January 2015.

[LEE 13] LEE K., LIU L., "Scaling queries over big RDF graphs with semantic hash partitioning", *Proceedings of the VLDB Endowment*, vol. 6, no. 14, pp. 1894–1905, 2013.

[LU 91] LU H., SHAN M.-C., TAN K.-L., "Optimization of multi-way Join queries for parallel execution", *Proceedings of the 17th International Conference on Very Large Data Bases*, pp. 549–560, Barcelona, September 1991.

[MAL 10] MALEWICZ G., AUSTERN M.H., BIK A.J. *et al.*, "Pregel: a system for large-scale graph processing", *ACM SIGMOD*, pp. 135–146, 2010.

[NAA 16] NAACKE H., CURÉ O., AMANN B., SPARQL query processing with Apache Spark, working paper or preprint, 2016.

[NAA 17] NAACKE H., AMANN B., CURÉ O., "SPARQL graph pattern processing with Apache Spark", *SIGMOD 2017*, Graph Data-management Experiences & Systems (GRADES Workshop), Chicago, May 2017.

[NEJ 02] NEJDL W., WOLF B., QU C. *et al.*, "EDUTELLA: a P2P networking infrastructure based on RDF", *Proceedings of the Eleventh International World Wide Web Conference, WWW 2002*, Honolulu, pp. 604–615, May 2002.

[NEU 10] NEUMANN T., WEIKUM G., "The RDF-3X engine for scalable management of RDF data", *VLDB Journal*, vol. 19, no. 1, pp. 91–113, 2010.

[ÖZS 11] ÖZSU M.T., VALDURIEZ P., *Principles of distributed database systems*, Springer Science & Business Media, 2011.

[ÖZS 16] ÖZSU M.T., "A survey of RDF data management systems", *Frontiers of Computer Science*, vol. 10, pp. 1–15, 2016.

[PÉR 06] PÉREZ J., ARENAS M., GUTIERREZ C., "Semantics and Complexity of SPARQL", *The Semantic Web – ISWC 2006*, Springer, Berlin, Heidelberg, pp. 30–43, Athens, USA, November 2006.

[RAM 00] RAMAKRISHNAN R., GEHRKE J., *Database Management Systems*, 2nd edition, Osborne/McGraw-Hill, Berkeley, 2000.

[ROH 10] ROHLOFF K., SCHANTZ R.E., "High-performance, massively scalable distributed systems using the MapReduce software framework: the SHARD triple-store", *SPLASH Workshop*, p. 4, 2010.

[SCH 14] SCHÄTZLE A., PRZYJACIEL-ZABLOCKI M., NEU A. *et al.*, "Sempala: interactive SPARQL query processing on Hadoop", *The Semantic Web-ISWC 2014*, pp. 164–179, Riva del Garda, October 2014.

[SCH 16] SCHÄTZLE A., PRZYJACIEL-ZABLOCKI M., SKILEVIC S. *et al.*, "S2RDF: RDF querying with SPARQL on spark", *Proceedings of the VLDB Endowment*, vol. 9, no. 10, 2016.

[STO 10] STONEBRAKER M., ABADI D.J., DEWITT D.J. *et al.*, "MapReduce and parallel DBMSs: friends or foes?", *Communications of the ACM*, vol. 53, no. 1, pp. 64–71, 2010.

[TSI 12] TSIALIAMANIS P., SIDIROURGOS L., FUNDULAKI I. *et al.*, "Heuristics-based query optimisation for SPARQL", *15th International Conference on Extending Database Technology (EDBT)*, pp. 324–335, Berlin, March 2012.

[W3C 14] W3C (World Wide Web Consortiom), "RDF 1.1 Concepts and Abstract Syntax", Report, available at: http://w3.org/TR/rdfll-concepts, 2014.

[WEI 08] WEISS C., KARRAS P., BERNSTEIN A., "Hexastore: sextuple indexing for semantic web data management", *PVLDB*, vol. 1, no. 1, pp. 1008–1019, 2008.

[WU 14] WU B., ZHOU Y., YUAN P. *et al.*, "SemStore: a semantic-preserving distributed RDF triple store", *Proceedings of the 23rd ACM International Conference on Information and Knowledge Management*, pp. 509–518, Shanghai, November 2014.

[ZAH 10] ZAHARIA M., CHOWDHURY M., FRANKLIN M.J. *et al.*, "Spark: cluster computing with working sets", *USENIX HotCloud Workshop*, 2010.

[ZAH 12] ZAHARIA M., CHOWDHURY M., DAS T. *et al.*, "Resilient distributed datasets: a fault-tolerant abstraction for in-memory cluster computing", *USENIX NSDI Symposium*, pp. 15–28, 2012.

[ZEN 13] ZENG K., YANG J., WANG H. *et al.*, "A distributed graph engine for web scale RDF data", *Proceedings of the VLDB Endowment*, vol. 6, no. 4, pp. 265–276, 2013.

[ZOU 14] ZOU L., ÖZSU M.T., CHEN L. *et al.*, "gStore: a graph-based SPARQL query engine", *The VLDB Journal*, vol. 23, no. 4, pp. 565–590, 2014.

3

Doing Web Data: from Dataset Recommendation to Data Linking

3.1. Introduction

The chapter introduces the main concepts of interest in the context of data linking. We start by an overview of the Semantic Web.

3.1.1. *The Semantic Web vision*

With the advent of the World Wide Web (WWW), accessing information has become quicker and simpler via Web documents which are part of a "global search space". In this version of the Web, knowledge is accessible by traversing hypertext links using Web browsers. In recent years, this global information space of connected documents is currently evolving into a global Web of data – the Semantic Web – where both data and documents are linked. Tim Berners-Lee, the inventor of the WWW, defined the Semantic Web as "not a separate Web but an extension of the current one, in which information is given well-defined meaning, better enabling computers and people to work in cooperation" [BER 01].

The transition of the current document-oriented Web to a Web of interlinked data has led to the creation of a global data space that contains many billions of

---

Chapter written by Manel ACHICHI, Mohamed BEN ELLEFI, Zohra BELLAHSENE and Konstantin TODOROV.

NoSQL Data Models: Trends and Challenges, First Edition. Edited by Olivier Pivert.

linked data entities.[1] In other words, linked data can be seen as a deployment path for the Semantic Web.

In the following, we present an overview of existing visions of what the linked data life cycle looks like going through different challenges to construct and link knowledge graphs, while respecting linked data best practices [BIZ 09].

3.1.2. *Linked data life cycles*

We start by providing Tim Berners-Lee's vision of the five-star linked open data system[2], which is also a W3C recommendation[3], where the system has to ensure (the numbers in brackets correspond to "stars", or in other words, the degree of quality of linked open data): (1) availability of data on the Web, in whatever format; (2) its availability as machine-readable structured data (e.g. as CSV not as scanned image of table); (3) its availability in a non-proprietary format (i.e. CSV format instead of Microsoft Excel); (4) publishing using open standards from the W3C recommendation, RDF and SPARQL; and (5) links to other linked open data, whenever feasible.

For this purpose, a number of linked data life cycle visions have been adopted by the Semantic Web community, where data goes through a number of stages to be considered as five-star linked data. Figure 3.1 shows four visions of different linked data life cycles, which were proposed by Hyland *et al.* [HYL 11], Hausenblas *et al.* [HAU 12], Villazon-Terrazas *et al.* [VIL 11], as well as the DataLift vision [SCH 12] and the LOD2 linked open data life cycle [AUE 10]. Since there is no standardized linked data life cycle, the main stages of publishing a new dataset as linked data can be summarized as follows:

– extracting and transforming information from the raw data source to RDF data. Mainly, the data source can be in an unstructured, structured or semi-structured format, i.e. CSV, XML file, a relational database, etc. In this stage, the data does not include vocabulary modeling, namespace assignment nor links to existing datasets;

1 The LOD cloud diagram is published regularly at http://lod-cloud.net. and an overview of the LOD cloud (2017) can be checked in http://lod-cloud.net/versions/2017-02-20/lod.svg.

2 Linked open data (LOD) is linked data which is released under an open license as defined in http://5stardata.info/en/.

3 https://dvcs.w3.org/hg/gld/raw-file/default/glossary/index.html.

– assigning namespaces to all resources, notably to make them accessible via their URIs;

– modeling the RDF data by reusing relevant existing vocabulary terms whenever applicable;

– hosting the linked dataset and its metadata publicly, and making it accessible;

– linking the newly published dataset to other datasets already published as linked data on the Web.

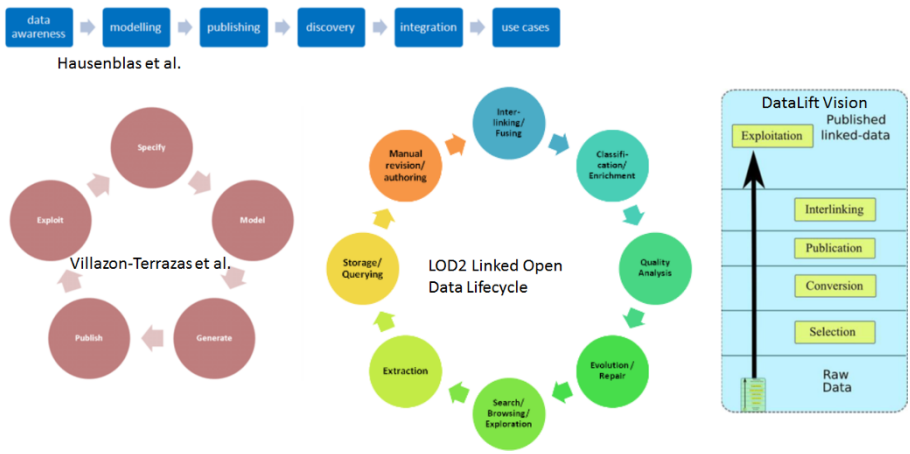


Figure 3.1. *Examples of government linked data life cycles. For a color version of this figure, see www.iste.co.uk/pivert/nosql.zip*

It is worth noting that the different stages of the linked data life cycle are not in a strict sequence nor do they exist in isolation; however, they are in a process of mutual enrichment. Some stages in the linked data life cycles are not completely automated and may need human effort either for linked data publishing, maintenance or usage. Hence, in recent years, an increasing number of works have shown an interest in the development of research approaches, standards, technology and tools for supporting different stages of the linked data life cycle.

For instance, linked data publishers face the major challenge of providing and maintaining a data store (in the Semantic Web field, we talk of large

triple stores[4]) that is highly effective and ensures the scalability of data. Currently, there exists a wealth of works contributing to this central problem by developing distributed data storage solutions which can work in server clusters scattered in the cloud and handle massive amounts of triples, e.g. the Hadoop Distributed File System (HDFS)[5] for storing very large amounts of data bytes (on the terabyte and petabyte scales).

Another current challenge concerns the linked data modeling process which requires a huge effort from metadata designers, in particular on the issues raised by the need to identify suitable terms from published vocabularies in order to reuse them following linked data modeling best practices.

Further challenges include semantic link discovery between different RDF datasets, which is manually unfeasible, considering the large amount of data available on the Web. Usually, among the different kinds of semantic links that can be established, the default option is to set `owl:sameAs` links between different URIs that refer to the same real objects. For example, DBpedia uses the URI http://dbpedia.org/resource/Montpellier to identify the city of *Montpellier*, while Geonames uses the URI http://www.geonames.org/2992166 to identify *Montpellier*. Data linking techniques and tools are often used to deal with this problem. However, this task still requires human involvement, notably on: (i) the identification of candidate datasets to link to, where the search for the targets should be done almost by an exhaustive search of all datasets in the different catalogues, which is manually not feasible; and (ii) the strenuous task of instance mapping configuration between different datasets.

We note that some stages in the linked data life cycles proceed automatically, notably the automatic extraction and transformation of raw data, which has led to the publication of large amount of information on the Web in the form of linked datasets. However, automatic approaches have raised many questions regarding the quality, the currency and the completeness of the contained information. Hence, the major challenge during this process concerns mainly the assessment of data quality.

4 https://www.w3.org/wiki/LargeTripleStores.

5 https://hadoop.apache.org/docs/r1.2.1/hdfsdesign.html.

Several issues arise after publishing linked data with regard to both data publishers and data consumers. On the one hand, data publishers (or rather, maintainers) have a responsibility to ensure a continued maintenance of the published dataset in terms of quality, i.e. access, dynamicity (different versions, mirrors), and so on. On the other hand, from the linked data consumers viewpoint, there is the need to:

– find and retrieve suitable information from different linked open datasets;

– integrate and reuse this knowledge;

– ensure continued feedback for data maintainers.

3.1.3. *Chapter overview*

As explained previously, this chapter will address the data linking challenge which, as shown in Figure 3.2, includes: (i) the dataset selection task (in particular, we address the problem of candidate datasets discovery aimed at finding target datasets to link to [see section 3.2]); and (ii) the data linking challenge. More precisely, we focus on the problem of determining which resources refer to the same real-world entity (see sections 3.3 and 3.4). The two challenges that we address can be summarized by the expression "looking for a needle in a haystack", either when searching for target datasets in large amount of data available on linked data catalogues[6], or when looking for similar instances between two selected datasets, implying the creation of `owl:sameAs` statements between them.

For the sake of clarity, we note that by *dataset*, we mean the RDF description of a dataset in accordance with the definition in the Vocabulary of Interlinked Datasets (VoID)[7]: "A dataset is a set of RDF triples that are published, maintained or aggregated by a single provider". In the same context, the term *linkset* is defined in line with the VoID linkset definition – "A collection of RDF links between two void:Datasets", i.e. triples, in which the subject belongs to a different dataset than that of the object.

The remainder of the chapter is organized as follows: section 3.2 presents state-of-the-art approaches on candidate dataset identification, dealing with

6 Linked data catalogues such as http://datahub.io/.

7 http://vocab.deri.ie/void.

the challenge of linking data. Section 3.3 describes in detail the various data heterogeneity types that hinder the linking task. In section 3.4, we provide the different techniques that are applied by the majority of data linking tools and approaches. We then compare and discuss different state-of-the-art tools/approaches according to these techniques. Finally, section 3.5 provides a conclusion and a discussion of open issues.

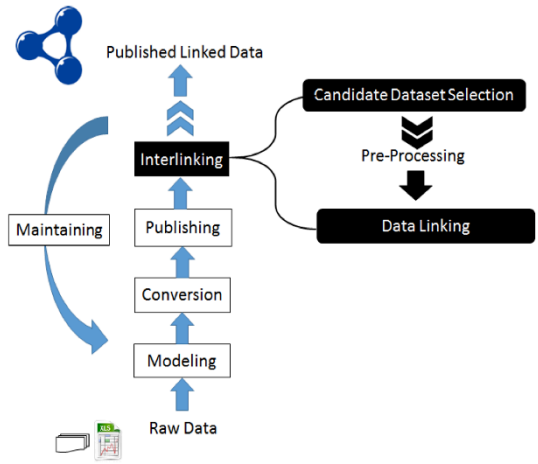


Figure 3.2. *Data linking challenge positioning in the linked data life cycle*

3.2. Datasets recommendation for data linking

Coreference resolution is a common thread across many communities, which is referred to as entity matching, entity disambiguation, cross-document coreference, duplicate record detection or record linkage. These terms all describe the process of determining the presence of different and heterogeneous descriptions of the same real-world objects and also the process of determining links and relations among these descriptions in order to make their correspondence explicit. Coreference resolution can build on a large body of related work across different communities. For example within database communities, we refer the interested reader to the works of Winkler *et al.* [WIN 06] on record linkage and Elmagarmid *et al.* [ELM 07] for duplicate detection. In the natural language processing field, we cite the survey of Soon *et al.* [SOO 01] where coreference resolution can be seen as the task

of finding all expressions that refer to the same entity in a text. The focus of this paper is coreference resolution in the domain of linked web data, where we refer the reader to the survey of Ferrara *et al.* [FER 13].

The purpose of this section is to emphasize the challenges of finding candidate RDF datasets to link to, also known as *candidate datasets identification*, *selection* or *recommendation*. We start by defining the process of dataset recommendation followed by an overview of related work.

3.2.1. *Process definition*

In a data linking scenario, where the user has no or little prior knowledge of the existing datasets on the Web, the first challenge is to identify potential target datasets that may contain identical instances as those contained in a given source dataset. For this purpose, let us take a step back from the naive methods which have been usually adopted, i.e. applying one of the two following solutions: (i) using brute force for combining all the possible pairs of datasets to the interlinking tool; and (ii) requesting the user to select the most suitable datasets following their intuition and background knowledge.

With the rapid growth of the Web of data, and specifically the LOD cloud, an exhaustive search of all existing datasets in available catalogues is manually unfeasible. Hence, the most common linking tradition is limited to targeting popular and cross-domain datasets such as DBpedia [AUE 07] and GeoNames [WIC 12]. However, many other potential LOD datasets have been ignored which have led to an inequitable distribution of links and consequently, a limited semantic consumption in the linked data graph.

The proposed solution for easing this task is the recommendation of a set of candidate datasets to be linked, which consequently reduces considerably the human effort in searching candidate datasets. Recommender systems in general have been an active research area for more than a decade. Typically, recommendation approaches are based on explicit score ratings for candidate items. This scoring can produce an ordered list of suitable results or even more to reduce the search space to the top n most suitable recommendations.

With respect to finding relevant datasets on the Web, several application scenarios have been proposed, from which we cite federated search and data linking.

Federated search has come from the information retrieval area to meet the need of searching multiple data sources with one query. Hartig and Ozsu [HAR 14] presented a tutorial on linked data query processing. They provided basics about linked data SPARQL[8] queries, data source selection (which concerns us here), and query optimization strategies. In this context, we cite the works of [WAG 13] and [WAG 14] which present techniques of querying distributed data sources based on well-known data mining strategies. In other words, these techniques contextually identify related datasets which correspond to the information needed by the user queries. A used feedback-based approach to incrementally identify new datasets for domain-specific linked data applications is proposed in [DEO 12]. User feedback is used as a way to assess the relevance of the candidate datasets. Furthermore, we cite LODVader [BAR 16], a framework for LOD Visualization, Analytics and Discovery, which proposes to compare datasets using the Jaccard coefficient based on *rdf:type*, *owl:Classes* and general predicates.

In the following, we cite approaches that have been devised for candidate dataset recommendation and which are directly relevant to the data linking task.

3.2.2. *Dataset recommendation for data linking based on a Semantic Web index*

Nikolov *et al.* [NIK 11] proposed a keyword-based search approach to identify candidate sources for data linking. The approach consists of two steps: (i) searching for potentially relevant entities in other datasets using as keywords randomly selected instances over the literals in the source dataset, and (ii) filtering out irrelevant datasets by measuring semantic concept similarities obtained by applying ontology matching techniques.

3.2.3. *Dataset recommendation for data linking based on social networks*

Leme *et al.* [LEM 13] presented a ranking method for datasets with respect to their relevance to the interlinking task. The ranking is based on Bayesian

8 The SPARQL protocol and RDF query language: https://www.w3.org/TR/rdf-sparql-protocol/.

criteria and on the popularity of the datasets, which affects the generality of the approach (the well-known cold-start problem). The authors extend this work and overcome this drawback in [LOP 14] by exploring the correlation between different sets of features – properties, classes and vocabularies – and the links to compute new rank score functions for all the available linked datasets.

3.2.4. *Dataset recommendation for data linking based on domain-specific keywords*

Mehdi *et al.* [MEH 14] proposed a method to automatically identify relevant public SPARQL endpoints from a list of candidates. First, the process needs a set of domain-specific keywords as input, which are extracted from a local source or can be provided manually by an expert. Then, using natural languages processing techniques and query expansion techniques, the system generates a set of queries that seek exact literal matches between the introduced keywords and the target datasets, i.e. for each term supplied to the algorithm, the system runs a matching with a set of eight queries: {original-case, proper-case, lower-case, upper-case} * {no-lang-tag, @en-tag}. Finally, the produced output consists of a list of potentially relevant SPARQL endpoints of datasets for linking. In addition, an interesting contribution of this technique is the bindings returned for the subject and predicate query variables, which are recorded and logged when a term match is found on some SPARQL endpoints. These records are particularly useful in the linking step.

3.2.5. *Dataset recommendation for data linking based on topic modeling*

A recent approach is presented by Röder *et al.* [RÖD 16], where the authors present *Tapioca*, a linked dataset search engine for topical similarity of datasets. Topics are frequent schema classes and properties extracted from the dataset metadata. The similarity of two datasets is defined as the similarity of their topic distributions, which are extracted using the latent Dirichlet allocation – a generative model for the creation of natural language documents.

3.2.6. *Dataset recommendation for data linking based on topic profiles*

The main goal of a recommender system is to reduce the cost of identifying candidate datasets for the interlinking task. Some systems may provide recommendations with high quality in terms of both precision and recall, but only over a small portion of datasets (as is the case in [LEM 13]). In this context, we cite a topic-based approach introduced in [ELL 16a], which focuses on the LOD search space size reduction challenge. This recommendation approach relies on the direct relatedness of datasets as emerging from the topic – dataset bipartite graph[9] produced through the profiling method of Fetahu *et al.* [FET 14]. The main intuition behind this approach is the consideration of topic profiles that provide reliable connectivity indicators – *the connectivity behavior* – even in cases where the underlying topic profiles might be noisy. Our assumption is that even poor or incorrect topic annotations will serve as reliable relatedness indicators when shared among datasets. This approach consists of a processing pipeline that combines suitable techniques for dataset sampling, topic extraction and topic relevance ranking. For further understanding of topic profiles, we provide an example for a particular resource, http://data.linkededucation.org/resource/lak/conference/edm2012/paper/21, where the system starts by extracting all the literal values from the corresponding entities by the use of DBpedia Spotlight[DAI 13], i.e. "Learning"[10] and "Student"[11] resources where the system proceeds to extract their corresponding DBpedia categories (which will be referred to henceforth as representative topic profiles), i.e. "Cognition"[12], "Academia"[13] and "Education"[14], "Intelligence"[15].

A series of experiments demonstrated the effectiveness of this approach in terms of common evaluation measures for recommender systems with an

9 The current version of the topic dataset profile graph contains 76 datasets and 185,392 topics and it is accessible via the following SPARQL endpoint: http://data-observatory.org/lod-profiles/profile-explorer.

10 http://dbpedia.org/resource/Learning.

11 http://dbpedia.org/resource/Student.

12 http://dbpedia.org/resource/Category:Cognition.

13 http://dbpedia.org/resource/Category:Academia.

14 http://dbpedia.org/resource/Category:Education.

15 http://dbpedia.org/resource/Category:Intelligence.

average recall up to 81%, translating to a search space reduction where the approach achieved a reduction of the original (LOD) search space of up to 86% on average.

3.2.7. *Dataset recommendation for data linking based on intensional profiling*

In this section, we discuss a candidate dataset recommendation approach which skips the learning step and adopts the notion of *intensional profile* - a set of schema concept descriptions representing the dataset [ELL 16b]. The intuition is the following: datasets that share at least one pair of semantically similar concepts are likely to contain at least one pair of instances to be linked by an `owl:sameAs` statement. The recommendation approach consists of two steps: (1) the identification of a cluster of datasets that share schema concepts with respect to the source dataset, and (2) the ranking of datasets in each cluster with respect to their relevance.

The first step identifies concept labels that are semantically similar by using a similarity measure based on the frequency of term co-occurrence in a large corpus (the Web) combined with a semantic distance based on the WordNet without relying on string matching techniques [HAN 13]. For example, this allows us to recommend to a dataset annotated by "school" and one annotated by "college". In this way, we form the clusters of "comparable datasets" for each source dataset. The intuition is that for a given source dataset, any of the datasets in its cluster is a potential target dataset for interlinking.

The second step focuses on ranking the datasets in the cluster with respect to their importance to the source dataset. This allows the evaluation of results in a more meaningful way and, of course, provides quality results to the user. The ranking criterium should not be based on the amount of schema overlap, because the potential to link instances can be found in datasets sharing 1 class or sharing 100. Therefore, we need a similarity measure on the profiles of the comparable datasets. We have proceeded by building a vector model for the document representations of the profiles and computing cosine similarities.

The approach was evaluated using the current topology of the LOD as the evaluation data. The different experiments described in [ELL 16b] show a high performance of the introduced recommendation approach with an average precision of 53% for a recall of 100%. Furthermore, as a result of

the recommendation process, the user is given not only candidate datasets for linking, but also pairs of classes in which to look for identical instances. This is an important advantage, allowing us to more easily run linking systems such as SILK [JEN 10] in order to verify the quality of the recommendation and perform the actual linking (see section 3.3).

3.2.8. *Discussion on dataset recommendation approaches*

None of the studies outlined in [NIK 11, MEH 14, RÖD 16] have evaluated the ranking measure in terms of precision/recall, except for [LOP 14], which, according to the authors, achieves a mean average precision of approximately 60% and an excepted recall of 100%. However, a direct comparison to the topic-based and intension-based approaches seems unfair since the authors did not provide the set of considered datasets as sources and the corresponding ranking scores or the corresponding target list. Furthermore, in line with the considered state-of-the-art approaches, we highlight the efficiency of the intension-based recommendation in overcoming a series of complexity related problems, precisely, considering the complexity to generate the matching in [NIK 11], to produce the set of domain-specific keywords as input in [MEH 14] and to explore the set of features of all the network datasets in [LOP 14]. Hence, in the following, we limit our discussion to the two approaches: topic-based versus intension-based dataset recommendation.

The current version of the topic dataset profile graph contains 76 datasets and $185,392$ topics. Working with this already annotated subset of existing datasets is not sufficient and would limit the scope of the topic-based recommendations significantly. In addition, the number of the profiled datasets compared to the number of topics is very small, which in turn appears to be problematic in the recommendation process due to the high degree of topic diversity, leading to a lack of discriminability. One way of approaching this problem would be to index all LOD datasets by applying the original profiling algorithm [FET 14]. However, given the complexity of this processing pipeline – consisting of resource sampling, analysis, entity and topic extraction for a large amount of resources – it is not efficient enough, specifically given the constant evolution of Web data, calling for frequent re-annotation of datasets.

The experimental results reported in [ELL 16b] show that the intension-based approach outperforms the topic-based recommendation approach in

terms of the considered evaluation metrics, achieving a 100% recall versus 81%, respectively, and an average precision of up to 53% versus 19%. The low performance of the topic-based approach can be explained by the weakness of the considered learning data – topology of LOD Cloud (2014), i.e. the amount of explicitly declared links in the LOD cloud is certain but far from being complete as be considered as a ground truth. Subsequently, false positive items (recommendations that are considered as false by the evaluation data) would have not been used even if they had been recommended, i.e. they are uninteresting or useless to the user. For this reason, a large amount of false positives occur, which in reality are likely to be relevant recommendations.

While the intension-based approach ensures a better performance and less complexity, we believe that a better ground truth for the topic-based approach may lead to much richer learning data and thus a significantly improved its ranking performance.

3.3. Challenges of linking data

In previous sections, we have seen different methods which are proposed for the selection of candidate datasets for linking with a given (presumably yet unpublished) RDF graph. Provided that this task has been performed successfully, we now have at hand two RDF graphs to link, or else to discover the related entities across these two graphs. A relation of particular interest for the Semantic Web community is that of identity, given by the OWL predicate `owl:sameAs`. However, a given real-world entity may be described differently or even with complementary information in different data sources. Hence, the automatic discovery of identity links may become particularly difficult considering the heterogeneity of data on the Web. In this section, we focus on a number of real-world issues that may arise when comparing resources across datasets, and we illustrate these issues by the means of a fictional example given in Figure 3.3.

Let us imagine that the composer Ludwig van Beethoven is described by two different data sources without making any assumption about the way the data are structured in a general form nor about the used properties describing the composer. When the same entity is heterogeneously described (see the example in Figure 3.3), the comparison becomes much more complex. One of the main difficulties stems from the fact that data may be incomplete. This

would raise the question about identity criteria for comparison – *between which attributes is the comparison done*? Variations in how such attributes are valued or structured may lead to missing true positive links or may produce false positive ones. As shown in Figure 3.3, the descriptions can be expressed in different natural languages, with different vocabularies or with different values. Hence, the comparison can be hindered due to the variations in the expressions according to three dimensions: *value*-based, *ontological* and *logical* dimension. Indeed, the linking quality improves when various aspects of heterogeneity between two resources are handled. In the following, we analyze the most recurrent aspects of heterogeneity by identifying a set of techniques that can be applied to them.

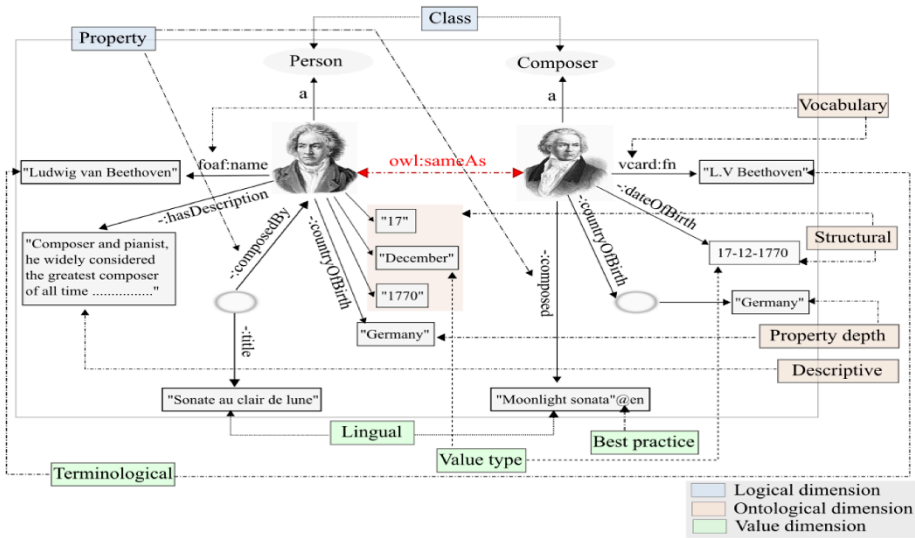


Figure 3.3. *Issues occurring during the linking task. For a color version of this figure, see www.iste.co.uk/pivert/nosql.zip*

3.3.1. *Value dimension*

In the open context of the Web, certain properties of resources are filled by textual (string literal) values expressed in a natural language or numerical values. Any attribute valued with textual or numerical information may potentially raise matching issues. This characterizes the heterogeneities at the

data value level. In the following, we define a list of heterogeneity types of what we call *value dimension*:

1) *Terminological heterogeneity*. A description about an entity could present syntactical or semantic variations with respect to another description about the same entity. These variations concern terms, referring here to a word or to a set of words. This heterogeneity can be expressed in: (i) variation of terms to represent the same concept (synonymy); (ii) variation of the meaning of a term representing different concepts (polysemy); or (iii) variation in spelling (acronyms or abbreviations). The problems of synonymy and polysemy can be solved using lexical databases such as WordNet [MIL 95] which is composed of sets of synonyms called *synsets* where each term belongs to one or more classes. Each *synset* represents a particular concept and is accompanied by a description of the meaning it represents. Indeed, a word should be disambiguated to be compared. The disambiguation consists of assigning the most appropriate meaning for each word in a text according to the context in which it is found. Many works proposed solutions allowing us to find the full form of an acronym or an abbreviation – among others, consider [YAM 11, LI 15]. This issue can be observed between "Ludwig van Beethoven" and "L.V. Beethoven" in our example (Figure 3.3). As we can see, the full name values of the two instances are distinct with the first source preferring to name a person by his/her first name and last name, while in the second source, a person is named using the initials of his/her first name and the full last name. Indeed, the comparison between such values becomes much more complex.

2) *Lingual heterogeneity*. Note that users sometimes publish their data in their native language, leading to a plethora of natural languages used in data descriptions. To better clarify the impact of this fact on the matching decision, consider the example of BNF (French National Library) [SIM 13] and Freebase [BOL 07] that make their existing data available as RDF in their own language, i.e. in French and English respectively. Thereby, if we consider the same entity of Ludwig van Beethoven, we may end up not being able to compare correctly (if it is not impossible) its two representations particularly when applying string similarity measures. This problem is of crucial importance in the open Web of data. Hence, it becomes necessary to discover links among RDF data with multilingual values. To overcome this problem, a few studies have proposed methods for automatic cross-lingual data linking [SCH 09, LES 14, LES 15], considerably reducing the complexity of the comparison task. Machine translation is performed by

[SCH 09, LES 14] to make two descriptions in different languages comparable using the Google Translator API[16] and the Bing Translator API[17], respectively. A more recent study [LES 15] proposed a data linking method based on the BabelNet multilingual lexicon to bridge the language gap. The authors define the resources as vectors of BabelNet identifiers where each of them represents a sense (concept identifier) of a term. The similarity between these vectors is then computed. TR-ESA [NAR 17] is a cross-lingual data linking tool which matches resources whose descriptions are written in different languages. Each resource description is translated into English using the Bing Translator API[18]. Then, a Wikipedia-based representation (a set of concepts) is generated for each resource. Then, these representations are indexed. The main drawback of such approaches is that they do not deal with acronyms in the compared descriptions.

The difficulty of this issue is demonstrated in Figure 3.4, where two resources with multilingual object values are compared. In fact, the data property values in this example are expressed in English and in Chinese (Traditional), which makes it impossible to find a correspondence using string-based similarity metrics without using machine translation or external multilingual resources.

3) *Trangressions of best practices*. Data representations on the value level can differ depending on the degree to which the Semantic Web best practices are respected in the data publishing process. In Figure 3.3, this can be illustrated through the titles of Beethovens work, where one of them ("*Moonlight sonata*") is annotated by the language in which it is given but not the other (the presence/absence of a language tag). Another problem may occur on the values by introducing inappropriate symbols that convey no information. A good practice would be to ignore what we do not know due to the acceptance of the open-world assumption (missing information is not necessarily wrong). This practice is not always respected, where symbols such as "#" can be used to indicate an unknown value as for example when comparing the object values "29-07-1990" and "##-##-1990". Indeed, this may hinder the matching decision. Recently, best practices for the creation, linking and use of multilingual linked data were elaborated by the multilingual linked open data community group[19].

16 http://code.google.com/apis/ajaxlanguage/.

17 http://datamarket.azure.com/dataset/bing/microsofttranslator.

18 http://datamarket.azure.com/dataset/bing/microsofttranslator.

19 https://www.w3.org/community/bpmlod/.

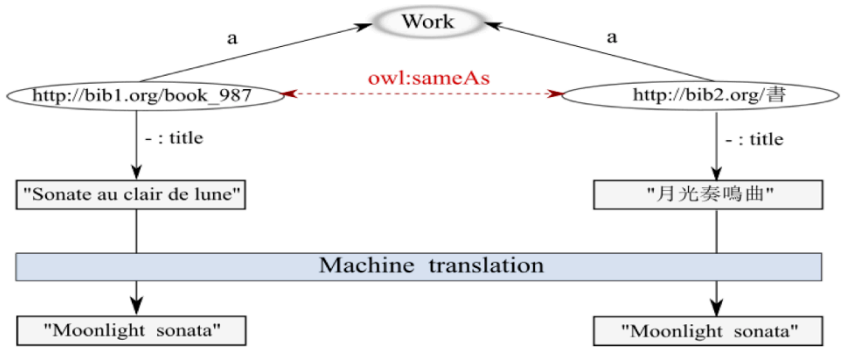


Figure 3.4. *RDF resources described in different languages*

4) *Value-type heterogeneity*. An efficient linking tool is able to deal with different value types expressing the same information. This heterogeneity type concerns differences in encoding of data. Often values are expressed in different formats or by using different value types, as for example, representing an "*age*" value as a string (i.e. "*twenty-six*") or as a number (26). This property describes the month of the year part of the birth date of a person (e.g. February). It is included mostly to ease problems related to structural heterogeneity in the representation of the birth date values. For example, some sources would not represent the date as a date format (consider the example of "17-12-1770"), but would rather represent the same information as a string ("*17 December 1770*"). The main challenge is to provide a means for semantic unification. The birth date would impact the quality of matching results if it is not taken into account. For instance, a possible solution consists of retrieving individual resource values in the RDF graph, transforming different object values in a given format, and standardizing the retrieved information to compare them in a uniform manner. Two data generators aiming at evaluating whether the data linking tools deal with this issue have been proposed: the Semantic Publishing Instance Matching Benchmark (SPIMBENCH) [GAN 15] and LANCE [SAV 15]. They aim to produce different benchmarks by transforming the source instances based on their values, structures and semantics. Some of the problems addressed by both SPIMBENCH and LANCE are the use of different dates ("1948-12-21" vs. "December 21, 1948"), genders ("Male" vs. "M") and number formats ("1.3" vs. "1.30").

3.3.2. *Ontological dimension*

This dimension refers to variations with respect to the properties or classes of instances to compare. We identify four main heterogeneity problems:

1) *Vocabulary heterogeneity*. Classes and properties are often described by using different vocabularies by different data publishers, because the semantics of a given class or a property can be interpreted differently according to the application and the intension. This problem is even more complicated in the context of the open Web of data where all resources are not necessarily described in the same manner. Indeed, different data publishers may differently interpret the semantics of attributes when they decide how to model their data. Indeed, it is not uncommon that the open nature of the Web is one of the main causes of different uses of properties. In fact, anyone can define their own ontology and can describe their own classes. Let us consider the example of the property providing the information about the full name of Beethoven.

$< i_1$, foaf:name, "**Ludwig van Beethoven**"$>$
$< i_2$, vcard:name, "**Ludwig van Beethoven**"$>$
$< i_3$, foaf:name, "**Beethoven**"$>$

For a given data source, such information could be described by the property *foaf:name* (i_1 and i_3), while for another data source, it could be described by the property *vcard:name* (i_2).

Description	Vocabulary	Property
Full Name	contact[20]	http://www.w3.org/2000/10/swap/pim/contact#fullName
	FOAF[21]	http://xmlns.com/foaf/0.1/name
	DBpedia[22]	http://dbpedia.org/ontology/name
	VCard[23]	http://www.w3.org/2006/vcard/ns#Name

Table 3.1. *Vocabularies describing the full name of a person*

Beyond the fact that the same information can be described by different properties (see Table 3.1), the same property can be used to describe different information. In fact, the instances have been described with the same property to describe the full name and the last name, respectively. Yet in both cases, the use of the property *foaf:name* is correct regarding the FOAF ontology[20]. Given

20 http://xmlns.com/foaf/spec/#term_name.

the large amount of existing vocabularies, it becomes necessary to propose a solution aimed at finding correspondences between properties conveying the same information. We note that the linked open vocabularies[21] (LOV) are a catalog containing more than 600 vocabularies in the linked open data cloud. A vocabulary in LOV provides a set of terms (classes and properties) describing a given type of entities. LOV facilitates the reuse of vocabularies through a SPARQL endpoint[22] for retrieving types and their properties. LOV is a good initiative, but note that, to date, it is not exhaustive and has a characteristic of being incomplete (e.g. Yago, Freebase or MusicBrainz are not included in LOV).

2) *Structural heterogeneity*. The description of an entity can be done at different levels of granularity. To take a brief example, consider the birth date of Ludwig van Beethoven. It can be described in a single information field (as a value of the property *vcard:bday*[23]) "17 December 1770" or split over several information fields (multiple properties) "17" (day), "December" (month) and "1770" (year). The last three values are parts of the property *vcard:bday*. Comparing two instances presenting this heterogeneity may impact the matching decision. A possible solution consists of identifying the sub-properties (day, month and year) as belonging to a given type (date) and aggregating them or in decomposing one value into several values to make the properties comparable at the same level of granularity. To the best of our knowledge, the problem of structural heterogeneity is not treated in the literature. However, it is partially resolved using inverted indexes and NLP (natural language processing) techniques in some data linking approaches [JEN 10, LES 14, LES 15, RON 12, SHA 16]. The method consists of indexing each resource by its literals collected at a distance[24] $n >= 1$ in the RDF graph. Then, a vector space model is used to represent each resource description as word feature vector. The resources sharing similar vectors (similar words) are considered to be equivalent or to be linking candidates. By doing so, the resources are compared with respect to their literals without taking into account the properties describing them. The main drawback of this process is the loss of precision when comparing resources

21 http://lov.okfn.org/dataset/lov/.

22 http://lov.okfn.org/dataset/lov/sparql.

23 https://www.w3.org/2006/vcard/ns#bday.

24 A distance in an RDF graph is defined as the minimal number of edges (properties) connecting two resources or a resource and a literal.

represented as bundles of words instead of comparing their literals in pairs (those defined with equivalent properties).

3) *Property depth heterogeneity*. This type of heterogeneity regards the difference in property modeling. In a given dataset, a property can be specified directly (at a distance $n = 1$) through a literal value, while in another dataset, the same information is given by a longer property chain including several triples. In our example (see Figure 3.3), we can observe this problem between the two resources, where the name of the country of birth is a literal that is directly assigned to the resource describing Ludwig van Beethoven, while for the other resource, the same value is assigned through a literal to another resource, which is the URI of the country itself. In fact, we can observe that for the first resource, the country of birth is defined through a data type property, while for the second one, it is defined through an object property which is described by the same value of place of birth "*Germany*". This problem can also be solved by indexing the scope of literals describing each resource. Namely, for each entity the distance to which the literals are collected can be set. However, this is a limitation given the fact that in this context, the further we get from the resource, the more likely it is that we collect noisy information.

4) *Descriptive heterogeneity*. A resource can have several types (concepts) or it can be described with more information (a larger set of properties) in one dataset than in another, as we can see in our example (see Figure 3.3). We can observe that a resource contains more information denoted by a descriptive text (biography) about Ludwig van Beethoven through the property *-:hasDescription*, while the other resource, referring to the same real-world person, is described by a much more austere set of properties. It is obvious that comparing these two resources by this property will not identify any equivalence between them, namely for approaches considering the whole description (all the literals) of a resource or for approaches relying on key properties to compare the resources. In fact, for the former approaches [LES 14, LES 15], the property *-:hasDescription* decreases the similarity value between the two resources, while it is identified as a key property by most of the second approaches [SYM 14, SYM 11, SOR 15]. A key is a set of properties for which there is no two resources sharing the same values for these properties. We note that for a given dataset, several keys may be identified where several combinations of properties uniquely identify the resources.

Let us consider the example given in Figure 3.5 of two musical works *mw_1* and *mw_2* in an RDF dataset. Even if these works seem similar (the same title and the same composer), they refer to two different entities, hence the need to identify which properties, called keys, to compare in order to decide whether they are equivalent work or not. Relying on the key definition, we can deduce that the property *catalogue* is a key. However, a question that arises is: *what about the property* ***genre****?* Discovering keys under the *open-world assumption* (OWA) considers that a property is a key if there are no two resources sharing at least one common value for this property [SYM 14, SYM 11]. On the contrary, discovering keys under the *close-world assumption* (CWA) considers that a property is a key if there is no two resources sharing all the values for this property [SOR 15]. The former supposes that the description of a dataset is complete, while the latter does not.

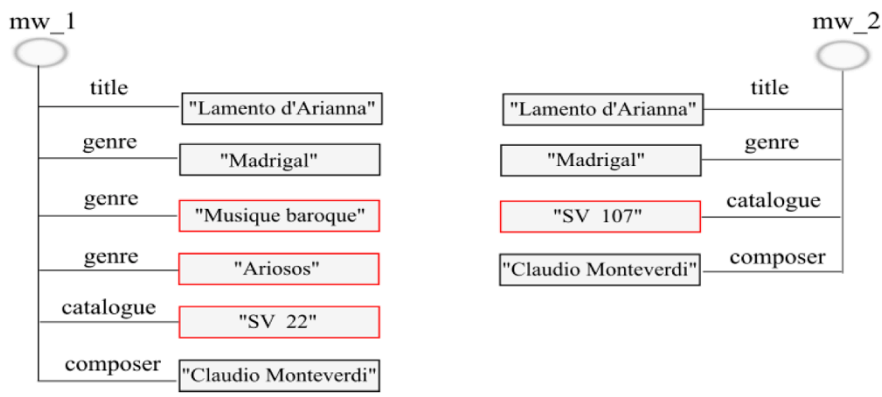


Figure 3.5. *Property keys identification in an RDF dataset. For a color version of this figure, see www.iste.co.uk/pivert/nosql.zip*

3.3.3. *Logical dimension*

This heterogeneity problem refers to the fact that the equivalence between two pieces of information across two datasets is implicit but can be inferred by the help of reasoning methods. We outline the following two main heterogeneity problems:

1) *Class heterogeneity*. This type of heterogeneity regards the class hierarchy level. This is typically the case for two resources belonging to different classes for which an explicit or an implicit hierarchical relationship

is defined (the concepts "*Person*" and "*Composer*", in Figure 3.3, show this issue). Moreover, two instances referring to the same object can belong to two different subclasses of the same class.

2) *Property heterogeneity*. At this level, the equivalence between two values is deduced after performing a reasoning task on properties. Two resources $\mathtt{r}_1$ and $\mathtt{r}_2$ referring to the same entity can have two properties that are semantically reversed (i.e. the properties `composed` and `composedBy`). In this case, these two properties convey the same information, as shown in the example in Figure 3.3:

$< \mathtt{r}_1, \mathtt{composed}, \text{“}\mathtt{Moonlight\ sonata}\text{”} >$

$< \mathtt{r}_3, \mathtt{composedBy}, \mathtt{r}_2 >$

$< \mathtt{r}_3, \mathtt{title}, \text{“}\mathtt{Sonate\ au\ clair\ de\ lune}\text{”} > .$

Here, the instance comparison process has to go beyond the value and property levels by comparing an explicitly specified value and an implicitly specified one between the two entities. Another example of this problem is given by two instances having "*26*" and "*29 July 1990*" as values of *age* and *birthdate* properties, respectively.

3.4. Techniques applied to the data linking process

Identity link discovery (also called *linkset* discovery) requires a three step process to identify equivalent resources across different datasets: prepare data (preprocessing, step 1), align resources (instance matching, step 2) and fix erroneous links generated between some of them (post-processing, step 3). First, the resources need to be represented in a uniform manner. This preprocessing proves necessary when we deal with different vocabularies, when resources are valued by using different languages, or when the number of resources and properties to be compared is too high. To establish links, it is important to compare resources regarding their values. However, the comparison can be done at different levels going from the URI of resources to the description of their neighborhoods in the RDF graph. Finally, once equivalent resources are connected, some systems perform an additional step to evaluate the generated links and therefore to filter some of them identified as erroneous. Several tools and approaches have been surveyed in a recent work [ACH 16], which classified them along these three main steps of data linking process. We note that for each of these steps, a set of techniques has been used

by different data linking tools. An overview of several of the commonly used techniques is presented in the following section.

3.4.1. *Data linking techniques*

Various techniques from different computer science fields are adapted and applied to the data linking problem. We outline several of the main groups of techniques, drawing the readers attention to the fact that a single approach commonly combines several techniques from different groups in its workflow. These techniques, used at different steps of the linking task, are shown in Table 3.2, which summarizes and compares the main data linking tools and approaches.

Machine learning groups a set of techniques that are applied to infer class properties of data instances based on statistical knowledge. These techniques are divided in two main groups – supervised and unsupervised learning methods. In supervised learning, the algorithm is given examples in order to learn a function that allows for the categorization of an unknown individual into one out of a set of predefined classes. Unsupervised learning, also known as clustering, groups individuals together in order to form classes, without prior class knowledge. Note that these techniques do not require *a priori* linguistic knowledge and they are applied on corpora for which no external resources (for example, dictionary or ontology resources) have been developed.

Clustering aims at forming groups of data items based on their similarity with respect to their properties. These entities may be of different types: terms, documents or any given data entity represented as a set of features. In the Web data field, resources sharing similar properties are likely to be interlinked. Therefore, applying clustering methods on a set of instances from different datasets reduces the search space of matching candidates. In [RON 12, ARA 11], the clustering algorithm is used based on the assumption that the resources referring to the same entities are often similar in their description. Both of them start by grouping resources whose distance is very small. Then, their matching algorithm is applied iteratively on similar candidate resources, avoiding the comparison of dissimilar ones.

Binary classification is a *supervised* learning technique, which consists of assigning elements to one out of two given classes based on training data. In data interlinking, this technique can be used to assign pairs of resources

into one of the two predefined classes: (1) a *positive class*, which defines the category of pairs of resources to be matched (the resources of each pair are considered as identical); (2) a *negative class*, where the resources of each pair are considered as different (they will not be interlinked). As sufficient amount of training data is provided, this technique can be used to infer the class (as a match or non-match) of an unseen pair of resources. In [RON 12], aside from the manually generated links, existing matching instance pairs in LOD can also be used to train the classifier.

Linguistic techniques: These techniques perform a linguistic analysis of the textual information describing resources based on knowledge of the language and its structure. Most of the linguistic techniques exploit syntactic, lexical or morphological knowledge. These techniques can be used in the data interlinking task for data preprocessing, addressing a number of problems, such as polysemy or multilingualism.

Word-sense disambiguation (WSD) is a problem of the NLP field, which consists of identifying the appropriate meaning of a word with respect to its context ("*apple*" may have two different senses depending on the context - a *fruit* or a *company*). When performing data linking, this technique can be used to render entities comparable by resolving polysemous terms. In particular, the tools applying this technique [LES 15] explore contextual information to assign the appropriate meaning for a term.

Lexicon resource exploitation: a lexicon is a linguistic resource, commonly used in *information retrieval*, defining terms or structured knowledge bases. For instance, WordNet [MIL 95] is a lexical database of English words organized into synsets (sets of synonyms) where each synset contains a set of semantically close words and their grammatical categories (noun, verb, adjective and adverb). Semantic relations between synsets, such as hypernymy or meronymy, are defined explicitly. This permits word sense disambiguation, and also measuring semantic distances between terms. Multilingual lexical databases can be applied to manage multilingualism by describing all RDF entities across two data sets via a single pivot language [LES 15].

Machine translation is an NLP method, which consists of automatically translating a piece of text from one language to another. In data linking, applying this technique is very important if we have instances described in different languages. Often, textual information is compared across resources and the similarity measures that exploit this information take, input terms in one single language. Through machine translation, instances, or their textual

descriptions, are made comparable (see *lingual heterogeneity* in section 3.3 for more details).

String matching: using this technique, we compute the similarity D (often with $D \in [0, 1]$) between two strings. The result $D = 1$ means that we have an *exact match* between the two strings. If $D > \sigma$ ($\sigma \in (0, 1)$), then we have an *approximate matching* between them with the value D being used as a confidence value. In data linking, this technique is commonly used to measure the correspondence of a resource's property values, in the case of string literals. We note that most matching approaches apply this operation (see Table 3.2). In the example hereafter, we have two RDF triples that describe the same resource, the composer Ludwig van Beethoven:

⟨⟨http ://..../Ludwig_van_Beethoven⟩, name, "L.V. Beethoven"⟩

⟨⟨http ://..../Ludwig_van_Beethoven⟩, name, "L. van Beethoven"⟩

The string matching algorithm computes the similarity value between the two strings "L.V. Beethoven" and "L. van Beethoven". For this purpose, several measures have been proposed. As an illustration, we present three of the most commonly used measures.

– *Levenshtein distance* or *Edit distance* [EUZ 07] is the cost (i.e. the minimum number of operations) required to transform one string to another. Several edit operations are defined, such as: *insertion*, *deletion* or *substitution*. For instance, the Levenshtein distance between the words $A =$ "$L.V.\ Beethoven$" and $B =$ "$L.\ van\ Beethoven$" is 3 and it is computed as follows:

• adding the whitespace character between the first character "." and the character "v" of the word A;

• substitution of the second character ".", of the word A, by character "a";

• adding the character "n" (after the added character "a") to the word A.

– *Jaccard distance* [EUZ 07] is the ratio between the number of characters in common between two strings on the total number of characters, which is defined by the formula:

$$J(A, B) = \frac{|A \cap B|}{|A \cup B|},$$

where A and B are the two considered strings. For instance, the Jaccard similarity coefficient between the two words $A =$ "$L.V.\ Beethoven$" and $B =$ "$L.\ van\ Beethoven$" is computed as follows:

$$J(\text{"}L.V.\ Beethoven\text{"}, \text{"}L.\ van\ Beethoven\text{"}) = 13 \div 14 = 0.92.$$

– *Jaro distance* [EUZ 07] is based on the common characters between two strings A and B. It is defined by the formula:

$$J(A,B) = \frac{1}{3}(\frac{m}{|A|} + \frac{m}{|B|} + \frac{m-t}{m}),$$

where $|A|$ and $|B|$ are the length of the strings A and B, respectively, m denotes the number of corresponding characters and t is the ratio of their transpositions. In other words, the number of all matching characters (in different sequence order) between the two words defines t. Giving the example of two words $A =$ "$L.V.\ Beethoven$" and $B =$ "$L.\ van\ Beethoven$" : $m = 13$ (9 characters are matched) and $t = 0$. Thus, the Jaro distance between these words is computed as follows: $(1 \div 3) \times ((13 \div 14) + (13 \div 16) + ((13 - 0) \div 13)) = 0.91$.

Graph-based: a graph is defined as a set of entities called *vertices*, and a set of *edges* that are defined as pairs of vertices. We often think of an RDF data set as a graph formed by a number of connected triples, with its entities being the vertices and its predicates the edges. In data linking, graph traversal techniques from graph theory [BON 76] are called upon in order to collect information that can be used to describe resources and further compute the similarity between them. For each resource, the information collected where the graph traversal distance is greater than 1 is called contextual knowledge. The approach proposed by Raimond *et al.* [RAI 08] matches two resources, relying on the similarity between them and also on the similarity between their neighbors.

Keys identification: in data linking, for a given class, a key is identified by a property or a set of properties such that there do not exist two instances that refer to different real-world objects and which have the same values for all of these properties. Therefore, two instances, belonging to the same class, across two datasets, can be matched with an owl:sameAs link if they share the same values for the class key. Key identification reduces considerably the search space and is often applied as a data preprocessing step (see *descriptive heterogeneity* in section 3.3 for more details).

Feature-based techniques [TVE 77]: this group of techniques of data representation, used in information retrieval, consists of representing a document (or any other data element for that purpose) in a model by using a set of features (terms) that describe this document. In the context of data interlinking, this technique is used to index each resource by a

document of terms (called *virtual document* or *pseudo-document*), in which each term is a part of a string literal collected within a given distance to the resource in its RDF graph [LES 14, LES 15, RON 12]. Two documents $D_1 = \{t_1, t_2, .., t_n\}$ and $D_2 = \{t'_1, t'_2, .., t'_m\}$ are represented by vectors such as $D_1 = [v_1, v_2, .., v_o]$ ($o = n$ if $n > m$ or $o = m$ otherwise) and $D_2 = [v'_1, v'_2, .., v'_o]$ ($o = n$ if $n > m$ or $o = m$ otherwise), where v_i (v'_i) represents the weight of the term t_i (t'_i respectively) considering the document D_1 (D_2 respectively). Here, the objective is to provide a similarity comparison in a structure, called *similarity vector*, that will be understood and processed by learning algorithms. In data interlinking, we distinguish two main feature representations:

– *Vector model* [SAL 75]: for the similarity computation between documents, one of the best known weighting schemes and commonly used is TF–IDF (term frequency–inverse document frequency). It is based on the frequency (number of occurrences) of a term in the document and across the set of documents in the corpus. To compute the distance between pairs of documents (i.e. resources), several similarity measures can be used. The cosine similarity [EUZ 07] is the most common one; it calculates the cosine of the angle between two vectors.

– *Boolean model*: in this representation type, the similarity vector v produced is binary, i.e. each similarity value is either 0 or 1. The value 1 means that we have an *exact match* between two terms t_i and t'_i.

3.4.2. *Discussion*

Table 3.2 provides a comparison between the surveyed tools/approaches classified according to the three steps of a data linking process presented in section 3.4, i.e. preprocessing, data linking and post-processing steps. As we can see, each of them can perform more than one step. We refer to such tools/approaches as hybrid. For each step of a matching process, an approach performs a set of techniques. Moreover, each tool/approach is described by its principle (inputs and outputs), the specificity of its application to a particular field (music, LOD), its management of multilingualism, and, in the case where it is implemented in a tool: its degree of automation and its participation in the Ontology Alignment Evaluation Initiative[25] (OAEI) campaign. In this table, the symbol "/" means that the information is unavailable.

25 http://oaei.ontologymatching.org/.

<table>
<tr><th>Approaches</th><th>Data Linking Step</th><th>Technique</th><th>Input</th><th>Output</th><th>Domain</th><th>Multilingualism</th><th>Degree of Automation</th><th>Evaluation</th></tr>
<tr><td>Atencia et al. [ATE 12]
SAKey [SYM 14]</td><td rowspan="4">Preprocessing</td><td rowspan="2">Keys identification</td><td rowspan="6">RDF datasets</td><td rowspan="2">Keys set</td><td rowspan="6">LOD</td><td rowspan="5">No</td><td>/</td><td rowspan="4">No</td></tr>
<tr><td>KD2R [SYM 11]
ROCKER [SOR 15]</td><td>Automatic</td></tr>
<tr><td>EAGLE [NGO 12]</td><td rowspan="2">Link specification Learning</td><td rowspan="2">Link specification</td><td rowspan="3">Semi-automatic</td></tr>
<tr><td>ActiveGenLink [ISE 13]</td></tr>
<tr><td>SLINT [NGU 12]</td><td rowspan="3">Preprocessing
Matching</td><td>Keys identification
Blocking
String matching
Clustering
String matching</td><td rowspan="3">Linkset</td><td rowspan="2">Automatic</td></tr>
<tr><td>SERIMI [ARA 11]</td><td>Yes(2011)</td></tr>
<tr><td>Lesnikova et al.
[LES 15]</td><td>WSD
Multilingual lexical resources
Feature-based</td><td>Yes</td><td>/</td><td>/</td></tr>
<tr><td>Rong et al. [RON 12]</td><td>Feature-based
Clustering
Binary classification</td><td>No</td><td>Semi-automatic</td><td>No</td></tr>
</table>

Lesnikova *et al.* [LES 14]		Machine translation Feature-based				Yes	/	/
Raimond *et al.* [RAI 08]	Matching	Graph-based			Music	No	/	/
SILK [JEN 10]		Link specification	Link specification		LOD		Semi-automatic	No
LIMES [NGO 11]								
Legato [ACH 17]	Preprocessing Matching Post-processing	Cleaning Feature-based Link repairing	RDF datasets					Yes(2017)
RDF-AI [SCH 09]	Preprocessing Matching Post-processing	Machine translation String matching Inconsistency checking	RDF datasets + Link specification	Linkset/ Merged datasets				No
KnoFuss [NIK 07]	Matching Post-processing	String matching One-to-one filter		Linkset				
Paulheim *et al.* [PAU 14]	Post-processing	Outlier detection	RDF datasets	RDF datasets			/	/
Guizol *et al.* [GUI 13]		Partitioning	Bibliographic records	Link validation	Bibliographic knowledge			

Table 3.2. *Summary table of the main data linking tools/approaches*

We can note that string matching techniques are used by every approach in the data linking step as they are the obvious techniques for comparing resource values. For preparing data in the preprocessing step, the used techniques differ according to the purpose of the approach that can be either *search space reduction* or *representing the resources* in a uniform manner. However, it seems clear that there are few approaches performing the final step of post-processing. This can surely be explained by the fact that most of the approaches trust their process of matching. The same remark applies, in fact, to multilingualism where only few approaches tackle this issue. However, it is obvious that this criterion presents a crucial problem which will be resolved particularly to match equivalent resources described in multiple languages. Finally, we note that, compared with the total number of proposed instance matching systems, very few of them participated in the OAEI evaluation campaign.

3.5. Conclusion

Efficiently discovering candidate datasets for linking and the links between these candidates are both challenging tasks, given the size and diversity of the Web of data. In this chapter, we first focus on the identification of candidate datasets for data linking where we provide an overview of various existing approaches leading to a comprehensive discussion on the topic. Given a source data set to be linked, and once the target dataset is identified, it is important to deal with different heterogeneity problems that may occur between these two datasets, such as differences in descriptions on the value, ontological or logical level in order to compare the resources they contain efficiently. In this context, we identified and then provided the possible solutions to these heterogeneities that exist in the literature.

The datasets connected by `owl:sameAs` links form a non-oriented graph of very large size. The absence of strong connectivity (due to lack of `owl:sameAs` links) prevents the use of the LOD as a ground truth to evaluate the quality of results proposed by a dataset recommendation system. The identification of the related components in this graph is of great importance in order to allow the recommendation systems to be evaluated, as these related components may constitute benchmarks for evaluating the quality of referral systems. Hence, an open issue with regard to the candidate dataset recommendation task is to develop a reliable and complete ground truth in order to provide a common

benchmark to the community. One possible solution may be to use crowd sourcing techniques.

On the data linking level, we provided an overview of the different techniques applied on each step in service of the global linking task. We consider the linking process as a pipeline composed of preprocessing, data linking and post-processing steps. Finally, we described and compared different state-of-the-art approaches and tools according to these steps and to the surveyed techniques. As a conclusion to our study, we note that during the past years, significant progress has been made in the field with numerous off-the-shelf tools now available to the data community at large. However, we also outline that more effort is needed in order to ensure matching tools can cope with certain more difficult and less studied heterogeneity types. Particularly, the value, ontological and logical heterogeneity dimensions must be paid more attention in future work. The challenge of linking multilingual data also remains largely unexplored.

3.6. Bibliography

[ACH 16] ACHICHI M., BELLAHSENE Z., TODOROV K., "A survey on web data linking", *Ingénierie des Systèmes d'Information*, vol. 21, nos 5-6, pp. 11–29, 2016.

[ACH 17] ACHICHI M., BELLAHSENE Z., TODOROV K., "Legato results for OAEI 2017", *Proceedings of the 12th International Workshop on Ontology Matching co-located with the 16th International Semantic Web Conference (ISWC 2017)*, Vienna, pp. 146–152, October 2017.

[ARA 11] ARAUJO S., HIDDERS J., SCHWABE D. *et al.*, Serimi-resource description similarity, rdf instance matching and interlinking, Reports, arXiv preprint arXiv:1107.1104, 2011.

[ATE 12] ATENCIA M., DAVID J., SCHARFFE F., "Keys and pseudo-keys detection for web datasets cleansing and interlinking", *Knowledge Engineering and Knowledge Management: Proceedings of the 18th International Conference EKAW, Galway*, pp. 144–153, October 2012.

[AUE 07] AUER S., BIZER C., KOBILAROV G. *et al.*, "DBpedia: A Nucleus for a Web of Open Data", *The Semantic Web, 6th International Semantic Web Conference, 2nd Asian Semantic Web Conference*, Busan, pp. 722–735, November 2007.

[AUE 10] AUER S., LEHMANN J., "Creating knowledge out of interlinked data", *Semantic Web*, vol. 1, nos 1–2, pp. 97–104, 2010.

[BAR 16] BARON NETO C., MÜLLER K., BRÜMMER M. *et al.*, "LODVader: an interface to LOD visualization, analyticsand DiscovERy in real-time", *Proceedings of the 25th International Conference Companion on World Wide Web*, Quebec, pp. 163–166, April 2016.

[BER 01] BERNERS-LEE T., HENDLER J., LASSILA O. *et al.*, "The semantic web", *Scientific American*, vol. 284, no. 5, pp. 28–37, 2001.

[BIZ 09] BIZER C., HEATH T., BERNERS-LEE T., "Linked data – the story so far", *International Journal on Semantic Web and Information Systems*, vol. 5, no. 3, pp. 1–22, 2009.

[BOL 07] BOLLACKER K.D., COOK R.P., TUFTS P., "Freebase: a shared database of structured general human knowledge", *Proceedings of the Twenty-Second AAAI Conference on Artificial Intelligence*, Vancouver, pp. 1962–1963, July 2007.

[BON 76] BONDY J.A., MURTY U.S.R., *Graph theory with applications*, vol. 290, Macmillan, 1976.

[DAI 13] DAIBER J., JAKOB M., HOKAMP C. *et al.*, "Improving efficiency and accuracy in multilingual entity extraction", *I-SEMANTICS 2013 – 9th International Conference on Semantic Systems, ISEM '13*, Graz, pp. 121–124, September 2013.

[DEO 12] DE OLIVEIRA H.R., TAVARES A.T., LÓSCIO B.F., "Feedback-based data set recommendation for building linked data applications", *Proceedings of the 8th ISWC*, Boston, pp. 49–55, November 2012.

[ELL 16a] ELLEFI M.B., BELLAHSENE Z., DIETZE S. *et al.*, "Beyond established knowledge graphs–recommending web datasets for data linking", *Web Engineering – 16th International Conference, ICWE*, Lugano, pp. 262–279, July 2016.

[ELL 16b] ELLEFI M.B., BELLAHSENE Z., DIETZE S. *et al.*, "Dataset recommendation for data linking: an intensional approach", *The Semantic Web. Latest Advances and New Domains – 13th International Conference, ESWC 2016*, Heraklion, pp. 36–51, May–June 2016.

[ELM 07] ELMAGARMID A.K., IPEIROTIS P.G., VERYKIOS V.S., "Duplicate record detection: a survey", *IEEE Transactions on Knowledge and Data Engineering*, vol. 19, no. 1, pp. 1–16, 2007.

[EUZ 07] EUZENAT J., SHVAIKO P., *Ontology Matching*, Springer, 2007.

[FER 13] FERRARAM A., NIKOLOV A., SCHARFFE F., "Data linking for the semantic web", in SHETH A. (ed.), *Semantic Web: Ontology and Knowledge Base Enabled Tools, Services, and Applications*, pp. 169–200, Information Science References, 2013.

[FET 14] FETAHU B., DIETZE S., PEREIRA NUNES B. *et al.*, "A scalable approach for efficiently generating structured dataset topic profiles", *Proceedings of the 11th ESWC*, Springer, 2014.

[GAN 15] GANGEMI A., LEONARDI S., PANCONESI A. (eds), *Proceedings of the 24th International Conference on World Wide Web Companion, WWW 2015*, Florence, May 2015.

[GUI 13] GUIZOL L., CROITORU M., LECLÈRE M., "Aggregation semantics for link validity", *Research and Development in Intelligent Systems XXX, Incorporating Applications and Innovations in Intelligent Systems XXI Proceedings of AI-2013, the Thirty-third SGAI International Conference on Innovative Techniques and Applications of Artificial Intelligence*, Cambridge, pp. 359–372, December 2013.

[HAN 13] HAN L., KASHYAP A.L., FININ T. *et al.*, "UMBC_EBIQUITY-CORE: semantic textual similarity systems", *Proceedings of the *SEM*, Atlanta, pp. 44–52, June 2013.

[HAR 14] HARTIG O., OZSU M.T., "Linked data query processing", *2014 IEEE 30th International Conference on Data Engineering*, pp. 1286–1289, Chicago, March–April 2014.

[HAU 12] HAUSENBLAS M., CYGANKIAK R., "Linked Data Life cycles", http://linked-data-life-cycles.info/, 2012.

[HYL 11] HYLAND B., TERRAZAS B., CAPADISLI S., Cookbook for Open Government Linked Data, Report , W3C Task Force-Government Linked Data Group, 2011.

[ISE 13] ISELE R., BIZER C., "Active learning of expressive linkage rules using genetic programming", *Journal of Web Semantics*, vol. 23, pp. 2–15, 2013.

[JEN 10] JENTZSCH A., ISELE R., BIZER C., "Silk – generating RDF links while publishing or consuming Linked Data", *9th International Semantic Web Conference (ISWC'10)*, Shanghai, November 2010.

[LEM 13] LEME L.A.P.P., LOPES G.R., NUNES B.P. *et al.*, "Identifying candidate datasets for data interlinking", *Proceedings of the 13th ICWE*, Aalborg, pp. 354–366, July 2013.

[LES 14] LESNIKOVA T., DAVID J., EUZENAT J., "Interlinking English and Chinese RDF data sets using machine translation", *Proceedings of the 3rd Workshop on Knowledge Discovery and Data Mining Meets Linked Open Data co-located with 11th Extended Semantic Web Conference (ESWC 2014)*, Crete, May 2014.

[LES 15] LESNIKOVA T., DAVID J., EUZENAT J., "Interlinking English and Chinese RDF data using BabelNet", *Proceedings of the 2015 ACM Symposium on Document Engineering, DocEng 2015*, Lausanne, pp. 39–42, September 2015.

[LI 15] LI C., JI L., YAN J., "Acronym disambiguation using word embedding", *Proceedings of the Twenty-Ninth AAAI Conference on Artificial Intelligence*, Austin, pp. 4178–4179, January 2015.

[LOP 14] LOPES G., PAES LEME L.A., NUNES B. *et al.*, "Two approaches to the dataset interlinking recommendation problem", *Proceedings of 15th on WISE 2014*, Thessalaniki, pp. 324–339, October 2014.

[MEH 14] MEHDI M., IQBAL A., HOGAN A. *et al.*, "Discovering domain-specific public SPARQL endpoints: a life-sciences use-case", *Proceedings of the 18th IDEAS 2014*, Porto, pp. 39–45, July 2014.

[MIL 95] MILLER G.A., "WordNet: A Lexical Database for English", *Communication ACM*, vol. 38, no. 11, pp. 39–41, 1995.

[NAR 17] NARDUCCI F., PALMONARI M., SEMERARO G., "Cross-lingual link discovery with TR-ESA", *Information Sciences*, vol. 394, pp. 68–87, 2017.

[NGO 11] NGOMO A.N., AUER S., "LIMES – a time-efficient approach for large-scale link discovery on the web of data", *IJCAI 2011, Proceedings of the 22nd International Joint Conference on Artificial Intelligence*, Barcelona, pp. 2312–2317, July 2011.

[NGO 12] NGOMO A.N., LYKO K., "EAGLE: efficient active learning of link specifications using genetic programming", *The Semantic Web: Research and Applications – 9th Extended Semantic Web Conference, ESWC 2012*, Heraklion, pp. 149–163, May 2012.

[NGU 12] NGUYỄN K., ICHISE R., LE B., "SLINT: a schema-independent linked data interlinking system", *Proceedings of the 7th International Workshop on Ontology Matching*, Boston, pp. 1–12, November 2012.

[NIK 07] NIKOLOV A., UREN V.S., MOTTA E., "KnoFuss: a comprehensive architecture for knowledge fusion", *Proceedings of the 4th International Conference on Knowledge Capture (K-CAP 2007)*, Whistler, pp. 185–186, October 2007.

[NIK 11] NIKOLOV A., D'AQUIN M., "Identifying relevant sources for data linking using a semantic web index", *WWW2011 Workshop on Linked Data on the Web*, Hyderabad, March 2011.

[PAU 14] PAULHEIM H., "Identifying wrong links between datasets by multi-dimensional outlier detection", *Proceedings of the Third International Workshop on Debugging Ontologies and Ontology Mappings, WoDOOM 2014, co-located with 11th Extended Semantic Web Conference (ESWC 2014)*, Anissaras/Hersonissou, pp. 27–38, May 2014.

[RAI 08] RAIMOND Y., SUTTON C., SANDLER M.B., "Automatic interlinking of music datasets on the semantic web", *LDOW'08*, Beijing, April 2008.

[RÖD 16] RÖDER M., NGOMO A.-C.N., ERMILOV I. *et al.*, "Detecting similar linked datasets using topic modelling", *Proceedings of the 13th International Semantic Web Conference*, Heraklion, pp. 3–19, May–June 2016.

[RON 12] RONG S., NIU X., XIANG E.W. *et al.*, "A machine learning approach for instance matching based on similarity metrics", *Proceedings of the 11th International Semantic Web Conference*, pp. 460–475, November 2012.

[SAL 75] SALTON G., WONG A., YANG C.-S., "A vector space model for automatic indexing", *Communications of the ACM*, vol. 18, no. 11, pp. 613–620, Boston, 1975.

[SAV 15] SAVETA T., DASKALAKI E., FLOURIS G. *et al.*, "LANCE: piercing to the heart of instance matching tools", *Proceedings of the 14th International Semantic Web Conference*, Bethlehem, pp. 375–391, October 2015.

[SCH 09] SCHARFFE F., LIU Y., ZHOU C., "RDF-AI: an architecture for RDF datasets matching, fusion and interlink", *Proceedings of the IJCAI 2009 Workshop on Identity, Reference, and Knowledge Representation (IR-KR)*, Pasadena, July 2009.

[SCH 12] SCHARFFE F., ATEMEZING G., TRONCY R. *et al.*, "Enabling linked-data publication with the datalift platform", *Proceedings of the AAAI Workshop on Semantic Cities*, Toronto, July 2012.

[SHA 16] SHAO C., HU L., LI J. *et al.*, "RiMOM-IM: a novel iterative framework for instance matching", *Journal of Computer Science and Technology*, vol. 31, no. 1, pp. 185–197, 2016.

[SIM 13] SIMON A., WENZ R., MICHEL V. *et al.*, "Publishing bibliographic records on the web of data: opportunities for the BnF (French National Library)", *The Semantic Web: Semantics and Big Data, 10th International Conference, ESWC 2013*, Montpellier, pp. 563–577, May 2013.

[SOO 01] SOON W.M., NG H.T., LIM D.C.Y., "A machine learning approach to coreference resolution of noun phrases", *Computational Linguistics*, vol. 27, no. 4, pp. 521–544, 2001.

[SOR 15] SORU T., MARX E., NGOMO A.N., "ROCKER: a refinement operator for key discovery", *Proceedings of the 24th International Conference on World Wide Web, WWW 2015*, Florence, pp. 1025–1033, May 2015.

[SYM 11] SYMEONIDOU D., PERNELLE N., SAÏS F., "KD2R: a key discovery method for semantic reference reconciliation", *On the Move to Meaningful Internet Systems: OTM 2011 Workshops – Confederated International Workshops and Posters: EI2N+NSF ICE, ICSP+INBAST, ISDE, ORM, OTMA, SWWS+MONET+SeDeS, and VADER 2011*, Hersonissos, pp. 392–401, October 2011.

[SYM 14] SYMEONIDOU D., ARMANT V., PERNELLE N. *et al.*, "SAKey: scalable almost key discovery in RDF data", *The Semantic Web - ISWC 2014 - 13th International Semantic Web Conference, Proceedings, Part I*, pp. 33–49, Riva del Garda, Italy, 19–23 October 2014.

[TVE 77] TVERSKY A., "Features of similarity", *Psychological Review*, vol. 84, no. 4, p. 327, American Psychological Association, 1977.

[VIL 11] VILLAZÓN-TERRAZAS B., VILCHES-BLÁZQUEZ L.M., CORCHO O. *et al.*, "Methodological guidelines for publishing government linked data", *Linking Government Data*, pp. 27–49, Springer, 2011.

[WAG 13] WAGNER A., HAASE P., RETTINGER A. *et al.*, "Discovering related data sources in data-portals", *Proceedings of the First International Workshop on Semantic Statistics, co-located with the the International Semantic Web Conference*, Sydney, October 2013.

[WAG 14] WAGNER A., HAASE P., RETTINGER A. *et al.*, "Entity-based data source contextualization for searching the web of data", *Proceedings of the 1st International Workshop on Dataset PROFIling & fEderated Search for Linked Data co-located with the 11th Extended Semantic Web Conference*, Anissaras, May 2014.

[WIC 12] WICK M., VATANT B., "The Geonames Geographical Database", http://geonames.org, 2012.

[WIN 06] WINKLER W.E., Overview of Record Linkage and Current Research Directions, Report, U.S. Census Bureau, 2006.

[YAM 11] YAMAMOTO Y., YAMAGUCHI A., BONO H. *et al.*, "Allie: a database and a search service of abbreviations and long forms", *Database*, vol. 2011, article ID bar013, 2011.

4

Big Data Integration in Cloud Environments: Requirements, Solutions and Challenges

4.1. Introduction

Over the past years, two important concepts have emerged in the computing world: Big Data and Cloud Computing. According to the NIST Big Data Public Working Group[1], Big Data is data that exceed the capacity or capability of current or conventional methods and systems. In [ZIK 11], IBM defines the term Big Data as information that cannot be processed or analyzed using traditional processes or tools. It is mainly based on the three-Vs model, where the three Vs refer to the volume, velocity and variety properties. We define these properties as follows:

– *Volume*: this denotes the processing of large amounts of information. Over the past decades, several high technologies have appeared and they accompany people in their everyday life. If they can track and record something, they typically do it. For instance, simple actions (e.g. taking your smartphone out of your pocket, checking in for a plane, scanning your badge into work, buying a song on iTunes, etc.) generate events and data. Managing this big volume of data may be overwhelming for an organization and several challenges arise. To deal with this, we can find a plethora of modern solutions

Chapter written by Rami SELLAMI and Bruno DEFUDE.

1 http://bigdatawg.nist.gov/uploadfiles/M0392v13022325181.pdf.

NoSQL Data Models: Trends and Challenges, First Edition. Edited by Olivier Pivert.

to store, manage and analyze data in order to gain a better understanding of our data and use it efficiently.

– *Velocity*: this signifies the increasing rate at which data flows. IBM considers that this property means how quickly the data is collected, stored and retrieved [ZIK 11]. It is noteworthy that the velocity follows the evolution of the volume characteristic and, where it concerns fast-moving data, we can call this streaming data or complex event processing. Whether we can handle the data velocity or not, it will help researchers and business experts in making valuable decisions and using data efficiently. To deal with data velocity issues, some researchers suggest conducting data sampling and data streaming analysis.

– *Variety*: this refers to the diversity of data stores and data structures. With the emergence of new technologies (e.g. Cloud Computing, sensors, smart devices, social networks, etc.), the resultant data have become complex since it is a combination of structured, semi-structured and unstructured data. To deal with this characteristic, we today find several kinds of data stores (e.g. relational data stores, NoSQL data stores, etc.) allowing us to store those heterogeneous data. In addition, there exist plenty of approaches and solutions using data integration based on either a mediator/wrapper or a unique API, and supporting heterogeneity.

In addition to these three characteristics, several people have also proposed adding more Vs to this basic definition. For example, we can cite the veracity that is widely proposed and represents the quality of data (accuracy, freshness, consistency, etc.). There is also the data volatility representing for how long the data is valid and for how long it should be stored.

In parallel with the emergence of the Big Data era, Cloud Computing has appeared as a new computing paradigm enabling on-demand and scalable provision of resources, platforms and software as services. Cloud Computing is often presented at three levels [BAU 11]: Infrastructure-as-a-Service (IaaS), giving access to an abstracted view of the hardware; Platform-as-a-Service (PaaS), providing programming and execution environments to the developers; and Software-as-a-Service (SaaS), enabling software applications to be used by the Cloud's end users.

Owing to its scaling and elasticity properties, Cloud Computing provides interesting execution environments for several emerging applications such as Big Data (data-intensive). These applications interact with various

heterogeneous data stores and question the "one size fits all" data management principles [STO 05]. This promotes the use of specialized Cloud data management infrastructures, also known as NoSQL systems. These solutions are able to perform orders of magnitude better than the traditional (and generic) relational database management systems (DBMS). This implies that data-intensive Cloud applications usually need to access and interact with different relational and NoSQL data stores with heterogeneous APIs, data models and query languages. A possible remedy to this high heterogeneity between data stores is data integration [LEN 02, DOA 12]. This refers to a set of techniques and approaches used (1) to combine data residing in different data stores and (2) to provide users with a unified view of these data. This unified view is called in general global schema. There are a plenty of techniques and approaches which allow data integration, but we can roughly classify them into two categories: mediation-based systems and federation-based systems. Mediation is a classical approach for querying heterogeneous sources and it is usually based on two main parts: the mediator and the adapter (also called the wrapper) [GAR 97]. This kind of system is based on a global schema and there are some substantial works proposing a schema mapping-based solution to integrate heterogeneous data. We also find the federated database systems [HEI 85, SHE 90]. These are database management systems that transparently map multiple autonomous database management systems into one single unit.

Integrating Big Data today is very challenging and cumbersome. The classical data integration techniques are not suitable and do not support the new requirements of applications in a Big Data and Cloud Computing context. For example, classical approaches do not support NoSQL data stores and do not consider Big Data characteristics during query optimization. For this purpose, we propose to focus, in this chapter, on the existing solutions of the state of the art supporting Big Data integration in Cloud environments.

The rest of this chapter is organized as follows. Section 4.2 analyzes Big Data integration requirements in Cloud environments. We use these requirements to study and analyze the state of the art in sections 4.3, 4.4, 4.5 and 4.6. In each section, we present a set of substantial works and conclude with a comparative analysis of them all. Section 4.7 gives a global analysis and comparison of all of the works and ends with the presentation of some open issues. Section 4.8 provides a conclusion.

4.2. Big Data integration requirements in Cloud environments

Most of the data-intensive applications deployed in cloud environments question the "one size fits all" [STO 05] data management principles and promote the use of specialized Cloud data management infrastructures, also known as NoSQL systems. These solutions are able to perform orders of a magnitude that is better than the traditional (and generic) relational DBMS. NoSQL, which stands for Not only SQL, refers to a new generation of data stores. Unlike relational data stores, they are not standardized. Indeed, each NoSQL data store has its own data model, its own query language, its own API, etc. They are mostly open-source, schema-less, largely distributed database systems that enable rapid, *ad-hoc* organization and analysis of extremely high-volume, disparate data types [SAD 12]. They mainly come from web companies developing very highly intensive web applications such as Facebook, Amazon and Twitter. NoSQL data stores are sometimes referred to as Cloud databases, non-relational databases or Big Data databases.

To store and query data, each NoSQL data store has its own mechanism. We can find some NoSQL systems that provide a simple interface/API. Some others use declarative query languages (e.g. SQL-like, SPARQL-like, etc). NoSQL data stores are classified into four families of data stores depending on their underlying data model: graph, key/value, document and column. For example, CouchDB, MongoDB and Cloudant belong to the family of document data stores.

Moreover, to support massive data, new distributed models of computation such as Map/Reduce [DEA 04] or Spark [ZAH 10] have been proposed, allowing data-parallelism at a very large scale. These models are implemented into the Hadoop software platform [WHI 15]. Hadoop is the leading open-source platform for Big Data, including a large set of tools which support Big Data processing: data ingestion tools such as Squoop, storage of Big Data in file systems (HDFS) or in structured data stores (such as Hive), batch processing (Map Reduce or Spark), stream processing (Apache Storm or Spark Streaming) and declarative languages such as Pig [OLS 08] or HiveQL [THU 09].

Against this background, readers may notice the high heterogeneity between both (1) the different NoSQL data stores and (2) the relational, Hadoop-based and NoSQL data stores. This heterogeneity is a barrier to the users of these data stores since they should find capabilities of each data

store with respect to their requirements. Then, they should be able to decide which data store to choose. For various and specific requirements, users are sometimes obliged to interact with multiple data stores at the same time. This phenomenon is popularly referred to as the polyglot persistence. Hence, the process of data store discovery and selection becomes more sophisticated. As a first requirement for Big Data integration, we emphasize the following:

R_1*: automatic data store selection and discovery*. In general, Cloud environments support several data stores for each category of data store (relational, NoSQL and Hadoop-based). For instance CouchDB as an implementation for a document data store and MySQL and Postgres for relational data stores. The deployment of an application has to take into account the discovery and the selection of the specific data stores to use. Finally, the process of deploying these data stores together with the application may be quite complex and automation tools are needed.

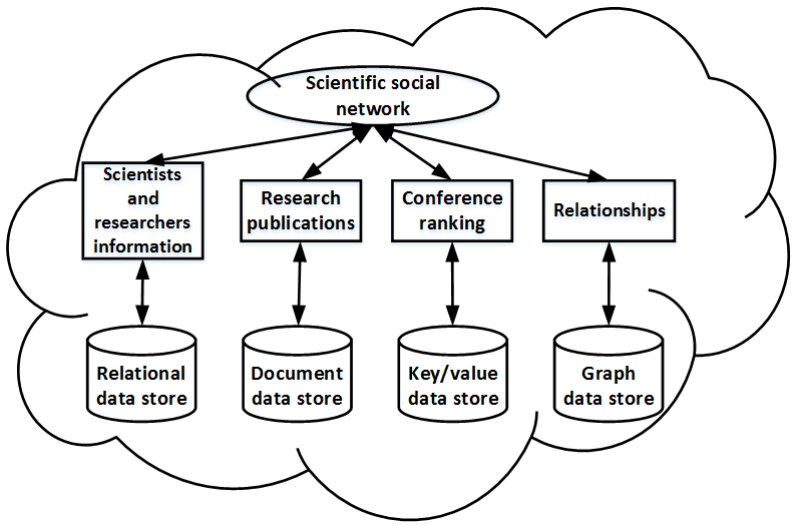


Figure 4.1. *Using multiple data stores in a cloud environment*

When the developer discovers and selects multiple data stores, it is necessary to integrate these stores in order to cover the heterogeneity between the data. In Figure 4.1, we show an example of this situation in which a scientific social network interacts with four heterogeneous data stores: one relational data store and three NoSQL ones (i.e. a key/value, a document and

a graph). The main advantage would be to use specialized data stores, well adapted to the specific requirements of the scientific social network. More precisely, we expect to have:

– one collection named *Person* describing researchers' personal data and their affiliations stored in a relational data store;

– one collection named *Dblp* describing articles with their meta-data and full text stored in a document data store;

– one collection named *ConferenceRanking* describing the rank of conferences stored in a key-value data store;

– one collection named *ScientificRelationship* describing the different relationships between researchers stored in a graph data store. In this example, the querying of the collection *ScientificRelationship* will be supported by a graph-based query language that will be more adapted and expressive than SQL, for example.

It is quite complex and tedious for programmers to interact with each data store using its native API and query language (in this example they are very different). The second requirement is R_2*: unique access for all data stores*. This can be supported, for example, by a unique API capable of manipulating all data stores in the same way that the JDBC API is capable of manipulating all relational data stores.

Even if requirement R_2 is fulfilled, the programmer is faced with an important number of data stores and he/she has to explicitly interact with each data store independently. Against this, we define the third requirement, which is R_3*: transparent access for all data stores*. For example, let us suppose that the *scientific social network* at some point needs to retrieve the affiliation and the name of authors with at least one paper published in an "A" ranked conference. In our scenario, it is quite challenging since it involves one relational and two different NoSQL data stores. In fact, since *Dblp*, *ConferenceRanking* and *Person* use different data models, the developers have to write a program (e.g in Java) to interact with each data store with proprietary APIs and implement the join operation in memory using iterations. This remains (1) purely programmatic, (2) not optimal and naive and (3) time consuming. If all of the data stores are in some way integrated in a "global schema", it is possible to use a declarative language such as SQL to express our query as a combination of joins between the three collections and a selection

on the rank attribute. In the context of Cloud Computing and Big Data, it is not so easy because some data stores can be schema-less (e.g NoSQL data stores or Hadoop's data store as HDFS) and do not support a declarative query language.

Global queries, which are queries targeting data from different data stores, need a specific engine for processing and optimizing their execution. Considering the Big Data and Cloud context, optimization needs to consider new criteria such as minimizing data movement between nodes and data transformations. The fourth requirement is R_4*: global query processing and optimization*.

4.3. Automatic data store selection and discovery

4.3.1. *Introduction*

Choosing one or multiple data stores based on data requirements is a very important step before integrating heterogeneous data stores and deploying and running applications in a Cloud environment. In this context, requirement R_1*: automatic data store selection and discovery* can be refined in sub-requirements R_{11}*: definition of the application needs and requirements towards data*, R_{12}*: exposure of data stores capabilities* and R_{13}*: definition of matching and negotiation techniques between the application needs and data store capabilities*.

In this section, we present the current state of the art. There are some works which propose only solutions based on the use of models (e.g. contract, manifest, XML schemes, etc.) [CAR 12, SNI 12, ZHA 12]. In general, these solutions are standards used on the Cloud. Their models are used to express the application requirements and the data services capabilities. These solutions do not enable effective matching techniques (i.e. the sub-requirement R_{13}). Unlike these approaches, we nowadays find solutions which enable the three sub-requirements at the same time [RUI 11, RUI 12, TRU 11, TRU 12, VU 12, WIT 12, SEL 15].

4.3.2. *Model-based approaches*

Cloud Application Management for Platforms [CAR 12]: CAMP is a specification defined for application management, including packaging and

deployment, in the PaaS level of Cloud environments. Indeed, CAMP provides the application developers with a set of interfaces and artifacts based on the REST architecture in order to manage the application deployment and their use of the PaaS resources. Concerning the data storage/application relationship, this standard allows us to discover the application needs and the data store requirements. The PaaS resource model of CAMP focuses on defining the capabilities of either applications or data storage resources. These resources are *Platform*, *Assembly Template*, *Application Component Template*, *Platform Component Template* and *Capabilities and Requirements*.

Cloud Data Management Interface [SNI 12]: The Storage Networking Industry Association (SNIA) has defined a standard for IaaS in the Cloud. This standard is referred to as the Cloud Data Management Interface (CDMI). Based on the REST architecture [RIC 13], CDMI allows us to create, retrieve, update and delete data in the cloud. In addition, it enables us to describe and discover the available capabilities of the cloud storage offerings but at a lower level (it is more infrastructure-oriented than platform-oriented). Hence, users need to complement it using the data stores proprietary API in the PaaS level.

An ontology-based system for Cloud infrastructure services discovery [ZHA 12]: Zhang *et al.* propose the Cloud Computing Ontology (CoCoOn) which enables us to denote functional and non-functional concepts, attributes and relations of infrastructure services in Cloud environments. Using the CoCoOn ontology, Cloud providers expose a description of their services. In addition, they propose Cloud Recommender, which is a system which implements their ontology in a relational model. This system enables the selection of infrastructure services using a single SQL query to match user requests to the service descriptions. Although their solution is interesting, users cannot discover multiple resources in a single query. The matching is also very simple.

4.3.3. *Matching-oriented approaches*

An automated approach to Cloud storage service selection [RUI 11, RUI 12]: Ruiz-Alvarez *et al.* propose an automatic approach to selecting a cloud storage service according to the application requirements and the storage services capabilities. For this purpose, they define an XML schema based on

a machine-readable description of the capabilities of each storage system. In this model, they define a set of storage characteristics that a Cloud service may provide. For instance, they enable us to describe the cost, the performance and the region of a given service. The goal of this XML schema is twofold: (i) expressing the storage needs of consumers using high-level concepts, and (ii) enabling the matching between consumer requirements and the offerings data storage systems. In order to satisfy the XML schema, the authors propose a mathematical model which tackles the data allocation problem in the context of Cloud Computing. This mathematical model is helpful since it guides the best storage service selection. Indeed, it tries to meet application requirements and the capabilities of Cloud services. In addition, it computes the optimal cost of network allocation. To do so, the authors define an objective function that consists of the linear combination between the cost, the latency and the bandwidth. Although this solution is interesting, the authors consider only single data store applications in their work.

Data contracts for Cloud-based data marketplaces [TRU 11, TRU 12, VU 12]: Truong *et al.* propose a model and specify data concerns in data contracts to support risk-aware data selection and utilization. For this purpose, they define an abstract model to specify a data contract and the main data contract terms. Moreover, they propose some algorithms and techniques in order to enforce the data contract usage. They present a data contract compatibility evaluation algorithm and define how to construct, compose and exchange a data contract. In [TRU 11], they introduce their model for exchanging data agreements in the Data-as-a-Service (DaaS) based on a new type of service, which is called Data Agreement Exchange as a Service (DAES). This model is called Description Model for DaaS (DEMODS) [VU 12]. It is noteworthy that Truong *et al.* propose this data contract to discover data and not the data stores. This will not help developers to choose the appropriate data stores for their applications.

Cloud service selection based on variability modeling [WIT 12]: Wittern *et al.* propose using *Cloud features modeling* based on variability modeling in order to present user requirements and Cloud services capabilities. A feature model is a directed graph in which vertices denote the Cloud features and edges define the relationships between the features. These features represent either the application requirements or the data services capabilities. Based on that, they also define a Cloud service selection process as a methodology for decision-making. Although this approach is automatic and

dynamic, it does not support the discovery and the selection of multiple services at the same time.

Automatic resources discovery for multiple data store-based applications [SEL 15]: Sellami *et al.* propose a manifest-based approach to automatically discover and select multiple data stores in Cloud environments. Indeed, the developer should express his/her requirements in terms of data stores in an abstract application manifest. Each Cloud environment exposes in an offer manifest one or multiple offers that would support the application requirements. A matching algorithm takes both manifests as input and selects the most appropriate offer to the application.

4.3.4. *Comparison*

In this section, we present a comparison of the works presented above. To do so, we fix a set of seven criteria to and compare these works. Criterion 1 corresponds to sub-requirement R_{11} (definition of the application needs and requirements towards data). Criterion 2 corresponds to sub-requirement R_{12} (exposure of data store capabilities) and criterion 3 corresponds to sub-requirement R_{13} (definition of matching and negotiation techniques between the application needs and data store capabilities). We also analyze if these works are declarative (criterion 4) and automatic (criterion 5) in order to ease the description of the requirements and the capabilities in terms of data stores. The capability to address multi-data store environments is given by criterion 6. Finally, criterion 7 corresponds to the Cloud Computing level of the solution. This analysis is showcased in Table 4.1. In this table, the $+$ is used to say that the proposed work completely fulfills a given criterion and the *o* is used for partial fulfillment. The $-$ is used to say that the related work does not propose a solution for a given criterion. It is worth noting that we use this notation throughout this chapter to compare the studied works.

Against this analysis, we conclude that the majority of the studied works mainly support the requirements R_{11} and R_{12}. However, a minority takes into account the requirement R_{13}. In addition, these works do not propose describing application requirements and data store capabilities in a descriptive model even if this modeling ensures more automaticity in data store discovery. Finally, except the work of Sellami *et al.* [SEL 15], the other works do not support multiple data store discovery.

Studied solutions \ Criteria	R_{11}	R_{12}	R_{13}	**Declarative**	**Automatic**	**Multiple data stores**	**Cloud computing level**
CAMP [CAR 12]	+	+	−	−	−	−	PaaS
CDMI [SNI 12]	+	−	−	−	−	−	IaaS
Zhang *et al.* [ZHA 12]	+	+	−	−	−	−	IaaS
Ruiz-Alvarez *et al.* [RUI 11, RUI 12]	+	+	+	+	+	−	PaaS
Truong *et al.* [TRU 12, TRU 11, VU 12]	+	−	*o*	+	+	−	IaaS
Wittern *et al.* [WIT 12]	+	+	+	−	+	−	IaaS
Sellami *et al.* [SEL 15]	+	+	+	+	+	+	PaaS

Table 4.1. *Comparison of the related works enabling automatic data store selection and discovery*

4.4. Unique access for all data stores

4.4.1. *Introduction*

In some cases, applications want to explicitly store and manipulate their data in multiple data stores. Applications already know the set of data stores to use and how to distribute their data on these sources. However, in order to simplify the development process, application developers do not want to manipulate different proprietary APIs, especially when interacting with multiple data stores (e.g. relational, NoSQL, etc.). Two classes of solutions can be used in this case. The first is based on the definition of a neutral API capable of supporting access to the different data stores. The second class is based on the model-driven architecture and engineering methodology [POO 01].

In this section, we focus exclusively on the first class of solutions to ease access to multiple data stores (especially NoSQL and relational data stores). Stonebraker [STO 11] exposes the problems and the limits that a user may encounter while using NoSQL data stores in particular. These problems derive from the lack of standardization and the absence of a unique query language

and API, not only between NoSQL data stores but also between relational and NoSQL data stores. To rule out these problems, there are nowadays substantial research works which propose solutions to provide transparent access to heterogeneous data stores [SEL 14, HAS 10, ONG 14, POL 12, ATZ 12, CAB 13]. All of these works are based on the definition of a neutral API in the same spirit as JDBC [FIS 03] for relational data stores. We start this section by presenting [SEL 14] in detail as a representative example of these works and then we will describe the other works.

4.4.2. *ODBAPI: a unified REST API for relational and NoSQL data stores*

Based on their unified data model (see section 4.5.3), Sellami *et al.* [SEL 14] propose a generic resource model defining the different concepts used in each category of data stores (i.e. NoSQL and relational). These resources are managed by ODBAPI. This API is designed to provide an abstraction layer and seamless interaction with data stores deployed in Cloud environments. Developers can express and execute CRUD (Create, Retrieve, Update and Delete) and complex queries in a uniform way regardless of the type of the data store, whether it is relational or NoSQL. The authors propose to support three types of queries within ODBAPI: (1) simple CRUD queries on a single data store, (2) complex queries on a single data store and (3) complex queries on multiple data stores. An overview of ODBAPI is given in Figure 4.2. The figure is divided into four parts, which we introduce below:

– *Data stores*: first of all, we have the deployed data stores that a developer may interact with during his/her application coding. In the figure, we showcase that a developer may use a relational data store, a document data store (that is, CouchDB) and a key/value data store (that is, Riak).

– *Proprietary APIs and drivers*: second, we find the proprietary API and driver of each data store implemented by ODBAPI. For instance, in our API implementation we use the JDBC API and MySQL driver to interact with a relational DBMS.

– *ODBAPI interface*: the third part of Figure 4.2 represents the ODBAPI interface and the different implementations of each data store. In fact, it represents the shared part between all the integrated data stores and it provides a unique view of the application side. It contains specific implementations of

each data store. The current version of the ODBAPI implementation includes four data stores: (1) relational DBMS, (2) CouchDB, (3) MongoDB and (4) Riak. The addition of a new data store requires the implementation of a new specific driver for ODBAPI.

– *ODBAPI operations*: these operations can be structured in three categories: metadata operations which return information on the data store, CRUD operations which manipulate entities and query operations which support complex queries. Arguments and results of operations are structured using JSON, which allows the exchange of data between heterogeneous data stores.

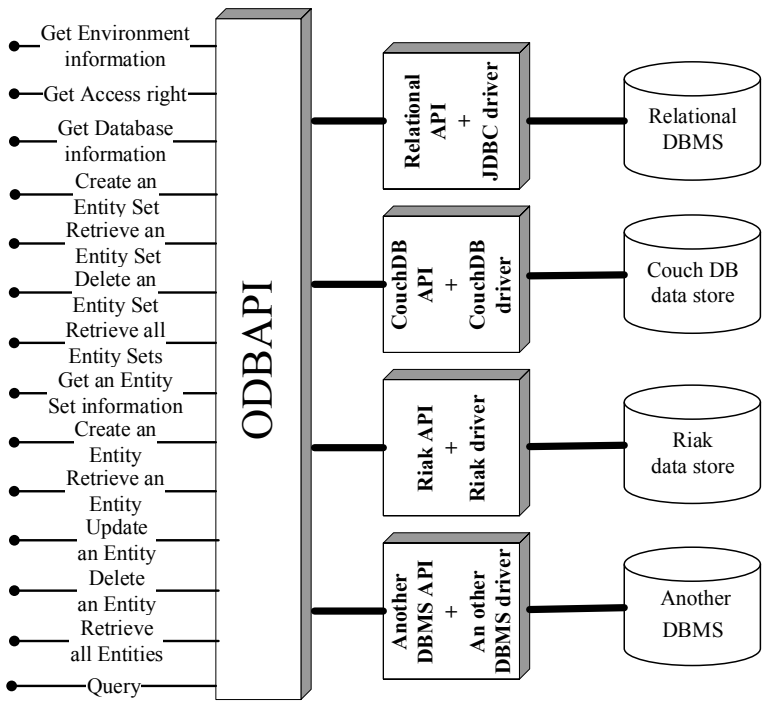


Figure 4.2. *An overview of ODBAPI*

4.4.3. *Other works*

A REST-based API for Database-as-a-Service systems [HAS 10]: Haselmann *et al.* present a universal REST-based API concept. This API

allows us to interact with different Database-as-a-Services (DaaSs) whether they are based on relational or NoSQL data stores. They propose a new terminology of different concepts in either type of data store. They introduce the terms entity, container and attribute to represent, respectively, (i) an information object similar to a tuple in a relational data store, (ii) set of information objects equivalent to a table and (iii) the content of an information object. These terms represent the resources targeted by their API. This API enables either CRUD operations or complex queries execution. However, the authors just describe the API and do not give any details about its implementation. In addition, their resource model is not generic to each category of data store.

OData, an Open Data Protocol [ONG 14]: OData is a REST-based web protocol which allows data querying and updating by building and consuming RESTful APIs. Operations may be either CRUD operations or some complex queries expressed using the OData-defined query language. OData enables us to publish and edit resources via web clients within a corporate network and across the Internet using simple HTTP messages. Resources are identified using URIs and defined based on an abstract data model. Data are represented using a JSON-based format or an XML-based format. Even if this approach seems quite promising, it is more a specification than an implementation especially for the management of complex queries. The query language is also not well defined.

Spring Data Framework [POL 12]: the Spring Data Framework provides some generic abstractions to handle different types of NoSQL and relational data stores. These abstractions are refined for each data store. In addition, they are based on a consistent programming model using a set of patterns and abstractions defined by the Spring Framework. Nevertheless, adding a new data store is not so easy and the solution is strongly linked to the Java programming model.

SOS, a uniform access to non-relational data stores [ATZ 12]: Atzeni *et al.* propose a common programming interface to seamlessly access NoSQL and relational data stores referred to as Save Our Systems (SOS). SOS is a database access layer between an application and the different data stores. To do so, the authors define a common interface to access different NoSQL data stores and a common data model to map application requests to the target data store. They argue that SOS can be extended to integrate relational data stores; meanwhile, there is no proof of the efficiency and extensibility of their system.

ONDM, Object NoSQL Datastore Mapper [CAB 13]: ONDM is a framework aiming to facilitate persistent object storage and retrieval in NoSQL data stores. In fact, it offers to NoSQL-based application developers an Object Relational Mapping-like (ORM-like) API (e.g. JPA API). However, ONDM does not take into account relational data stores.

4.4.4. *Comparison*

We propose a comparison of the studied works based on criteria directly derived from our main objective, which is supporting multiple data store-based application developers in Cloud environments (see Table 4.2). This can be provided by ensuring a unique access for relational and NoSQL data stores. For this purpose, we fix a set of six requirements. Indeed, we check if the proposed solutions provide unique API based on the REST architecture. The API should be RESTful since we are integrating data in Cloud environments. Then, we verify whether these works take into account relational and NoSQL data stores or not. Afterwards, we make sure that these works allow the portability of the source code of the application. Portability introduces the possibility of using different programming languages. Finally, we check the extendability of the proposed APIs. By extendability, we mean the possibility of adding a new data store to the API.

Criteria / Studied solutions	**Unique API**	**REST-based API**	**NoSQL data stores**	**Relational data stores**	**Portability**	**Extendability**
ODBAPI [SEL 14]	+	+	+	+	+	+
Haselmann *et al.* [HAS 10]	+	+	+	+	−	−
OData [ONG 14]	+	+	+	+	+	−
Spring Data Framework [POL 12]	+	+	+	+	−	−
SOS [ATZ 12]	+	−	+	−	−	−
ONDM [CAB 13]	+	−	+	−	−	−

Table 4.2. *Comparison of the related works which ensure unique access to multiple data stores*

Based on the comparison given above, we can conclude that there are many works proposing a neutral API for heterogeneous data stores. These works differ in that they do not support the same type of data stores and for the moment there is no generic solution supporting all families of NoSQL data stores (especially graph data stores). Complex queries are not always supported. These works propose operations limited to one single data store at a time. This requires some kind of "global" layer.

4.5. Unified data model and query languages

4.5.1. *Introduction*

The cornerstone of multiple data store integration solutions is some kind of unified data model. The latter is used to define what is commonly referred to as the "global" schema. This schema provides developers with transparent access to the different data stores. The definition of such a global schema has been widely addressed in the context of the relational data model, of the object-oriented data model [CAR 95] and of the semi-structured data models [PAP 95, MCH 97] (including XML-based data models). In all cases, the global schema is the result of an integration process which produces one single schema from several ones. This is a very complex process, especially when the data models are heterogeneous and when the number of data stores to integrate is high. To support the heterogeneity of the data models, a generic model is chosen that has enough expressivity to describe all of the varieties of the input data models. Historically, this generic model has been the relational one, then semi-structured and sometimes semantic models (inspired by ontology formalism such as RDF or description logic).

It is very difficult to integrate a large number of data stores (several hundreds to thousands). In this context, peer-to-peer data integration systems (e.g Piazza [HAL 04]) propose an approach where the global schema is replaced by a set of partially global schemas. In Piazza, all data stores (or nodes) are relational ones. A Piazza node stores local relations plus semantic mappings to some remote Piazza nodes. A semantic mapping from node A to node B is a transformation that is able to rewrite a query expressed on A's schema to an equivalent query expressed on B's schema. Each node may have different semantic mappings on different nodes and consequently does not have the same view of the other nodes: that is the reason why there is

no global (and shared) schema. Queries are evaluated by a distributed query rewriting engine using recursively local relations and semantic mappings.

The global query language can be either an integrated query language based on a unified data model, or some kind of federated query language which allows us to compose results returned by sub-queries expressed in the native query language of the target data store.

In this section, we present some substantial work proposing different unified data models to manage heterogeneous data integration. We start by introducing classical data integration approaches [CAR 95, PAP 95, MCH 97]. It is noteworthy that we do not take into account these solutions in our analysis since they were not designed for NoSQL data stores and Cloud environments. Then, we introduce the new generation of data integration approaches that support NoSQL data stores and the Cloud Computing era [KIM 92, KIM 94, KOS 10, ONG 14, SUN 13, MIC 15, SEL 16b, SEL 17, DOU 06, DOU 10], focusing on first [SEL 17].

4.5.2. *Data models of classical data integration approaches*

The GARLIC data model [CAR 95] is an extension of the ODMG-93 object model. The main building blocks of this model are objects and values. Each object is uniquely identified and has a type expressed in the data model through an object interface. The latter is composed of a set of attributes, relationships and methods. The values can be either base values (e.g. integers, strings, object references, etc.) or structured values (i.e. interfaces without identity). The extension of the ODMG-93 data model [BAN 94] enables us to support the management of integrity constraints and object references.

The Lightweight Object Repository (LORE) is a database management system for semi-structured data. Its query language is referred to as LOREL, which has a syntax closer to a select-from-where syntax and its data model is called the Object Exchange Model (OEM) [PAP 95, MCH 97]. The latter is a simple nested object model that is represented using a labeled and directed graph. It is also a self-describing model since it is based on labeling in order to describe objects' meaning. This avoids defining a fixed schema in advance and enables us to ensure more flexibility during data modeling. In other words, we can say that each object represents its own schema. Each object in the OEM model is structured by four fields: a label, a type, a value and an object ID.

These works based on semi-structured models are clearly a good starting point to support the integration of NoSQL data stores, since they have a data model with similar characteristics.

4.5.3. *A global schema to unify the view over relational and NoSQL data stores*

Sellami *et al.* [SEL 16a, SEL 16b, SEL 17] propose to use a global data schema to easily integrate relational and NoSQL data stores. Nevertheless, this global schema is minimum and represents a collection of entity sets that are accessible by an application. This choice is deliberate in order to keep the characteristics of NoSQL data stores. It is composed of two key elements: the unified data model and the refinement rules. First, the unified data model offers a unified view of potentially heterogeneous collections and enables us to execute simple queries (i.e. CRUD operations). We illustrate this model in Figure 4.3, based on five concepts. For ease of understanding of the data model, we present some examples in Figure 4.4, which describe three *EntitySets*: *dblp* is of type *document*, *Conference ranking* is of type *key/value*, and *Person* is of type *relational*.

– *The Attribute concept*: this represents an attribute in a data store. In Figure 4.4, the elements *personName*, *Rank* and *year* are concepts of type *Attribute*.

– *The Entity concept*: an *Entity* is a set of one or multiple *Attributes*. In Figure 4.4, we show *Entities* in the *EntitySet* called *Conference ranking*, identified by the *Attribute Conference*.

– *The EntitySet concept*: an *EntitySet* is a set of one or multiple concepts of type *Entity*. In Figure 4.4, *dblp*, *Conference ranking* and *Person* represent *EntitySets* of type document collection, Key/Value database and relational table, respectively.

– *The Database concept*: a *Database* contains one or multiple concepts of type *EntitySet*. In Figure 4.1, we showcase the scientific social network interacting with four *Databases*: a document data store, a relational data store, a key/value data store and a graph data store. For example, the document and relational *Databases* may contain, respectively, the *EntitySets dblp* and *Person* illustrated in Figure 4.4.

– *The Environment concept*: the root concept in our model is *Environment*. This concept represents a pool of concept of type *Database* and an application can choose some of them to interact with. As a concrete example, we can give

the example of the Cloud environment in which the scientific social network interacts with the four data stores depicted in Figure 4.1.

Figure 4.3. *Unified data model*

Figure 4.4. *Examples of EntitySet concepts of type document, key/value and relational*

Second, the refinement rules enrich the unified data model using inter-entity set relationships. This helps application programmers to correctly express their multi-data store queries. For instance, we can define correspondence rules to express a possible join between relational and NoSQL data stores or to remove semantic ambiguities between attributes. Although this unified data model is very interesting, it does not take into account graph data stores that are a very specific NoSQL type.

Based on this unified data model, Sellami *et al.* propose their query language based on a query algebra defined on *EntitySets/Entities*. This query algebra is simple and is composed of the classical unary operations (e.g. selection and projection) and binary operations (e.g. Cartesian product, join, union, etc.). For manipulating complex attributes like those defined in their unified model, more complex algebra can be used, notably N1NF (Non first Normal Form) algebra [ABI 86]. Indeed, the authors define two kinds of operations coming from the N1NF algebra. These operations, called nest and unnest, are implemented in ODBAPI (see section 4.4.2).

In Listing 4.1, we give the example of a join query between *person* and *dblp EntitySets* using ODBAPI.

```
> POST /odbapi/query
> Database−Type: database/VirtualDataStore
> Accept: application/json
> {
>   "select": ["personName", "personCountry", title],
>   "from": ["person", "dblp"],
>   "where": ["personId <3", "personName in author"]
> }

< HTTP/1.1 200 OK
< Content−Type: application/json
< {
<  "data":
<       [
<         {
<           "personName": "Leslie Valiant",
<           "personCountry":"United Kingdom",
<           "title": "Robust logics"
<         },
<         {
<           "personName": "John Hopcroft",
<           "personCountry":"United States"
<           "title": "Manipulation−Resistant
<            Reputations Using Hitting Time"
<         }
<       ]
< }
```

Listing 4.1. *Example of ODBAPI query and answer*

In the body of his/her query, the user expresses the query. He/she specifies in the element *select* the attributes *personName*, *personCountry* and *title* that he/she wants to project as a result to the query. Then, he/she expresses in the element *from* the name of the *person* and *dblp EntitySets* to join. Finally, he/she defines in the element *where* the predicate of the selection, that is, *personId* < 3 and the logical expression of join *personName in author*. The response of this request is the status code `200 OK` and the answer is written in JSON format in Listing 4.1. The approach of Sellami *et al.* [SEL 17] to evaluating and optimizing this kind of query will be described in section 4.6.3.

4.5.4. *Other works*

The generalized data model of Cloudy [KOS 10]: Cloudy is a modular Cloud storage system based on a generic data model. It can be customized to meet application requirements. Cloudy offers to an application several external interfaces (key/value, SQL, XQuery, etc.) while supporting different storage engines (such as BerkeleyDB or in-memory hash-map). Each Cloudy node in the system is an autonomous running process which executes the entire Cloudy stack. Its generic data model is defined as a set of fields that allows us to identify, transfer and query data. It contains (1) a key to uniquely identify data, (2) information about the data structure, (3) the data itself and (4) some metadata. Data are handled using three kinds of operation: read, write and delete. Even though Cloudy's data model is generic, it does not support other types of NoSQL data stores and it does not enable execution of complex queries. In addition, this data model does not take into account the modeling of data stores, where data are stored, and the environment in which they are deployed. This is very important in order to evaluate and optimize query execution across integrated data.

SQL++ data model [ONG 14]: SQL++ is a semi-structured query language which enables interaction with relational and JSON native data stores. Its data model is a super set of JSON format and SQL data model (especially relational tables). SQL++ extends JSON with bags and maps, while it extends SQL with arrays, maps and arbitrary composition of complex values and heterogeneity. Indeed, a SQL++ array/bag may contain heterogeneous elements comprising a mix of tuples, scalars and nested bags/arrays/maps. The SQL++ query

language is backward compatible with SQL in order to make it easy to understand and to encourage developers to use it. It enables us to define two variants of queries. On the one hand, a Select-From-Where (SFW) query takes a bag as input and produces another bag as output. On the other hand, an expression query produces arbitrary values as output, and consequently, it is fully composable with SFW queries. Queries in SQL++ are evaluated using an environment which includes a set of configurations and bindings. Although SQL++ is very interesting, it does not support all kinds of NoSQL data stores (namely key/value, column and graph data stores).

Heterogeneous data resource collaborative management model [SUN 13]: Tao Sun *et al.* propose a heterogeneous data resource collaborative management model in Cloud environment. It enables the update and balancing of workloads, security management and some monitoring methods. This model includes four main components: (1) the physical storage layer that stores the heterogeneous data in relational, NoSQL and all file-based data stores; (2) the data resource network layer that allows the management and the use of Cloud storage services by abstracting all of the physical nodes (i.e. data stores) into logical nodes and interconnecting these nodes – it is noteworthy that each logical node may store various types of data; (3) the data conversion layer that unifies the data resource formats by transforming them into GroupDB database center's format; (4) the GroupDB data management layer that enables data management (e.g. data integration, data fusion, etc.). Nevertheless, authors remain superficial in their proposed model and abstract their data in a high-level manner. Indeed, they ignore the representation of data collections (e.g. tables, document collections, etc.) and data (tuples, documents, attributes, etc.). These omissions prevent users from expressing complex declarative queries.

xR2RML: a non-relational database to RDF mapping language [MIC 15]: xR2RML is a language enabling the description of mappings of various types of data stores to RDF. It takes into account relational, XML, object-oriented and some NoSQL data stores. It is an extension of R2RML that belongs to the W3C Recommendation and it is a language for expressing mappings from relational data stores to RDF datasets. Once the RDF dataset is created, users can seamlessly query it using a declarative query language. Nevertheless, using this approach, users cannot express complex queries, especially when it

comes to interacting with NoSQL data stores. In addition, converting result sets (especially SQL result sets) into RDF is likely to be quite inefficient.

Data integration of NoSQL stores using mappings [CUR 11, CUR 13]: Curé *et al.* [CUR 11] propose a data integration system to enable querying NoSQL and relational data stores. Their approach considers the following assumptions: (1) the global schema is a standard relational data model since the majority of end-users are familiar with this model and its query language, and (2) the sources can be of type relational or NoSQL. To support the execution queries over NoSQL data stores, they define a mapping language to map attributes of the data stores to the global schema and a Bridge Query Language (BQL) to rewrite queries. BQL is an intermediate query language filling the gap between SQL and the NoSQL constructions. It is based on filters and iterations. BQL queries are transformed into native NoSQL programs. In a second step, Curé *et al.* [CUR 13] extend their solution using an Ontology-Based Data Access (OBDA) approach. They replace the relational data model with an ontology-based model and BQL with SPARQL[2]. This allows us to use the reasoning capabilities of ontology. Although their proposal is promising, there are some functionalities lacking (i.e. no query optimization at the global level, no complex query execution, etc.). The approach is interesting but is limited to key/value and document data stores. Administrators also need to define the mappings.

Data models in Ontology-Based Data Access (OBDA) approaches [GIE 15, BAG 14]: The key ingredient of OBDA approaches is the use of ontologies to represent data and the data stores. In the same approach, it is possible to define one or multiple ontologies to integrate data coming from heterogeneous sources. The ontologies in this kind of solution play the role of data models and, in general, they are classified and constructed in a Global-as-View (GAV) manner. To query the data in a ODBA approach, users need to express their queries using the SPARQL query language. As an example of solutions, we can cite Curé *et al.* [CUR 13] as well as Ontop [GIE 15, BAG 14], which is an OBDA-based framework to integrate relational data stores. It exposes relational data stores as virtual RDF graphs by linking the terms in the ontology to the data stores through mappings acquired using the mapping language R2RML. Then the resulting RDF graph is queried using SPARQL. The Ontop

2 http://www.w3.org/TR/rdf-sparql-query/.

framework is composed of four inputs: (1) an ontology to uniquely represent data stores, (2) a set of mappings to link data stores to the ontology, (3) the integrated data stores and (4) the queries written in SPARQL.

The common data model of CloudMdsQL [KOL 16]: CloudMdsQL is an example of an approach without a global schema. Programmers manipulate each data store using the native query language of their data store and a global query language is used to define multi-data store queries. This global language is based on a functional approach allowing programmers to combine the results of (sub-)queries on a single data store, even if these sub-queries are expressed in different languages (native query language of the data stores or Python expressions). The results are represented in a neutral format based on the relational model, allowing the exchange of data between heterogeneous data stores.

The relational data model has been chosen for its simplicity and its expressiveness. Indeed, they define three kinds of table expressions: (1) the native table expressions (expressed via a data store's native query mechanism), (2) the SQL table expressions and (3) the embedded blocks of Python statements which produce relations. Data structured in these tables may be formatted using two kinds of data type: scalar and composite. In addition, data are managed using classical relational operators (i.e. the projection, the restriction, the join, etc.) and two other operators. The first one is defined to support data and metadata transformation and the second one is defined to optimally execute nested queries. Below, we introduce a query expressing a join between two data stores:

```
T1(x int, y int)@DB1 = {select x, y from A}
T2(x int, z string)@DB2 = {* db.B.find( {$lt: {x, 10}, {x:1, z:1, _id:0}} *}
SELECT T1.x, T2.z
FROM T1, T2
WHERE T1.x = T2.x AND T1.y <= 3
```

Listing 4.2. *Example of CloudMdsQL query*

This query uses two table expressions, T1 and T2. T1 is a SQL table expression located on DB1, which is a relational data store, whereas T2 is a native table expression located on DB2, which is a MongoDB data store. The query computes the join of T1 and T2 and another selection on T1. We can notice that, even if T2 is defined by a native MongoDB query, its signature is described in the (relational) common data model of CloudMdsQL.

The main advantage of CloudMdsQL is that the underlying data stores can be queried using their own query language with all its expressiveness. It has the disadvantage of introducing a new query language for programmers (they have to master both the different native query languages and the functional language).

4.5.5. *Comparison*

In this section, we present an overview of the studied solutions for unifying data models, and discuss the features of these works based on the comparison shown in Table 4.3. In this comparison, we consider the following set of criteria. Firstly, we take into account whether the data model has a dynamic aspect or not. We characterize a data model by dynamicity if it represents the data, the data stores where they are stored and the environment where these data stores are deployed. In addition, we highlight whether the studied solutions enable us to unify relational and NoSQL data models. Then, we present the use of a global schema and of a global query language. Finally, we address whether the solutions specifically address cloud environments. In Table 4.3, we show whether each solution satisfies our criteria or not.

Criteria / Studied Solutions	Relational data stores	NoSQL data stores	Dynamicity	Global schema	Global query language	Cloud computing
Sellami et al. [SEL 16a]	+	+	+	Unified data model + correspondence rules	SQL	+
Cloudy data model [KOS 10]	+	−	−	Generic data model	External unique interface	+
SQL++ data model [ONG 14]	+	−	−	JSON++	SQL++	−
Tao Sun et al. [SUN 13]	+	+	+	Heterogeneous data resource management model	Uniform query language	+
xR2RML [MIC 15]	+	+	−	RDF	Declarative	−
Ontop [GIE 15, BAG 14]	+	+	+	RDF	SPARQL	−
Curé [CUR 11, CUR 13]	+	+	+	Relational/RDF	SQL	−
CloudMdsQL [KOL 16]	+	+	+	Relational	Functional	+

Table 4.3. *Comparison of the related works unifying data models*

From this table, we can observe that most of the works do define a global and integrated schema. This schema is defined using the relational data model, RDF structures or some specific models. A global query language is also

integrated, to query the global schema (SQL like or SPARQL). A few works do not impose a global schema: they replace it by the definition of the set of accessible collections of data. In one work [SEL 16a], a set of correspondence rules can be added. CloudMdsQL proposes a functional federated query language allowing the composition of results of sub-queries expressed using native query languages. This means we can preserve the expressiveness of data stores and express multi-data store queries. This comes at the price of the use of a new language. Finally, the majority of works support at least one type of NoSQL data store (i.e. key/value, document, column or graph).

4.6. Query processing and optimization

4.6.1. *Introduction*

In this section, we analyze how global (or multi-data stores) queries are processed. They are done partly by the data stores and partly by an external component (global query engine). This query engine can be centralized or distributed and needs statistics from the different data stores to perform efficient optimizations. Of course, the query processing only somewhat depends on the underlying global query language. We distinguish two categories: integrated query languages (based on an integrated global schema) and federated query languages (without a global schema). The first part of this section describes federated query language works [KOL 16, DUG 15] and the second part describes integrated query language works [SEL 17, BAJ 11, LIA 16, VIL 13, ZHU 11, DEW 13, LEF 14].

4.6.2. *Federated query language approaches*

CloudMdsQL, Querying heterogeneous Cloud data stores with a functional query language: Valduriez *et al.* [KOL 16] propose CloudMdsQL, which is an example of a federated system. It is based on a functional language approach allowing users to combine the results of (sub-)queries on a single data store, even if these sub-queries are expressed in different languages (the native query language of the data stores or Python expressions). The main advantage of this system is that the underlying data stores can be queried using their own query language with all its expressiveness. Even if sub-queries use different query languages, their combination needs a common data model and an associated query language (for more details, see section 4.5.4).

The CloudMdsQL query engine is fully distributed on a set of nodes storing the different data stores in a peer-to-peer style. Each query engine node is composed of two parts, a master part and a worker part. Each master or worker is able to exchange data or query sub-plans between them. A master is able to analyze and optimize CloudMdsQL queries and to control sub-plans execution by workers. A worker locally optimizes a sub-plan, executes it and sends the results either to another worker or to the master.

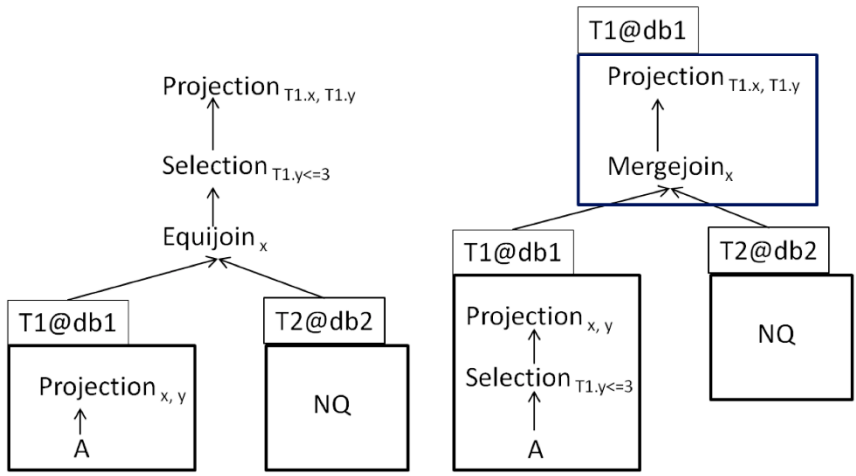
A query is sent to a specific master using some load balancing strategies. This master is in charge of producing an optimized query execution plan serialized as a JSON-based object allowing the exchange between worker nodes. The query is first decomposed in sub-trees, each of them being associated with (and executed by) a certain data store. These sub-trees are connected by a common query plan that will be handled by the query engine. This initial decomposition may be modified by the optimization step, which will possibly push some operations from the common query plan to the sub-trees or pull some operations from the sub-trees to the common query plan (if a data store does not support these operations). The optimization is done using a simple cost model and information stored in a catalog (replicated at all master nodes). The optimizer needs accurate statistics that are not easy to obtain for heterogeneous and autonomous data stores. Statistics can be gathered either running periodically probing queries or using cost functions. In CloudMdsQL, these cost functions can be defined by database administrators or even by users. A simple exhaustive search strategy is used to explore all possible plan permutations. The objectives of this optimization are (1) to minimize local execution time in the data stores (pushing down select operators, selecting the optimal join algorithm between hash, merge, nested loop or bind join) and (2) to minimize communication cost by reducing data transfers between workers. This query execution plan (QEP) is transmitted to a worker in charge to control its execution. This execution can be done by a single worker but, in most cases, by several workers encapsulating different data stores. Each worker can be considered as a simple runtime database processor on top of a data store and is composed of three modules:

– The *query execution controller*: it controls the execution of a query plan interacting with the local operator engine or with other workers if part of the query needs to be handled by other nodes.

– The *operator engine*: it executes the operators on data retrieved from the local data store (through the wrapper), from another worker or from the table storage. It may store intermediate results in the table storage.

– The *wrapper*: it translates calls from generic to specific in the local data store API. It writes results in table storage or delivers it to the operator.

We consider our example of a simple CloudMdsQL query (see Listing 4.2) using two tables T1 and T2 and suppose that T1 is stored on node *db1* and T2 is stored on node *db2*. At a first step, this query is decomposed as a non-optimized query plan (see Figure 4.5(a)), where T1 is computed on *db1*, T2 as a native query on MongoDB is computed on *db2* and the rest of the query (equijoin on *x*, selection on *y* and projection on *x* and *y*) is done outside. After optimization (see Figure 4.5(b)), the selection on *y* is pushed down on node *db1*, and the equijoin is replaced by a merge join. This merge join is computed on node *db1* together with the final projection. *db1* has been chosen because it is able to compute joins, which thus reduces the communicating cost (there is no need to transfer nor transform T1 data).



(a) Initial query plan for a CloudMdsQL query

(b) Optimal query plan for a CloudMdsQL query

Figure 4.5. *Examples of query plans for a CloudMdsQL query [KOL 16]*

BigDAWG, a new view of federated databases [DUG 15]: The authors introduce a new class of multi-store systems called polystores. A polystore system pursues three goals: (1) supporting location transparency for the storage of objects, (2) semantic completeness that does not lose any capabilities natively provided by an underlying data store and (3) enabling

users to access objects stored in a single data store using different interfaces. In order to support these goals, [DUG 15] proposes the notion of an island. An island of information is a collection of data stores accessed with a single query language. More precisely, an island defines a data model, a query language and a set of data stores, each of them able to translate a query expressed in the island's query language into its native one. It is possible to express a query on multiple islands, if the target objects are stored in different islands (that is, in data stores present in different islands). In this case, users have to express explicitly how the data should be cast from one island data model to another. [ELM 15] demonstrates an implementation of this vision with two cross-system islands Myria and D4M. Myria is based on relational algebra extended with iteration and it is used on top of SciDB (a DBMS for scientific data) and Postgres. D4M is based on associative arrays, which allows us to model matrices, graphs and spreadsheets. It is used on top of SciDB, Accumulo (a key-value store) and Postgres. Polystore systems are clearly a nice and ambitious vision. Current implementations are only partial ones and it lacks more declarative query languages for cross system islands.

4.6.3. *Integrated query language approaches*

Virtual data stores to integrate relational and NoSQL data stores: Sellami *et al.* [SEL 17] propose a mediator/wrapper approach based on an integrated schema. A common data model and an associated query algebra have been defined to express and process declarative multi data stores queries. These queries are sent to a mediator, which decomposes them into sub-queries that are sent to the different data stores using ODBAPI (see section 4.4.2). Wrappers encapsulate the different data stores, taking as input ODBAPI queries and transforming them into native queries, which are then locally optimized and processed. The wrappers send the results back to another node (either the mediator or another wrapper in order to reduce data movement and transformations). Optimization is done by the mediator in two steps. In the first step, selections and projections are pushed down to the local data stores to reduce the size of the data to exchange. In the second step, an optimal distributed plan is constructed by a dynamic programming method trying to minimize not only I/O and CPU costs, but also data shipping and transformation. For that, the composition of the intermediate results returned by data stores can be done not only by the mediator but also by other nodes if

they have the capability. Its main advantage is simplicity, since programmers can use just one simple query language without manipulating explicit transformations or different native query languages. However, expressivity is limited to those of the ODBAPI query algebra, and specific constructs of NoSQL graph data stores (e.g. path queries) are not supported.

Hadapt, an efficient processing of data warehousing queries in a split execution environment [BAJ 11]: Many works have focused on the integration of Hadoop-based stores with relational ones. In Hadapt, a relational DBMS is associated with each node in a Hadoop cluster. Map/reduce is the basis of the evaluation of queries, but optimization is done by having the corresponding data partitions in each store co-located on the same node. In fact, map/reduce jobs cannot exploit data placement because they manipulate blocks of data without any knowledge about their content. Hadapt constructs and exploits partitioning information to optimize queries. The problem of this approach is that partitioning is predefined and must be consistent on both Hadoop and relational stores.

Querying relational and NoSQL databases in SQL [LIA 16, VIL 13, ZHU 11]: A data adapter [LIA 16] is designed to efficiently transform NoSQL data into relations and to support SQL querying. Three modes are proposed: blocking transformation, blocking dump and direct access. These modes differ on the synchronization guarantees supported between the NoSQL data and the relational ones that may produce inconsistencies due to the write operations on NoSQL data processed during the transformation process. The main problem with this approach is the high cost of data transformation. [VIL 13] extends a relational DBMS by implementing new scan operators able to efficiently process basic key/value operations of NoSQL data stores. BigIntegrator [ZHU 11] is a mediation-based system enabling the integration of relational and Cloud-based (i.e. Bigtable) data stores. In this system, authors propose using SQL as a query language in order to join data stored in Bigtable and relational data stores, and they define a RDBMS wrapper and a Bigtable wrapper for each kind of integrated data store. In addition, they propose the BigIntegrator query processor layer that plays the role of the mediator. This layer enables query re-writing into trees and Datalog queries, query optimization and algebra operation-based execution plan generation. The key ingredients of this layer are (1) the absorber manager that takes the Datalog query and, for each source predicate referenced in the query, calls the corresponding absorber of its wrapper, and (2) the finalizer manager that takes

the algebra expression and, for each access filter operator referenced in the algebra expression, calls the corresponding finalizer of its wrapper. Despite its importance, the BigIntegrator system has some limitations. Indeed, it suffers from a lack of genericity since it supports just one particular NoSQL data store (i.e. Bigtable). Besides, if a user wants to integrate a new data store, he/she has to define a well-tailored wrapper to this data store, an absorber manager and a finalizer manager.

All these works [LIA 16, VIL 13, ZHU 11] are limited to a single data store and do not support distributed query processing on a set of heterogeneous data stores.

Polybase, split query processing over Hadoop clusters using SQL standards [DEW 13]: The authors propose an extension of SQL server Parallel Data Warehouse (PDW) that can query data stored either in PDW or in Hadoop file systems (HDFS). Polybase exposes HDFS data using an external table mechanism. A key component of the architecture is the HDFS bridge, which is deployed on each node of the PDW cluster and can read/write data from HDFS to SQL server supporting parallelism and repartitioning. Polybase uses a cost-based optimization algorithm allowing to choose the best way to evaluate a hybrid query plan (i.e. a query plan involving data both from PDW and HDFS). Considering various parameters estimated with statistics extracted from HDFS files, Polybase can either push HDFS data into PDW and then evaluate the SQL query or execute part of the SQL query as map/reduce jobs (in this case, we avoid the cost of moving and transforming data). Compared to Hadapt, Polybase supports a larger class of optimization but supposes the knowledge of accurate statistics on external HDFS files.

MISO, MultIStore Online tuning [LEF 14]: MISO addresses the physical design of multi-store systems composed of Hadoop-based systems and data warehouse-based systems (i.e. parallel DBMS). In order to reduce data movement during query processing, it decides what data to materialize in which store using online tuning. The basic idea is to materialize intermediate results produced by Hadoop jobs or multi-store query processing and to strategically store them to optimize subsequent queries. Materialization comes at a low cost because it uses data already constructed from previous queries. The only costs to consider are the transfer cost and the storage cost. MISO formalizes this physical tuning as an optimization problem (it minimizes a cost function depending on a query workload and a set of materialized results)

solved by a variant of a dynamic programming-based knapsack solution. Experiments not only show the effectiveness of the approach but also illustrate some trade-off issues (frequency of reorganization and cost). It is also not really adapted to ad-hoc queries (the query workload is supposed to be known).

4.6.4. *Comparison*

Table 4.4 analyzes and compares these different approaches using a set of well-defined criteria. The first criterion corresponds to the type of supported architecture (mediation, federation or simple adaptation). The second and third ones correspond to, respectively, the type of the supported data stores (relational, Hadoop or NoSQL) and the number of data stores (just one source per type or multiple sources per type). The query language criterion describes the type of query language (global or federated) and the style of the language (SQL-like, functional, etc.). The last criteria describe the optimization part: the placement corresponds to optimization due to the placement of data and movement corresponds to optimization trying to minimize data movement.

Criteria / Studied Solutions	Architecture	Type of data stores	Multi-data stores	Query language	Placement	Movement
CloudMdsQL [KOL 16]	Federation	Relational NoSQL	+	• Federated • Functional	−	+
BigDAWG [DUG 15]	Federation	Relational NoSQL (possibly all types)	+	• Federated • D4M • Myria	+	+
Sellami et al. [SEL 17]	Mediation	Relational NoSQL	+	• Global • SQL-like	−	+
Hadapt [BAJ 11]	−	Relational Hadoop	−	• − • SQL	static	−
Data-adapter [LIA 16]	Adaptation	Relational NoSQL	−	• − • SQL	−	−
Polybase [DEW 13]	Adaptation	Relational Hadoop	−	• − • SQL	+	−
MISO [LEF 14]	−	Relational Hadoop	−	• − • SQL	−	Table materialization

Table 4.4. *Comparison of approaches ensuring query processing and optimization*

From this table, we can observe that works which fully support multi-data stores use mediation or federation as the architecture. Mediation-based systems offer a unified (relational) view of the different data stores and SQL as a global query language, while federated systems have no unified view and support a dedicated federated query language. The other systems use a simple adaptation architecture, that is, they transform a non-relational data source into a relational format that can query both relational and non-relational sources with SQL. Hadapt tries to couple relational DBMS and Hadoop, increasing the efficiency of Hadoop by exploiting information coming from the relational data source. Optimization is done by most of the systems but is very specific to each approach.

4.7. Summary and open issues

In this section, we propose to conduct a global summary of the most relevant studied solutions. We will exclusively focus on works supporting at least two requirements of Big Data integration requirements in cloud environments, that is, R_1 – selection, discovery and deployment of data stores, R_2 – uniform access to any data store, R_3 – integrated view on data stores and R_4 – global query processing and optimization. To do so, we present a global summary in section 4.7.1 and introduce a set of open issues in section 4.7.2.

4.7.1. *Summary*

In this section, we provide a global summary and comparison between all the works studied throughout this chapter (see Table 4.5). To do so, we define three groups of criteria. The first one represents the requirements that we identified in section 4.2 for Big Data integration in cloud environments (R_1, R_2, R_3 and R_4). The second one enables us to check if the studied works sufficiently support Big Data and Cloud Computing features. The third one illustrates the main dimensions of the proposed solutions: which type of data integration architecture (federation, mediation or adaptation), how it supports the heterogeneity of data structures, how it supports the heterogeneity of query languages and, finally, which criteria are used for query optimization. It is noteworthy that we have not discussed works that only focus on a single requirement.

Criteria / Studied Solutions	R_1	R_2	R_3	R_4	Cloud computing	Big Data	Data integration architecture	Data heterogeneity	Query languages heterogeneity	Query optimization
Curé *et al.* [CUR 13]	−	+	+	+	−	−	Mediation	Relational NoSQL	• Global • SPARQL	−
Ontop [GIE 15, BAG 14]	−	+	+	+	−	−	Mediation	Relational NoSQL	• Global • SPARQL	−
Polybase [DEW 13]	−	+	+	+	−	+	Adaptation	Relational Hadoop	• Global • SQL	Cost based
BigDAWG [DUG 15]	−	+	+	+	−	+	Mediation	Relational NoSQL	• Federated • D4M, Myria	Cost based
Sellami *et al.* [SEL 16a]	+	+	+	+	+	+	Mediation	Relational NoSQL	• Global • SQL-like	Cost based
CloudMdsQL [KOL 16]	−	+	−	+	+	+	Federation	Relational NoSQL	• Federated • Functional	Cost based

Table 4.5. *Global summary*

Based on Table 4.5, we can see that there is no work that fulfills all the criteria. Indeed, the requirement R_1 is only partly supported by Sellami *et al.* [SEL 16a]. If Big Data is considered in many works, there are just a few works that explicitly consider the Cloud Computing context (i.e. CloudMdsQL and Sellami *et al.*). Considering data heterogeneity, some works use the relational model as a pivot model: CloudMdsQL uses the relational model to store intermediate results, and Polybase uses the relational model to structure data stored in Hadoop. Sellami *et al.* use a model inspired from semi-structured models. Finally, some works like Curé *et al.* and Ontop use ontologies to structure and remove ambiguities over heterogeneous data. The proposed solutions differ in the way they offer a uniform view on top of heterogeneous data stores and the way they define some kinds of global query languages. Many works use an integrated schema, unifying the different data stores, but, considering the diversity of the data stores, it is quite difficult – even impossible – to be generic. In most cases, the proposed approaches do not consider all types of data stores (e.g. graph data stores are in general not supported). CloudMdsQL adopts another approach, keeping the different data stores in their native model without integration. Sellami *et al.* use

an intermediate solution describing the different data stores with the same model but without integration (though there are correspondence rules to allow inter-data store relationships). The global query language depends mainly on the choice made for data integration. Systems supporting an integrating schema propose an associated query language (e.g. SQL-like, or SPARQL for ontology-based approach). CloudMdsQL proposes a functional language composing sub-queries expressed using the native language of the different data stores. BigDAWG goes a step further in supporting different global models and associated query languages (for the moment, an array-based view with D4M and an extended relational view with Myria). Optimization is done by many systems, but it is quite difficult without accurate statistics, which are difficult to acquire with heterogeneous and autonomous systems. Some works try to address Big Data issues in minimizing data movement between the different nodes. The concept of polystore systems introduced by BigDAWG is very innovative, but difficult to implement. Optimization is difficult to handle and is very specific to each view. Moreover, no completely declarative query language has been defined so far: in Myria or D4M data stores need to be explicitly cast during query execution.

4.7.2. *Open issues*

Big Data and Cloud Computing have clearly complicated the classical approaches of data integration: the large number of data and the high level of heterogeneity of data stores introduce a higher level of complexity. There are thus two opposing goals. The first one is to have highly integrated data stores, which can be accessed using a declarative query language, whereas the second one is to preserve the specificity of the different data stores and the use of their native query languages. Many works focus on the first goal but do not preserve the expressivity of the native query languages. Moreover, the high number of data stores also impacts the integration process which is very difficult to achieve. Few works like CloudMdsQL focus on the second goal, but they introduce a new global query language combining sub-queries expressed with native query languages and this mixture is not so declarative nor easy to use for programmers. The definition of such a global declarative query language is clearly an open issue.

Optimization is the ‘holy grail’ of database management and, in the context of Big Data integration, it is clearly a major challenge. First, it is difficult

to get accurate statistics from autonomous data stores. Second, new criteria have to be considered, such as minimization of data movement and data transformations. Finally, each data store may be implemented on a cluster and not a single server, which improves the complexity of the optimization (e.g the partitioning of data is much more complex with data possibly distributed on different nodes).

Updates are in general not considered in data integration systems even if they are quite important. It is the same thing for Big Data integration. The different data stores may use different transaction and consistency models. Relational data stores respect the ACID properties in order to try to exhibit strong consistency and availability of their data. However, for NoSQL data stores, the problem is even more complex because of the lack of a common and clear concept of transaction. In fact, some NoSQL data stores just support transactions limited to one record and, in general, they suppose that updates of a transaction are limited to one server node to avoid using synchronization protocols such as a two-phase commit. NoSQL data stores are generally based on a simple model of consistency (compared to ACID properties) called BASE (Basically Available, Soft-State and Eventually Consistent). These properties come from the CAP theorem stating that a distributed system cannot ensure consistency, availability and partition tolerance at the same time [BRE 00, BRE 12, ABA 12]. The problem of eventually consistent models is that programmers need to explicitly program the behavior of the application if part of the work of the transaction has a failure. [BER 13] addresses this problem of client-centric consistency on top of eventually consistent distributed data stores (e.g. Amazon S3). It involves a middleware service running on the same server as the application and providing the same behavior as a causally consistent data source even in the presence of failures or concurrent updates. This service uses vector clocks and client-side caching to ensure client-centric properties. The main interest of this proposal is that it comes at a low cost and it is transparent for programmers. Defining consistency models for distributed NoSQL systems is also an open issue.

As part of Big Data integration, it would certainly be worthwhile to take into account the management and the supervision of data quality. Classical data quality models and tools are intended to work with relational data stores. In fact, they are based on (1) static data, (2) a very small number of rules and (3) specific metrics to evaluate and measure data quality. However, when it comes to dealing with data quality in a Big Data integration environment,

new challenges and hurdles appear [SAH 14]. Indeed, we face voluminous and high-velocity data. Hence, the number of rules should increase and new metrics should be (re)defined. It is quite impossible for administrators to extract rules manually, but rules can be learned from the data itself. If rules are violated by the data, there are two main approaches to deal with inconsistencies: inconsistent data are removed [KHA 15] or queries are answered in an approximate manner without repairing the database [RAZ 11].

4.8. Conclusion

In this chapter, we defined four Big Data integration requirements in cloud environments: R_1 (selection, discovery and deployment of data stores), R_2 (uniform access to any data stores), R_3 (integrated view on data stores) and R_4 (global query processing and optimization). Even if data integration has been widely studied in the past, Big Data and Cloud Computing introduce new problems that question classical approaches. The use of an integrated global schema and an associated query language, for instance, is deeply questionable and some works are based on a minimum global schema. We have analyzed the literature regarding each requirement and presented a summary of this analysis. Our summary includes a comparison between studied works on the basis of the criteria that we addressed in our research objectives. We concluded this article by introducing a global summary concering the most relevant studied solutions and defining some open issues covering different requirements and other open problems such as updates and data quality.

4.9. Bibliography

[ABA 12] ABADI D., "Consistency tradeoffs in modern distributed database system design: CAP is only part of the story", *IEEE Computer*, vol. 45, no. 2, pp. 37–42, 2012.

[ABI 86] ABITEBOUL S., BIDOIT N., "Non first normal form relations: an algebra allowing data restructuring", *Journal of Computer and System Sciences*, vol. 33, no. 3, pp. 361–393, 1986.

[ATZ 12] ATZENI P., BUGIOTTI F., ROSSI L., "Uniform access to non-relational database systems: the SOS platform", *Advanced Information Systems Engineering – 24th International Conference, CAiSE 2012*, pp. 160–174, Gdansk, 25–29 June 2012.

[BAG 14] BAGOSI T., CALVANESE D., HARDI J. *et al.*, "The ontop framework for ontology based data access", *The Semantic Web and Web Science – 8th Chinese Conference, CSWS 2014*, Revised selected papers, pp. 67–77, Wuhan, 8–12 August 2014.

[BAJ 11] BAJDA-PAWLIKOWSKI K., ABADI D.J., SILBERSCHATZ A. *et al.*, "Efficient processing of data warehousing queries in a split execution environment", *Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD 2011*, pp. 1165–1176, Athens, 12–16 June 2011.

[BAN 94] BANCILHON F., FERRAN G., "ODMG-93: The object database standard", *IEEE IEEE Technical Committee on Data Engineering*, vol. 17, no. 4, pp. 3–14, 1994.

[BAU 11] BAUN C., KUNZE M., NIMIS J. *et al.*, *Cloud Computing – Web-Based Dynamic IT Services*, Springer, 2011.

[BER 13] BERMBACH D., KUHLENKAMP J., DERRE B. *et al.*, "A middleware guaranteeing client-centric consistency on top of eventually consistent datastores", *2013 IEEE International Conference on Cloud Engineering, IC2E 2013*, pp. 114–123, San Francisco, 25–27 March 2013.

[BRE 00] BREWER E.A., "Towards robust distributed systems (abstract)", *Proceedings of the Nineteenth Annual ACM Symposium on Principles of Distributed Computing*, p. 7, Portland, 16–19 July 2000.

[BRE 12] BREWER E., "CAP twelve years later: How the "rules" have changed", *Computer*, vol. 45, no. 2, pp. 23–29, 2012.

[CAB 13] CABIBBO L., ONDM: an Object-NoSQL Datastore Mapper, http://www.dia.uniroma3.it/cabibbo/pub/ondm-demo-draft.pdf, 2013.

[CAR 95] CAREY M.J., HAAS L.M., SCHWARZ P.M. *et al.*, "Towards heterogeneous multimedia information systems: the garlic approach", *Proceedings of RIDE-DOM '95, Fifth International Workshop on Research Issues in Data Engineering – Distributed Object Management*, pp. 124–131, Taipei, 6–7 March 1995.

[CAR 12] CARLSON M., CHAPMAN M., HENEVELD A. *et al.*, Cloud Application Management for Platforms, http://cloudspecs.org/CAMP/CAMP_v1_0.pdf, August 2012.

[CUR 11] CURÉ O., HECHT R., DUC C.L. *et al.*, "Data integration over NoSQL stores using access path based mappings", *Proceedings of Database and Expert Systems Applications – 22nd International Conference, DEXA 2011*, pp. 481–495, Toulouse, August 29–September 2, 2011.

[CUR 13] CURÉ O., KERDJOUDJ F., FAYE D. *et al.*, "On the potential integration of an ontology-based data access approach in NoSQL stores", *IJDST*, vol. 4, no. 3, pp. 17–30, 2013.

[DEA 04] DEAN J., GHEMAWAT S., "MapReduce: simplified data processing on large clusters", *6th Symposium on Operating System Design and Implementation OSDI 2004*, pp. 137–150, San Francisco, 6–8 December 2004.

[DEW 13] DEWITT D.J., HALVERSON A., NEHME R.V. *et al.*, "Split query processing in polybase", *Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD 2013*, pp. 1255–1266, New York, 22–27 June 2013.

[DOA 12] DOAN A., HALEVY A.Y., IVES Z.G., *Principles of Data Integration*, Morgan Kaufmann, 2012.

[DOU 06] DOU D., LEPENDU P., "Ontology-based integration for relational databases", *Proceedings of the 2006 ACM Symposium on Applied Computing (SAC)*, pp. 461–466, Dijon, 23–27 April 2006.

[DOU 10] DOU D., QIN H., LEPENDU P., "Ontograte: towards automatic integration for relational databases and the semantic web through an ontology-based framework", *International Journal of Semantic Computing*, vol. 4, no. 1, pp. 123–151, 2010.

[DUG 15] DUGGAN J., ELMORE A.J., STONEBRAKER M. *et al.*, "The BigDAWG Polystore System", *SIGMOD Record*, vol. 44, no. 2, pp. 11–16, 2015.

[ELM 15] ELMORE A.J., DUGGAN J., STONEBRAKER M. *et al.*, "A demonstration of the BigDAWG polystore system", *PVLDB*, vol. 8, no. 12, pp. 1908–1911, 2015.

[FIS 03] FISHER M., ELLIS J., BRUCE J.C., *JDBC API Tutorial and Reference*, 3rd edition, Pearson Education, 2003.

[GAR 97] GARCIA-MOLINA H., PAPAKONSTANTINOU Y., QUASS D. *et al.*, "The TSIMMIS approach to mediation: data models and languages", *Journal of Intelligent Information Systems*, vol. 8, no. 2, pp. 117–132, 1997.

[GIE 15] GIESE M., SOYLU A., VEGA-GORGOJO G. *et al.*, "Optique: zooming in on big data", *IEEE Computer*, vol. 48, no. 3, pp. 60–67, 2015.

[HAL 04] HALEVY A.Y., IVES Z.G., MADHAVAN J. *et al.*, "The Piazza peer data management system", *IEEE Transactions on Knowledge and Data Engineering*, vol. 16, no. 7, pp. 787–798, 2004.

[HAS 10] HASELMANN T., THIES G., VOSSEN G., "Looking into a REST-based universal API for database-as-a-service systems", *12th IEEE Conference on Commerce and Enterprise Computing, CEC 2010*, pp. 17–24, Shanghai, 10–12 November 2010.

[HEI 85] HEIMBIGNER D., MCLEOD D., "A federated architecture for information management", *ACM Transactions on Information Systems*, vol. 3, no. 3, pp. 253–278, 1985.

[KHA 15] KHAYYAT Z., ILYAS I.F., JINDAL A. *et al.*, "BigDansing: a system for big data cleansing", *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*, Melbourne, pp. 1215–1230, May 31–June 4 2015.

[KIM 92] KIM W., "On unifying relational and object-oriented database systems", *ECOOP '92, Proceedings of the European Conference on Object-Oriented Programming*, pp. 1–18, Utrecht, June 29–July 3 1992.

[KIM 94] KIM W., "UniSQL/X unified relational and object-oriented database system", *Proceedings of the 1994 ACM SIGMOD International Conference on Management of Data, Minneapolis, Minnesota*, p. 481, 24–27 May 1994.

[KOL 16] KOLEV B., VALDURIEZ P., BONDIOMBOUY C. *et al.*, "CloudMdsQL: querying heterogeneous cloud data stores with a common language", *Distributed and Parallel Databases*, vol. 34, no. 4, pp. 463–503, 2016.

[KOS 10] KOSSMANN D., KRASKA T., LOESING S. *et al.*, "Cloudy: a modular cloud storage system", *PVLDB*, vol. 3, no. 2, pp. 1533–1536, 2010.

[LEF 14] LEFEVRE J., SANKARANARAYANAN J., HACIGÜMÜS H. *et al.*, "MISO: souping up big data query processing with a multistore system", *International Conference on Management of Data, SIGMOD 2014*, pp. 1591–1602, Snowbird, 22–27 June 2014.

[LEN 02] LENZERINI M., "Data integration: a theoretical perspective", *Proceedings of the Twenty-first ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*, pp. 233–246, Madison, 3–5 June 2002.

[LIA 16] LIAO Y.-T., ZHOU J., LU C.-H. *et al.*, "Data adapter for querying and transformation between SQL and NoSQL database", *Future Generation Computer Systems*, Special Issue on Big Data in the Cloud, vol. 65, pp. 111–121, 2016.

[MCH 97] MCHUGH J., ABITEBOUL S., GOLDMAN R. *et al.*, "Lore: a database management system for semistructured data", *SIGMOD Record*, vol. 26, no. 3, pp. 54–66, 1997.

[MIC 15] MICHEL F., DJIMENOU L., FARON-ZUCKER C. *et al.*, "Translation of heterogeneous databases into RDF, and application to the construction of a SKOS taxonomical reference", *Web Information Systems and Technologies – 11th International Conference, WEBIST 2015*, Revised selected papers, pp. 275–296, Lisbon, 20–22 May 2015.

[OLS 08] OLSTON C., REED B., SRIVASTAVA U. *et al.*, "Pig latin: a not-so-foreign language for data processing", *Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD 2008*, pp. 1099–1110, Vancouver, 10–12 June 2008.

[ONG 14] ONG K.W., PAPAKONSTANTINOU Y., VERNOUX R., The SQL++ Semi-structured Data Model and Query Language: A Capabilities Survey of SQL-on-Hadoop, NoSQL and NewSQL Databases, Report, CoRR abs/1405.3631, 2014.

[PAP 95] PAPAKONSTANTINOU Y., GARCIA-MOLINA H., WIDOM J., "Object exchange across heterogeneous information sources", *Proceedings of the Eleventh International Conference on Data Engineering*, pp. 251–260, Taipei, 6–10 March 1995.

[POL 12] POLLACK M., GIERKE O., RISBERG T. *et al.* (eds), *Spring Data*, O'Reilly Media, 2012.

[POO 01] POOLE J.D., "Model-driven architecture: vision, standards and emerging technologies", *Workshop on Metamodeling and Adaptive Object Models, Presentation, ECOOP'01*, Budapest, Hungary, 2001.

[RAZ 11] RAZNIEWSKI S., NUTT W., "Completeness of queries over incomplete databases", *PVLDB*, vol. 4, no. 11, pp. 749–760, 2011.

[RIC 13] RICHARDSON L., RUBY S., AMUNDSEN M., *RESTful Web APIs*, O'Reilly, 2013.

[RUI 11] RUIZ-ALVAREZ A., HUMPHREY M., "An automated approach to cloud storage service selection", *Proceedings of the 2nd International Workshop on Scientific Cloud Computing, ScienceCloud '11*, pp. 39–48, San Jose, 8 June 2011.

[RUI 12] RUIZ-ALVAREZ A., HUMPHREY M., "A model and decision procedure for data storage in cloud computing", *12th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing, CCGrid 2012*, pp. 572–579, Ottawa, 13–16 May 2012.

[SAD 12] SADALAGE P.J., FOWLER M., *NoSQL distilled: A brief guide to the emerging world of polyglot persistence*, Addison-Wesley Professional, 2012.

[SAH 14] SAHA B., SRIVASTAVA D., "Data quality: the other face of big data", *IEEE 30th International Conference on Data Engineering*, pp. 1294–1297, Chicago, March 31–April 4, 2014.

[SEL 14] SELLAMI R., BHIRI S., DEFUDE B., "ODBAPI: a unified REST API for relational and NoSQL data stores", *2014 IEEE International Congress on Big Data*, Anchorage, pp. 653–660, 27 June–2 July 2014.

[SEL 15] SELLAMI R., VEDRINE M., BHIRI S. *et al.*, "Automating resources discovery for multiple data stores cloud applications", *CLOSER 2015 – Proceedings of the 5th International Conference on Cloud Computing and Services Science*, pp. 397–405, Lisbon, 20–22 May 2015.

[SEL 16a] SELLAMI R., Supporting multiple data stores based applications in cloud environments, PhD thesis, University of Paris-Saclay, 2016.

[SEL 16b] SELLAMI R., BHIRI S., DEFUDE B., "Supporting multi data stores applications in cloud environments", *IEEE Transaction Services Computing*, vol. 9, no. 1, pp. 59–71, 2016.

[SEL 17] SELLAMI R., DEFUDE B., "Complex queries optimization and evaluation over relational and NoSQL data stores in cloud environments", *IEEE Transactions on Big Data*, Early access, 2017.

[SHE 90] SHETH A.P., LARSON J.A., "Federated database systems for managing distributed, heterogeneous, and autonomous databases", *ACM Computer Survey*, vol. 22, no. 3, pp. 183–236, 1990.

[SNI 12] SNIA, "Cloud data management interface", Version 1.0.2, http://snia.org/sites/default/files/CDMI %20v1.0.2.pdf, June 2012.

[STO 05] STONEBRAKER M., ÇETINTEMEL U., "'One size fits all': an idea whose time has come and gone (abstract)", *Proceedings of the 21st International Conference on Data Engineering, ICDE 2005*, pp. 2–11, Tokyo, 5–8 April 2005.

[STO 11] STONEBRAKER M., "Stonebraker on data warehouses", *Communication ACM*, vol. 54, no. 5, pp. 10–11, 2011.

[SUN 13] SUN T., WANG X., "Research on heterogeneous data resource management model in cloud environment", *International Journal of Database Theory and Application*, vol. 6, no. 5, pp. 141–152, 2013.

[THU 09] THUSOO A., SARMA J.S., JAIN N. *et al.*, "Hive – a warehousing solution over a map-reduce framework", *PVLDB*, vol. 2, no. 2, pp. 1626–1629, 2009.

[TRU 11] TRUONG H.L., DUSTDAR S., GÖTZE J. *et al.*, "Exchanging data agreements in the DaaS model", *2011 IEEE Asia-Pacific Services Computing Conference, APSCC 2011*, pp. 153–160, Jeju, 12–15 December 2011.

[TRU 12] TRUONG H.L., COMERIO M., PAOLI F.D. *et al.*, "Data contracts for cloud-based data marketplaces", *IJCSE*, vol. 7, no. 4, pp. 280–295, 2012.

[VIL 13] VILAÇA R., CRUZ F., PEREIRA J. *et al.*, "An effective scalable SQL engine for NoSQL databases", *Distributed Applications and Interoperable Systems – 13th IFIP WG 6.1 International Conference, DAIS 2013*, pp. 155–168, Florence, 3–5 June 2013.

[VU 12] VU Q.H., PHAM T.V., TRUONG H.L. *et al.*, "DEMODS: a description model for data-as-a-service", *IEEE 26th International Conference on Advanced Information Networking and Applications, AINA, 2012*, pp. 605–612, Fukuoka, 26–29 March 2012.

[WHI 15] WHITE T., *Hadoop: The Definitive Guide*, 4th edition, O'Reilly, 2015.

[WIT 12] WITTERN E., KUHLENKAMP J., MENZEL M., "Cloud service selection based on variability modeling", *Service-Oriented Computing – 10th International Conference, ICSOC 2012*, pp. 127–141, Shanghai, 12–15 November 2012.

[ZAH 10] ZAHARIA M., CHOWDHURY M., FRANKLIN M.J. *et al.*, "Spark: cluster computing with working sets", *2nd USENIX Workshop on Hot Topics in Cloud Computing, HotCloud'10*, Boston, 22 June 2010

[ZHA 12] ZHANG M., RANJAN R., HALLER A. *et al.*, "An ontology-based system for Cloud infrastructure services' discovery", *8th International Conference on Collaborative Computing: Networking, Applications and Worksharing, CollaborateCom 2012*, pp. 524–530, Pittsburgh, 14–17 October 2012.

[ZHU 11] ZHU M., RISCH T., "Querying combined cloud-based and relational databases", *2011 International Conference on Cloud and Service Computing, CSC 2011*, pp. 330–335, Hong Kong, 12–14 December 2011.

[ZIK 11] ZIKOPOULOS P., EATON C., *Understanding Big Data: Analytics for Enterprise Class Hadoop and Streaming Data*, McGraw-Hill Osborne Media, 2011.

5

Querying RDF Data: a Multigraph-based Approach

5.1. Introduction

Resource description framework (RDF) is a standard for the conceptual description of knowledge. The RDF data are cherished and exploited by various domains such as life sciences, Semantic Web and social networks. Furthermore, its integration at Web scale compels RDF management engines to deal with complex queries in terms of both size and structure. Popular examples are provided by Google, which exploits the so-called *knowledge graph* to enhance its search results with semantic information gathered from a wide variety of sources, or by Facebook, which implements the so-called *entity graph* to fuel its search engine and provide further information extracted, for instance, by Wikipedia. Another example is provided by recent question–answering systems [CAB 12, ZOU 14a] that automatically translate natural language questions in SPARQL queries and successively retrieve answers by considering the available information in the different linked open data sources. In all these examples, complex queries (in terms of size and structure) are generated to ensure the retrieval of all the required information. Since the use of large knowledge bases that are commonly stored as RDF triplets is becoming a common way to ameliorate a wide range of applications,

---

Chapter written by Vijay INGALALLI, Dino IENCO and Pascal PONCELET.

NoSQL Data Models: Trends and Challenges, First Edition. Edited by Olivier Pivert.

efficient querying of RDF data sources using SPARQL[1] (query language conceived to query RDF data) is becoming crucial for modern information retrieval systems.

With the ever-increasing advantage of representing real-world data of various domains in RDF format, the following two vital challenges have been faced by the RDF data management community: (1) the automatically generated queries cannot be bounded in their structural complexity and size (e.g. the DBPEDIA SPARQL Benchmark [MOR 11] contains some queries having more than 50 triplets [ALU 14a]); (2) the queries generated by retrieval systems (or by any other applications) need to be efficiently answered in a reasonable amount of time. Modern RDF data management systems, such as *x-RDF-3X* [NEU 10] and *Virtuoso* [ERL 12], are designed to address the scalability of SPARQL queries, but they still have problems answering large and structurally complex SPARQL queries [ALU 14b].

In order to address these challenges, in this chapter, we discuss a graph-based RDF querying engine, AMBER [ING 16] (Attributed Multigraph Based Engine for RDF querying), which involves two steps: an offline step, where RDF data are transformed into multigraph and are indexed, and an online step, where an efficient approach to answer a SPARQL query is proposed. First, RDF data are represented as a multigraph, where subjects/objects constitute vertices and multiple edges (predicates) can appear between the same pair of vertices. Then, new indexing structures are conceived to efficiently access RDF multigraph information. Finally, by representing the SPARQL queries multigraphs too, the query answering task can be reduced to the problem of subgraph homomorphism. To deal with this problem, AMBER uses an efficient approach that exploits the structural properties of the multigraph query as well as the indices previously built on the multigraph structure. Experimental evaluation over popular RDF benchmarks show the quality in terms of time performances and robustness of our proposal. In this chapter, we focus only on the `SELECT/WHERE` clause of the SPARQL language[2], which constitutes the most important operation of any RDF query engine.

1 http://www.w3.org/TR/sparql11-overview/.

2 http://www.w3.org/TR/sparql11-overview/.

5.2. Related work

In order to efficiently answer SPARQL queries, many stores and APIs inspired by the relational model were proposed [ERL 12, BRO 02, NEU 10, CAR 04]. x-RDF-3X [NEU 10], inspired by modern RDBMS, representing RDF triples as a large three-attribute table. The RDF query processing is boosted using an exhaustive indexing schema coupled with statistics over the data. Also, Virtuoso [ERL 12] strongly exploits the RDBMS mechanism in order to answer SPARQL queries. Virtuoso is a column-store based system that uses sorted multi-column column-wise compressed projections. Furthermore, these systems build table indexing using standard B-trees. Jena [CAR 04] supplies API for manipulating RDF graphs. Jena exploits multiple-property tables that permit multiple views of graphs and vertices, which can be used simultaneously.

The database community has recently started to investigate RDF stores based on graph data management techniques [DAS 14, ZOU 14b, KIM 15]. The work in [DAS 14] addresses the problem of supporting property graphs as RDF, since the majority of graph databases are based on the property graph model. The authors introduce a property graph to the RDF transformation scheme and propose three models to address the challenge of representing the key/value properties of property graph edges in RDF. gStore [ZOU 14b] applies graph pattern-matching techniques using the filter-and-refinement strategy to answer SPARQL queries. It employs an indexing schema, named VS*-tree, to concisely represent the RDF graph. Once the index is built, it is used to find promising subgraphs that match the query. Finally, exact subgraphs are enumerated in the refinement step. Turbo_Hom++ [KIM 15] is an adaptation of a state-of-the-art subgraph isomorphism algorithm (Turbo$_{ISO}$ [HAN 13]) to the problem of SPARQL queries. Exploiting the standard graph isomorphism problem, the authors relax the injectivity constraint to handle the graph homomorphism, which is the RDF pattern-matching semantics. Unlike the proposed AMbER, Turbo_Hom++ does not index the RDF graph, while gStore concisely represents RDF data through a VS*-tree.

5.3. Background and preliminaries

In this section, we provide basic definitions on the interplay between RDF and its multigraph representation. Later, we explain how the task of answering SPARQL queries can be reduced to a multigraph homomorphism problem.

5.3.1. *RDF data*

According to the W3C standards[3], RDF data are represented as a set of triples $<S, P, O>$, as shown in Figure 5.1(a), where each triple $<s, p, o>$ consists of the following three components: a *subject*, a *predicate* and an *object*. Further, each component of the RDF triple can be either of the two forms: an *IRI* (Internationalized Resource Identifier) or a literal. For brevity, an *IRI* is usually written with a prefix (e.g. <http://dbpedia.org/resource/isPartOf> is written as "x:isPartOf"), whereas a literal is always written with double quotes (e.g. "90000"). While a subject s and a predicate p are always an *IRI*, an object o can be either an *IRI* or a literal.

RDF data can also be represented as a directed graph where, given a triple $<s, p, o>$, the subject s and the object o can be treated as vertices and the predicate p forms a directed edge from s to o, as depicted in Figure 5.1(b). Furthermore, to underline the difference between an *IRI* and a literal, we use standard rectangles and arcs for the former, while we use beveled corners and edges (no arrows) for the latter.

5.3.1.1. *Data multigraph representation*

Motivated by the graph representation of RDF data (Figure 5.1(b)), we take it a step further by transforming it into a data multigraph G, as shown in Figure 5.1(c).

Let us consider an RDF triple $<s, p, o>$ from the RDF tripleset $<S, P, O>$. To transform the RDF tripleset into data multigraph G, we set the following four protocols: we always treat the subject s as a vertex; a predicate p is always treated as an edge; we treat the object o as a vertex only if it is an *IRI* (e.g. vertex v_2 corresponds to object "x:London") and when the object is a literal, we combine the object o and the corresponding predicate p to form a tuple $<p, o>$ and assign it as a vertex attribute to the subject s (e.g. <"y:hasCapacityOf", "90000"> is assigned to vertex v_4). Every vertex is assigned a null value {-} in the vertex attribute set. However, to realize this in the realm of graph management techniques, we maintain three different dictionaries, whose elements are a pair of "key" and "value", and a mapping function that links them. The three dictionaries depicted in Table 5.1 are a

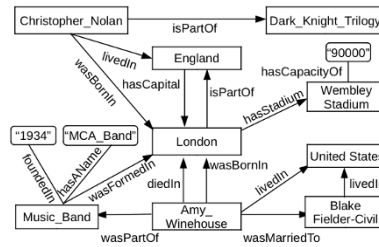
3 http://www.w3.org/TR/2014/REC-rdf11-concepts-20140225/.

vertex dictionary (Table 5.1(a)), an edge-type dictionary (Table 5.1(b)) and a vertex attribute dictionary (Table 5.1(c)). In all the three dictionaries, an RDF entity represented by a "key" is mapped to a corresponding "value", which can be a vertex/edge/attribute identifier. Thus, by using the mapping functions $\mathcal{M}_v$, $\mathcal{M}_e$ and $\mathcal{M}_a$ for vertex, edge-type and vertex attribute mapping, respectively, we obtain a directed, vertex attributed data multigraph G (Figure 5.1(c)), which is formally defined as follows.

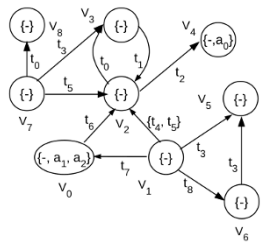
Prefixes: x= http://dbpedia.org/resource/ ; y=http://dbpedia.org/ontology/

Subject	*Predicate*	*Object*
x:London	y:isPartOf	x:England
x:England	y:hasCapital	x:London
x:Christophar_Nolan	y:wasBornIn	x:London
x:Christophar_Nolan	y:LivedIn	x:England
x:Christophar_Nolan	y:isPartOf	x:Dark_Knight_Trilogy
x:London	y:hasStadium	x:WembleyStadium
x:WembleyStadium	y:hasCapacityOf	"90000"
x:Amy_Winehouse	y:wasBornIn	x:London
x:Amy_Winehouse	y:diedIn	x:London
x:Amy_Winehouse	y:wasPartOf	x:Music_Band
x:Music_Band	y:hasName	"MCA_Band"
x:Music_Band	y:FoundedIn	"1994"
x:Music_Band	y:wasFormedIn	X:London
x:Amy_Winehouse	y:livedIn	x:United States
x:Amy_Winehouse	y:wasMarriedTo	x:Blake Fielder-Civil
x:Blake Fielder-Civil	y:livedIn	x:United States

(a) RDF tripleset



(b) Graph representation of RDF data



(c) Equivalent multigraph G

Figure 5.1. *a) RDF data in n-triple format; b) graph representation and c) attributed multigraph G*

DEFINITION 5.1 (Directed, Vertex Attributed Multigraph).– *A directed, vertex attributed multigraph G is defined as a 4-tuple (V, E, L_V, L_E), where V is a set of vertices, $E \subseteq V \times V$ is a set of directed edges with $(v, v') \neq (v', v)$, L_V is a labeling function that assigns a subset of vertex attributes A to the set of*

vertices V and L_E is a labeling function that assigns a subset of edge types T to the edge set E.

To summarize, an RDF tripleset is transformed into a data multigraph G, whose elements are obtained by using the mapping functions as already discussed. Thus, the set of vertices $V = \{v_0, \ldots, v_m\}$ is the set of mapped subject/object *IRI*, and the labeling function L_V assigns a set of vertex attributes $A = \{\text{-}, a_0, \ldots, a_n\}$ (mapped tuple of predicate and object literal) to the vertex set V. The set of directed edges E is a set of pair of vertices (v, v') that are linked by a predicate, and the labeling function L_E assigns the set of edge types $T = \{t_0, \ldots, t_p\}$ (mapped predicates) to the set of directed edges E. The edge set E maintains the topological structure of the RDF data. Furthermore, the mapping of object literals and the corresponding predicates as a set of vertex attributes results in a compact representation of the multigraph. As depicted in Figure 5.1(a), all the object literals and the corresponding predicates are reduced to a set of vertex attributes. For example, the pair <y:hasCapacityOf, "90000"> is mapped to the vertex attribute a_0; similarly, <y:wasFoundedIn, "1994"> and <y:hasName, "MCA_Band"> are mapped to attributes a_1 and a_2, respectively.

5.3.2. *SPARQL query*

A SPARQL query usually contains a set of triple patterns, similarly to RDF triples, except that any of the subject, predicate and object may be a variable, whose bindings are to be found in the RDF data[4]. In the current work, we address the SPARQL queries with a "SELECT/WHERE" option, where the predicate is always instantiated as an *IRI* (Figure 5.2(a)). The `SELECT` clause identifies the variables to appear in the query results, while the `WHERE` clause provides triple patterns to match against the RDF data.

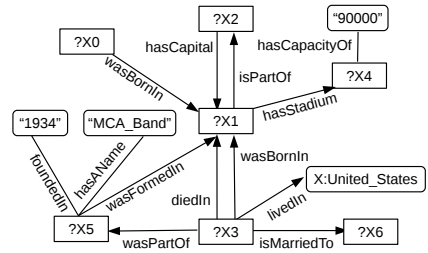
5.3.2.1. *Query multigraph representation*

In any valid SPARQL query (as in Figure 5.2(a)), every triplet has at least one unknown variable $?X$, whose bindings are to be found in the RDF data. It should now be easy to observe that a SPARQL query can be represented in the form of a graph as in Figure 5.2(b), which in turn is transformed into query multigraph Q (as in Figure 5.2(c)).

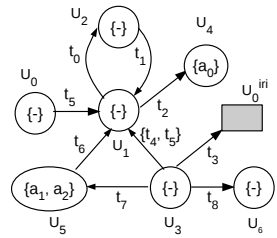
4 http://www.w3.org/TR/2008/REC-rdf-sparql-query-20080115/.

```
SELECT ?X0 ?X1 ?X2 ?X3 ?X4 ?X5 ?X6 WHERE {
    ?X0   y:livedIn          ?X1 .
    ?X1   y:isPartOf         ?X2 .
    ?X2   y:hasCapital       ?X1 .
    ?X1   y:hasStadium       ?X4 .
    ?X3   y:wasBornIn        ?X1 .
    ?X3   y:diedIn           ?X1 .
    ?X3   y:isMarriedTo      ?X6 .
    ?X3   y:wasPartOf        ?X5 .
    ?X5   y:wasFormedIn      ?X1 .
    ?X4   y:hasCapacity      "90000" .
    ?X5   y:hasName          "MCA_Band" .
    ?X5   y:foundedIn        "1934" .
    ?X3   y:livedIn          x:United States . }
```

(a) SPARQL Query G



(b) Graph representation of SPARQL



(c) Equivalent Multigraph Q

Figure 5.2. *a) SPARQL query representation; b) graph representation and c) attributed multigraph Q*

In the query multigraph representation, each unknown variable $?X_i$ is mapped to a vertex u_i that forms the vertex set U component of the query multigraph Q (e.g. $?X_6$ is mapped to u_6). Since a predicate is always instantiated as an *IRI*, we use the edge-type dictionary in Table 5.1(b), to map the predicate to an edge-type identifier $t_i \in T$ (e.g. "isMarriedTo" is mapped to t_8). When an object o_i is a literal, we use the attribute dictionary (Table 5.1(c)), to find the attribute identifier a_i for the predicate-object tuple $<p_i, o_i>$ (e.g. $\{a_0\}$ forms the attribute for vertex u_4). Further, when a subject or an object is an *IRI*, which is a not a variable, we use the vertex dictionary (Table 5.1(a)), to map it to an *IRI*-vertex u_i^{iri} (e.g. "x:United_States" is mapped to u_0^{iri}) and maintain a set of IRI vertices R. Since this vertex is not a variable and a real vertex of the query, we portray it differently by a shaded square-shaped vertex. When a query vertex u_i does not have any vertex attributes associated with it (e.g. u_0, u_1, u_2, u_3, u_6), a null attribute $\{\text{-}\}$ is assigned to it. On the contrary, an *IRI*-vertex $u_i^{iri} \in R$ does not have any attribute. Thus, a SPARQL query is transformed into a query multigraph Q.

s/o	$\mathcal{M}_v(s/o)$
x:Music_Band	v_0
x:Amy_Winehouse	v_1
x:London	v_2
x:England	v_3
x:WembleyStadium	v_4
x:United States	v_5
x:Blake Fielder-Civil	v_6
x:Christopher_Nolan	v_7
x:Dark_Knight_Trilogy	v_8

(a) Vertex Dictionary

p	$\mathcal{M}_e(p)$
y:isPartOf	t_0
y:hasCapital	t_1
y:hasStadium	t_2
y:livedIn	t_3
y:diedIn	t_4
y:wasBornIn	t_5
y:wasFormedIn	t_6
y:wasPartOf	t_7
y:wasMarriedTo	t_8

(b) Edge-type Dictionary

$<p, o>$	$\mathcal{M}_a(<p, o>)$
<y:hasCapacityOf, “90000">	a_0
<y:wasFoundedIn, “1994">	a_1
<y:hasName, “MCA_Band">	a_2

(c) Attribute Dictionary

Table 5.1. *Dictionary look-up tables for vertices, edge-types and vertex attributes*

In this work, we always use the notation V for the set of vertices of G, and U for the set of vertices of Q. Consequently, a data vertex $v \in V$ and a query vertex $u \in U$. Also, an incoming edge to a vertex is positive (default) and an outgoing edge from a vertex is labeled negative (“-”).

5.3.3. *SPARQL querying by adopting multigraph homomorphism*

As we recall, the problem of SPARQL querying is addressed by finding the solutions to the unknown variables $?X$ that can be bound with the RDF data entities so that the relations (predicates) provided in the SPARQL query are respected. In this work, to harness the transformed data multigraph G and the query multigraph Q, we reduce the problem of SPARQL querying to a sub-multigraph homomorphism problem. The RDF data are transformed into data multigraph G, and the SPARQL query is transformed into query multigraph Q. Let us now recall that finding SPARQL answers in the RDF data is equivalent to finding all the sub-multigraphs of Q in G that are homomorphic. Thus, let us now formally introduce homomorphism for a vertex attributed, directed multigraph.

DEFINITION 5.2.– *Sub-multigraph homomorphism. Given a query multigraph* $Q = (U, E^Q, L_U, L_E^Q)$ *and a data multigraph* $G = (V, E, L_V, L_E)$*, the sub-multigraph homomorphism from* Q *to* G *is a surjective function* ψ*:* $U \rightarrow V$ *such that:*

1) $\forall u \in U, L_U(u) \subseteq L_V(\psi(u))$ *and*

2) $\forall (u_m, u_n) \in E^Q$*,* $\exists\ (\psi(u_m), \psi(u_n)) \in E$*, where* (u_m, u_n) *is a directed edge, and* $L_E^Q(u_m, u_n) \subseteq L_E(\psi(u_m), \psi(u_n))$*.*

Thus, by finding all the sub-multigraphs in G that are homomorphic to Q, we enumerate all possible homomorphic embeddings of Q in G. These embeddings contain the solution for each of the query vertex that is an unknown variable. Thus, by using the inverse mapping function $\mathcal{M}_v^{-1}(v_i)$ (referring to the vertex dictionary in Table 5.1(a)), we find the bindings for the SPARQL query. The decision problem of subgraph homomorphism is NP-complete. This standard subgraph homomorphism problem can be seen as a particular case of sub-multigraph homomorphism, where both the labeling functions L_E and L_E^Q always return the same subset of edge types for all the edges in both Q and G. Thus, the problem of sub-multigraph homomorphism is at least as hard as subgraph homomorphism. Further, the subgraph homomorphism problem is a generic scenario of subgraph isomorphism problem, where the injectivity constraints are slackened [KIM 15].

5.4. AMBER: a SPARQL querying engine

The proposed AMBER (Attributed Multigraph Based Engine for RDF querying) contains two different stages: (1) an offline stage, during which RDF data are transformed into multigraph G and then a set of index structures $\mathcal{I}$ are constructed that capture the necessary information contained in G; (2) an online stage, during which a SPARQL query is transformed into a multigraph Q, and then by exploiting the subgraph matching techniques along with the already built index structures $\mathcal{I}$, the homomorphic matches of Q in G are obtained.

Given a multigraph representation Q of a SPARQL query, AMBER decomposes the query vertices U into a set of core vertices U_c and satellite vertices U_s. Intuitively, a vertex $u \in U$ is a core vertex if the degree of the vertex is more than one; on the contrary, a vertex u with degree one is a satellite vertex. For example, in Figure 5.2(c), $U_c = \{u_1, u_3, u_5\}$ and

$U_s = \{u_0, u_2, u_4, u_6\}$. Once decomposed, we run the sub-multigraph matching procedure on the query structure spanned only by the core vertices. However, during the procedure, we also process the satellite vertices (if available) that are connected to a core vertex that is being processed. For example, while processing the core vertex u_1, we also process the set of satellite vertices $\{u_0, u_2, u_4\}$ connected to it; however, the core vertex u_5 has no satellite vertices to be processed. In this way, as the matching proceeds, the entire structure of the query multigraph Q is processed to find the homomorphic embeddings in G. The set of indexing structures $\mathcal{I}$ is extensively used during the process of sub-multigraph matching. The homomorphic embeddings are finally translated back to the RDF entities using the inverse mapping function $\mathcal{M}_v^{-1}$, as discussed in section 5.3.

5.5. Index construction

Given a data multigraph G, we build the following three different indices: (i) an inverted list $\mathcal{A}$ for storing the set of data vertex for each attribute in $a_i \in A$; (ii) a trie index structure $\mathcal{S}$ to store features of all the data vertices V and (iii) a set of trie index structures $\mathcal{N}$ to store the neighborhood information of each data vertex $v \in V$. For brevity of representation, we ensemble all the three index structures into $\mathcal{I} := \{\mathcal{A}, \mathcal{S}, \mathcal{N}\}$. During the query matching procedure (the online step), we access these indexing structures to obtain the candidate solutions for a query vertex u. Formally, for a query vertex u, the *candidate solutions* are a set of data vertices $C_u = \{v | v \in V\}$ obtained by accessing $\mathcal{A}$, $\mathcal{S}$ or $\mathcal{N}$, denoted as $C_u^{\mathcal{A}}$, $C_u^{\mathcal{S}}$ or $C_u^{\mathcal{N}}$, respectively.

5.5.1. *Attribute index*

The set of vertex attributes is given by $A = \{a_0, \ldots, a_n\}$ (section 5.3), where a data vertex $v \in V$ might have a subset of A assigned to it. We now build the vertex attribute index $\mathcal{A}$ by creating an inverted list, where a particular attribute a_i has the list of all the data vertices in which it appears.

Given a query vertex u with a set of vertex attributes $u.A \subseteq A$, for each attribute $a_i \in u.A$, we access the index structure $\mathcal{A}$ to fetch a set of data vertices that have a_i. Then, we find a common set of data vertices that have the entire attribute set $u.A$. For example, considering the query vertex u_5 (Figure 5.2(c)), it has an attribute set $\{a_1, a_2\}$. The candidate solutions for

u_5 are obtained by finding all the common data vertices, in $\mathcal{A}$, between a_1 and a_2, resulting in $C^{\mathcal{A}}_{u_5} = \{v_0\}$.

5.5.2. *Vertex signature index*

The index $\mathcal{S}$ captures the edge-type information from the data vertices. For a lucid understanding of this indexing schema, we formally introduce the notion of vertex signature that is defined for a vertex $v \in V$, which encapsulates the edge information associated with it.

DEFINITION 5.3.– *Vertex signature. For a vertex $v \in V$, the vertex signature σ_v is a multiset containing all the directed multi-edges that are incident on v, where a multi-edge between v and a neighboring vertex v' is represented by a set that corresponds to the edge types. Formally, $\sigma_v = \bigcup_{v' \in N(v)} L_E(v, v')$, where $N(v)$ is the set of neighborhood vertices of v and $\cup$ is the union operator for the multiset.*

Data vertex	Signature	Synopses							
v	σ_v	f_1^+	f_2^+	f_3^+	f_4^+	f_1^-	f_2^-	f_3^-	f_4^-
v_0	$\{\{-t_6\},\{t_7\}\}$	1	1	-7	7	1	1	-6	6
v_1	$\{\{-t_3\},\{-t_7\},\{-t_8\},\{-t_4,-t_5\}\}$	0	0	0	0	2	5	-3	8
v_2	$\{\{-t_0\},\{t_1\},\{-t_2\},\{t_5\},\{t_6\},\{t_4,t_5\}\}$	2	4	-1	6	1	2	0	2
v_3	$\{\{t_0\},\{t_3\},\{-t_1\}\}$	1	2	0	3	1	1	-1	1
v_4	$\{\{t_2\}\}$	1	1	-2	2	0	0	0	0
v_5	$\{\{t_3\},\{t_3\}\}$	1	1	-3	3	0	0	0	0
v_6	$\{\{t_8\},\{-t_3\}\}$	1	1	-8	8	1	1	-3	3
v_7	$\{\{-t_0\},\{-t_3\},\{-t_5\}\}$	0	0	0	0	1	3	0	5
v_8	$\{\{t_0\}\}$	1	1	0	0	0	0	0	0

Table 5.2. *Vertex signatures and the corresponding synopses for the vertices in the data multigraph G (Figure 5.1(c))*

The index $\mathcal{S}$ is constructed by tailoring the information supplied by the vertex signature of each vertex in G. To extract some interesting features, let us observe the vertex signature σ_{v_2} as supplied in Table 5.2. To begin with, we can represent the vertex signature σ_{v_2} separately for the incoming and outgoing multi-edges as $\sigma^+_{v_2} = \{\{t_1\},\{t_5\},\{t_6\},\{t_4,t_5\}\}$ and $\sigma^-_{v_2} = \{\{-t_0\}\{-t_2\}\}$, respectively. Now we observe that $\sigma^+_{v_2}$ has four distinct multi-edges and $\sigma^-_{v_2}$ has two distinct multi-edges. Now, let us assume that we want to find candidate solutions for a query vertex u. The data vertex v_2 can be a match for u only

if the signature of u has at most four incoming ("+") edges and at most two outgoing ("-") edges; else, v_2 cannot be a match for u. Thus, more similar features (e.g. maximum cardinality of a set in the vertex signature) can be proposed to filter out irrelevant candidate vertices. Thus, for each vertex v, we propose to extract a set of features by exploiting the corresponding vertex signature. These features constitute a *synopsis*, which is a surrogate representation that approximately captures the vertex signature information.

The synopsis of a vertex v contains a set of features F, whose values are computed from the vertex signature σ_v. In this background, we propose four distinct features: f_1 – the maximum cardinality of a set in the vertex signature; f_2 – the number of unique dimensions in the vertex signature; f_3 – the minimum index value of the edge type and f_4 – the maximum index value of the edge type. For f_3 and f_4, the index values of edge type are nothing but the position of the sequenced alphabet. These four basic features are replicated separately for outgoing (negative) and incoming (positive) edges, as seen in Table 5.2. Thus, for the vertex v_2, we obtain $f_1^+ = 2$, $f_2^+ = 4$, $f_3^+ = -1$ and $f_4^+ = 7$ for the incoming edge set and $f_1^- = 1$, $f_2^- = 2$, $f_3^- = 0$ and $f_4^- = 2$ for the outgoing edge set. Synopses for the entire vertex set V for the data multigraph G are depicted in Table 5.2.

Once the synopses are computed for all data vertices, an R-tree is constructed to store all the synopses. This R-tree constitutes the vertex signature index $\mathcal{S}$. A synopsis with $|F|$ fields forms a leaf in the R-tree. When a set of possible candidate solutions are to be obtained for a query vertex u, we create a vertex signature σ_u in order to compute the synopsis and then obtain the possible solutions from the R-tree structure.

The general idea of using an R-tree is as follows. A synopsis F of a data vertex spans an axes-parallel rectangle in an $|F|$-dimensional space, where the maximum coordinates of the rectangle are the values of the synopses fields $(f_1, \ldots, f_{|F|})$ and the minimum coordinates are the origin of the rectangle (filled with zero values). For example, a data vertex represented by a synopsis with two features $F(v) = [2, 3]$ spans a rectangle in a two-dimensional space in the interval range $([0, 2], [0, 3])$. Now, if we consider synopses of two query vertices, $F(u_1) = [1, 3]$ and $F(u_2) = [1, 4]$, we observe that the rectangle spanned by $F(u_1)$ is wholly contained in the rectangle spanned by $F(v)$ but $F(u_2)$ is not wholly contained in $F(v)$. Thus, u_1 is a candidate match, whereas u_2 is not.

LEMMA 5.1.– *Querying the vertex signature index $\mathcal{S}$ constructed with synopses guarantees to output at least the entire set of candidate solutions.*

PROOF.– *Consider the field $f_1^{\pm}$ in the synopsis that represents the maximum cardinality of the neighborhood signature. Let σ_u be the signature of the query vertex u and $\{\sigma_{v_1}, \ldots, \sigma_{v_n}\}$ be the set of signatures on the data vertices. By using f_1, we need to show that $C_u^{\mathcal{S}}$ has at least all the valid candidate matches. Since we are looking for a superset of query vertex signatures and we are checking the condition $f_1^{\pm}(u) \leq f_1^{\pm}(v_i)$, where $v_i \in V$, a vertex v_i is pruned if it does not match the inequality criterion, because it can never be an eligible candidate. This analogy can be extended to the entire synopsis, since it can be applied disjunctively.* ■

Formally, the set of candidate solutions for a query vertex u can be written as $C_u^{\mathcal{S}} = \{v | \forall_{i \in [1,\ldots,|F|]} f_i^{\pm}(u) \leq f_i^{\pm}(v)\}$, where the constraints are met for all the $|F|$-dimensions. Since we apply the same inequality constraint to all the fields, we negate the fields that refer to the minimal index value of the edge type (f_3^+ and f_3^-) so that the rectangular containment problem still holds. Further, to respect the rectangular containment, we populate the synopsis fields with "0" values in case the signature does not have either a positive or negative edge in it, as seen for v_1, v_3, v_4, v_5 and v_7.

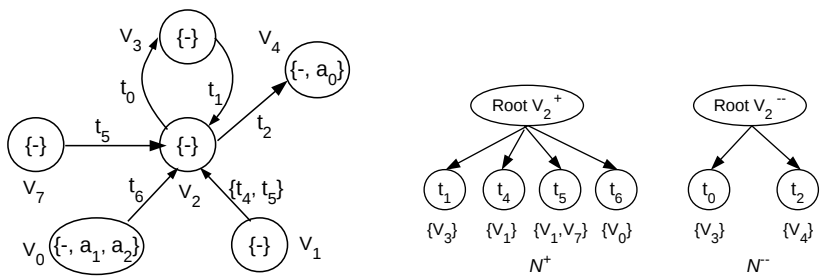
For example, if we want to compute the possible candidates for a query vertex u_0 in Figure 5.2(c), whose signature is $\sigma_{u_0} = \{-t_5\}$, we compute the synopsis [0 0 0 0 1 1 5 5]. Now we look for all those vertices that subsume this synopsis in the R-tree, whose elements are depicted in Table 5.2, which gives us the candidate solutions $C_{u_0}^{\mathcal{S}} = \{v_1, v_7\}$, thus pruning the rest of the vertices.

The $\mathcal{S}$ index helps to prune the vertices that do not respect the edge-type constraints. This is crucial since this pruning is performed for the initial query vertex, and hence many candidates are cast away, thereby avoiding unnecessary recursion during the matching procedure. For example, for the initial query vertex u_0, whose candidate solutions are $\{v_1, v_7\}$, the recursion branch is run only on these two starting vertices instead of the entire vertex set V.

5.5.3. *Vertex neighborhood index*

The *vertex neighborhood index* $\mathcal{N}$ captures the topological structure of the data multigraph G. The index $\mathcal{N}$ comprises 1-neighborhood trees built for each data vertex $v \in V$. Since G is a directed multigraph, and each vertex $v \in V$ can have both incoming and outgoing edges, we construct two separate index structures $\mathcal{N}^+$ and $\mathcal{N}^-$ for incoming and outgoing edges, respectively, that constitute the structure $\mathcal{N}$.

To understand the index structure, let us consider the data vertex v_2 from Figure 5.1(c), shown separately in Figure 5.3(a). For this vertex v_2, we collect all the neighborhood information (vertices and multi-edges) and represent this information via a tree structure, built separately for incoming ("+") and outgoing ("-") edges. Thus, the tree representation of a vertex v contains the neighborhood vertices and the corresponding multi-edges, as shown in Figure 5.3(b), where the vertices of the tree structure are represented by the edge types.



(a) Neighborhood structure of v_2 (b) OTIL structure for v_2

Figure 5.3. *Building Neighborhood Index for data vertex* v_2

In order to construct an efficient tree structure, we take inspiration from Terrovitis *et al.* [TER 06] to propose the structure – Ordered Trie with Inverted List (OTIL). To construct the OTIL index as shown in Figure 5.3(b), we insert each ordered multi-edge that is incident on v at the root of the trie. We consider a data vertex v_i, with a set of n neighborhood vertices $N(v_i)$. Now, for every pair of incoming edge $(v_i, N^j(v_i))$, where $j \in \{1, \ldots, n\}$, there exists a multi-edge $\{t_i, \ldots, t_j\}$, which is inserted into the OTIL structure $\mathcal{N}^+$. Similarly, for every pair of outgoing edge $(N^j(v_i), v_i)$, there exists a multi-edge $\{t_m, \ldots, t_n\}$, which is inserted into the OTIL structure

$\mathcal{N}^-$ maintaining two OTIL structures that constitute $\mathcal{N}$. Each multi-edge is ordered (w.r.t. increasing edge-type indexes), before inserting into the respective OTIL structure, and the order is universally maintained for all data vertices. Further, for every edge type t_i, we maintain a *list* that contains all the neighborhood vertices $N^+(v_i)$/$N^-(v_i)$ that have the edge type t_i incident on them.

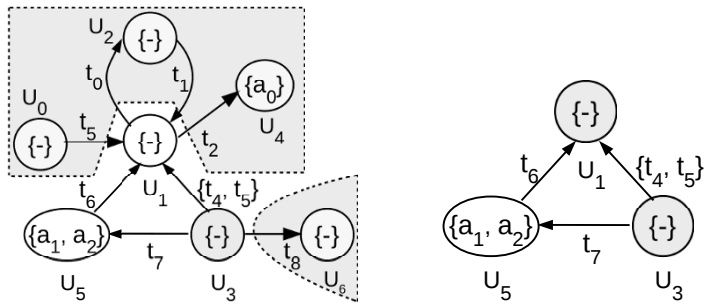
To understand the utility of $\mathcal{N}$, let us consider an illustrative example. Considering the query multigraph Q in Figure 5.2(c), let us assume that we want to find the matches for the query vertices u_1 and u_0 in order. Thus, for the initial vertex u_1, let us say, we have found the set of candidate solutions, which is $\{v_2\}$. Now, to find the candidate solutions for the next query vertex u_0, it is important to maintain the structure spanned by the query vertices, and this is where the indexing structure $\mathcal{N}$ is accessed. Thus, to retain the structure of the query multigraph (in this case, the structure between u_1 and u_0), we have to find the data vertices that are in the neighborhood of already matched vertex v_2 (a match for vertex u_1), which has the same structure (edge types) between u_1 and u_0 in the query graph. Thus, to fetch all the data vertices that have the edge type t_5, which is directed toward v_2 and hence "+", we access the neighborhood index trie $\mathcal{N}^+$ for vertex v_2, as shown in Figure 5.3. This gives us a set of candidate solutions $C^{\mathcal{N}}_{u_0} = \{v_1, v_7\}$. It is easy to observe that, by maintaining two separate indexing structures $\mathcal{N}^+$ and $\mathcal{N}^-$, for both incoming and outgoing edges, we can reduce the time to fetch the candidate solutions.

Thus, in a generic scenario, given an already matched data vertex v, the edge direction "+" or "-" and the set of edge types $T' \subseteq T$, the index $\mathcal{N}$ will find a set of neighborhood data vertices $\{v' | (v', v) \in E \wedge T' \subseteq L_E(v', v)\}$ if the edge direction is "+" (incoming), while $\mathcal{N}$ returns $\{v' | (v, v') \in E \wedge T' \subseteq L_E(v, v')\}$ if the edge direction is "-" (outgoing).

5.6. Query matching procedure

In order to follow the working of the proposed query matching procedure, we formalize the notion of *core* and *satellite* vertices. Given a query graph Q, we decompose the set of query vertices U into a set of *core* vertices U_c and a set of *satellite* vertices U_s. Formally, when the degree of the query graph $\Delta(Q) > 1$, $U_c = \{u | u \in U \wedge deg(u) > 1\}$; however, when $\Delta(Q) = 1$, i.e. when the query graph is either a vertex or a multi-edge, we choose one query vertex at random as a *core* vertex and hence $|U_c| = 1$. The remaining

vertices are classified as satellite vertices, whose degree is always 1. Formally, $U_s = \{U \backslash U_c\}$, where for every $u \in U_s$, $deg(u) = 1$. The decomposition of the query multigraph Q is depicted in Figure 5.4, where the satellite vertices are separated (vertices under the shaded region in Figure 5.4(a)), in order to obtain the query graph that is spanned only by the core vertices (Figure 5.4(b)). Thus, during query decomposition, satellite vertices are separated from the query graph, leaving a core graph and a set of satellite vertices; the original query structure is preserved by storing the edge information that links each satellite vertex and the corresponding core vertex in the query graph spanned by core vertices.



(a) Query graph with satellite vertices (shaded)

(b) Query graph spanned by core vertices

Figure 5.4. *Decomposing the query multigraph into* core *and* satellite *vertices*

The proposed AMBER-Algo (Algorithm 5.3) only performs recursive sub-multigraph matching procedure on the query structure spanned by U_c, as seen in Figure 5.4(b). Since the entire set of satellite vertices U_s is connected to the query structure spanned by the core vertices, AMBER-Algo processes the satellite vertices while performing sub-multigraph matching on the set of core vertices. Thus, during the recursion, if the current $core$ vertex has $satellite$ vertices connected to it, the algorithm directly retrieves a list of possible matching for such $satellite$ vertices and it includes them in the current partial solution. Each time the algorithm executes a recursion branch with a solution, the solution not only contains a data vertex match v_c for each query vertex belonging to U_c but it also contains a set of matched data vertices V_s for each query vertex belonging to U_s. Each time a solution is found, we can generate

not just one, but a set of embeddings through the Cartesian product of the matched elements in the solution.

Since finding SPARQL solutions is equivalent to finding homomorphic embeddings of the query multigraph, the homomorphic matching allows different query vertices to be matched with the same data vertices. We recall that there is no injectivity constraint in the sub-multigraph homomorphism as opposed to the sub-multigraph isomorphism [KIM 15]. Thus, during the recursive matching procedure, we do not have to check if the potential data vertex has already been matched with previously matched query vertices. This is an advantage when we are processing satellite vertices: we can find matches for each satellite vertex independently without the necessity to check for a repeated data vertex.

Before getting into the details of the AMBER-Algo, we first explain how a set of candidate solutions is obtained when there is information that is only associated with the vertices. Then, we explain how a set of candidate solutions is obtained when we encounter the satellite vertices.

5.6.1. *Vertex-level processing*

To understand the generic query processing, it is necessary to understand the matching process at vertex level. Whenever a query vertex $u \in U$ is being processed, we need to check whether u has a set of attributes A associated with it or if any *IRI*s are connected to it (see section 5.3.2).

To process an arbitrary query vertex, we propose a PROCESSVERTEX procedure, depicted in Algorithm 5.1. This algorithm is only invoked when a vertex u has at least either a set of vertex attributes or any *IRI* associated with it. The PROCESSVERTEX procedure returns a set of data vertices $CandAtt_u$, which are matchable with u; if $CandAtt_u$ is empty, then the query vertex u has no matches in V. As seen in Lines 1–2, when a query vertex u has a set of vertex attributes, i.e. $u.A \neq \emptyset$, we obtain the candidate solutions C_u^A by invoking the QUERYATTINDEX procedure, which accesses the index $\mathcal{A}$ as explained in section 5.5.1. For example, the query vertex u_5 with vertex attributes $\{a_1, a_2\}$ can only be matched with the data vertex v_0; thus, $C_{u_5}^A = \{v_0\}$.

Algorithm 5.1. PROCESSVERTEX($u, Q, \mathcal{A}, \mathcal{N}$)

1: **if** $u.A \neq \emptyset$ **then**
2: $\quad C_u^A = \text{QUERYATTINDEX}(\mathcal{A}, u.A)$
3: **if** $u.R \neq \emptyset$ **then**
4: $\quad C_u^I = \bigcap_{u_i^{iri} \in u.R}(\ \text{QUERYNEIGHINDEX}(\mathcal{N}, L_E^Q(u, u_i^{iri}), u_i^{iri})\ )$
5: $CandAtt_u = C_u^A \overline{\cap} C_u^I$ /* `Find common candidates` */
6: **return** $CandAtt_u$

When a query vertex u has *IRI*s associated with it, i.e. $u.R \neq \emptyset$ (Lines 3–4), we find the candidate solutions C_u^I by invoking the QUERYNEIGHINDEX procedure. As seen in section 5.3.2, a vertex u is connected to an *IRI* vertex u_i^{iri} through a multi-edge $L_E^Q(u, u_i^{iri})$. An *IRI* vertex u_i^{iri} always has only one data vertex v that can match. Thus, the candidate solutions C_u^I are obtained by invoking the QUERYNEIGHINDEX procedure, which fetches all the neighborhood vertices of v that respect the multi-edge $L_E^Q(u, u_i^{iri})$. The procedure is invoked until all the IRI vertices $u.R$ are processed (Line 4). Considering the example in Figure 5.2(c), u_3 is connected to an *IRI*-vertex u_0^{iri}, which has a unique data vertex match v_5, through the multi-edge $\{-t_3\}$. Using the neighborhood index $\mathcal{N}$, we look for the neighborhood vertices of v_5 that have the multi-edge $\{-t_3\}$, which gives us the candidate solutions $C_{u_3}^I = \{v_1\}$. Finally, in Line 5, the merge operator $\overline{\cap}$ returns a set of common candidates $CandAtt_u$ only if $u.A \neq \emptyset$ and $u.R \neq \emptyset$. Otherwise, C_u^A or C_u^I are returned as $CandAtt_u$.

5.6.2. *Processing satellite vertices*

In this section, we provide insights into processing a set of satellite vertices $U_{sat} \subseteq U_s$ that are connected to a core vertex $u_c \in U_c$. This scenario results in a structure that appears frequently in SPARQL queries called star structure [GUB 14, HUA 11]. A typical star structure depicted in Figure 5.5 has a core vertex $u_c = u_1$ and a set of satellite vertices $U_{sat} = \{u_0, u_2, u_4\}$ connected to the core vertex. For each candidate solution of the core vertex u_1, we process u_0, u_2, u_4 independently of each other, since there is no structural connectivity

(edges) among them, although they are only structurally connected to the core vertex u_1.

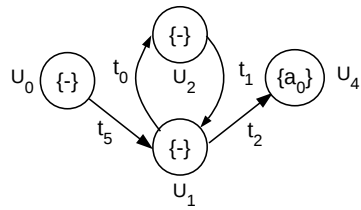


Figure 5.5. *A star structure in the query multigraph Q*

LEMMA 5.2.– *For a given star structure in a query graph, each satellite vertex can be independently processed if a candidate solution is provided for the core vertex u_c.*

PROOF.– *Consider a core vertex u_c that is connected to a set of satellite vertices $U_{sat} = \{u_0, \ldots, u_s\}$, through a set of edge types $T' = \{t_0, \ldots, t_s\}$. Let us assume v_c is a candidate solution for the core vertex u_c, and we want to find candidate solutions for $u_i \in U_{sat}$ and $u_j \in U_{sat}$, where $i \neq j$. Now, the candidate solutions for u_i and u_j can be obtained by fetching the neighborhoods of the already matched vertex v_c that respect the edge types $t_i \in T'$ and $t_j \in T'$, respectively. Since two satellite vertices u_i and u_j are never connected to each other, the candidate solutions of u_i are independent of that of u_j. This analogy applies to all the satellite vertices.* ■

Given a core vertex u_c, we initially find a set of candidate solutions $Cand_{u_c}$ using the index $\mathcal{S}$. Then, for each candidate solution $v_c \in Cand_{u_c}$, the set of solutions for all the satellite vertices U_{sat} that are connected to u_c are returned by the MATCHSATVERTICES procedure described in Algorithm 5.2. The set of solution tuple M_{sat} defined in Line 1 stores the candidate solutions for the entire set of satellite vertices U_{sat}. Formally, $M_{sat} = \{[u_s, V_s]\}_{s=1}^{|U_{sat}|}$, where $u_s \in U_{sat}$ and V_s is a set of candidate solutions for u_s. In order to obtain candidate solutions for u_s, we query the neighborhood index $\mathcal{N}$ (Line 3); the QUERYNEIGHINDEX function returns all the neighborhood vertices of already matched v_c by considering the multi-edge in the query multigraph $L_E^Q(u_c, u_s)$. As every query vertex $u_s \in U_{sat}$ is processed, the solution set M_{sat} that

contains candidate solutions grows until all the satellite vertices have been processed (Lines 2–8).

Algorithm 5.2. MATCHSATVERTICES($\mathcal{A}, \mathcal{N}, Q, U_{sat}, v_c$)

1: SET: $M_{sat} = \emptyset$, where $M_{sat} = \{[u_s, V_s]\}_{s=1}^{|U_{sat}|}$
2: **for all** $u_s \in U_{sat}$ **do**
3: $Cand_{u_s} =$ QUERYNEIGHINDEX($\mathcal{N}, L_E^Q(u_c, u_s), v_c$)
4: $Cand_{u_s} = Cand_{u_s} \cap$ PROCESSVERTEX($u_s, Q, \mathcal{A}, \mathcal{N}$)
5: **if** $Cand_{u_s} \neq \emptyset$ **then**
6: $M_{sat} = M_{sat} \cup (u_s, Cand_{u_s})$ /* `Satellite solutions` */
7: **else**
8: **return** $M_{sat} := 0$ /* `No solutions possible` */
9: **return** M_{sat} /* `Matches for satellite vertices` */

In Line 4, the set of candidate solutions $Cand_{u_s}$ are refined by invoking Algorithm 5.1 (VERTEXPROCESSING). After the refinement, if there are finite candidate solutions, we update the solution M_{sat}; else, we terminate the procedure as there can be no matches for a given matched vertex v_c. The MATCHSATVERTICES procedure performs two tasks: (1) it checks whether the candidate vertex $v_c \in Cand_{u_s}$ is a valid matchable vertex; and (2) it obtains the solutions for all the satellite vertices.

5.6.3. *Arbitrary query processing*

Algorithm 5.3 shows the generic procedure we develop to process arbitrary queries. We recall that for an arbitrary query Q, we define two different types of vertexes: a set of core vertices U_c and a set of satellite vertices U_s. The QUERYDECOMPOSE procedure in Line 1 of Algorithm 5.3. performs this decomposition by splitting the query vertices U into U_c and U_s, as observed in Figure 5.4. To process arbitrary query multigraphs, we perform recursive sub-multigraph matching procedure on the set of core vertices $U_c \subseteq U$; during the recursion, satellite vertexes connected to a specific core vertex are

processed too. Since the recursion is performed on the set of core vertices, we propose a few heuristics for ordering the query vertices.

Ordering of the query vertices forms one of the vital steps for subgraph matching algorithms [KIM 15]. In any subgraph matching algorithm, the embeddings of a query subgraph are obtained by exploring the solution space spanned by the data graph. But since the solution space itself can grow exponentially in size, we are compelled to use intelligent strategies to traverse the solution space. In order to achieve this, we propose a heuristic procedure VERTEXORDERING (Line 2, Algorithm 5.3) that uses two ranking functions.

The first ranking function r_1 relies on the number of satellite vertices connected to the core vertex, and the query vertices are ordered with the decreasing rank value. Formally, $r_1(u) = |U_{sat}|$, where $U_{sat} = \{u_s | u_s \in U_s \wedge (u, u_s) \in E(Q)\}$. A vertex with more satellite vertices connected to it is rich in structure and hence it would probably yield fewer candidate solutions to be processed under recursion. Thus, in Figure 5.4, u_1 is chosen as an initial vertex. The second ranking function r_2 relies on the number of incident edges on a query vertex. Formally, $r_2(u) = \sum_{j=1}^{m} |\sigma(u)^j|$, where u has m multi-edges and $|\sigma(u)^j|$ captures the number of edge types in the j^{th} multi-edge. Again, U_c^{ord} contains the ordered vertices with the decreasing rank value r_2. Further, when there are no satellite vertices in the query Q, this ranking function is of high priority. Despite the use of any ranking function, the query vertices in U_c^{ord}, when accessed in sequence, should be structurally connected to the previous set of vertices. If two vertices have the same rank, the rank with lower priority determines which vertex wins. Thus, for the example in Figure 5.4, the set of ordered core vertices is $U_c^{ord} = \{u_1, u_3, u_5\}$.

The first vertex in the set U_c^{ord} is chosen as the initial vertex u_{init} (Line 3), and subsequent query vertices are chosen in sequence. The candidate solutions for the initial query vertex $CandInit$ are returned by QUERYSYNINDEX procedure (Line 4), which are constrained by the structural properties (neighborhood structure) of u_{init}. By querying the index $\mathcal{S}$ for initial query vertex u_{init}, we obtain the candidate solutions $CandInit \in V$ that match the structure (multi-edge types) associated with u_{init}. Although some candidates in $CandInit$ may be invalid, all valid candidates are present in $CandInit$, as deduced in Lemma 5.1. Furthermore, PROCESSVERTEX procedure is invoked to obtain the candidate solutions according to vertex attributes and *IRI* information, and only then are the common candidates retained.

Algorithm 5.3. AMBER-Algo $(\mathcal{I}, Q)$

1: QUERYDECOMPOSE: Split U into U_c and U_s
2: U_c^{ord} = VERTEXORDERING(Q, U_c)
3: $u_{init} = u | u \in U_c^{ord}$
4: $CandInit$ = QUERYSYNINDEX$(u_{init}, \mathcal{S})$
5: $CandInit = CandInit \cap$ PROCESSVERTEX$(u_{init}, Q, \mathcal{A}, \mathcal{N})$
6: FETCH: $U_{init}^{sat} = \{u | u \in U_s \wedge (u_{init}, u) \in E(Q)\}$
7: SET: $Emb = \emptyset$
8: **for** $v_{init} \in CandInit$ **do**
9: SET: $M = \emptyset, M_s = \emptyset, M_c = \emptyset$
10: **if** $U_{init}^{sat} \neq \emptyset$ **then**
11: $M_{sat} =$ MATCHSATVERTICES$(\mathcal{A}, \mathcal{N}, Q, U_{init}^{sat}, v_{init})$
12: **if** $M_{sat} \neq \emptyset$ **then**
13: **for** $[u_s, V_s] \in M_{sat}$ **do**
14: UPDATE: $M_s = M_s \cup [u_s, V_s]$
15: UPDATE: $M_c = M_c \cup [u_{init}, v_{init}]$
16: $Emb = Emb \cup$ HOMOMORPHICMATCH$(M, \mathcal{I}, Q, U_c^{ord})$
17: **else**
18: UPDATE: $M_c = M_c \cup (u_{init}, v_{init})$
19: $Emb = Emb \cup$ HOMOMORPHICMATCH$(M, \mathcal{I}, Q, U_c^{ord})$
20: **return** Emb /* `Homomorphic embeddings of query multigraph` */

Before getting into the algorithmic details, we explain how the solutions are handled and how we process each query vertex. We define M as a set of tuples, whose i^{th} tuple is represented as $M_i = [m_c, M_s]$, where m_c is a solution pair for a core vertex and M_s is a set of solution pairs for the set of satellite vertices that are connected to the core vertex. Formally, $m_c = (u_c, v_c)$, where u_c is the core vertex and v_c is the corresponding matched vertex; M_s is a set of solution

pairs, whose j^{th} element is a solution pair (u_s, V_s), where u_s is a satellite vertex and V_s is a set of matched vertices. In addition, we maintain a set M_c whose elements are the solution pairs for all the core vertices. Thus, during each recursion branch, the size of M increases until it reaches the query size $|U|$; once $|M| = |U|$, homomorphic matches are obtained.

For all the candidate solutions of initial vertex $CandInit$, we perform recursion to obtain homomorphic embeddings (Lines 8–19). Before getting into recursion, for each initial match $v_{init} \in CandInit$, if it has satellite vertices connected to it, we invoke the MATCHSATVERTICES procedure (Lines 10–11). This step not only finds solution matches for satellite vertices (if present), but also checks if v_{init} is a valid candidate vertex. If the returned solution set M_{sat} is empty, then v_{init} is not a valid candidate and hence we continue with the next $v_{init} \in CandInit$; else, we update the set of solution pairs M_s for satellite vertices and the solution pair M_c for the core vertex (Lines 12–15) and invoke the HOMOMORPHICMATCH procedure (Lines 17). On the contrary, if there are no satellite vertices connected to u_{init}, we update the core vertex solution set M_c and invoke the HOMOMORPHICMATCH procedure (Lines 18–19).

In the HOMOMORPHICMATCH procedure (Algorithm 5.4), we fetch the next query vertex from the set of ordered core vertices U_c^{ord} (Line 4). Then, we collect the neighborhood vertices of already matched core query vertices and the corresponding matched data vertices (Lines 5–6). As we recall, the set M_c maintains the solution pair $m_c = (u_c, v_c)$ of each matched core query vertex. The set N_q collects the already matched core vertices $u_c \in M_c$ that are also in the neighborhood of u_{nxt}, whose matches have to be found. Further, N_g contains the corresponding matched query vertices $v_c \in M_c$. As the recursion proceeds, we find those matchable data vertices of u_{nxt} that are in the neighborhood of all the matched vertices $v \in N_g$ so that the query structure is maintained. In Line 7, for each $u_n \in N_q$ and the corresponding $v_n \in N_g$, we query the neighborhood index $\mathcal{N}$ to obtain the candidate solutions $Cand_{u_{nxt}}$ that are in the neighborhood of already matched data vertex v_n and have the multi-edge $L_E^Q(u_n, u_{nxt})$, obtained from the query multigraph Q. Finally (Line 7), the set of candidate solutions that are common for every $u_n \in N_q$ are retained in $Cand_{u_{nxt}}$.

Algorithm 5.4. HOMOMORPHICMATCH($M, \mathcal{I}, Q, U_c^{ord}$)

1: **if** $|M| = |U|$ **then**
2: **return** GENEMB(M)
3: $Emb = \emptyset$
4: FETCH: $u_{nxt} = u | u \in U_c^{ord}$
5: $N_q = \{u_c | u_c \in M_c\} \cap adj(u_{nxt})$
6: $N_g = \{v_c | v_c \in M_c \wedge (u_c, v_c) \in M_c\}$, where $u_c \in N_q$
7: $Cand_{u_{nxt}} = \bigcap_{n=1}^{|N_q|}$(QueryNeighIndex($\mathcal{N}, L_E^Q(u_n, u_{nxt}), v_n$))
8: $Cand_{u_{nxt}} = Cand_{u_{nxt}} \cap$ PROCESSVERTEX($u_{nxt}, Q, \mathcal{A}, \mathcal{N}$)
9: **for** each $v_{nxt} \in Cand_{u_{nxt}}$ **do**
10: FETCH: $U_{nxt}^{sat} = \{u | u \in V_s \wedge (u_{nxt}, u) \in E(Q)\}$
11: **if** $U_{nxt}^{sat} \neq \emptyset$ **then**
12: $M_{sat} =$ MATCHSATVERTICES($\mathcal{A}, \mathcal{N}, Q, U_{nxt}^{sat}, v_{nxt}$)
13: **if** $M_{sat} \neq \emptyset$ **then**
14: **for** every $[u^s, V^s] \in M_{sat}$ **do**
15: UPDATE: $M_s = M_s \cup [u^s, V^s]$
16: UPDATE: $M_c = M_c \cup (u_{nxt}, v_{nxt})$
17: $Emb = Emb \cup$ HOMOMORPHICMATCH($M, \mathcal{I}, Q, U_c^{ord}$)
18: **else**
19: UPDATE: $M_c = M_c \cup (u_{nxt}, v_{nxt})$
20: $Emb = Emb \cup$ HOMOMORPHICMATCH($M, \mathcal{I}, Q, U_c^{ord}$)
21: **return** Emb

Further, the candidate solutions are refined using the PROCESSVERTEX procedure (Line 8). Now, for each of the valid candidate solutions $v_{nxt} \in Cand_{u_{nxt}}$, we recursively call the HOMOMORPHICMATCH procedure. When the next query vertex u_{nxt} has no satellite vertex attached to it, we update the core vertex solution set M_c and call the recursion procedure (Lines 19–20). But when u_{nxt} has satellite vertices attached to it, we obtain the candidate

matches for all the satellite vertices by invoking the MATCHSATVERTICES procedure (Lines 11–12); if there are matches, we update both the satellite vertex solution M_s and the core vertex solution M_c and invoke the recursion procedure (Line 17).

Once all the query vertices have been matched for the current recursion step, the solution set M contains the solutions for both core and satellite vertices. Thus, when all the query vertices have been matched, we invoke the GENEMB function (Line 2), which returns the set of embeddings that are updated in Emb. The GENEMB function treats the solution vertex v_c of each core vertex as a singleton and performs Cartesian product among all the core vertex singletons and satellite vertex sets. Formally, $Emb_{part} = \{v_c^1\} \times \cdots \times \{v_c^{|U_c|}\} \times V_s^1 \times \cdots \times V_c^{|U_s|}$. Thus, the partial set of embeddings Emb_{part} is added to the final result Emb.

5.7. Experimental analysis

In this section, we report on our extensive experiments on two RDF data sets. We evaluate the time performance and the robustness of AMBER w.r.t., the state-of-the-art competitors by varying the size and structure of the SPARQL queries. Experiments are carried out on a 64-bit Intel Core i7-4900MQ @ 2.80GHz, with 32GB memory, running Linux OS - Ubuntu 14.04 LTS. AMBER is implemented in C++.

5.7.1. *Experimental setup*

We compare AMBER with the four standard RDF engines: *Virtuoso-7.1* [ERL 12], *x-RDF-3X* [NEU 10], *Apache Jena* [CAR 04] and *gStore* [ZOU 14b]. For all these competitors, we use the source code available on the website or obtained by the authors. Another recent work, *Turbo_HOM++* [KIM 15], has been excluded, since it is not publicly available.

For experimental analysis, we use two RDF data sets: *DBPEDIA* and *YAGO*. *DBPEDIA* constitutes the most important knowledge base for the Semantic Web community. Most of the data available in this data set come from the Wikipedia Infobox. *YAGO* is a real-world data set built from factual information obtained from the *Wikipedia* and *WordNet* semantic network. The times required to build the multigraph database as well as to construct the

indexes are reported in Table 5.3(b). We note that the database building time and the corresponding size are proportional to the number of triples. Regarding the indexing structures, we underline that both building time and size are proportional to the number of edges. For instance, *DBPEDIA* has the highest number of edges (∼15M) and, as a result, AMBER uses more time and space to build and store its data structure.

Dataset	# Triples	# Vertices	# Edges	# Edge types
DBPEDIA	33 071 359	4 983 349	14 992 982	676
YAGO	35 543 536	3 160 832	10 683 425	44

(a) Statistics of datasets

Dataset	Database		Index $\mathcal{I}$	
	Building Time	Size	Building Time	Size
DBPEDIA	307	1300	45.18	1573
YAGO	379	2400	29.1	1322

(b) Database and index construction time (seconds) and memory usage (Mbytes)

Table 5.3.

5.7.2. *Workload generation*

In order to test the scalability and robustness of different RDF engines, we generate the query workloads considering a similar setting as in [GUB 14, ALU 14a, HAN 13]. We generate the query workload from the respective RDF data sets, which are available as RDF triplesets. Specifically, we generate two types of query sets: a star-shaped and a complex-shaped query set; furthermore, both query sets are generated for varying sizes (say k) ranging from 10 to 50 triples, in steps of 10.

To generate star-shaped or complex-shaped queries of size k, we pick an initial entity at random from the RDF data. Now, to generate star queries, we check if the initial entity is present in at least k triples in the entire benchmark, to verify whether the initial entity has k neighbors. If so, we choose those k triples at random; thus, the initial entity forms the central vertex of the star structure and the rest of the entities form the remaining star structure, connected by the respective predicates. To generate complex-shaped queries of size k, we navigate in the neighborhood of the initial entity through the predicate links until we reach size k. In both query types, we inject

some object literals as well as constant *IRI*s; the remaining *IRI*s (subjects or objects) are treated as variables. However, this strategy could choose some very unselective queries [GUB 14]. In order to address this issue, we set a maximum time constraint of 60 s for each query. If the query is not answered in time, then it will not be considered for the final average (a similar procedure is usually employed for graph query matching [HAN 13] and RDF workload evaluation [ALU 14a]). We report the average query time as well as the percentage of unanswered queries (considering the given time constraint) to study the robustness of the approaches.

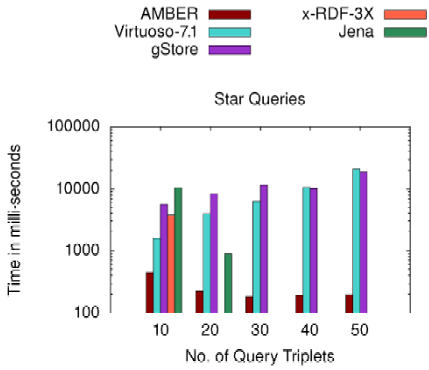
5.7.3. *Comparison with RDF engines*

In this section, we report and discuss the results obtained by different RDF engines. For each combination of query type and benchmark, we report the following two plots by varying the query size: the average time and the corresponding percentage of unanswered queries for the given time constraint. We recall that the average time per approach is computed only on the set of queries that were answered.

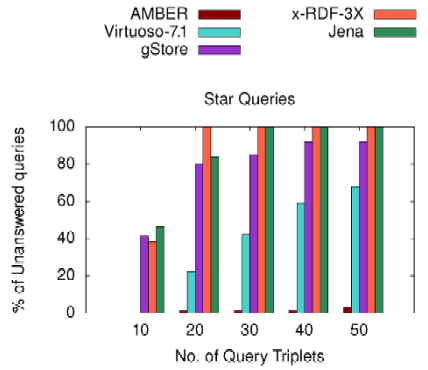
The experimental results for *DBPEDIA* are depicted in Figure 5.6. The time performance (averaged over 200 queries) for *star-shaped* queries (Figure 5.6(a)) affirm that AMbER clearly outperforms all the competitors. Furthermore, the robustness of each approach, evaluated in terms of percentage of unanswered queries within the stipulated time, is shown in Figure 5.6(b). For the given time constraint, *x-RDF-3X* and *Jena* are unable to output results for size 20 and 30 onward, respectively. Although *Virtuoso* and *gStore* output results until query size 50, their time performance is still poor. However, as the query size increases, the percentages of unanswered queries for both *Virtuoso* and *gStore* keep on increasing from $\sim$0% to 65% and from $\sim$45% to 95%, respectively. On the contrary, AMbER answers >98% of the queries, even for queries of size 50, establishing its robustness.

Analyzing the results for *complex-shaped* queries, we observe that in Figure 5.6(c), *x-RDF-3X* and *Jena* are the slowest engines; *Virtuoso* and *gStore* perform better than them but nowhere close to AMbER. We further observe that *x-RDF-3X* and *Jena* are the least robust as they do not output results for size 30 onward (Figure 5.6(d)); on the contrary, AMbER is the most robust engine as it answers >85% of the queries even for size 50. The percentages of

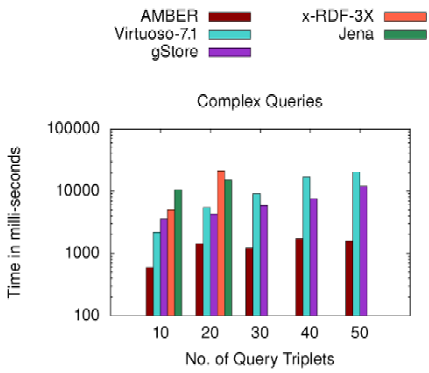
unanswered queries for *Virtuoso* and *gStore* increase from 0% to ∼80% and from 25% to ∼70%, respectively, as we increase the size from 10 to 50.



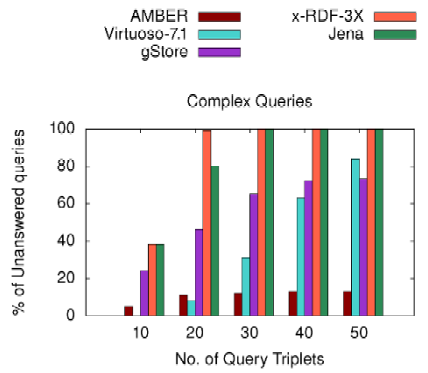
(a) Time performance (Star queries)



(b) % Unanswered queries (Star queries)



(c) Time performance (Complex queries)

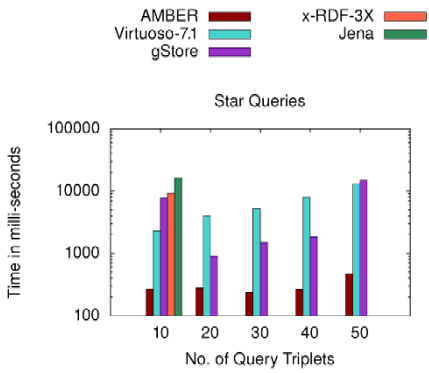


(d) % Unanswered queries (Complex queries)

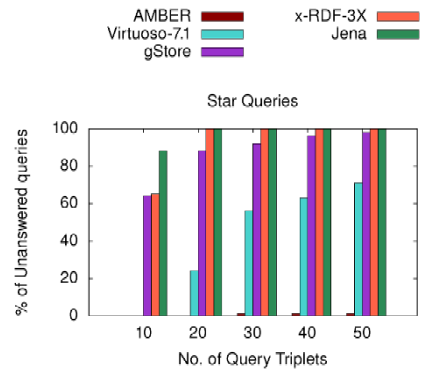
Figure 5.6. *Evaluation of time performance and robustness for DBPEDIA. For a color version of this figure, see www.iste.co.uk/pivert/nosql.zip*

The results for *YAGO* are reported in Figure 5.7. For *star-shaped* queries, the time performance of AMBER is 1–2 orders of magnitude higher than that of its nearest competitor *Virtuoso,* as observed in Figure 5.7(a), and the performance remains stable even with increasing query size (Figure 5.7(b)). *x-RDF-3X* and *Jena* are not able to output results for size 20 onward. As observed for *DBPEDIA*, *Virtuoso* seems to become less robust with the increasing query size. For size 20–40, the time performance of *gStore* seems

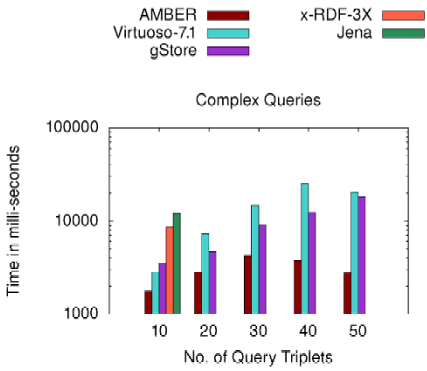
to be higher than that of *Virtuoso*, due to the fewer queries that are being considered. Conversely, AMBER is able to supply answers most of the time (>98%). It can be observed from the results for *complex-shaped* queries that AMBER is still the best in time performance, as seen in Figure 5.7(c); *Virtuoso* and *gStore* are the closest competitors. Only for sizes 10 and 20, *Virtuoso* seems more robust than AMBER. *Jena* and *x-RDF-3X* do not answer queries for size 20 onward, as seen in Figure 5.7(d).



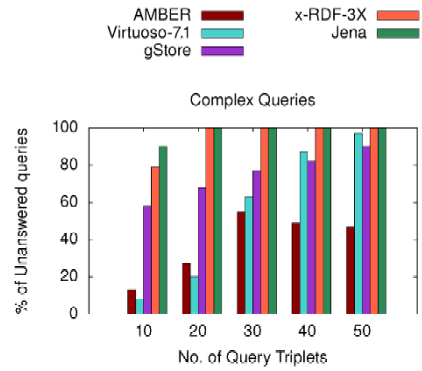
(a) Time performance (Star queries)



(b) % Unanswered queries (Star queries)



(c) Time performance (Complex queries)



(d) % Unanswered queries (Complex queries)

Figure 5.7. *Evaluation of time performance and robustness for YAGO. For a color version of this figure, see www.iste.co.uk/pivert/nosql.zip*

To summarize, we observe that *Virtuoso* is robust for *complex-shaped* smaller queries (10–20), but fails for larger (>20) queries. *x-RDF-3X* fails

for queries with size larger than 10. *Jena* shows reasonable behavior until size 20 but not from size 30 onward. *gStore* shows a reasonable behavior for size 10, but its robustness deteriorates from size 20 onward. AMbER clearly outperforms the state-of-the-art approaches in terms of time performance and robustness.

5.8. Conclusion

In this chapter, a multigraph-based engine AMbER has been proposed in order to answer SPARQL queries. The multigraph representation has given us the following two advantages: (1) it enables us to construct lightweight indexing structures that ameliorate the time performance of AMbER; (2) the graph representation itself motivates us to exploit recent graph management techniques. The proposed engine AMbER has been tested over large RDF triplestores. We have observed that AMbER outperforms its state-of-the-art competitors on two main aspects: (1) its robustness with respect to the query size and (2) its excellent time performance.

5.9. Acknowledgment

This work was funded by Labex NUMEV (NUMEV, ANR-10-LABX-20).

5.10. Bibliography

[ALU 14a] Aluç G., Hartig O., Özsu M.T. *et al.*, "Diversified stress testing of RDF data management systems", *ISWC*, pp. 197–212, 2014.

[ALU 14b] Aluç G., Özsu M.T., Daudjee K., "Workload matters: why RDF databases need a new design", *PVLDB*, vol. 7, no. 10, pp. 837–840, 2014.

[BRO 02] Broekstra J., Kampman A., van Harmelen F., "Sesame: a generic architecture for storing and querying RDF and RDF schema", *ISWC*, pp. 54–68, 2002.

[CAB 12] Cabrio E., Cojan J., Aprosio A.P. *et al.*, "QAKiS: an open domain QA system based on relational patterns", *ISWC*, 2012.

[CAR 04] Carroll J.J., Dickinson I., Dollin C. *et al.*, "Jena: implementing the semantic web recommendations", *WWW*, pp. 74–83, 2004.

[DAS 14] Das S., Srinivasan J., Perry M. *et al.*, "A tale of two graphs: property graphs as RDF in oracle", *EDBT*, pp. 762–773, 2014.

[ERL 12] Erling O., "Virtuoso, a hybrid RDBMS/graph column store", *IEEE Technical Committee on Data Engineering*, vol. 35, no. 1, pp. 3–8, 2012.

[GUB 14] GUBICHEV A., NEUMANN T., "Exploiting the query structure for efficient join ordering in SPARQL queries", *EDBT*, pp. 439–450, 2014.

[HAN 13] HAN W., LEE J., LEE J., "Turbo$_{\text{iso}}$: towards ultrafast and robust subgraph isomorphism search in large graph databases", *SIGMOD*, pp. 337–348, 2013.

[HUA 11] HUANG J., ABADI D.J., REN K., "Scalable SPARQL querying of large RDF graphs", *PVLDB*, vol. 4, no. 11, pp. 1123–1134, 2011.

[ING 16] INGALALLI V., IENCO D., PONCELET P. *et al.*, "Querying RDF data using a multigraph-based approach", *EDBT: Extending Database Technology*, 2016.

[KIM 15] KIM J., SHIN H., HAN W. *et al.*, "Taming subgraph isomorphism for RDF query processing", *PVLDB*, vol. 8, no. 11, pp. 1238–1249, 2015.

[MOR 11] MORSEY M., LEHMANN J., AUER S. *et al.*, "DBpedia SPARQL benchmark performance assessment with real queries on real data", *ISWC*, pp. 454–469, 2011.

[NEU 10] NEUMANN T., WEIKUM G., "x-RDF-3X: fast querying, high update rates, and consistency for RDF databases", *PVLDB*, vol. 3, no. 1, pp. 256–263, 2010.

[PRU 08] PRUD'HOMMEAUX E., SEABORNE A., SPARQL query language for RDF, W3C recommendation, W3C, January 2008, http://www.w3.org/TR/2008/REC-rdf-sparql-query-20080115/.

[TER 06] TERROVITIS M., PASSAS S., VASSILIADIS P. *et al.*, "A combination of trie-trees and inverted files for the indexing of set-valued attributes", *CIKM*, ACM, pp. 728–737, 2006.

[ZOU 14a] ZOU L., HUANG R., WANG H. *et al.*, "Natural language question answering over RDF: a graph data driven approach", *SIGMOD Conference*, pp. 313–324, 2014.

[ZOU 14b] ZOU L., ÖZSU M.T., CHEN L. *et al.*, "gStore: a graph-based SPARQL query engine", *VLDB Journal*, vol. 23, no. 4, pp. 565–590, 2014.

6

Fuzzy Preference Queries to NoSQL Graph Databases

6.1. Introduction

The motivations for integrating user preferences into database queries are manifold [HAD 11]. First, it has become desirable to offer more expressive query languages that can be more faithful to what a user intends to say. Second, the introduction of preferences in queries provides a basis for rank-ordering the retrieved items, which is especially valuable in cases of large sets of items satisfying a query. Third, a classical query may also have an empty set of answers, while a relaxed (and thus less restrictive) version of the query might be matched by some items. A great deal of work has been carried out on preference queries in relational databases [STE 11] and more specifically concerning the fuzzy querying of *relational* databases, see, for instance, [PIV 12], which led in particular to a fuzzy extension of SQL, called SQLf [BOS 95]. However, even though relational databases are still widely used, the need to handle *complex* data has led to the emergence of other types of data models. Thus, a new concept has started to attract much attention in the database world, namely that of *graph databases* (see [ANG 08, WOO 12, ANG 12]). The basic purpose of graph databases is to efficiently manage networks of entities, where each node is described by a set of characteristics (e.g. a set of attributes) and each edge represents a link

Chapter written by Arnaud CASTELLTORT, Anne LAURENT, Olivier PIVERT, Olfa SLAMA and Virginie THION.

NoSQL Data Models: Trends and Challenges, First Edition. Edited by Olivier Pivert.

between entities. Such a database model has many potential applications, for example, modeling social networks, RDF data, cartographic databases and bibliographic databases.

Graph databases raise new challenges in terms of flexible querying, since the following two aspects can be involved in the preferences that a user may express: (1) the content of the nodes/edges and (2) the structure of the graph itself. Furthermore, graph database management systems still lack query languages with a clear syntax and semantics [BAR 13].

In this chapter, we present a framework that makes it possible to introduce fuzzy preferences in queries over graph databases. We first introduce, in section 6.2, the notion of graph databases and preliminary notions concerning fuzzy set theory. Then, in section 6.3, we present, following a common perspective, two contributions of the scientific literature that allow for a flexible querying of crisp and fuzzy graph databases. These contributions choose the same approach which consists in extending a query language in order to introduce fuzzy preferences in graph pattern queries. Both these approaches consider the extension of the Cypher query language implemented in the Neo4j graph database management system. Implementation issues are exposed in section 6.4, and related work is discussed in section 6.5. Conclusions drawn from this chapter are presented in section 6.6.

6.2. Preliminary statements

We first introduce some background notions concerning graph databases in section 6.2.1 and then fuzzy set theory in section 6.2.2.

6.2.1. *Graph databases*

We present the graph data model in section 6.2.1.1 and explain how to query such a graph database through pattern queries in section 6.2.1.2. We then present the Cypher query language in section 6.2.1.3, which is a concrete language that makes it possible to query a graph database in the Neo4j graph database management system.

6.2.1.1. *The graph data model*

A graph database model is a model in which the data structures for the instances and/or the schema are modeled as a directed, possibly labeled graph

or generalizations of the graph structure, where data manipulation is expressed by graph-oriented operations and type constructors [ANG 08]. Such a model makes it possible to naturally model networks of entities, where each node is described by a set of characteristics (for instance, a set of attributes) and each edge represents a link between entities, for example, modeling social networks, RDF data, cartographic databases and bibliographic databases.

There are different models for graph databases that consider different variations of the definition of a graph, for example, flat graph, hypergraph, hypernodes, directed or undirected edges and simple or complex relations. We refer the reader to [ANG 08] for an overview of graph data models and [ANG 12] for a comparison of some of them. We consider here the model that makes it possible to represent data as a finite directed graph with labeled edges, where properties may be embedded in nodes and edges. Such a model is referred to as the *attributed graph* model (a.k.a. the *property graph* model). An instance of this model is simply called *graph data* in the following.

Let us formally define the concept of *data graph* that we consider in the following. We first assume the existence of the following pairwise disjoint sets: a set V of nodes, a set $\mathcal{E}$ of labels for (directed) edges and a set $\mathcal{A}$ of attribute names. Definition 6.1 is a formal definition of data graph based on the attributed graph model.

DEFINITION 6.1 (Data Graph).– *A* data graph*, a.k.a.* graph database*, G is a quadruple (V, R, κ, ζ), where V is a finite set of nodes, $R = \{r_e \mid e \in \mathcal{E}$ and $r_e \subseteq V \times V\}$ is a set of labeled edges between nodes of V, and κ (resp. ζ) is a function in $\mathcal{A} \times V$ (resp. $\mathcal{A} \times (V \times \mathcal{E} \times V)$) assigning attributed values to nodes (resp. edges) of G.*

We suppose that each node n is identified by an id attribute denoted by $n.id$. Nodes are assumed to be typed. If n is a node of V, then $Type(n)$ denotes its type.

EXAMPLE 6.1.– *Figure 6.1, referred to as $\mathcal{DB}$ in the following, is an example of a data graph inspired from an excerpt of DBLP*[1]*. This graph contains scientific publications and authors' names. In $\mathcal{DB}$, the nodes* `WWW12`*,* `Pods11` *and* `Pods13` *are of type* `Conference`*; the nodes* `PodsBBV11`*,* `PodsTU13`*,*

1 http://www.informatik.uni-trier.de/ley/db/.

`PodsB13`, `TodsS81` *and* `WWW_VVT12` *are of type* `Article`*; the nodes* `Pods`, `Tods` *and* `WWW` *are of type* `Series` *and the other nodes are of type* `Author`. *Attribute values attached to each node/edge are denoted by a set of key–value pairs. For instance, the set* `{year:2013, pages:9}` *attached to the node* `PodsTU13` *assigns to this node the value* `2013` *to its attribute* `year` *and the value* `9` *to its attribute* `pages`.

6.2.1.2. *Querying a graph database*

Most of the query languages for graph databases are based on graph pattern matching. Roughly speaking, a query is a graph pattern that is found in the graph database. The evaluation process consists in binding elements of the pattern in subgraphs of the database.

A graph pattern query takes the form of a graph where variables and conditions can occur on nodes and edges. The answer to such a query $\mathcal{P}$ over a graph database $\mathcal{G}$ is the largest set of subgraphs of $\mathcal{G}$ defined by $\{g \in \mathcal{P}(\mathcal{G}) \mid g \text{ "matches" } \mathcal{P}\}$, where $\mathcal{P}(\mathcal{G})$ is the set of subgraphs of $\mathcal{G}$ and g "matches" $\mathcal{P}$ iff there is a homomorphism h from nodes and labels of $\mathcal{P}$ to g [GAL 06, BAR 14] and each node $h(n)$ (resp. label $h(e)$) satisfies its associated conditions.

A query may denote a simple path connecting two nodes, a regular path query (path defined by a regular expression) or more expressively a conjunctive regular path query (a graph pattern where each edge is a regular path expression) possibly including specific navigational capabilities, for example, inverse traversal, nested regular expressions or memory registers (see [BAR 13, LIB 13, BAR 14]).

EXAMPLE 6.2.– *Figure 6.2 is a simple example of a graph pattern (query) that aims at retrieving the authors with an article published in the* Pods *conference after* 2010 *and another article in the* `WWW` *conference. The elements* `au1`, `ar1`, `ar2`, `s1` *and* `s2` *are query variables. Each* `WHERE` *clause associated with a node denotes a condition over the node. The elements* `Conference`, `Author` *and* `Article` *are node types.*

The answer to this query over the data graph $\mathcal{DB}$ (Figure 6.1) is the set of subgraphs of Figure 6.3, which are the subgraphs of $\mathcal{DB}$ that match the pattern.

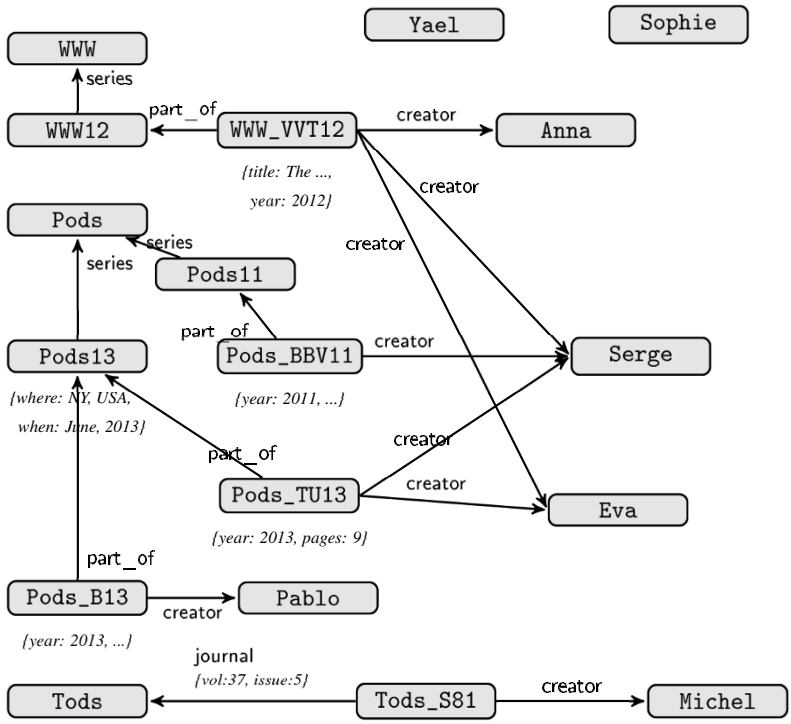


Figure 6.1. *A data graph $\mathcal{DB}$ inspired from DBLP*

6.2.1.3. *The Neo4j Cypher query language*

A graph database management system enables the management of graph databases, facilitating efficient storage and querying of such data. Let us mention the existing systems AllegroGraph [ALL 18], DataStax Enterprise Graph [DAT 18], InfiniteGraph [INF 18], Neo4j [NEO 18], OrientDB [ORI 18] and Sparksee [SPA]. Among these, Neo4j supports the attributed graph model (see section 6.2.1.1) and provides a query language, called Cypher [NEO 13], based on graph pattern matching (see section 6.2.1.2) which supports five of the essential kinds of graph queries [ANG 12].

Cypher [NEO 13] is based on ASCII-art for graph pattern representation. A graph pattern expressed in the *Cypher* formalism consists of a set of expressions of the form

```
(n1:Type1)-[exp]->(n2:Type2)
```

where n1 and n2 are node variables, Type1 and Type2 are node types and exp describes the relationship between n1 and n2.[2] Such an expression denotes a path satisfying the form exp going from a node of type Type1 to a node of type Type2. A simple path expression could be e:lab, where e is an edge variable and lab is a label of $\mathcal{E}$, denoting that an edge labeled by e goes from n1 to n2. All the arguments are individually optional, so the merest form of an expression is ()-[]->(), denoting a path made of two nodes connected by any edge. Additional constraints, like conditions over node and edge variables, may be expressed on node and edge variables in a WHERE clause.

EXAMPLE 6.3.– *Query 6.1 is an example of a Cypher graph pattern query.*

```
1 MATCH
2    (ar1:Article)-[:part_of]->()-[:series]->(s1),
3    (ar2:Article)-[:part_of]->()-[series]->(s2),
4    (ar1)-[:creator]->(au1:Author),
5    (ar2)-[:creator]->(au1:Author)
6 WHERE s1.name='WWW' AND s2.name='Pods' AND ar2.year>2010
7 RETURN au1, ar1, ar2
```

Query 6.1. *Pattern of Figure 6.2, expressed in the Cypher formalism*

Lines 1–6 of Query 6.1 including the MATCH *and* WHERE *clauses constitute the graph pattern. This pattern is the one graphically represented in Figure 6.2. The* RETURN *clause of Line 7 makes it possible to define which parts of the pattern should be returned (nodes, relationships or their properties) and the matches of the node variables* au1, ar1 *and* ar2 *here.*

Figure 6.4 is a screenshot of the Rabbithole Console [RAB] which makes it possible to query, using the Cypher language, a Neo4j data graph. This figure shows the result of Query 6.1 applied to the data graph of Figure 6.1. The answers are the instantiations of the variables au1, ar1 *and* ar2 *in the data subgraphs that match the graph pattern defined in Lines 1–6 (these subgraphs are given in Figure 6.3).*

2 Regular expressions are rather simple in version 3.0 of Cypher but one can easily imagine a future extension of the language that makes it possible to define more complex expressions.

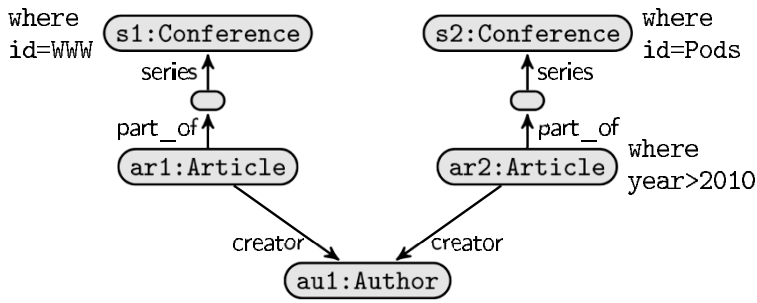


Figure 6.2. *Example of graph pattern $\mathcal{P}$*

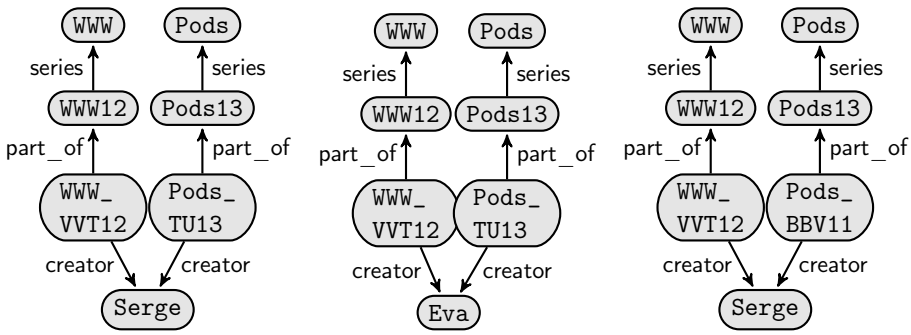
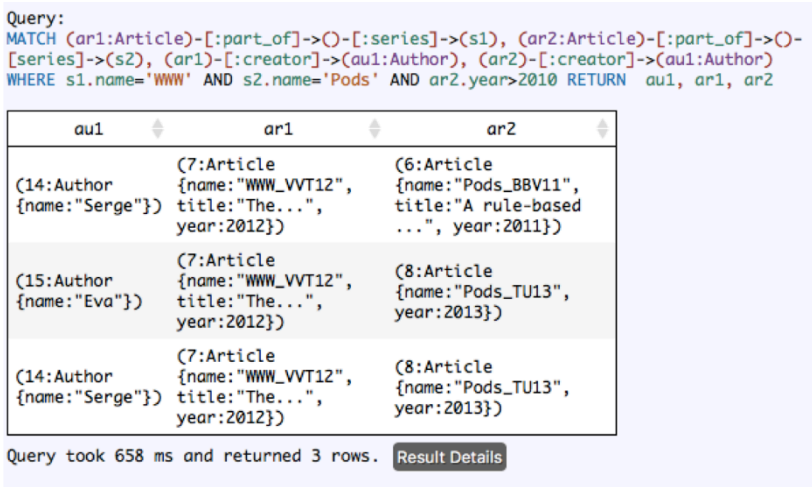


Figure 6.3. *Subgraphs of $\mathcal{DB}$ matching pattern of Figure 6.2*

More generally, Cypher offers some more evolved functionalities, for instance, using the following clauses: the `WITH` clause allows query parts to be chained together, piping the results from the one to be used as starting points or criteria in the next; the `UNWIND` clause expands a list into a sequence of rows; the `MERGE` clause ensures that a pattern exists in the graph or creates it if it does not exist; the `CREATE` clause creates nodes and relationships; the `SET` clause updates labels on nodes and properties on nodes and relationships; the `DELETE` clause removes graph elements (nodes, relationships and paths) and the `REMOVE` clause removes properties and labels from graph elements.

We introduced here only some of the features of the Cypher language. For a comprehensive presentation of Cypher, the reader is referred to [NEO 13, CYP].



```
Query:
MATCH (ar1:Article)-[:part_of]->()-[:series]->(s1), (ar2:Article)-[:part_of]->()-
[series]->(s2), (ar1)-[:creator]->(au1:Author), (ar2)-[:creator]->(au1:Author)
WHERE s1.name='WWW' AND s2.name='Pods' AND ar2.year>2010 RETURN  au1, ar1, ar2
```

au1	ar1	ar2
(14:Author {name:"Serge"})	(7:Article {name:"WWW_VVT12", title:"The...", year:2012})	(6:Article {name:"Pods_BBV11", title:"A rule-based ...", year:2011})
(15:Author {name:"Eva"})	(7:Article {name:"WWW_VVT12", title:"The...", year:2012})	(8:Article {name:"Pods_TU13", year:2013})
(14:Author {name:"Serge"})	(7:Article {name:"WWW_VVT12", title:"The...", year:2012})	(8:Article {name:"Pods_TU13", year:2013})

Query took 658 ms and returned 3 rows. Result Details

Figure 6.4. *Answers of Query 6.2 applied to the data graph of Figure 6.1 (screenshot of the Rabbithole Neo4j Console)*

6.2.2. *Fuzzy set theory*

Fuzzy set theory was introduced by Zadeh [ZAD 65] for modeling classes or sets whose boundaries are not clear-cut. For such objects, the transition between full membership and full mismatch is gradual rather than quick. Typical examples of such fuzzy classes are those described using adjectives of the natural language, such as *young, cheap, recent* and *fast*. Formally, a fuzzy set F on a referential U is characterized by a membership function $\mu_F : U \rightarrow [0,\ 1]$, where $\mu_F(u)$ denotes the grade of membership of u in F. In particular, $\mu_F(u) = 1$ reflects full membership of u in F, whereas $\mu_F(u) = 0$ expresses absolute non-membership. The condition $0 < \mu_F(u) < 1$ indicates partial membership. The following two crisp sets are of particular interest when defining a fuzzy set F: the core $C(F) = \{u \in U \mid \mu_F(u) = 1\}$, which gathers the *prototypes* of F, and the support $S(F) = \{u \in U \mid \mu_F(u) > 0\}$.

In practice, the membership function associated with F is often of a trapezoidal shape. Then, F is expressed by the quadruplet $(A,\ B,\ a,\ b)$, where $C(F) = [A,\ B]$ and $S(F) = [A - a,\ B + b]$, as depicted in Figure 6.5. Following this shape, fuzzy ascending or decreasing membership functions

may be defined in order to define fuzzy terms like *short* in Figure 6.6 or *recent* in Figure 6.7.

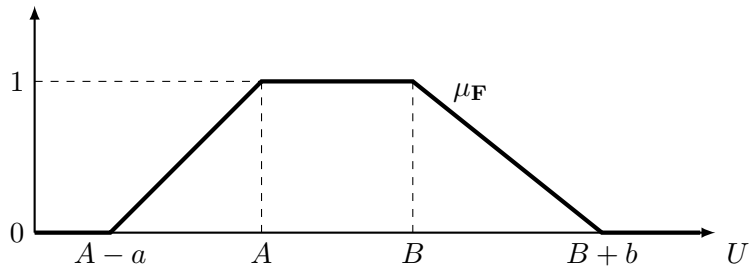


Figure 6.5. *Typical trapezoidal membership function*

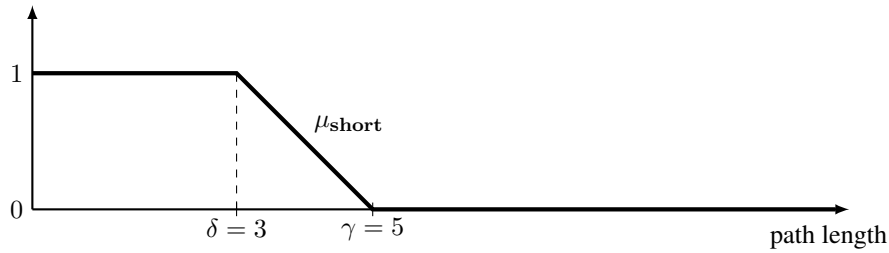


Figure 6.6. *Representation of the fuzzy term* $short$

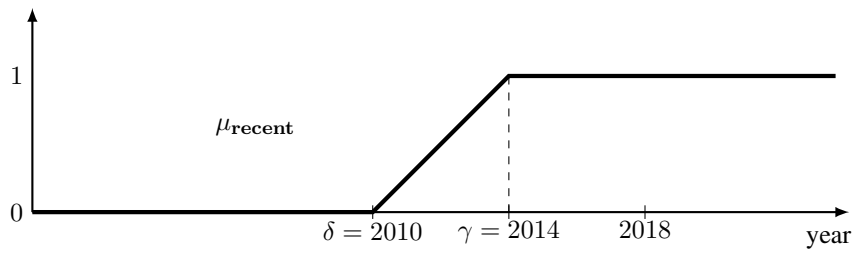


Figure 6.7. *Representation of the fuzzy term* $recent$

The α-cut of a fuzzy set F, denoted by F^{α}, is an ordinary set of elements whose degree of satisfaction is at least equal to α: $F^{\alpha} = \{u \in U | \mu_F(u) \geq \alpha\}$.

Thus, $C(F)$ and $S(F)$ are two particular α-cuts of F, where α = 1 and 0^+, respectively.

Let F and G be two fuzzy sets on the universe U, such that $F \subseteq G$ iff $\mu_F(u) \leq \mu_G(u), \forall u \in U$. The complement of F, denoted by F^c, is defined by $\mu_{F^c}(u) = 1 - \mu_F(u)$. Furthermore, $F \cap G$ (resp. $F \cup G$) is defined as follows: $\mu_{F \cap G}(u) = \min(\mu_F(u), \mu_G(u))$ (resp. $\mu_{F \cup G}(u) = \max(\mu_F(u), \mu_G(u))$). As usual, the logical counterparts of the theoretical set operators $\cap$, $\cup$ and the complementation operator correspond to the conjunction $\wedge$, disjunction $\vee$ and negation $\neg$ (see [DUB 00] for more details).

6.3. Fuzzy preference queries over graph databases

In the following, we present two scientific contributions that introduce fuzzy preferences in pattern queries over graph databases. In section 6.3.1. We present a pioneer work that extends the Cypher language in order to express fuzzy preference queries over attributes (a.k.a. properties), nodes and relationships in order to query a *crisp* graph database. Then, in section 6.3.2. We present a more general framework that makes it possible to express fuzzy preference pattern queries in order to query a fuzzy graph database.

6.3.1. *Fuzzy preference queries over crisp graph databases*

In this section, we address fuzzy selection (READ) queries over regular NoSQL Neo4j graph databases. We introduce a new query language built on top of Cypher to address flexible queries, which we call CYPHERF. We claim that fuzziness can be handled at the following three levels: over attributes, over nodes and over relationships.

The CYPHERF extension affects the `MATCH`, `WHERE` and `RETURN` clauses from Cypher by allowing for the application of fuzzy terms to attributes, nodes and relationships occurring in the query graph pattern, some examples of which are given below.

6.3.1.1. *Fuzzy conditions over attributes*

Fuzzy queries over attributes are defined by linguistic terms corresponding to fuzzy sets and fuzzy comparators (e.g. approximately equal and much

greater than). The queries are different depending on whether the attributes being addressed are linked to a node or a relationship.

In the WHERE clause, it is possible to search for *recent* papers in some databases, where *recent* is the fuzzy term defined in Figure 6.7 (for the sake of simplicity, the fuzzy labels and membership functions are denoted by the same words hereafter). Then, the answers may be ordered by recentness over this attribute by specifying it in the ORDER BY clause. The RECENT function takes a numerical value as input and outputs a numerical value ranging $[0, 1]$.

Syntactically, the CYPHERF language makes it possible to apply fuzzy conditions to the attributes that appear in the WHERE, RETURN and ORDER BY clauses.

EXAMPLE 6.4.– *Query 6.2 retrieves recent papers, ordered by recentness.*

```
1 MATCH (art:Article)
2 WHERE RECENT(art.year) > 0
3 RETURN art
4 ORDER BY RECENT(art.year) DESC
```

Query 6.2. *Query retrieving recent papers ordered by recentness*

The answer to this query applied to the data graph of Figure 6.1 is given in Figure 6.8.

art
(8:Article {name:"Pods_TU13", year:2013})
(9:Article {name:"Pods_B13", year:2013})
(7:Article {name:"WWW_VVT12", title:"The...", year:2012})
(6:Article {name:"Pods_BBV11", title:"A rule-based ...", year:2011})

Query took 1 ms and returned 4 rows. Result Details

Figure 6.8. *Answers of Query 6.2 applied to the data graph of Figure 6.1*

It is also possible to integrate fuzzy labels in the MATCH clause.

EXAMPLE 6.5.– *Query 6.3 allows us to retrieve authors of recent articles, ordered by the recentness of the article.*

```
1 MATCH (art:Article {RECENT(art.year)>0})-[: creator]->(a:Author)
2 RETURN art, a
3 ORDER BY RECENT(art.year) DESC
```

Query 6.3. *Authors of recent papers*

In the RETURN clause, no selection will be operated, but fuzzy labels can be added in order to show the users the degree to which some values match fuzzy sets.

EXAMPLE 6.6.– *Query 6.4 allows us to retrieve authors of recent articles, ordered by the recentness of the article.*

```
1 MATCH (art:Article)-[:creator]->(a:Author)
2 RETURN art, a, RECENT(art.year) AS 'Satisfaction_degree'
3 ORDER BY Satisfaction_degree DESC
```

Query 6.4. *Fuzziness in the RETURN clause*

The answer to this query applied to the data graph of Figure 6.1 is given in Figure 6.9.

When considering graph data, graphical representations are also of great interest for helping the user to understand and analyze data. For instance, Figure 6.10 shows how the result of Query 6.4 (authors of recent articles) is displayed in the Rabbithole Console [RAB], overprinted on the table that displays the answers to this query. The nodes that belong to the answers to the query appear in red (the blue nodes belong to the initial graph but do not belong to the answers). Here, the visualization exhibits two cliques, which form the network associated with the elements of the answers. This example demonstrates the interest of a graphical display.

ar1	au1	Satisfaction_degree
(8:Article {name:"Pods_TU13", year:2013})	(14:Author {name:"Serge"})	0.75
(8:Article {name:"Pods_TU13", year:2013})	(15:Author {name:"Eva"})	0.75
(9:Article {name:"Pods_B13", year:2013})	(16:Author {name:"Pablo"})	0.75
(7:Article {name:"WWW_VVT12", title:"The...", year:2012})	(13:Author {name:"Anna"})	0.5
(7:Article {name:"WWW_VVT12", title:"The...", year:2012})	(14:Author {name:"Serge"})	0.5
(7:Article {name:"WWW_VVT12", title:"The...", year:2012})	(15:Author {name:"Eva"})	0.5
(6:Article {name:"Pods_BBV11", title:"A rule-based ...", year:2011})	(14:Author {name:"Serge"})	0.25

Query took 3 ms and returned 7 rows. Result Details

Figure 6.9. *Answers to Query 6.2 applied to the data graph of Figure 6.1*

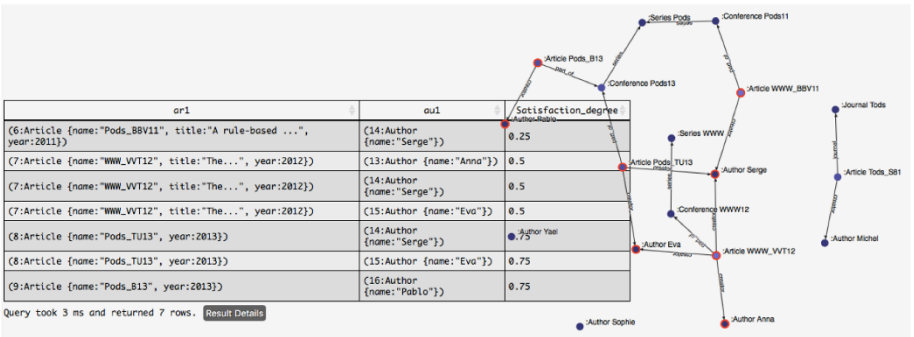


Figure 6.10. *Displaying the result of a Cypher Query. For a color version of this figure, see www.iste.co.uk/pivert/nosql.zip*

It would thus be interesting to investigate how fuzzy queries over graphs may be displayed, showing for every object to which extents it matches the result. For this purpose, a possible solution would be to use the work on fuzzy graph representation and distorted projection from the literature as done in anamorphic maps [GRI 80].

6.3.1.2. *Fuzzy conditions over nodes*

Dealing with fuzzy queries over nodes makes it possible to retrieve similar nodes.

Syntactically, the CYPHERF language allows us to apply fuzzy conditions to the attributes that appear in the `WHERE` clause, the `RETURN` clause or the `ORDER BY` clause. For instance, it is possible to retrieve similar hotels, as shown in Query 6.5.

```
1 MATCH (a1:Author),(a2:Author)
2 WITH a1 AS Aut1, a2 AS Aut2, SimilarTo(Aut1,Aut2) AS sim
3 WHERE sim > 0.7
4 RETURN Aut1,Aut2,sim
```

Query 6.5. *Getting Similar Authors*

In this framework, the link between nodes is based on the definition of measures between the descriptions. Such measures merge the several attributes that the nodes embed. Similarity measures may, for instance, be used and authors may all the more be considered as their ages and common research topics are similar.

6.3.1.3. *Fuzzy conditions over relationships*

As for nodes, such queries may be based not only on attributes but also the graph structure.

In Cypher, the structure of the pattern being searched is mostly defined in the `MATCH` clause. The first attempt to extend pattern matching to fuzzy pattern matching is to consider chains and depth matching. Chains are defined in Cypher in the `MATCH` clause with consecutive links between objects. If a node a is linked to an object b at depth 2, then the pattern is written as $(a) - [*2] - > (b)$. If a link between a and b without regard to the depth in-between is searched for, then it is written $(a) - () - > (b)$. The mechanism also applies when searching for objects linked through a range of nodes (e.g. between 3 and 5): $(a) - [*3..5] - > (b)$.

We now introduce fuzzy descriptors to define extended patterns, where depth is imprecisely described. It is then possible to search, for example, for customers linked through *almost 3* hops. The syntax $**$ indicates a fuzzy linker, as illustrated by Query 6.6.

```
MATCH (a1:Author)-[:KNOWS**almost3]->(a2:Author)
RETURN a1,a2
```

Query 6.6. *Fuzzy Patterns*

This is related to fuzzy tree and graph mining [LÓP 09], where some patterns emerge from several graphs even if they do not occur in exactly the same way everywhere regarding the structure.

Another possibility is to not consider chains but patterns where several links from and to nodes exist.

In the current example, *common* journals may, for instance, be considered as common when they are chosen by many people. This is similar to the way famous people are detected if they are followed by many people on social networks. In this example, a conference is popular if a large proportion of authors submit papers to it. In Cypher, such queries are defined by using aggregators. For instance, Query 6.7 retrieves articles created by at least two persons.

```
MATCH (art:Article)-[:CREATED_BY]->(a:Author)
WITH a AS Aut, count(*) AS cpt
WHERE cpt>1
RETURN Aut
```

Query 6.7. *Aggregation*

Such crisp queries can be extended to consider fuzziness, as proposed in Query 6.8.

```
MATCH (art:Article)-[:CREATED_BY]->(a:Author)
WITH a AS Aut, count(*) AS cpt
WHERE COMMON(cpt) > 0
RETURN Aut
```

Query 6.8. *Aggregation*

All fuzzy clauses described in this section can be combined using conjunctive and disjunctive operators. The question that rises is to implement them in the existing Neo4j engine, which is further discussed in section 6.4.

6.3.2. *Fuzzy preference queries over fuzzy graph databases*

By nature, data may express gradual information (rather than Boolean information). Friendship, influence and collaboration are examples of gradual relations that should be associated with a degree quantifying the extent to which a relation holds between two resources. The crisp graph model may then be extended into the notion of a *fuzzy data graph,* where a degree may be attached to edges in order to express the "intensity" of any kind of gradual relationship (e.g. *likes*, *is friends with* and *is about*). In [PIV 14, PIV 15], the authors considered a more general framework for introducing preferences in graph pattern queries over fuzzy graph data, where preferences may concern the data or the structure of the graph. After describing the formal concept of the fuzzy graph and exhibiting relevant features for fuzzy preferences over such a model (sections 6.3.2.1 and 6.3.2.2 respectively), we introduce this framework, which includes a theoretical contribution defining the concepts considered (presented in section 6.3.2.3) and an instantiation taking the form of an extension of the Cypher language (presented in section 6.3.2.4).

6.3.2.1. *Fuzzy graphs*

In its basic form, a *graph* is a pair (V, R), where V is a set and R is a relation on V. The elements of V (resp. R) correspond to the vertices (resp. edges) of the graph. Similarly, any fuzzy relation ρ on a set V can be regarded as defining a weighted graph or fuzzy graph [ROS 75], where the edge $(x,\ y) \in V \times V$ has weight or strength $\rho(x,\ y) \in [0,\ 1]$.

As noted in [YAG 13], the fuzzy relation ρ may be viewed as a fuzzy subset on $V \times V$, which allows us to use much of the formalism of fuzzy sets. For example, we can say that $\rho_1 \subseteq \rho_2$ if $\forall(x,\ y),\ \rho_1(x,\ y) \leq \rho_2(x,\ y)$. Some notable properties that can be associated with fuzzy relations are reflexivity ($\rho(x,\ x) = 1,\ \forall x$), symmetry ($\rho(x,\ y) = \rho(y,\ x)$) and transitivity ($\rho(x,\ z) \geq \max_y \min(\rho(x,\ y),\ \rho(y,\ z))$).

An important operation on fuzzy relations is composition. We assume that ρ_1 and ρ_2 are two fuzzy relations on V. Thus, composition $\rho = \rho_1 \circ \rho_2$

is also a fuzzy relation on V s.t. $\rho(x, z) = \max_y \min(\rho_1(x, y), \rho_2(y, z))$. The composition operation can be shown to be associative: $(\rho_1 \circ \rho_2) \circ \rho_3 = \rho_1 \circ (\rho_2 \circ \rho_3)$. The associativity property allows us to use the notation $\rho^k = \rho \circ \rho \circ \ldots \circ \rho$ for the composition of ρ with itself $k - 1$ times. In addition, following [YAG 13], we define ρ^0 to be s. t. $\rho^0(x, y) = 0, \forall(x, y)$.

Fuzzy graphs as defined above may be generalized to the case where a fuzzy set of vertices is considered. Then, denoting by F the fuzzy subset of V considered, the corresponding fuzzy graph is defined as (V, F, ρ_F). In this case, we let ρ_F be a relation on V defined as $\rho_F(x, y) = \min(\rho(x, y), \mu_F(x), \mu_F(y))$, where μ_F denotes the membership function attached to F. In the following, we only consider the simple case of a crisp set of vertices.

If ρ is symmetric, (V, ρ) is said to be an undirected graph. Otherwise, we shall refer to (V, ρ) as a directed graph. Without loss of generality, we consider directed graphs in the following.

6.3.2.2. *Relevant features for fuzzy preferences over fuzzy graphs*

We now describe the main elements that may appear relevant in a fuzzy query addressed to a graph database. The following two types of preferences should be considered: those on content and those on structure.

Preferences on the node content. The idea is to express flexible conditions about attributes associated with nodes and/or vertices of the graph. An example is: "find the people who are *young*, *highly educated* and live in *Eastern Europe*" (assuming that each node contains information about the age, education level, address, etc., of the person it corresponds to). Compound conditions may also be expressed using a large range of fuzzy connectives. We do not get into more detail as this aspect has been studied in depth in section 6.3.1 whose concepts can easily be extended to fuzzy graphs.

Preferences on the graph structure. Hereafter, we describe the concepts of fuzzy graph theory that appear the most relevant in a perspective of graph database querying. We denote a *fuzzy graph* by $G = (V, \rho)$.

Strength of a path. A path p in G is a sequence $x_0 \rightarrow x_1 \rightarrow \ldots \rightarrow x_n$ $(n \geq 0)$ s. t. $\rho(x_{i-1}, x_i) > 0$, $1 \leq i \leq n$ where n is the number of links in the path. The *strength* of the path is defined as:

$$ST(p) = \min_{i=1..n} \rho(x_{i-1}, x_i). \qquad [6.1]$$

In other words, the strength of a path is defined as the weight of the weakest edge of the path. Two nodes for which there exists a path p with $ST(p) > 0$ between them are called *connected*. It is possible to show that $\rho^k(x, y)$ is the strength of the strongest path from x to y containing at most k links. Thus, the strength of the strongest path joining any two vertices x and y (using any number of links) may be denoted by $\rho^\infty(x, y)$.

Length and *distance*. The *length* of a path $p = x_0 \rightarrow x_1 \rightarrow \ldots \rightarrow x_n$ in the sense of ρ is a concept defined in [ROS 75] as:

$$Length(p) = \sum_{i=1}^{n} \frac{1}{\rho(x_{i-1}, x_i)}. \quad [6.2]$$

Clearly, $Length(p) \geq n$ (it is equal to n if ρ is Boolean, in particular if G is a crisp graph). We can then define the *distance* between two nodes x and y in G as:

$$\delta(x, y) = \min_{all\ paths\ x\ to\ y} Length(p). \quad [6.3]$$

It is the length of the shortest path from x to y. It can be shown that δ is a metric [ROS 75].

α-cut of a relation. This is defined as $\rho^\alpha = \{(x, y) \mid \rho(x, y) \geq \alpha\}$ where $\alpha \in]0, 1]$. We note that ρ^α is a crisp relation.

Preference combination. Different types of connectives may be considered for combining conditions about the content or the structure of the graph: "flat" (min, max, arithmetic mean, etc.), weighted (weighted mean, OWA, quantified proposition, etc.; see [FOD 00]) or hierarchical.

In [YAG 13], different types of fuzzy preference criteria that appear relevant in the context of graph databases are discussed, without getting into the detail of how to express them using a formal query language.

6.3.2.3. *Fuzzy preference queries over fuzzy graph databases*

The only contributions that extend a graph pattern query language in order to express fuzzy preferences over the structure of a fuzzy graph are [PIV 14, PIV 15, PIV 16b]. Let us recall again here that a crisp graph is a special case of fuzzy graph, meaning that the query extension proposed in these works can also be used for querying a crisp graph.

We are interested in fuzzy graph databases, where nodes and edges can carry data (e.g. key–value pairs in attributed graphs; see section 6.2.1). In [PIV 15], the authors proposed an extension of the definition of a *fuzzy graph* into that of a *fuzzy data graph* (which can also be seen as a fuzzy extension of Definition 6.1).

DEFINITION 6.2 (Fuzzy data graph).– *Assuming the existence of a set $\mathcal{E}$ of labels, a* fuzzy data graph *G is a quadruple (V, R, κ, ζ), where V is a finite set of nodes (each node n is identified by $n.id$), $R = \bigcup_{e \in E} \{\rho_e : V \times V \rightarrow [0, 1]\}$ is a set of labeled fuzzy edges between nodes of V and κ (resp. ζ) is a function assigning an attributed value to nodes (resp. edges) of G.*

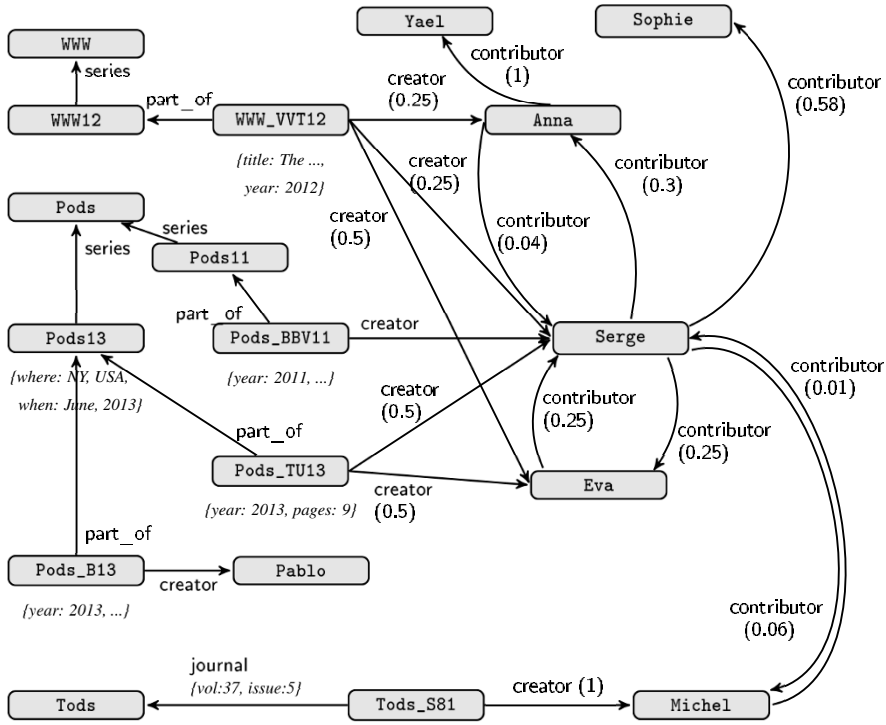


Figure 6.11. *A fuzzy data graph $\mathcal{FDB}$ inspired from DBLP*

EXAMPLE 6.7.– *Figure 6.11 is an example of a fuzzy data graph. This fuzzy data graph is an extension of the data graph of Figure 6.1, with some fuzzy edges (with a degree in brackets). In this example, the degree associated*

with `A-contributor->` `B` *denotes the extent to which author* `A` *contributed to the works of author* `B`*, calculated by the proportion of journal papers co-written by* `A` *and* `B` *in the total number of journal papers written by* `B`*. The degree associated with* `A-creator->` `B` *denotes the extent to which author* `A` *participated in the publication of work by author* `B`*. This degree may be given by the authors themselves. Crisp edges are special cases of fuzzy edges for which the degree is equal to* `1` *(this is considered as being the default degree value associated with edges).*

We now move to the definition of queries in which fuzzy preferences may appear. To this aim, we define the notion of *fuzzy graph pattern* [PIV 14] applied to a possibly *fuzzy* data graph. This notion is an extension of the crisp *graph pattern* notion [FAN 12] shown to have good properties for practical implementation.

The notion of *fuzzy regular expression*, which extends the notion of regular expression, should be introduced first. Definition 6.3 defines its syntax, and Definition 6.4 defines its interpretation.

DEFINITION 6.3 (Fuzzy regular expression).– *A* fuzzy regular expression *is an expression of the form:*

$$F ::= e \mid F{\cdot}F \mid F \cup F \mid F^* \mid F^{Cond}$$

where:

– $e \in \mathcal{E} \cup \{\_\}$ *denotes an edge labeled by* e*, with the wildcard symbol (underscore) denoting any label in* $\mathcal{E}$*;*

– $F{\cdot}F$ *denotes a concatenation of expressions;*

– $F \cup F$ *denotes alternative expressions;*

– F^* *denotes the repetition of an expression;*

– F^{Cond} *denotes paths* p *satisfying* F *and the condition* $Cond$*, which is a Boolean combination of atomic formulas of the form* `Prop is Fterm`*, where* `Prop` *is a property defined on* p *and* `Fterm` *denotes a predefined or user-defined fuzzy term like* `short` *(see Figure 6.6 (resp. Figure 6.7), which gives a membership function associated with the fuzzy term* short *(resp.* recent*)).*

In the following, we limit properties to $\{\texttt{ST}, \texttt{Length}\}$ *denoting resp.* $ST(p)$ *(see equation [6.1]) and* $Length(p)$ *(see equation [6.2]). Examples of conditions of this form are* `Length is short` *and* `ST is strong`*. We note*

that Boolean conditions of the form `Prop op a`*, where* `a` *is a constant and* `op` *is a crisp comparator, are just a special case.*

In the following, giving a fuzzy regular expression f, f^+ is a shortcut notation for $f \cdot f^*$, f^k stands for $f \cdot f \cdots f$ with k occurrences of f and $f^{n,m}$ is a shortcut for $\bigcup_{i=n}^{m} f^i$.

A fuzzy regular expression is said to be *simple* if it is of the form e, where $e \in \mathcal{E} \cup \{\_\}$ (it denotes a *single* edge).

DEFINITION 6.4 (Fuzzy regular expression matching).– *Given a path p and a fuzzy regular expression exp, p* matches *exp* with a satisfaction degree of $\mu_{exp}(p)$ *defined as follows, according to the form of exp (in the following, f, f_1 and f_2 are fuzzy regular expressions):*

– exp is of the form e with $e \in \mathcal{E}$ (resp. "_"). If p is of the form $v_1 \xrightarrow{e'} v_1'$, where $e' = e$ (resp. where $e' \in \mathcal{E}$), then $\mu_{exp}(p) = 1$ else $\mu_{exp}(p) = 0$.

– exp is of the form $f_1 \cdot f_2$. Let P be the set of all pairs of paths (p_1, p_2) s.t. p is of the form $p_1 p_2$. We have: $\mu_{exp}(p) = \max_P(min(\mu_{f_1}(p_1), \mu_{f_2}(p_2)))$.

– exp is of the form $f_1 \cup f_2$. We have: $\mu_{exp}(p) = \max(\mu_{f_1}(p), \mu_{f_2}(p))$.

– exp is of the form f^*. If p is an empty path, then $\mu_{exp}(p) = 1$. Otherwise, we denote by P the set of all tuples of paths $(p_1, \cdots, p_n)$ $(n > 0)$ s.t. p is of the form $p_1 \cdots p_n$. We have: $\mu_{exp}(p) = \max_P(\min_{i \in [1..n]}(\mu_f(p_i)))$.

– exp is of the form f^{Cond}, where $Cond$ is a (possibly compound) fuzzy condition. We have: $\mu_{exp}(p) = \min(\mu_f(p), \mu_{Cond}(p))$, where $\mu_{Cond}(p)$ is the degree of satisfaction of $Cond$ by p.

Not matching is equivalent to matching with a degree 0.

EXAMPLE 6.8.– *Figure 6.12 represents some paths from the fuzzy data graph depicted in Figure 6.11 that somewhat match the following fuzzy regular expressions:*

– $e_1 = creator \cdot contributor^+$ is a fuzzy regular expression. All paths p_i ($i \in [1..4]$) of Figure 6.12 match e_1 with a satisfaction degree of $\mu_{e_1}(p_i) = 1$.

– $e_2 = (creator \cdot contributor^+)^{\texttt{ST>0.4}}$ is a fuzzy regular expression. Path p_4 is the only one of Figure 6.12 that matches e_2 (as $ST(p_1) = 0.3$, $ST(p_2) = 0.3$, $ST(p_3) = 0.01$ and $ST(p_4) = 0.58$), with $\mu_{e_2}(p_4) = 1$.

– $e_3 = creator \cdot (contributor^+)^{\texttt{Length is short}}$, *where short is the fuzzy term of Figure 6.6, is a fuzzy regular expression. Paths* p_1, p_2 *and* p_4 *of Figure 6.12 match* e_3 *with* $\mu_{e_3}(p_1) = 0.83$ *as* $\mu_{short}(1/0.3) = 0.83$ *(where* $1/0.3$ *is the length of path from* `Serge` *to* `Anna`*),* $\mu_{e_3}(p_2) = 0.67$ *as* $\mu_{short}(1/0.3 + 1) = 0.67$ *(where* $1/0.67$ *is the length of the short path from* `Serge` *to* `Yael`*) and* $\mu_{e_3}(p_4) = 1$ *as* $\mu_{short}(1/0.58) = 1$. *Path* p_3 *does not match* e_3 *as* $\mu_{short}(1/0.01) = 0$. ⋄

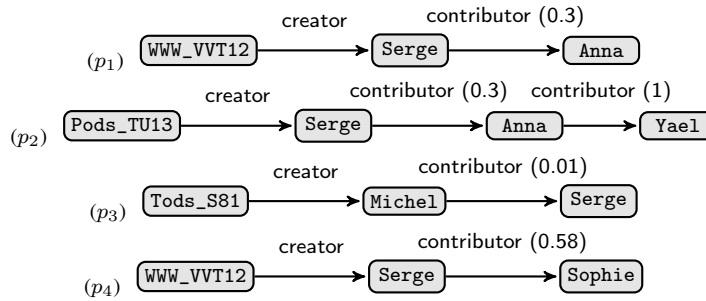


Figure 6.12. *Fuzzy regular expression matching*

We now introduce the notion of a *fuzzy graph pattern*, which is a directed crisp graph with conditions on nodes and edges, types on nodes and where edges are labeled by fuzzy regular expressions that denote paths. Definition 6.5 defines the syntax of a fuzzy graph pattern, and Definition 6.6 defines its interpretation.

DEFINITION 6.5 (Fuzzy graph pattern).– *Let* $\mathcal{F}$ *be a set of fuzzy terms. A* fuzzy graph pattern *is defined as a sextuple:* $\mathcal{P} = (V_\mathcal{P}, E_\mathcal{P}, f_e^{path}, f_n^{cond}, f_e^{cond}, f_n^{type})$, *where*

– $V_\mathcal{P}$ *is a finite set of nodes;*

– $E_\mathcal{P} \subseteq V_\mathcal{P} \times V_\mathcal{P}$ *is a finite set of edges, where* (u, u') *denotes an edge from* u *to* u'*;*

– f_e^{path} *is a function defined on* $E_\mathcal{P}$ *s. t. for each* (u, u') *in* $E_\mathcal{P}$, $f_e^{path}(u, u')$ *is a fuzzy regular expression;*

– f_n^{cond} *is a function defined on* $V_\mathcal{P}$ *s. t. for each node* u, $f_n^{cond}(u)$ *is a condition on attributes of* u*, defined as a combination of atomic formulas of the form* `A is Fterm`*, where* `A` *denotes an attribute and* `Fterm` *denotes a fuzzy term (e.g.* `year is recent`*).*

– f_e^{cond} *is the counterpart of* f_n^{cond} *for edges. For each* (u, u') *in* $E_{\mathcal{P}}$ *for which* $f_e^{path}(u, u')$ *is simple,* f_e^{cond} *is the condition on attributes of* (u, u')*; and*

– f_n^{type} *is a function defined on* $V_{\mathcal{P}}$ *s. t. for each node* u*,* $f_n^{type}(u)$ *is the type of* u*.*

In the following, we adopt a *Cypher* syntax for graph pattern representation.

EXAMPLE 6.9.– *We denote by* $\mathcal{P}$ *the fuzzy graph pattern below.*

```
MATCH
  (ar1:Article)-[part_of.series]->(s1),
  (ar2:Article)-[part_of.series]->(s2),
  (ar1)-[:creator]->(au1:Author),
  (ar2)-[:creator]->(au1:Author),
  (au1)-[(contributor+)|Length IS short]->(au2:Author)
WHERE s1.id=WWW AND s2.id=Pods AND ar2.year is recent.
```

Query 6.9. *Pattern expressed in Cypher*

Figure 6.13 is a graphical representation of pattern $\mathcal{P}$*, where the dashed edge denotes a path and the information in italic font denotes a node type or an additional condition on node or edge attributes. This pattern "models" information concerning authors (*`au2`*) who have, among their close contributors, an author (*`au1`*) who published a paper (*`ar1`*) in* `WWW` *as well as (*`ar2`*) in* `Pods` *recently (*`ar2.year is recent`*).* ◇

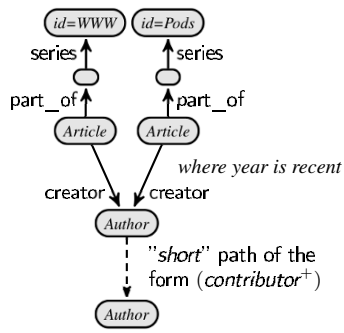


Figure 6.13. *Pattern* $\mathcal{P}$

DEFINITION 6.6 (Fuzzy graph pattern matching).– *A fuzzy data graph* $G = (V, R, \kappa, \zeta)$ matches *a fuzzy graph pattern* $\mathcal{P} = (V_\mathcal{P}, E_\mathcal{P}, f_e^{path}, f_n^{cond}, f_e^{cond}, f_n^{type})$ *with a satisfaction degree denoted by* $\mu_\mathcal{P}(G)$ *if there exists a binary relation* $S \subseteq V_\mathcal{P} \times V$ *representing an injective function from* $V_\mathcal{P}$ *to* V *such that (1) for each node* $u \in V_\mathcal{P}$*, there exists a node* $v \in V$ *s. t.* $(u, v) \in S$*; (2) for each edge* $(u, u') \in E_\mathcal{P}$*, there exist two nodes* v *and* v' *of* V *s. t.* $\{(u, v), (u', v')\} \subseteq S$ *and there is a path* p *in* G *from* v *to* v' *s. t.* p *matches* $f_e^{path}(u, u')$ *(we recall that in the case of matching, a satisfaction degree is associated, see Definition 6.4); (3) for each pair* $(u, v) \in S$*,* $\kappa(v) \vdash f_n^{cond}(u)$ *(the semantics of* $\vdash$ *is clear from the context here) and* $f_n^{type}(u) = Type(v)$ *and (4) the same reasoning is trivially applied to conditions on attributes for edges labeled with a simple fuzzy regular expression in* $E_\mathcal{P}$*, that is,* $\zeta(v, v') \vdash f_e^{cond}(u, u')$.

The value of $\mu_\mathcal{P}(G)$ *is the minimum of the satisfaction degrees produced by the mappings and conditions from (2), (3) and (4). If there is no relation* S *satisfying the previous conditions, then* $\mu_\mathcal{P}(G) = 0$*, that is,* G *does not match* $\mathcal{P}$.

EXAMPLE 6.10.– *Figure 6.14 gives the set of subgraphs of* $\mathcal{FDB}$ *matching the pattern* $\mathcal{P}$ *of Example 6.9. We note for the following that* $\mu_{recent}(2011) = 0.25$*,* $\mu_{recent}(2013) = 0.75$*,* p *is the path going from* $au1$ *to* $au2$*. Let us now consider the satisfaction degree associated with each graph of Figure 6.14. As the satisfaction degree is the minimum of the satisfaction degrees induced by Lines 5 and 8, we have:*

– $\mu_\mathcal{P}(g_1) = 0.75$ *(as* $\mu_{short}(Length(p)) = \mu_{short}(1.72) = 1$ *and* $\mu_{recent}(2013) = 0.75$*);*

– $\mu_\mathcal{P}(g_2) = 0.5$ *(as* $\mu_{short}(Length(p)) = \mu_{short}(4) = 0.5$ *and* $\mu_{recent}(2013) = 0.75$*);*

– $\mu_\mathcal{P}(g_3) = 0.33$*, (as* $\mu_{short}(Length(p)) = \mu_{short}(4.33) = 0.33$ *and* $\mu_{recent}(2013) = 0.75$*);*

– $\mu_\mathcal{P}(g_4) = 0.75$;

– $\mu_\mathcal{P}(g_5) = 0.5$*, (as* $\mu_{short}(Length(p)) = \mu_{short}(4) = 0.5$ *and* $\mu_{recent}(2013) = 0.75$*);*

– $\mu_\mathcal{P}(g_6) = 0.25$ *(as* $\mu_{short}(Length(p)) = \mu_{short}(1.72) = 1$ *and* $\mu_{recent}(2011) = 0.25$*);*

– $\mu_\mathcal{P}(g_7) = 0.25$;

– $\mu_{\mathcal{P}}(g_8) = 0.25$; *and*
– $\mu_{\mathcal{P}}(g_9) = 0.4$.$\diamond$

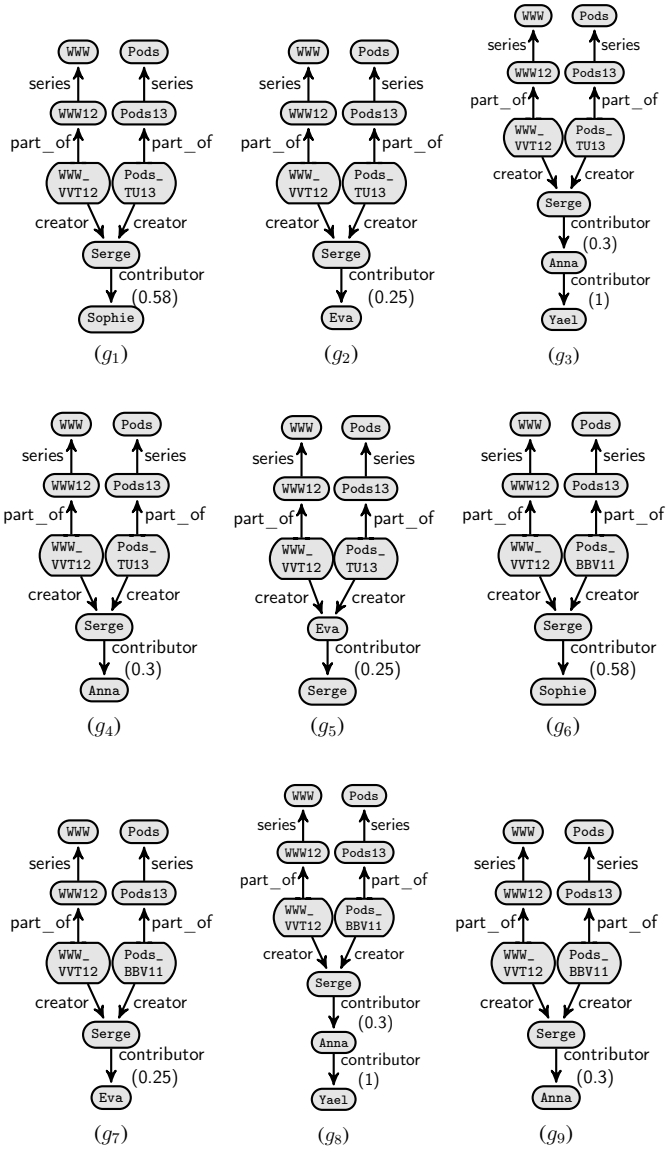


Figure 6.14. *Subgraphs of $\mathcal{FDB}$ matching $\mathcal{P}$*

6.3.2.4. *The* FUDGE *language*

The FUDGE language [PIV 14] is an extension of the Cypher language [NEO 13] that is used for querying graph databases in a crisp way in the Neo4j graph DBMS [NEO 18].

Given a fuzzy graph database $\mathcal{FDB}$, a selection query $\sigma_{\mathcal{P}}(\mathcal{DB})$ expressed in the FUDGE language is composed of the following:

1) A list of `DEFINE` clauses for fuzzy term declarations. If a fuzzy term `fterm` corresponds to a trapezoidal function with the four positions (abscissa) `A-a`, `A`, `B` and `B+b`, then the clause has the form `DEFINE fterm as (A-a,A,B,B+b)`. If `fterm` is a decreasing function like the term *short* of Figure 6.6, then the clause has the form `DEFINEDESC fterm as` (γ,δ) (there is the corresponding `DEFINEDESC` clause for increasing functions, like the term *recent* of Figure 6.7).

2) A `MATCH` clause of the form `MATCH pattern WHERE conditions`, where `pattern` denotes a the fuzzy graph pattern $\mathcal{P}$.

Query 6.10 is an example of a FUDGE query. The `DEFINEDESC` clause defines the fuzzy term *short* of Figure 6.6, and the next clause defines the fuzzy term *recent*. The pattern defined in Lines 4–10 is the one of Example 6.9.

```
1  DEFINEDESC short AS (3,5)
2  DEFINEASC recent AS (2010,2014)
3  IN
4  MATCH
5    (ar1:Article)-[part_of]->()->[series]->(s1),
6    (ar2:Article)-[part_of]->()->[series]->(s2),
7    (ar1)-[:creator]->(au1:Author),
8    (ar2)-[:creator]->(au1:Author),
9    (au1)-[(contributor+)|Length IS short]->(au2:Author)
10 WHERE s1.id=WWW AND s2.id=Pods AND ar2.year IS recent
```

Query 6.10. *A FUDGE query*

The FUDGE language is implemented in a system called SUGAR [PIV 16a], whose implementation issues are discussed in section 6.4.

An extension of the FUDGE language with fuzzy quantifiers over the structure of the fuzzy data graph was also proposed in [PIV 16b] but is not presented here.

6.4. Implementation challenges

Concerning the implementation issues, the following three main problems arise: (1) the implementation of the fuzzy data model if fuzzy data are considered; (2) the evaluation of queries extended by fuzzy preferences; and (3) the scalability of the query evaluation, which are discussed below.

6.4.1. *Modeling fuzzy databases*

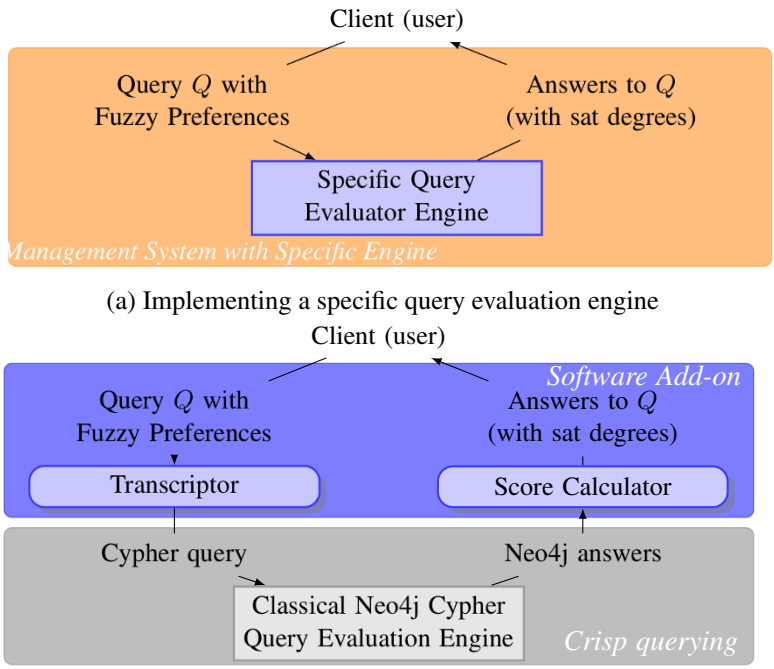
Existing graph database management systems make it possible to only model *crisp* property graph databases, so the problem of modeling *fuzzy* databases in such systems arises. In the case of the *property graph* model like that considered in Neo4j, a set of properties (key–value pairs) can be bound to a node or an edge. Properties usually denote embedded data and meta-data associated with nodes or edges. A simple mechanism may then be used to simulate fuzzy graph databases in this crisp property graph model: we only have to attach to each edge of the property graph a supplementary arbitrary property called *fdegree* carrying the degree value of the relation, supposing that *fdegree* now becomes a reserved keyword of the system.

6.4.2. *Evaluation of queries with fuzzy preferences*

Concerning the evaluation of extended queries (queries with fuzzy preferences introduced in section 6.3.1 and in section 6.3.2.3), two architectures may be thought of. A first solution consists of implementing a specific query evaluation engine inside the data management system. Figure 6.15.(a) is an illustration of this architecture. The advantage of this solution is that optimization techniques implemented directly in the query engine should make the system very efficient for query processing. An important downside is that the implementation effort is substantial, but the strongest objection for this solution is that the evaluation of an extended query in a distributed architecture would imply having an extended query evaluator engine available for each distant engine, which is a very strong architectural constraint.

An alternative, more realistic architecture consists of adding a software layer over a standard and possibly distant classical Neo4j engine. This layer is composed of the following two modules, which interact with a Neo4j crisp

engine (see Figure 6.15): the *Transcriptor* module and the *Score Calculator*. The former aims to translate a FUDGE query into a (crisp) Cypher query retrieves all needed data. This query is then sent to the crisp Neo4j engine. Then, the *Score Calculator* module extracts answers, calculates the satisfaction degree associated with each answer returned by the crisp engine and ranks the answers. Figure 6.15(b) is an illustration of this architecture.



(a) Implementing a specific query evaluation engine

(b) Adding a software layer

Figure 6.15. *Possible architectures for evaluating a query with fuzzy preferences*

As an example, the SUGAR software, which implements the FUDGE language, is based on the architecture of Figure 6.15(b). It is built from the Neo4j REPL Console Rabbithole [RAB] supporting Cypher queries. Figure 6.16 presents a screenshot of the GUI of SUGAR; this GUI is an extension of the Rabbithole one. It is composed of (1) a frame for visualizing the graph database and the results of a query; and (2) an input field frame for entering and executing a FUDGE query. SUGAR also provides logs that

trace the evaluation in an associated console, providing each intermediate result of the execution process: the crisp Cypher query obtained after the transcription stage (output of the *Transcriptor* module), the result of the crisp query evaluation (an intermediate result containing additional information needed for the score calculation) and the final result obtained after the score calculation (output of the *Score Calculator* module). The logs also provide the execution time associated with each stage of the evaluation. Figure 6.17 is an excerpt of execution logs associated with an evaluation of the FUDGE query from Query 6.10. An interesting observation concerns the *conciseness* of the FUDGE language. Indeed, the crisp Cypher query obtained after the transcription stage is very complex with respect to the FUDGE one.

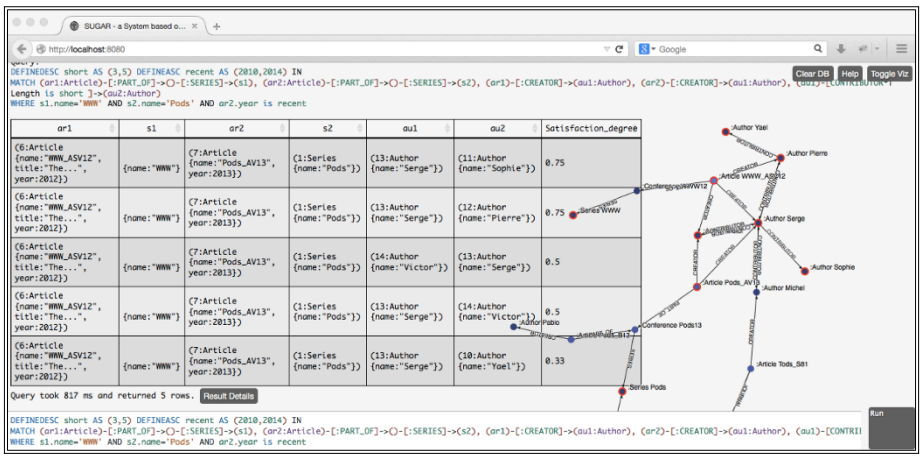


Figure 6.16. *Screenshot of the* SUGAR *system. For a color version of this figure, see www.iste.co.uk/pivert/nosql.zip*

6.4.3. *Scalability*

In [PIV 15], the authors discussed the cost of introducing flexibility in graph pattern queries. They showed that the first and third stages, which allow us to introduce flexibility in the query language in the SUGAR software based on the architecture of Figure 6.15(b), are strongly dominated in complexity by the crisp evaluation (the second stage). Therefore, introducing the flexibility as defined in the FUDGE language (see section 6.3) seems reasonable in terms of additional evaluation cost.

```
[SUGAR TRANSCRIPTOR MODULE] Generated Cypher query:
MATCH (ar1:Article)-[:PART_OF]->()-[:SERIES]->(s1),
 (ar2:Article)-[:PART_OF]->()-[:SERIES]->(s2), (ar1)-[:CREATOR]->(au1:Author),
 (ar2)-[:CREATOR]->(au1:Author), p1 = (au1)-[CONTRIBUTOR*]->(au2:Author)
WITH REDUCE(weight=0, edge IN relationships(p1)| weight + 1/edge.fdegree) AS length_au1_au2_p1,
     ar1 AS ar1, s1 AS s1, ar2 AS ar2, s2 AS s2, au1 AS au1, au2 AS au2
WHERE s1.name= WWW  AND s2.name= Pods  AND ar2.year >2010.0  AND length_au1_au2_p1<5.0
RETURN ar1,s1,ar2,s2,au1,au2,
     length_au1_au2_p1 AS calc_fuzzy_length_p1_short,
     ar2.year AS calc_fuzzy_ar2_year_recent
===> period of translating from fudge to cypher query : 1 ms <====

Intermediate result:
+------------------------------------------------------------------------------------------------------------
| ar1                                                    | s1                    | ar2                        ↩
        |
+------------------------------------------------------------------------------------------------------------
| Node[5]{name:"WWW_ASV12",title:"The...",year:2012} | Node[0]{name:"WWW"} | Node[6]{name:"Pods_AV13",year:2013} |
| Node[5]{name:"WWW_ASV12",title:"The...",year:2012} | Node[0]{name:"WWW"} | Node[6]{name:"Pods_AV13",year:2013} |
| Node[5]{name:"WWW_ASV12",title:"The...",year:2012} | Node[0]{name:"WWW"} | Node[6]{name:"Pods_AV13",year:2013} |
| Node[5]{name:"WWW_ASV12",title:"The...",year:2012} | Node[0]{name:"WWW"} | Node[6]{name:"Pods_AV13",year:2013} |
| Node[5]{name:"WWW_ASV12",title:"The...",year:2012} | Node[0]{name:"WWW"} | Node[6]{name:"Pods_AV13",year:2013} |
+------------------------------------------------------------------------------------------------------------
---------------------------------------------------------------------------------------------------------------------
| s2                      | au1                       | au2                       |calc_fuzzy_length_p1_short| ↩
       calc_fuzzy_ar2_year_recent|
---------------------------------------------------------------------------------------------------------------------
| Node[1]{name:"Pods"} | Node[12]{name:"Serge"}      | Node[13]{name:"Victor"} | 4.0                        | 2013        ↩
                  |
| Node[1]{name:"Pods"} | Node[12]{name:"Serge"}      | Node[10]{name:"Sophie"} | 1.7241379310344829         | 2013        ↩
                  |
| Node[1]{name:"Pods"} | Node[12]{name:"Serge"}      | Node[11]{name:"Pierre"} | 3.3333333333333335         | 2013        ↩
                  |
| Node[1]{name:"Pods"} | Node[12]{name:"Serge"}      | Node[9]{name:"Yael"}        | 4.333333333333334      | 2013        ↩
                  |
| Node[1]{name:"Pods"} | Node[13]{name:"Victor"} | Node[12]{name:"Serge"}  | 4.0                            | 2013        ↩
                   |
---------------------------------------------------------------------------------------------------------------------
===> period of calculating the intermediate crisp result : 145ms <===
Neo4j Engine : 5 rows 139 ms

[SUGAR SCORE CALCULATOR MODULE] Generate Result.
===> period of calculating the score :3ms <==

+++++++++++++++++++++++++++++++++ Summary ++++++++++++++++++++++++++++++ |
- FUDGE TRANSCRIPTOR  : 1 ms
- CRISP Evalutation   : 145 ms including 139 ms of the Neo4j engine
- FUDGE Calculator  : 3 ms
+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++ |
```

Figure 6.17. *Execution trace in the* SUGAR *system*

However, dealing with very large databases and complex queries requires that architectures be scalable. This may, for instance, rely on the current In-Memory architectures, as described in [CAS 17]. Such architectures rely on *Graph processing systems*, the most popular ones being used are GraphX, Giraph and GraphLab. These systems are based on ideas expressed in the Google Pregel paper [MAL 10] and on Map/Reduce concepts [DEA 08, GHE 03]. In [CAS 17], the authors showed how to exploit NoSQL Graph Databases and In-Memory architectures to extract summaries. The approach presented in [CAS 17] has been tested on databases with up to 36 million nodes and 73 million relationships.

6.5. Related work

As several models have been proposed to represent data having an implicit or explicit graph structure (see [ANG 08] for an overview of these models), the literature includes a variety of query languages for graphs. The authors of [ANG 08], [WOO 12] and [BAR 13] proposed complementary surveys of graph query languages defined in the past 25 years, including languages for querying graph-based object databases, semistructured data, social networks and Semantic Web data. The authors of [BAR 13] focused on theoretical query languages for graph databases and emphasized that graph database management systems still lack query languages with a clear syntax and semantics. This is the problem that we addressed here.

Functionalities that should be offered by a language for querying the topology of a crisp graph database are exhibited in [ANG 05, ANG 12, CIG 12, ANG 13, WOO 12]. We summarize these (non-exclusive) functionalities hereafter, focusing on selection statements of the DML part of the language. Given a graph data G, *Adjacency queries* test node adjacency, for example, check whether two nodes are adjacent, list all neighbors of a node; given a vertex, *Reachability queries* search for topologically related vertices in G, where vertices are reachable by a *fixed-length path*, a *regular simple path* or a *shortest path*; *Pattern matching queries* look for all subgraphs of G that are isomorphic to a given graph pattern and *Data queries* specify conditions on the data embedded in G. The framework that we proposed expresses some flexible adjacency, reachability (as a rooted path is a special case of graph pattern), pattern matching and data queries on fuzzy and crisp graph databases. As a satisfaction degree is attached to each answer, rank-ordering them is straightforward.

Concerning flexible querying, [YAG 13] discusses different types of fuzzy preference criteria that appear relevant in the context of graph databases, without going into detail regarding how to express them using a formal query language. There are three main approaches allowing for a flexible querying of graph databases: (1) *keyword-based query* approaches that completely ignore the data schema (see, e.g. [HE 07]), which lack expressiveness for most querying use cases [MAN 09]; (2) approaches that, given a "crisp" query, propose *approximate answers*, for instance, through the implementation of a query relaxation or an approximate matching mechanism (see, e.g. [KAN 01, BUC 08] or [MAN 09]); and (3) approaches allowing the user to introduce

flexibility when formulating the query. The approach we propose belongs to this latter family, for which many contributions concern the flexible extension of XPath [DAM 07, CAM 09, ALM 11] for querying semistructured data (data trees). Such navigational languages behave well for querying graph databases [LIB 13], but no flexible extension was proposed in this specific case.

The literature about preference queries to graph databases is not as abundant, since this issue has only recently started to attract attention. Flexible extensions for querying RDF data are the closest work. Some extensions have been proposed in the literature (see [PIV 16c] for a survey). To the best of our knowledge, the expression of fuzzy preferences involving both value-based and structural aspects over fuzzy RDF graphs was addressed only in [PIV 16b], where the authors present this contribution as being an adaptation of the work presented in this chapter.

6.6. Conclusion and perspectives

In this chapter, we presented a framework that makes it possible to introduce fuzzy preferences in graph pattern queries addressed to a crisp or fuzzy graph database. We discussed implementation issues for the implementation of such a framework. A proof of concept, which extends the Cypher query language of the Neo4j graph database management system, supports the theoretical contribution.

The presented work opens a lot of research perspectives. Some of them obviously concern increasing the expressivity of the query language by considering more complex preferences. For instance, some features very useful for structural network analysis could be offered. Interesting ones based on ordering and counting capabilities concern the distance as well as the indegree, outdegree or centrality of nodes. Other types of fuzzy quantified statements could also be considered.

Another extension of this work concerns the application of fuzzy preferences to the management of data quality in graph databases. In [RIG 17], the authors proposed a framework that makes it possible to take quality information embedded in a graph database into account, at query time. However, no fuzzy preference is considered in this work, which could be extended in this way.

6.7. Acknowledgment

This work was partially funded by the French DGE (*Direction Générale des Entreprises*) under the project ODIN (Open Data INtelligence).

6.8. Bibliography

[ALL 18] AllegroGraph, franz.com/agraph/allegrograph, 2018.

[ALM 11] Almendros-Jiménez J.M., Luna A., Moreno G., "A Flexible XPath-based Query Language Implemented with Fuzzy Logic Programming", *International Workshop on Rules and Rule Markup Languages for the Semantic Web*, Springer-Verlag, Berlin, Heidelberg, pp. 186–193, 2011.

[ANG 05] Angles R., Gutiérrez C., "Querying RDF Data from a Graph Database Perspective", *Proceedings of European Semantic Web Conference (ESWC)*, pp. 346–360, 2005.

[ANG 08] Angles R., Gutierrez C., "Survey of Graph Database Models", *ACM Computer Survey*, vol. 40, no. 1, pp. 1–39, 2008.

[ANG 12] Angles R., "A Comparison of Current Graph Database Models", *Proceedings of ICDE Workshops*, pp. 171–177, 2012.

[ANG 13] Angles R., Prat-Pérez A., Dominguez-Sal D. *et al.*, "Benchmarking database systems for social network applications", *Proceedings of the First International Workshop on Graph Data Management Experiences and Systems*, 2013.

[BAR 13] Barceló Baeza P., "Querying Graph Databases", *Proceedings of PODS*, pp. 175–188, 2013.

[BAR 14] Barceló P., Libkin L., Reutter J.L., "Querying Regular Graph Patterns", *Journal ACM*, vol. 61, no. 1, pp. 8:1–8:54, 2014.

[BOS 95] Bosc P., Pivert O., "SQLf: a relational database language for fuzzy querying", *IEEE Transaction on Fuzzy Systems*, vol. 3, pp. 1–17, 1995.

[BUC 08] Buche P., Dibie-Barthélemy J., Hignette G., "Flexible Querying of Fuzzy RDF Annotations Using Fuzzy Conceptual Graphs", *Proceedings of ICCS*, pp. 133–146, 2008.

[CAM 09] Campi A., Damiani E., Guinea S. *et al.*, "A Fuzzy Extension of the XPath Query Language", *Journal of Intelligent Information Systems*, vol. 33, no. 3, pp. 285–305, 2009.

[CAS 17] Castelltort A., Laurent A., "Exploiting NoSQL graph databases and in memory architectures for extracting graph structural data summaries", *International Journal of Uncertainty, Fuzziness and Knowledge-Based Systems*, vol. 25, no. 1, pp. 81–110, 2017.

[CIG 12] Ciglan M., Averbuch A., Hluchý L., "Benchmarking Traversal Operations over Graph Databases", *Proceedings of ICDE Workshops (ICDEW)*, pp. 186–189, 2012.

[CYP] Cypher, Refcard, https://neo4j.com/docs/cypher-refcard/current/.

[DAM 07] DAMIANI E., MARRARA S., PASI G., "FuzzyXPath: Using Fuzzy Logic an IR Features to Approximately Query XML Documents", *International Fuzzy Systems Association World Congress*, Springer, Berlin, Heidelberg, pp. 199–208, 2007.

[DAT 18] DATASTAX, https://www.datastax.com, 2018.

[DEA 08] DEAN J., GHEMAWAT S., "MapReduce: simplified data processing on large clusters", *Communications of the ACM*, vol. 51, no. 1, pp. 107–113, 2008.

[DUB 00] DUBOIS D., PRADE H., *Fundamentals of Fuzzy Sets*, Kluwer Academic, New York, 2000.

[FAN 12] FAN W., LI J., MA S. *et al.*, "Adding regular expressions to graph reachability and pattern queries", *Frontiers of Computer Science*, vol. 6, no. 3, pp. 313–338, 2012.

[FOD 00] FODOR J., YAGER R., "Fuzzy-set theoretic operators and quantifiers", in DUBOIS D., PRADE H. (eds), *Fundamentals of Fuzzy Sets*, Kluwer Academic, New York, 2000.

[GAL 06] GALLAGHER B., "Matching structure and semantics: A survey on graph-based pattern matching", *AAAI FS*, 2006.

[GHE 03] GHEMAWAT S., GOBIOFF H., LEUNG S.-T., "The Google file system", *ACM SIGOPS operating systems review*, vol. 37, no. 5, pp. 29–43, 2003.

[GRI 80] GRIFFIN T., "Cartographic transformations of the thematic map base", *Cartography*, vol. 11, no. 3, pp. 163–174, 1980.

[HAD 11] HADJALI A., KACI S., PRADE H., "Database preference queries – A possibilistic logic approach with symbolic priorities", *Annals of Mathematics and Artificial Intelligence*, vol. 63, nos 3–4, pp. 357–383, 2011.

[HE 07] HE H., WANG H., YANG J. *et al.*, "BLINKS: Ranked Keyword Searches on Graphs", *Proceedings of SIGMOD*, pp. 305–316, 2007.

[INF 18] INFINITEGRAPH, www.objectivity.com/products/infinitegraph, 2018.

[KAN 01] KANZA Y., SAGIV Y., "Flexible Queries over Semistructured Data", *Proceedings of PODS*, pp. 40–51, 2001.

[LIB 13] LIBKIN L., MARTENS W., VRGOČ D., "Querying Graph Databases with XPath", *Proceedings of ICDT*, pp. 129–140, 2013.

[LÓP 09] LÓPEZ F. D.R., LAURENT A., PONCELET P. *et al.*, "FTMnodes: Fuzzy tree mining based on partial inclusion", *Fuzzy Sets and Systems*, vol. 160, no. 15, pp. 2224–2240, 2009.

[MAL 10] MALEWICZ G., AUSTERN M.H., BIK A.J. *et al.*, "Pregel: a system for large-scale graph processing", *Proceedings of the 2010 ACM SIGMOD International Conference on Management of data*, pp. 135–146, 2010.

[MAN 09] MANDREOLI F., MARTOGLIA R., VILLANI G. *et al.*, "Flexible Query Answering on Graph-modeled Data", *Proceedings of EDBT*, pp. 216–227, 2009.

[NEO 13] NEO TECHNOLOGY, The Neo4j Manual v2.0.0, Cypher, 2013.

[NEO 18] NEO4S, www.neo4j.org, 2018.

[ORI 18] ORIENTDB, http://orientdb.com, 2018.

[PIV 12] PIVERT O., BOSC P., *Fuzzy Preference Queries to Relational Databases*, World Scientific, 2012.

[PIV 14] PIVERT O., THION V., JAUDOIN H. *et al.*, "On a Fuzzy Algebra for Querying Graph Databases", *26th IEEE International Conference on Tools with Artificial Intelligence, ICTAI 2014*, Limassol, Cyprus, pp. 748–755, 10–12 November 2014.

[PIV 15] PIVERT O., SMITS G., THION V., "Expression and efficient processing of fuzzy queries in a graph database context", *2015 IEEE International Conference on Fuzzy Systems, FUZZ-IEEE 2015*, Istanbul, Turkey, pp. 1–8, 2–5 August 2015.

[PIV 16a] PIVERT O., SLAMA O., SMITS G. *et al.*, "SUGAR: A graph database fuzzy querying system", *Tenth IEEE International Conference on Research Challenges in Information Science, RCIS 2016*, Grenoble, France, pp. 1–2, 1–3 June 2016.

[PIV 16b] PIVERT O., SLAMA O., THION V., "Fuzzy Quantified Structural Queries to Fuzzy Graph Databases", *Scalable Uncertainty Management - 10th International Conference, SUM 2016*, Nice, France, pp. 260–273, 21–23 September 2016.

[PIV 16c] PIVERT O., SLAMA O., THION V., "SPARQL Extensions with Preferences: A Survey", *Proceedings of the 31st Annual ACM Symposium on Applied Computing, SAC'16*, New York, USA, pp. 1015–1020, 2016.

[RAB] RABBITHOLE, http://neo4j.com/blog/rabbithole-the-neo4j-repl-console/.

[RIG 17] RIGAUX P., THION V., "Quality awareness over graph pattern queries", *Proceedings of the 21st International Database Engineering & Applications Symposium, IDEAS 2017*, Bristol, United Kingdom, pp. 90–97, 12–14 July 2017.

[ROS 75] ROSENFELD A., "Fuzzy graphs", in ZADEH L.A., FU K.S., TANAKA K. (eds), *Fuzzy Sets and their Applications to Cognitive and Decision Processes*, Academic Press, 1975.

[SPA] SPARKSEE, sparsity-technologies.com.

[STE 11] STEFANIDIS K., KOUTRIKA G., PITOURA E., "A Survey on Representation, Composition and Application of Preferences in Database Systems", *ACM Transaction Database Systems*, vol. 36, no. 3, pp. 19:1–19:45, August 2011.

[WOO 12] WOOD P.T., "Query languages for graph databases", *SIGMOD Record*, vol. 41, no. 1, pp. 50–60, 2012.

[YAG 13] YAGER R., "Social Network database Querying based on Computing with Words", in PIVERT O., ZADROZNY S. (eds), *Flexible Approaches in Data, Information and Knowledge Management*, Springer, 2013.

[ZAD 65] ZADEH L.A., "Fuzzy sets", *Information and control*, vol. 8, no. 3, pp. 338–353, 1965.

7

Relevant Filtering in a Distributed Content-based Publish/Subscribe System

7.1. Introduction

Sources of information have been multiplying on the Web for several years, especially due to the success of news portals and social networks that produce information in real time. These flows of information can be kept and processed, often in RSS [RSS 03] and Atom [GRE 07] formats. However, it turns out that nowadays the amount of data which has to be analyzed daily is so large [HME 11] that a user may miss information of interest. Thus, a given user can be lost with so many sources and the frequency of updates [TRA 14]. Pub/Sub (Publish/Subscribe) systems (Redis [CAR 13], Scribe [ROW 01], Siena [CAR 01], Echo [EIS 00]) have been designed to face the problem of aggregating and delivering information of interest (bookmarks and topics) to end users.

For these reasons, we advocate a content-based Publish/Subscribe paradigm for Web 2.0 syndication in which information consumers are decoupled (in both space and time) from feeds (produced flows of items) and instead express their interest with keyword-based subscriptions, which are computed using content-based filtering. However, even if information is filtered through the matching process, users remain flooded by notifications [HME 11]. Some propositions enhance the filtering process by removing redundant information (i.e. Novelty [ZHA 02, CLA 08]) and/or

Chapter written by Cédric DU MOUZA and Nicolas TRAVERS.

NoSQL Data Models: Trends and Challenges, First Edition. Edited by Olivier Pivert.

by taking into account information diversification of the delivered items (i.e. *Diversity* [DRO 09b, DRO 09a, PRI 08]), which is generally presented as a *top-k* issue. However, the Pub/Sub context discards traditional *top-k* approaches due to real-time notifications and the time constraint that cannot remove a notified item from the past.

Very few works have been proposed to take into consideration both relevance, novelty and diversity in a *real-time* Pub/Sub context. Our Pub/Sub system `FiND` [HME 15] addresses this with a two-step process: matching and filtering. For matching keyword-based subscriptions, we assume the existence of an index [HME 12]. The second step is the core of this chapter, and it aims at computing items' content already delivered to a user to filter new incoming items. The difficulty for a real-time Pub/Sub system is evaluating novelty and diversity on-the-fly for every incoming item.

This chapter focuses on the way to enhance the relevance of filtering and to integrate such a process in two different implementations: a centric-based version and a distributed version in a NoSQL environment. Our contributions in this paper are:

– definitions for novelty and diversity in this particular context, along with a proposal for a weighting score (*Term Discrimination Values, TDV*) adapted to the characteristics of items and subscriptions;

– an efficient filtering algorithm for *real-time* Pub/Sub systems based on novelty and diversity, which exploits redundancy between subscriptions' history. Two optimized implementations are proposed in centralized and distributed contexts;

– a validation which highlights the complementarity of novelty and diversity both in centralized and NoSQL environments;

– enhancement of *TDV* [WIL 85] computation by proposing incremental versions in a distributed environment.

The chapter is organized as follows. An overview on novelty and diversity processes in Pub/Sub techniques is detailed in section 7.2. Then, we present the data model (section 7.3) on which relevance in notifications is based and developed in section 7.4 by introducing novelty and diversity. Two distinct implementations are detailed in section 7.5 and compared in section 7.7. Section 7.6 deals with TDV score updates. We conclude in section 7.8.

7.2. Related work: novelty and diversity filtering

In Pub/Sub systems, users are often faced with an issue of notifications, which tend to be flooded by information. To enhance the quality of filtering, two properties of information were proposed: *novelty* and *diversity*. In filtering by novelty, the objective is to discard an item whose information has already been notified (truncate or a similar content). *Diversity* captures a complementary kind of redundancy since it measures whether the information contained in a given item is globally present in the set of recently notified items or not. We present here how these two properties are used in the literature, first on the Web and then in a Pub/Sub context.

When searching a document on the Web, we assume static and already known documents. The objective is to present to users the k most relevant and diverse documents matching a query. To achieve this, some models are based on probabilities for diversity [ANG 11] or on graphs for computing distance [DRO 12a]. Some of them propose to modify diversity measures by focusing on non-common attributes between items based on user-defined filters [YU 09], by defining a trade-off between similarity and diversity [SMY 01], by integrating entities and sentiment in a *Greedy* Max–Min algorithm [ABB 13], by defining time-based distances with a Gaussian similarity [KEI 12] or by comparing an item with the compression of all previous texts like the NCD distance [CAR 10]. Globally, these techniques allow us to compute large texts in a static top-k evaluation but cannot adapt to our context since we consider small items, which changes the relevance of previous methods. Moreover, real-time delivery of information is an important constraint that cannot be ignored.

Some approaches focus on continuous filtering, as in the Pub/Sub context, combined with top-k techniques. They may be based on fixed-size windows in order to guarantee the amount of items to keep in the system, like [DRO 12b], which uses a dynamic index to quickly find whether an item is diverse or not using a frequently updated snapshot of items; [GAB 04], which focuses only on novelty with extracted entities from items; [MIN 11], which presents an incremental approach for time-based diversification; or [PAN 12], which extracts topics from items for a simple coverage distance. However, fixed-size windows can hardly manage different notification rates for subscriptions. In fact, low rates will keep very old items to filter out incoming items and high rates will remove recent items, which should remove duplicates.

The closest approach to our solution, [DRO 09b] (detailed in section 7.5.1), uses top-k windows to compute diversity in real-time delivery. It is based on the interchange algorithm that notifies an item if exchanging that item with the previous top-k levels up the diversity. However, this solution can deliver items from previous windows if considered as non-diverse, or remove items from the past for future filtering steps. Our experiments illustrate that this approach tends to locally diversify information, but not over time. Moreover, keeping all items will lead to scale-up issues.

7.3. A Publish/Subscribe data model

Our Pub/Sub data model relies on the fact that published items are mainly text oriented. Hence, subscriptions are *long lasting* (continuous) queries under the form of keyword-based subscriptions. Whenever a news item is published, it gets evaluated against the set of subscriptions submitted to the system and, for every matching subscription, the corresponding subscriber is notified.

7.3.1. *Data model*

The set of stored subscriptions is denoted by $\mathcal{S}$ and their total number by $|\mathcal{S}|$. Each subscription $s \in \mathcal{S}$ includes a set of distinct terms from a vocabulary $\mathcal{V}_S = \{t_1, ..., t_n\}$. $\mathcal{I} = [I_1, ...I_m]$ denotes the feed of incoming items. Items $I \in \mathcal{I}$ are also formed by a set of terms ($I \subseteq \mathcal{V}_I$, with $\mathcal{V}_I$ the vocabulary of items). In this context, a match occurs if and only if all of the terms of a subscription s are also present in a news item I (i.e. broad match semantics).

Subscription	s_1	s_2	s_3	s_4
Terms	$t_1 \wedge t_2 \wedge t_4$	$t_1 \wedge t_3$	$t_1 \wedge t_2 \wedge t_5$	$t_2 \wedge t_4$

Table 7.1. *Example of keyword based subscriptions*

Consider the set of subscriptions $\mathcal{S}$ illustrated in Table 7.1. Matching item $I = \{t_2, t_4, t_5\}$ against $\mathcal{S}$ will result in the set of matched subscriptions $\mathcal{M} = \{s_4\}$ since t_2 and t_4 of s_4 are contained in I. The matching process has been fully developed in [HME 16] by proposing tree-based structures to evaluate subscriptions. This paper goes one step further by enhancing notifications by integrating a relevance feature.

Processing textual content requires taking into account term weights for items and subscriptions. However, traditional techniques in Information Retrieval [BAE 99] cannot be adapted to our context. In the following sections, we will develop the TDV weighting approach adapted to the Pub/Sub context, which is more convenient for the filtering step.

7.3.2. *Weighting terms in textual data flows*

The partial matching process and filtering quality addressed by our system must take into account a weighting of terms in order to compute scores. In view of the foregoing, this weighting must be computed at low cost as well as being relevant. Several term-weighting models are proposed in the literature like the Term Frequency (TF – frequency of a term in a document) combined with Inverse Documents Frequency (IDF – inverse frequency of a term in all documents) [BAE 99], the Term Discrimination Value (TDV – see below) [SAL 75] or the *Term Precision* (TP – number of relevant vs. non-relevant terms) [BOO 74].

In the context of this work, we rely on the TDV weighting function, which is more adapted to the quality of vocabularies in Web Syndications Systems. In fact, an item is a short set of terms where term frequencies cannot be used, so the tf/idf standard function is unsuitable (experimentally proved in section 7.5.1). Moreover, the TDV weighting function measures how a term helps to distinguish a set of documents (i.e. the term influence on the global entropy). Therefore, for our subscriptions set, neither a very frequent term (present in many subscriptions, so this term is not a selective filter for subscriptions), nor a very uncommon term (present in very few subscriptions, so assuming this is not a typo, it will probably never lead to a notification) have an important TDV. Finally, the simplicity of computation is all the more important, since we only have to compute a sum of weights for each subscription (section 7.5.1.2).

DEFINITION 7.1 (TDV).– *The discrimination value for a term* t_k *is the difference between the vector space density of the occurrence matrix with* t_k *and the density of the matrix without* t_k*. Therefore, assuming a similarity distance* $sim(I_1, I_2)$ *between items, like, for instance, the cosine of the*

Euclidian distance, we compute the density as the average pairwise similarity between distinct items:

$$\Delta(\mathcal{I}) = \frac{1}{|\mathcal{I}| \times (|\mathcal{I}| - 1)} \sum_{i=1}^{|\mathcal{I}|} \sum_{j=1 \wedge j \neq i}^{|\mathcal{I}|} sim(I_i, I_j)$$

Finally, the TDV for a term t_k is:

$$tdv(\mathcal{I}, t_k) = \Delta(\mathcal{I} - \{t_k\}) - \Delta(\mathcal{I})$$

For the sake of simplicity, we denote the TDV for the term t_k $tdv(t_k)$ instead of $tdv(\mathcal{I}, t_k)$. Based on this function, we can weigh the different terms of items and subscriptions. Each term weight w_i is the TDV $tdv(i)$ normalized by the sum of TDV of the query terms.

Of course, computing and updating the TDV is a real time-consuming process but it can be done in parallel with the filtering process. For clarity purpose, we focus now on the way to integrate TDV in novelty and diversity computation and then detail in section 7.6 the way to optimize TDV updates incrementally in a NoSQL environment.

7.4. Publish/Subscribe relevance

In top-k approaches, notified items are computed on the whole set of items, leading to delays of item delivery. In our approach, we consider the set of notified items for a given subscription, the so-called subscription *history*, and we use this history to filter out in real time the incoming item just after the matching process. This section presents our approach and the definitions adopted, and the instantiation is presented in section 7.5.

7.4.1. *Items and histories*

In our context, we define an item as a set of terms. Each term is associated with a term weight denoted by w_i, which is used to compute distances and similarities. To compute novelty and diversity, a Pub/Sub system must keep already notified items, also called subscription history H. Each one is a time-ordered set of items linked to a subscription. Each time an item is notified for a subscription, it is added to its history.

7.4.2. *Novelty*

The objective when filtering by novelty is to discard an item that does not contain new information with respect to items in the subscription history, i.e. an item I with a truncate or a similar content of a previous item I'. Since, in our context, history is time dependent, the measure of novelty *new(I, I')* should be asymmetric [ZHA 02] to test how new an incoming item is w.r.t. an existing one and not conversely. Finally, we define the novelty of an incoming item I with respect to an existing history H by comparing I with all items in H, one by one.

DEFINITION 7.2 (Novelty item-history).– *Given a history of items H, an item I and a novelty item–item measure new (like the one proposed by Definition 7.4), I is said to be new with respect to H iff:*

$$\forall I' \in H,\ new(I, I') \geq \alpha$$

We assume that the novelty threshold α is a parameter fixed for the user for his subscription according to the defined or required output rate of items. section 7.7 will show an impact on the filtering rate and quality, histories and performances in our system.

7.4.3. *Diversity*

Diversity captures a complementary kind of redundancy (towards novelty) since it measures whether the information contained in a given item is globally present in the set of notified items or not (segmented information). The objective is to detect whether an incoming item conveys new information regarding the subscription's history of notified items. The user's objective is to receive only the interesting items with different information; the items are filtered by their content. The degree of diversity of an item for a user w.r.t. its subscription history is measured as the amount of increase in the average pairwise distance *dist(I,I')* between the history's items [DRO 09a]. It can be observed that, to keep $D(H)$ and $D(H')$ (with I) comparable, we must remove from H an old item I_o before adding I. To satisfy diversity, criteria I must be on average more distant from all items in H than at least one of the items in H. We decided to choose I_o as the oldest item in H assuming that I_o is more likely to be the most distant item since its information is older and deprecated. Focusing on only one item of the history allows us to avoid the quadratic complexity and scale up the system.

DEFINITION 7.3 (Diversity of items).– *Assume a history of items H, where $D(H)$ is the average pairwise distance between its items. An item I improves the diversity of H (I_o the oldest item of H) if and only if:*

$$D(H \cup \{I\} - \{I_o\}) > D(H)$$

$$D(H) = \frac{1}{|H| * (|H| - 1)} \sum_{I \in H} \sum_{(I' \in H \wedge I' \neq I)} dist(I, I')$$

The two average distances must be comparable and so the number of items in histories must be identical. Otherwise, if we compare H to $H \cup I$, the new item I must be far more distant from the items in H to make it more diverse than in our proposition. It justifies our choice to interchange I with I_o, the more likely distant item since the difference of time makes information naturally more distant.

7.4.4. *An overview of the filtering process*

To resume our time-dependent filtering process, an incoming item I that matches a subscription must verify novelty and diversity with the subscription history H. First, the novelty of I is checked with H and, if at least one similarity is below the threshold α, it is discarded for H. Second, the diversity of H is compared with the diversity of $H \cup I - I_o$. If I increases the average distance, then it is notified and added to H.

A subscription is said to be satisfied by an item only if either the matching or filtering process is validated. For the matching process, it means that all subscriptions' terms are contained in the item. According to the filtering process, the item passes through both novelty and diversity.

7.4.5. *Choices of relevance*

7.4.5.1. *Novelty*

Novelty checks if information from I has already been delivered. For example, if I' contains I and appends additional information, then I is not new compared with I', but I' is new compared with I. Therefore, symmetric measures like the Jaccard measure are unsuitable. Consequently, we adopt the following measure, inspired by Newsjunkie [GAB 04], for the novelty of an item compared with another one.

DEFINITION 7.4 (Novelty item-item).– *Let α be a threshold of novelty, $\alpha \in [0,1]$, and I and I' two items. I is said to be new compared with I' if and only if:*

$$new(I, I') = \frac{\sum_{t \in (I \setminus I \cap I')} tdv(t)}{\sum_{t \in I} tdv(t)} \geq \alpha$$

This measure computes the weighted coverage of terms from I without taking into account the terms present in I' w.r.t the sum of weights of terms only present in I. Note that we chose the tdv value as the weight of terms according to the above discussion.

7.4.5.2. *Similarity in diversity*

To measure diversity, we need to compute the distance between items. Several distance measures are proposed in the literature to compute diversity on a set of documents. Most frequently used are Cosine [ZHA 02], Euclidean [DRO 12a, PRI 08] and Jaccard [DRO 12b], but we can also quote Pearson derived from Cosine, and Dice derived from Jaccard or Levenstein. For short items, Euclidean is known to produce more relevant results [BAV 10]. Thus, we consider in our system a diversity function based on an Euclidean distance weighted by TDV.

7.4.5.3. *Discussion for the choice of a distance function*

The Jaccard measure computes the distance as the ratio between the intersection and the union of the terms of two documents (here items). However, it does not take into account the importance of item terms. Conversely, the Cosine measure allows us to compute the distance between two vectors of term weights: the similarity of two documents corresponds to the correlation between the vectors (angle between the vectors). An important property of the Cosine similarity is its independence of document size. However, since our items are short as observed in [BAV 10], Cosine and Jaccard are not appropriate. Therefore, we focus on Euclidean distance. Indeed for the Euclidean distance, items are more distant if they have important terms not in common. As for the Cosine metric, in order to make our score independent of the items size, we normalized the Euclidean distance, as shown in the following definition.

DEFINITION 7.5 (Distance between two items).–

$$dist(I, I') = \sqrt{\sum_{t \in (I \cup I' \setminus I \cap I')} tdv^2(t)}$$

With the Euclidean distance, the contribution of a term which is shared by two items is null since the difference in their TDV is 0. Therefore, we compute only the terms not in common with the tdv^2 value.

7.5. Real-time integration of novelty and diversity

We must recall that, in the Pub/Sub context, queries (subscriptions) are stored to be evaluated on-the-fly. Therefore, all the optimization is done in such a way as to process millions of queries on incoming items. One of the main techniques is to factorize treatments since there is a high probability that queries share the same operations.

7.5.1. *Centralized implementation*

In this section, we present our solution to quickly filter out items based on novelty and diversity criteria. It also allows us to efficiently store and manage items' histories for all subscriptions.

7.5.1.1. *Shared history*

Since an item can belong to several histories, we need to avoid keeping all items. A simple solution consists in storing the last N notified items [ZHA 02] for each subscription and in factorizing histories by storing each item only once. However, the publication rate strongly differs from one source to another and this approach will impact the filtering quality. In fact, important items can be removed too quickly (active sources) or a highly filtering item could never disappear (rarely notified sources). We conclude that relevance of filtering will be impacted by these item-based histories.

To optimize memory consumption, we adopt a *shared history,* which is basically a time-based sliding window $\mathcal{W}$ that contains all items notified at

least once during the last period. Subscription histories are stored as ordered sets of pointers to related items in $\mathcal{W}$. Figure 7.1 presents an example of a sliding window $\mathcal{W}$ and two subscriptions S_1 and S_2 with their corresponding histories and pointers to the shared history.

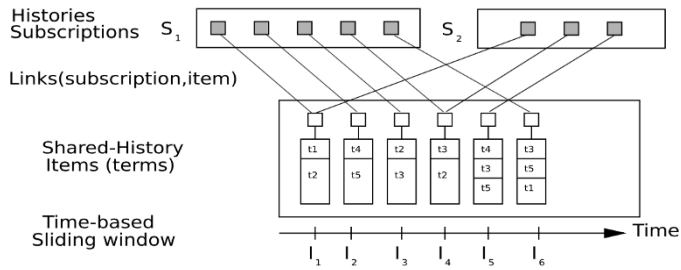


Figure 7.1. *Example of a sliding window*

7.5.1.2. *Shared history filtering algorithm*

Filtering by novelty and diversity with a large number of subscriptions that share common items poses a real optimization challenge. Indeed, a naïve algorithm, which checks first novelty, then diversity for an incoming item with the histories of all the subscriptions it matches, has the following cost (definitions 7.2 and 7.3):

$$C_{filter}(I,S) = \sum_{s \in S} \sum_{I' \in H_s} new(I,I') + \sum_{s \in \rho(S)} D(H_s \cup \{I_n\} - \{I_o\})$$

where S corresponds to the set of subscriptions matched by the incoming item I and $\rho(S)$ represents the ratio of S for which I satisfied the novelty threshold. Assume that term weights are computed and considered as constant such that the average history size is N_H items (number of computations per history) and the average item size is S_I (time for each computation is based on item size). Since the cost depends on the number of computations per history and the time for each computation is based on item size, the average complexity for this algorithm is:

$$C_{filter}(I,S) = O(|S|.N_H.S_I) + O(|\rho(S)|.(N_H.S_I)^2)$$

Experiments in Figure 7.6 show that the novelty has a filtering rate proportional to the chosen threshold α. This results in $|S| \sim |\rho(S)|$ and in a global quadratic complexity:

$$C_{filter}(I,S) = O(|S|.(N_H.S_I)^2)$$

To achieve web scaling, we propose to optimize the filtering algorithm by factorizing computations for different subscriptions. As explained previously, due to the quadratic complexity of the diversity computation, the novelty must be first evaluated. However, browsing H several times for processing similarities can be costly, especially as novelty does not filter enough (see section 7.7.3.1). In order to avoid the scanning of history twice, both filters are applied in one course. Algorithm 7.1 presents the processing with shared histories and optimized computations for novelty and diversity.

Algorithm 7.1. Novelty and diversity filtering on a history

Require: An item I, a history H and $\alpha \in [0, 1]$ novelty threshold

```
1:  sum_H ← 0; sum_I ← 0; I_o ← H[0]; //oldest item
2:  for all I' ∈ H do
3:      if I.getInfo(I') = null then
4:          N ← novelty(I,I'); d ← dist(I,I'); I.putInfo(I',N,d);
5:      else
6:          N ← I.getInfo(I').N; d ← I.getInfo(I').d;
7:      if N < α then
8:          return;
9:      else if I' != I_o then
10:         I.sum ← I.sum + d;
11: if I.sum > I_o.Sum then
12:     for all I' ∈ H do
13:         I'.sum ← I'.sum+I.getInfo(I').d
14:     H ← H ∪ I;
15:     Notify I;
```

The algorithm processes each item I' in H. Since I must be compared with I' each time it appears in a history, we compute $new(I, I')$ and $dist(I, I')$ only once to benefit from (I, I') co-occurrences. Thus, we check if this value has already been computed for another subscription. If not, we compute and register it (line 4), otherwise we just retrieve the stored value (line 6). If I is not new, the algorithm stops (line 8–9). Remember that diversity requires the computation of the average pairwise distance between items of H and,

due to this quadratic complexity, we evaluate novelty first. Then, we cumulate the distance (I, I') with others from H only if it is not the oldest item I_o (line 10–11). Second, as explained above, the diversity computation can be simplified to the comparison between the sum of distances from I_o and I (line 14). In that case, sums of distances are updated (line 15–16), I is added to the history and notified (line 18–19).

To summarize, our algorithm integrates two main optimizations. The first one exploits the high probability of computing several times the similarity and distance for each pair of items (I, I'). Further computations of pairs are constant and not dependent on item size. These values are stored during the filtering process of I and deleted when there is no more subscription to check. The gain depends on the co-occurrence ratio $\sigma \in [0, 1]$ of items in subscription histories, defined by the number of co-occurrences of item pairs checked during the filtering step on the total number of pairs required:

$$\sigma = 1 - \frac{\#cooccurrences}{\#pairs}$$

The second optimization deals with the computation of the density, which changes for each notified item. To avoid the quadratic complexity of computing the sum of pairwise distances in H, we propose to store computed sums of distances $I.sum$ for each item of the history H with all items received later. Then the density of H is the sum of $I.sum$. Since the oldest items are removed with their stored values, no other update has to be done for remaining items. Furthermore, the distance computations also benefit from the "σ" co-occurrence gain. Formally, $I.sum$ is a stored value equal to:

$$I.sum = \sum_{(I' \in H \wedge I'.\tau > I.\tau)} dist(I, I')$$

Since diversity is the comparison between $D(H')$ and $D(H)$ with $H' = H \cup \{I_n\} - \{I_o\}$, the result is that checking whether an incoming item increases the diversity or not may be simplified as follows:

$$\frac{2 \times \sum_{I' \in H'} I'.sum}{|H'| \times (|H'| - 1)} > \frac{2 \times \sum_{I' \in H} I'.sum}{|H| \times (|H| - 1)}$$

$$|H| = |H'| \Rightarrow \sum_{I_k \in H'} I_k.sum > \sum_{I_k \in H} I_k.sum$$

$$H' = H \cup \{I_n\} - \{I_o\} \Rightarrow \sum_{I_k \in H} I_k.sum + I.Sum - I_o.Sum > \sum_{I_k \in H} I_k.sum$$

Therefore, the diversity test consists in checking if:

$$I.sum > I_o.sum$$

To conclude, the complexity of our algorithm benefits from the co-occurrence between items for novelty and diversity, which results in the following linear complexity.

LEMMA 7.1 (Shared-history complexity).– *Algorithm 7.1 has a linear complexity w.r.t. the number of subscriptions matched by one item, the average history and the item size.*

PROOF.– Assume that σ denotes the average co-occurrence ratio between items, $|S|$ the number of subscriptions matched by the incoming item, N_H the average history size and S_I the average item size. Then with the shared-history management, the filtering cost given by Algorithm 7.1 is:

$$\begin{aligned} C_{filter}(I,S) &= C_{nov}(I,S) + C_{div}(I,\rho(S)) \\ &= O(|S|.\sigma.N_H.S_I) + O(\alpha|S|.\sigma.N_H.S_I) = O(|S|.N_H.S_I) \end{aligned}$$ ■

The complexity of computing $D(H)$ and $D(H \cup I)$ is about O($|H|$), while the complexity of computing the average pairwise distance between items of H is about O($|H|^2$). This optimization in computing the density reduces the processing time.

7.5.2. Distributed filtering

To complete our work on Pub/Sub systems, we try to compare our centralized implementation with a distributed version. To achieve this, we propose integration in the `MongoDB` NoSQL database, in which we must adapt our data model and the way to distribute the computation of Novelty and Diversity. `MongoDB` seems to be more adapted to this context since we need: 1) some flexibility with a document-oriented model, 2) control over where to place data in the cluster and 3) to guarantee the consistency

for proper filtering scores. Moreover, the implementation complexity of the similarity functions discards some other document-oriented NoSQL databases (Cassandra, DynamoDB, CouchBase).

First, we need to define a physical data model that takes into account the correlation between items and subscriptions in order to compute novelty and diversity. There are two possibilities: subscription-based (histories of items) and item-based (list of notified subscriptions).

7.5.2.1. *Subscription-based model*

The subscription-based model nests for each subscription all the history of items in a single document, as shown in Listing 7.1. The advantage is the accessibility to all items and a pre-computed sum of distances, and a simplification of the design of the distributed computation with the Map/Reduce functions since it corresponds to Algorithms 7.1 without the shared history. For an incoming item, 1) all subscriptions are queried (*Map function*), 2) novelty and diversity are evaluated and then 3) every notification leads to an update (append and remove) on the corresponding history and sum of distances (*Reduce function*). The interesting point of this algorithm is the distribution of the computation according to the number of subscriptions.

```
{"subID":1024, "terms":[{"tID":103, "tdv":0.073}, {"tID":1710, "tdv":0.090}],
 "history": [
  {"itemID":8, "sumDist":0.11,"time":21345, "terms":[{"tID":23,"tdv":0.005},"..." ]},
  {"itemID":12,"sumDist":0,  "time":45678, "terms":[{"tID":12,"tdv":0.068},"..." ]}
]}
{"subID":1260, "terms":[{"tID":101, "tdv":0.071}, {"tID":1514, "tdv":0.086}],
 "history": [
  {"itemID":5, "sumDist":0.20,"time":12345, "terms":[{"tID":15,"tdv":0.025},"..." ]},
  {"itemID":9, "sumDist":0.12,"time":34567, "terms":[{"tID":12,"tdv":0.068},"..." ]}
  {"itemID":12,"sumDist":0,  "time":45678, "terms":[{"tID":12,"tdv":0.068},"..." ]}
]}
```

Listing 7.1. *Subscription-based model*

However, items are highly redundant in the repository since one item is stored in every linked subscription history. This has a strong impact on the repository size and efficiency. It is worth noticing that updating documents, have a huge impact on efficiency in MongoDB. In fact, document growth requires reallocating blocks of data in the repository, which is a very costly process in a database (traditionally tackled with a PCTFREE). Our experiments will explore this point of view.

7.5.2.2. *Item-based model*

In the item-based model, each item is stored only once in the repository with all linked subscriptions that are nested (Figure 7.2). The advantage of this method is to maximize the evaluation of novelty and the distance for diversity since it is done only once for each pair of items.

Since updating stored items is an issue in NoSQL databases, we need to store an item once, and remove it when getting out of the time window. Consequently, we need to compute the sum of pairwise distances between items in the same history. Contrary to "*I.sum*" in the centralized system (Algorithm 7.1), the sum of distances as designed in Algorithm 7.1 must be set in preceding order (to avoid updates). Thus, each distance with preceding items is stored in the current item according to the corresponding subscription. For this, "distItems" stores for each itemID (the key) the distance with the local item (value), and this will help to recompute diversity of I_o versus I.

Listing 7.2 shows four items chronologically stored with nested subscriptions. We can see that subscription 1024 (resp. 1260) has notified items 8 and 12 (resp. 5, 9 and 12). Each item distance is stored with preceding stored items (e.g. in item 12, subscription 1260 has two distances with items 5 and 9). This method is more complex depending on the way to distribute the notification process. Algorithms 7.2 and 7.3 illustrate the Map/Reduce implementation of the distributed process. The Map function compares an incoming item with each stored one and emits for each corresponding subscription (in the history) information useful in the Reduce function. The latter reconstructs the spread history and checks the novelty.

Algorithm 7.2. Map Function of the item-based model

Require: A stored item I_s, an incoming item I_n

1: **if** I_s.time $<$ window_time **then** `return`;
2: N $\leftarrow$ `novelty`(I_s.terms, I_n.terms) ; D $\leftarrow$ `dist`(I_s.terms, I_n.terms)
3: **for all** nested subscriptions s **do**
4: **if** `match` (I_n.terms, s.terms) **then**
5: **if** N $< \alpha$ **then** `emit`($s.subID$, {"new": false})
6: **else** `emit`($s.subID$, {"new":true, "ID":I_s.itemID, "dist":D, "distItems":s.distItems, "time":d.time}) **end if**

Algorithm 7.3. Reduce Function of the item-based model

Require: List of grouped items l (for a subscription)

1: sum $\leftarrow$ 0; sum$_o$ $\leftarrow$ 0; oldest $\leftarrow$ $l[0]$
2: **for all** status s in l **do**
3: **if** s.new == false **then** return null **end if**
4: **if** oldest.time $>$ s.time **then** oldest $\leftarrow$ s **end if**
5: **for all** status s in $l - oldest$ **do**
6: sum $\leftarrow$ sum + s.dist ; sum$_o$ $\leftarrow$ sum$_o$ + s.distItems[s.ID]
7: **if** sum $>$ sum$_o$ **then** return l **end if**

```
{"itemID":5, "time":12345, "terms":[{"tID":12, "tdv":0.068}, "..." ],
  "subscriptions":[
    {"subID":1260, "terms":["..." ], "distItems":{}},
    {"subID":1411, "terms":["..." ], "distItems":{"3":0.310}}
]}
{"itemID":8, "time":23456, "terms":[{"tID":23, "tdv":0.005}, "..." ],
"subscriptions":[
    {"subID":1024, "terms":["..." ], "distItems":{}},
]}
{"itemID":9, "time":34567, "terms":[{"tID":15, "tdv":0.025}, "..." ],
  "subscriptions":[
    {"subID":1260, "terms":["..." ], "distItems":{"5":0.553}},
    {"subID":1411, "terms":["..." ], "distItems":{"5":0.610}}
]}
{"itemID":12, "time":45678, "terms":[{"tID":12, "tdv":0.068}, "..." ],
  "subscriptions":[
    {"subID":1024, "terms":["..." ], "distItems":{"8":0.441}},
    {"subID":1260, "terms":["..." ], "distItems":{"9":0.703, "5":0.503}},
    {"subID":1536, "terms":["..." ], "distItems":{}}
]}
```

Listing 7.2. *Item-based model*

The Map function (Algorithm 7.2) first checks if the stored item I_s remains in the time window (correct timestamp, line 1). Then it computes novelty and distance between I_s and the incoming item I_n (line 2). Then, for each nested subscription s, it checks the matching process (embedded terms, line 4). If the novelty N is lower than the threshold (line 5), it emits a *false* status to the Reduce function (the grouping key is the subscription ID) in order to avoid a notification; I_n can be new for other I_s and given to the Reduce function anyway. Finally, the distance D and the sum of distances $distItems$ are emitted (line 6) to compute novelty in the Reduce function.

The Reduce function (Algorithm 7.3) requires the recomposition of the subscriptions (grouping key "subID"). If a group occurs on a subscription ID, it means that at least one match occurred in the Map function step. Then we must check whether at least one novelty is false (line 3) and get the oldest item (line 4). Then compute the sums of distances for I and I_o (line 7). If the sum of distance levels up the diversity (line 9), the list of distances is returned to notify the item and store the item in the repository.

7.5.2.3. *Distributing strategies*

To make a choice between these two strategies, we must take into account the distribution process. It relies on a horizontal scaling of the database, where data are distributed across multiple shards (servers). The goal is to optimize distribution of computations and avoid network communications between the Map and Reduce functions (called *shuffle*).

The subscription-based strategy queries the subscriptions to check novelty and diversity in the *Map*. If an item is notified, it must be added to histories, and all other items must be updated (sumDist). The updated version is computed during the *Reduce,* but locally, if we distribute documents according to the subscription ID.

For the item-based strategy, the *Map* phase is applied on items and it benefits from the co-occurrence of items in subscription histories. Therefore, novelty and diversity distances are computed once. The *Reduce* phase aggregates histories for each matching subscription. The result provides the new item to insert. It generates more communications between shards when aggregating in the shuffle phase. However, we can enhance this by distributing documents according to the first subscription ID, which is sorted on the number of correlated items (more likely to group more items).

Naturally, the item-based strategy brings better performances especially when highly distributed. Avoiding document updates and factorizing the novelty and diversity in the *Map* phase decrease drastically the computation but the shuffle phase consumes the bandwidth for large numbers of subscriptions (number of aggregates). The experiments will confirm our conclusions (see section 7.7.5).

7.6. TDV updates

As discussed in section 7.3.2, TDV computation is a very time-consuming process. We need to extract similarities between all pairs of items in a collection for N times: one without all the terms, and N times by removing a term to compute the density of this term. Consequently, this process cannot be done in real time and must be evaluated in parallel with the filtering process. Initially, it took two days to compute 10M items. Even if a TDV does not evolve much over time, it relies on the evolution of the presence of a term compared with all of them, on all items. The computation step needs to start from the beginning each time we need to provide new TDV.

Our work to enhance the computation of TDV updates is twofold: i) adapt TDV computation techniques in an incremental process and ii) adapt our algorithm in a distributed context in order to scale up.

7.6.1. *TDV computation techniques*

TDV computation is a heavy process that requires evaluation of the density of a collection of items. It computes similarities between all pairs of items. Basically, to provide a TDV for a given term, densities must be computed twice, first with all the terms and second without the given term. Thus, the TDV must be computed for every term of the collection. We study, in this section, how to provide better solutions to compute TDV. In the literature, three main approaches are proposed. For conciseness, we only present the main ideas and comparisons between those techniques instead of giving precise algorithms, which can be found in [LAL 15].

7.6.1.1. *Naive approach*

[WIL 85] explains the natural way of computing the TDV of a term t_i. We need to extract two densities: the first one sums the similarities of all pairs of documents in the collection, while the second one computes this density after removing t_i from all documents. The TDV is the difference between two densities, with and without the given term. The TDV $\Delta(\mathcal{I})$ represents the relationship between documents $\mathcal{I}$.

The complexity of the algorithm is highly dependent on the number of terms N and the number of documents M: $2M^2N^2 + 2M^2N$.

7.6.1.2. *Centroid method*

Then, [WIL 85] proposes a simplified method by creating a centroid of the documents in the collection. This center of gravity is denoted by $\mathcal{G} = (G_1, ..., G_k, ...G_N)$, where G_k with w_{ik} (the weight of term k in the i^n document) is defined by:

$$G_k = \frac{\sum_{i=1}^{M} w_{ik}}{M}$$

For each term, this centroid provides the average occurrence. Then the TDV of a single term is computed by the similarity between each document and the centroid instead of each document of the collection. The complexity is then: $2MN^2 + 3MN$. We can notice that we save the combination between each pair of documents, but the TDV remains dependent on similarity computations between documents and the centroid.

7.6.1.3. *Cluster concept covering Method (C^{3}M)*

Another approach [CAN 90] proposes to compute TDV with a clustering method. C^{3}M algorithms place two documents in the same cluster if they are likely to answer the same query. The algorithm chooses a number of clusters by creating the probabilities to select the correlation between a term and an item.

The probabilities are computed by three main components, namely α, β and δ. Consider a matrix of term weights w_{ij} in all documents, where i is the term and j is the document. We compute α_i as the sum of weights for each term and β_j the weights for each document. Then, we produce a matrix of document term correlations $N \times N$. Each cell is the probability $delta_{ik}$ that shows how much the couple (i, k) is similar. This similarity is given by coupling weights and normalizing with the sum of weights α and β:

$$\alpha_i = \frac{1}{\sum_{i=1}^{N} w_i} \quad \beta_j = \frac{1}{\sum_{j=1}^{M} w_j} \quad \delta_{ik} = \alpha_i \times \sum_{l=1}^{N} w_{il} \times w_{kl} \times \beta_l$$

An interesting property is extracted from the diagonal of this matrix. In fact, $\delta_k k$ is the coupling similarity between a document and itself, and it therefore shows how much a document k is dissimilar from others.

Thus, the number of clusters $\mathcal{C}$ is the sum of the diagonal probabilities δ_{kk}. The relationship is: the more the clusters, the more the dissimilar documents.

The TDV is then the difference between the number of clusters $\mathcal{C}$ with term t_i and the number of clusters $\mathcal{C}_i$ without term t_i. The main idea is that the number of clusters is inversely proportional to the density of a collection. Therefore, the number of clusters and TDV are given by the following formulas:

$$\mathcal{C} = \sum_{k=1}^{M} \delta_{kk} \quad tdv(i) = \mathcal{C} - \mathcal{C}_i$$

The production of TDV is then extremely simplified, and thus, the complexity is reduced to: $MN^2 + 3MN$. Even if we earn comparisons between terms of documents, we must notice that this clustering method provides approximate results. In fact, this solution focuses only on documents in the same cluster, and not all the documents.

7.6.1.4. *Comparison of approaches*

The different complexities are resumed in the first column of Table 7.2, where N is the number of terms and M the number of items. We can notice that every technique is characterized by the correlation between documents and similarities between them (MN^2), even if the optimized techniques try to reduce this step. Specific gains are found in the following step by combining the matrices or resuming the contents of documents, which leads to the reduction of the second step.

	Static Approach	**Incremental Approach**
`Naive method`	$2M^2N^2 + 2M^2N$	$2MN^2 + 2MN$
`Centroid method`	$2MN^2 + 3MN$	$2MN^2 + 2MN + N$
`Clustering method`	$MN^2 + 3MN$	$2MN^2 + MN + 2N$

Table 7.2. *Complexities of the different methods*

7.6.2. *Incremental approach*

The above-mentioned techniques are dedicated to computing a whole static collection of documents. However, to follow its evolution in our highly dynamic context, we need to study an incremental way to compute TDV. To achieve this, we extract the minimum information to compute for a new incoming item (or set of items). In order to keep a constant size for the collection, we remove the oldest item for each add-on.

In the preceding techniques, each new item I_{new} leads to:

– The `naive method` which requires the computation of the similarities of the incoming items with the whole collection and deduces it with the old computed density:

$$\Delta_{new} = \Delta_{old} + \sum_{i=1}^{M} sim(d_i, I_{new})$$

with a complexity of $2MN^2 + 2MN$, the first scan of the collection M being saved.

– The `centroid method` which must update every changing term of I_{new} in centroid c_{new}. Then we compute the similarity of each document with c_{new}:

$$\Delta_{new} = \sum_{i=1}^{M} sim(d_i, c_{new})$$

with a complexity of $2MN^2 + 2MN + N$; a small gain is obtained since the centroid must be updated and recomputed.

– The `clustering method` which requires the recomputation of the number of clusters on each dimension, thus it must update probabilities. Those updates require an update of each probability in the matrix, focused on the incoming item, and then a recomputation of the number of clusters. This leads to a complexity of $2MN^2 + MN + 2N$, where updates cost more than computing the whole collection from the beginning.

7.6.2.1. *Comparison*

All the complexities from the four techniques in static and incremental approaches are resumed in Table 7.2. It summarizes in the first column the complexity of the TDV computation for all terms of a vocabulary, and second, its computation in an incremental way. We must notice that even if each dimension is huge, the number of items M remains the most critical one (it can be more than 100 times larger).

The naive method complexity is caused by density computation, which must be done N+1 times, whereas the centroid one is done with a unique vector. According to the clustering method, time is spent to compute the matrix once, and then three computations are done on this matrix. By taking into account the fact that the vocabulary evolves far less than the number of items, we can conclude that the naive method is not realistic. However, two others remain extremely time consuming.

7.6.3. *TDV in a distributed environment*

The last step of TDV update enhancement is the integration of those algorithms in a NoSQL database. Since the calculation of TDVs is very computational, we need to distribute it in a distributed environment in order to scale it up. To achieve this, we need to model each needed structure in the incremental approaches. First, data structures must be defined, and then algorithms developed in Map/Reduce functions.

Documents in the collection are very simple JSON documents in which term identifier tID and initial weights w are given:

```
1 {"docID":1, "terms":[{"tID":0,"w":0.33},{"tID":5,"w":0.33},
2                      {"tID":12,"w":0.33}]}
```

Adapting these algorithms requires computing several steps of parameters, matrices and constants. Moreover, in a distributed environment, we need to take care of the number and types of updates (time consuming in NoSQL), how distributed is the computation, and how to merge results before the Reduce function. Thus, we will compare the three approaches in incremental modes in a distributed environment.

7.6.3.1. *Naive method*

To compute the naive method in a distributed environment, we need to compute all the similarities between all the items. This approach is not realistic; in fact, NoSQL databases do not provide "join" operations, which are necessary to compute pairs, and making M queries on a collection is not a proper solution. Thus, a two-step algorithm is necessary to achieve it, and it is illustrated in Map/Reduce language in Figure 7.2.

The first step focuses on the term dimension (N), which is lower than the item one, as in [ELS 08] with similarity computations with Map/Reduce. It executes one query that creates for each term a list of (doc,weight). The second step computes the result of the previous step by summing the term-pair similarities for each term. The result of the previous step is the opposite of TDV, and a simple query gives the TDV of each term of the collection. The similarity is here simplified to the sum of common weighted terms between documents.

First step

```
map(){
  for(t in terms)
    emit(t.tID, {"docId":docId,
      "w":t.w});
}
```

```
reduce (key, values){
  return {"tID":key, "docs":values};
}
```

Second step

```
map(){
  for(d1 in docs)
    for(d2 in docs)
      if(d1.docID < d2.docID)
        emit(tID, d1.w*d2*w);
}
```

```
reduce (key, values){
  sum = 0;
  for(v in values)
    sum += v;
  return sum;
}
```

Figure 7.2. *Map/Reduce steps for the Naive Method*

We can see that huge lists of documents and weights are produced and aggregated for each term. Once the result of this step is computed and stored on several servers, the second step is computed with it. It provides for each pair of documents the sum of term similarities, and then computes the density (with and without the term). The obtained TDV is computed in a simpler way by computing only the sum of term pairs between documents. Therefore, TDVs will be flattened in comparison to the standard values. However, the computation of these values is far more efficient in this context.

The incremental approach adds or removes term weights for every term of the stored collection obtained in the first step. Then, it continues by: i) removing the first step; ii) adding term weights from the new item and removing the oldest one (two update queries on all term documents); and iii) computing the second step.

7.6.3.2. *Centroid method*

To compute the centroid method in a distributed environment, we need to compute the centroid on all the stored items. To achieve this, a two-step Map/Reduce process is defined (Figure 7.3).

The first step computes the centroid by summing the terms weight from all the items, creating a $centroidVector$ in the Reduce function, where each key "tID" is the sum of weights in the collection for this term. Once $centroidVector$ is obtained, the second step computes the similarities between all documents with this centroid. Each Map function produces a similarity with the centroid for a given term, after removing the term from the document (terms - t). The Reduce function sums the similarities of a key "tID"

to produce the density of the collection for this term. The result is the density Δ (`null` key) and every terms' (`tID`) density Δ_i. Recall that $tdv(i) = \Delta_i - \Delta$.

First step

```
map(){
  for(t in terms)
    emit(t.tID, t.w);
}
```

```
reduce (key, values){
  sum = 0;
  for(v in values) sum += v;
  return sum;
}
```

Second step

```
map(){
  centroid = [centroidVector];
  emit(null, sim(terms, centroid));
  for(t in terms)
    emit(t.tID, sim(terms-t, centroid));
}
```

```
reduce (key, values){
  sum = 0
  for(v in values) sum += v;
  return sum;
}
```

Figure 7.3. *Map/Reduce steps for the Centroid Method*

The incremental approach modifies the previous steps by: i) removing the first step, ii) adding the incoming item and removing the oldest one, iii) updating the centroid directly in main memory (no space consumption) and iv) computing the second step.

7.6.3.3. *Clustering method*

The distributed clustering method must compute the three components to define the number of clusters: α (document vector), β (term vector) and δ (dissimilarity matrix's diagonal). A two-step algorithm is then necessary as shown in Figure 7.4.

The first step focuses on α and β computation. In the Map function, we must distinguish types of computations by creating a specific key: type (α, β) and value (termID and docID). The Reduce functions then aggregate the values and produce all the information as output. Both vectors are recomposed locally.

The second step is optimized to compute either δ and every δ_i (without terms) to produce the number of clusters $\mathcal{C}$ and $\mathcal{C}_i$. The Map function produces an array of sums of term weights combined with β for all the terms. The Reduce function makes for each termID the sum of all δ_i, combined with α to give the final number of clusters. Final TDV are computed locally by making the difference between $\mathcal{C}$ (key "null") and every $\mathcal{C}_i$. The drawback of

this approach is to send very huge vectors (α and β) in the query to all the servers, which increases the network communication cost.

First step

```
map(){
  var alpha = 0;
  for(t in terms){
    alpha += t.w;
    emit({"t":"beta","v":t.tID}, t.w});
  }
  emit({"t":"alpha","v":docID}, alpha});
}
```

```
reduce (key, values){
  sum = 0;
  for(v in values)
    sum += v;
  return sum;
}
```

Second step

```
map(){
  sum = [];
  beta = [betaVector];
  for(t in terms){
    sum[0] += t.w ** 2 / beta[t.tID];
    sum[t.tID] += t.w ** 2 / beta[t.tID];
  }
  for(key => value in sum)
    if(key != 0)emit(key,sum[0]-sum[key]);
    else        emit(null,sum[0]);
}
```

```
reduce (key, values){
  sum = 0;
  alpha = [alphaVector];
  for(v in values)
    sum += v;
  return sum / alpha[key];
}
```

Figure 7.4. *Map/Reduce steps for the Clustering Method*

The incremental adaptation of this distributed algorithm requires updating α, β and δ. To achieve this, we: i) remove the first step, ii) update α and β locally and add the new item in the collection and iii) process the second step.

7.7. Experiments

In this section, we study the behavior of TDV computation, the filtering system in both centralized and NoSQL environments. We will also show the impact of several parameters (i.e. novelty threshold, diversity and size of the sliding window) with a real dataset of items. Finally, thanks to a user validation, we study the quality of our system with different settings and a periodic filtering based on a top-k approach.

7.7.1. *Implementation and description of datasets*

For our experiments, we used a subset from a real dataset of items acquired over an 8-month campaign from March to October 2010 [TRA 14]. Subscriptions were generated by using the ALIAS sampling method [WAL 77]. It produced 10M subscriptions that follow the distribution of term occurrences on the Web, and the Web query size reported in [BEI 04], based on the vocabulary of 1.5M distinct terms extracted from items. It is characterized among others by a maximum size equal to 12 terms and on average 2.2 terms. We implemented the filtering system with the standard Java v1.6.0_20. All experiments were run on a 3.60 GHz quad-core processor with 16 GB JVM memory.

7.7.2. *TDV updates*

In order to evaluate TDV updates using a static and incremental approach in a NoSQL environment, we stored our collection of items in a MongoDB database. To simulate the incremental insert of items we measured the average time on the last 10,000 items. We made our experiments on a cluster of 16 servers and on a subset of our dataset (640k items). To optimize the *reduce* steps of our treatment, we needed to choose which information to *shard* (organize data on several servers). By looking at reduced keys that are used on the second steps, it groups on the tID. However, each item has several terms and we chose the most popular term of each item (after a local sort) to be the most selective one. Thus, items are organized according to their most popular terms in the collection.

Figure 7.5 plots the performances (average time to process an item in milliseconds in log scale) of each approach by varying the amount of treated items. We must notice that the naive static algorithm is not shown since it took more than 800s to process each item. We can see that, in a static mode, the clustering method obtains better results than the centroid one, but equivalently around 640k items. In fact, the centroid method is less dependent on the number of items to process since every item only processes the centroid, while the clustering method has to provide the β vector whose size is the number of items. For the incremental implementations, the naive method grows up too fast, while the clustering method requires the processing of β vectors. Finally, the incremental centroid method in a NoSQL environment remains constant for each item.

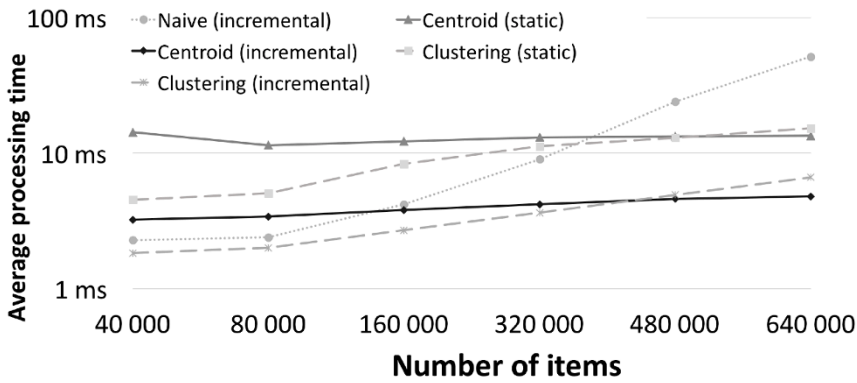


Figure 7.5. *TDV computation costs in a NoSQL environment. Static vs Incremental*

7.7.3. *Filtering rate*

In this section, we study the impact of the novelty threshold and diversity on the filtering rate, as well as of the number of subscriptions and the window size. The results presented in this section correspond to the average filtering rate (the number of notified items over the number of items that match the subscription) of the subscriptions satisfied at least once during the last day of the studied week.

7.7.3.1. *Impact of the novelty threshold*

Figure 7.6 shows the novelty's filtering rate when varying the novelty threshold for a window size of 24 hours (dashed lines). We observe that the filtering rate increases linearly with the novelty threshold. We notice that 38% of items are filtered when the novelty threshold is set to 50%, that is, when half of information is not redundant. We recall that item's novelty is based on its weighted coverage (definition 7.4). On average, only 20% of items that satisfy a subscription do not contain redundant information: 80% of the items are filtered out when the novelty threshold is equal to 100%.

7.7.3.2. *Impact of diversity*

Figure 7.6 also illustrates that filtering by diversity reduces the number of items to notify (solid line). Diversity acts as a strong filter since the filtering rate when considering only diversity (i.e. novelty threshold of 0%) is equal

to 64.34%. Figure 7.6 proves that novelty and diversity are complementary filters. It can be observed that the filtering rate slightly increases with the value of the novelty threshold if diversity is also considered (64.34% for a novelty threshold of 0% and up to 82% for a threshold of 100%). However, the benefit of having both filters is double since filtering by novelty further allows us to decrease the number of items to consider for the costly diversity computation.

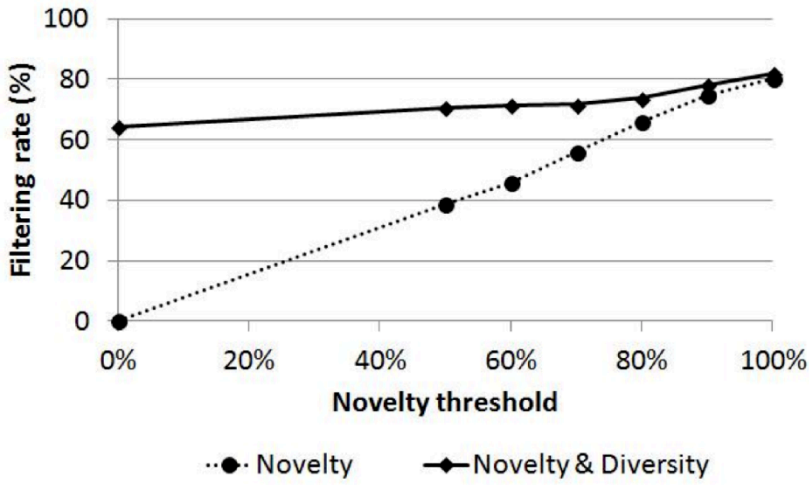


Figure 7.6. *Filtering rate by varying novelty threshold*

For the following experiments, we set the novelty threshold to 50% (best quality from Table 7.5) and take into account the diversity for the filtering process.

7.7.3.3. *Impact of window size*

Table 7.3 shows the filtering rate for different sliding window sizes. The size of the window affects the filtering rate: with a larger window size, the items stay longer in histories and are used to filter new items. Although larger sliding windows have an impact on the length of histories (see next section), items notified in large sliding windows stay for a long time in histories, but information remains diverse enough to generate new notifications. For the following experiments, the sliding window size is set to 24 hours.

window Size	Filtering Rate
12 H	61.06%
24 H	70.71%
48 H	75.93%

Table 7.3. *Filtering rate by varying window size*

7.7.3.4. *Impact of subscription size*

Table 7.4 presents, for each size of subscriptions, its distribution which follows the one from Web queries [BEI 04]. Most of the subscriptions are short (size less than 4). We can also note that the number of notified items by subscription decreases drastically with the subscription size: while short subscriptions are often matched (>500 items/day), large subscriptions are rarely notified (< 5 items/day).

$\|s\|$	# of subscriptions	Average # of statisfied items	Filtering rate
1	2 030 375	505.31	86.79%
2	1 804 265	21.94	57.71%
3	293 666	4.98	42.88%
>3	28 776	2.45	35.52%

Table 7.4. *Number of subscription & notifications w.r.t. subscription size*

According to our results, we can say that the filtering rate is highly dependent on diversity (present or not) as well as the novelty threshold. However, subscription size also has a significant impact.

7.7.3.5. *History size*

We capture the variation in history size over time. We get the average size every six hours over the studied week with three different sliding window sizes. Figure 7.7 shows this variation for a novelty threshold equal to 50%, the values presented are the average size of the history of subscriptions satisfied at least once in the first six hours of the week (3.35 million subscriptions). It should be noted that the size of the history at time τ is equal to the number of items in the window of the considered size p (items published after $\tau - p$). During the initialization phase, histories become larger with large sliding windows. The peak of each sliding window corresponds to the accumulation of items that

ends at window-size period (12h/24h/48h). The accumulation is due to the fact that empty histories do not play their filtering role. Indeed, since density keeps growing during this initialization step, there is almost no filtering by diversity and most items are notified. After this initialization period, the items which were greedily added to histories at the beginning go out the sliding window, which leads to a drastic decrease in the history size during one window-size period. First, items disappear which contributes to the gap between the peak and the depth exactly one period after the history size (12h/24h/48h). The same effect occurs with a lesser magnitude for the 12h and 48h sliding window-size; in fact a small sliding window empties quickly and must restart the density computation, while a large sliding window empties slowly and filters a lot. The 24h sliding window-size has a more stable behavior since it fills up and empties with an appropriate rate. The history now allows us to filter items by novelty and diversity and its size stabilizes. We also measured this variation with different novelty thresholds and confirmed these conclusions. The initialization phase corresponds to the diversification of histories.

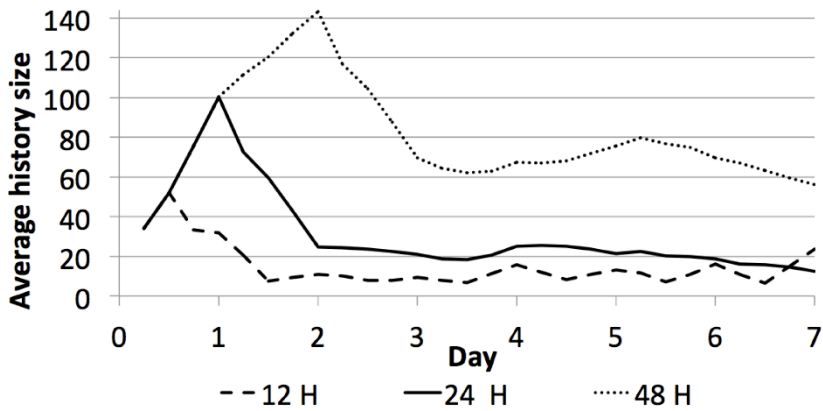


Figure 7.7. *Variation in history size over time*

Another conclusion from Figure 7.7 is that the history size is dependent on the window size. For 12h and 24h, the number of items in the history is globally equal to the window size (10/20), while the 48h sliding window-size is half more with 70. Even if old items contribute to diversifying the information, the filtering rate (Table 7.3) growth is not proportional to the window size, as with the history size which needs a greater number of items to filter diversity.

7.7.4. *Performance evaluation in the centralized environment*

As discussed in section 7.5.1.2, we present here three different implementations of our system: a NAÏVE approach without optimization, a CO-OCCURRENCE approach with the exploitation of the co-occurrence ratio σ of items, and a DIVERSITY approach that pre-computes and stores densities in every history.

7.7.4.1. *Memory requirements*

Since the CO-OCCURRENCE approach stores extra values only during the filtering process, the amount of space used by this implementation is equal to the NAÏVE implementation. Consequently, we present comparison only between the NAÏVE and the DIVERSITY implementations.

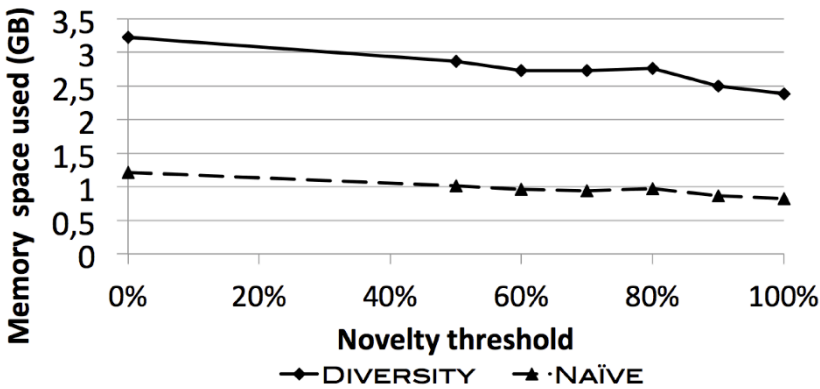


Figure 7.8. *Memory space versus novelty threshold*

Figure 7.8 shows the memory space used by the sliding window and subscription histories for various novelty thresholds. When the filtering rate is increasing, fewer items are stored in the sliding window; thus, it reduces memory consumption for both optimized and normal implementations. The DIVERSITY implementation requires more memory space since sums of distance scores are pre-computed and stored for each history. We observed that it requires consequently a memory space proportional to the size of histories, and so, inversely proportional to the filtering rate noticed in Figure 7.6. For instance, for a rate of 50%, we require 2.866MB of memory, while, for a rate of 100% (+16%), we require only 2.387 MB (–16.68%).

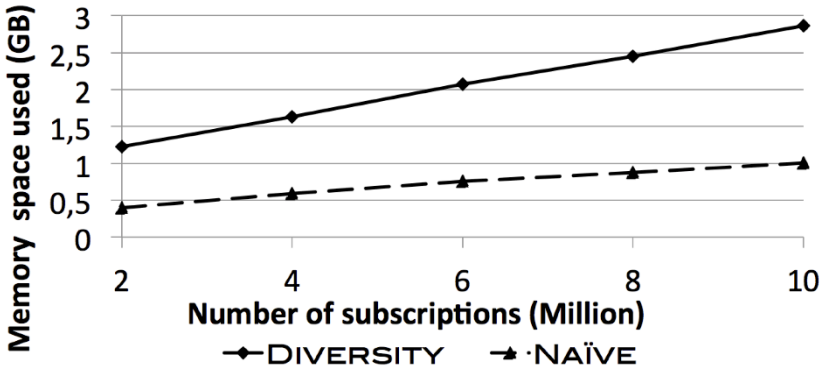


Figure 7.9. *Memory space versus number of subscriptions*

Figure 7.9 illustrates the variation in the memory consumption by varying the number of subscriptions. For this experiment, the filtering rate and the average sliding window size are fixed. We observe that the memory space increases linearly w.r.t. the number of subscriptions indexed in both implementations, since each history store information is linked to the sliding window. However, the DIVERSITY implementation requires more space to store extra-information compared with the NAÏVE version, but the ratio remains constant at 2.4. The NAÏVE implementation uses 399 MB (resp. 1009 MB), while the DIVERSITY optimization uses 1227 MB (resp. 2866 MB) for 2M (resp. 10M) of subscriptions.

7.7.4.2. *Processing time*

We now study the gain obtained by the optimizations of our system. Figure 7.10 shows that the average time (in log scale) decreases with the novelty threshold, and therefore, the history size. The NAÏVE implementation requires much more computing time especially for low novelty thresholds. The rationale lies in its CO-OCCURRENCE optimization, which reduces the number of similarities and distances computations. The NAÏVE implementation is on average five times more costly than the optimized ones, except for high thresholds where histories are short and few similarities/distances are computed. Moreover, the difference between CO-OCCURRENCE and DIVERSITY results decreases with the size of histories, which depends on the novelty threshold: the gain is 68% for a novelty threshold of 0%, and 13% for

a novelty threshold of 80%, due to the complexity of $O(1)$ (find $I_o.sum$) for the diversity computation.

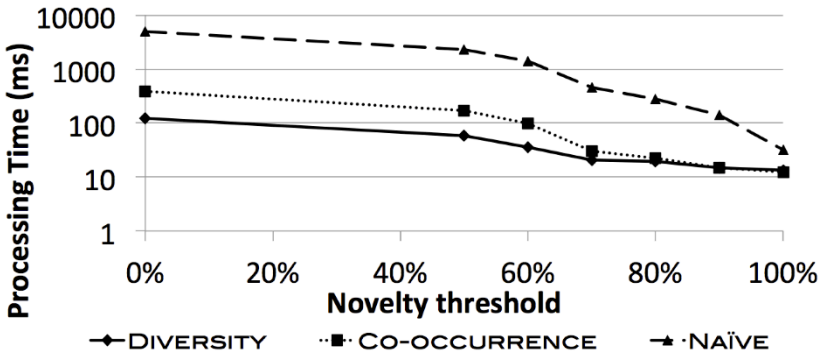


Figure 7.10. *Processing time when varying novelty threshold*

Since the processing time mainly relies on history size, it is also dependent on the sliding window size, especially for the NAÏVE and CO-OCCURRENCE implementations, where the growth of computation time is more important as shown in Figure 7.11. In fact, computation of diversity is dependent on sliding window size. On the other hand, the processing time for the DIVERSITY implementation exhibits a moderate increase, except for large window sizes (48h) where histories are larger, which means more distance computation and updates of sums. In fact, DIVERSITY stores sums of distances between items, while CO-OCCURRENCE implementation requires recomputing them. As larger windows filter more (Table 7.3), the distance computations are performed for each history, except for the DIVERSITY implementation which adds those values the first time an item is notified whatever the number of updates for histories is. On the other hand, the NAÏVE implementation computes distances each time, even if histories are updated. This requires 21–31 times more time than optimized solutions.

As we can see in Figure 7.12, the processing time increases linearly with the number of subscriptions for both optimizations, while the NAÏVE implementation increases very quickly since no co-occurrences between subscriptions are used. According to CO-OCCURRENCE and DIVERSITY implementations, it was expected to be sub-linear since similarity and distance computations between items are stored during the process to avoid its

recomputation. Therefore, with the growth of the number of subscriptions, the probability of having the same couple of items in different subscriptions increases. However, the gain for CO-OCCURRENCE is far more interesting (–93%) than DIVERSITY (–63%) since similarity and distance functions are very costly (compared with sums for diversities). Nevertheless, the DIVERSITY implementation needs 2.7 times less time on average than the CO-OCCURRENCE one.

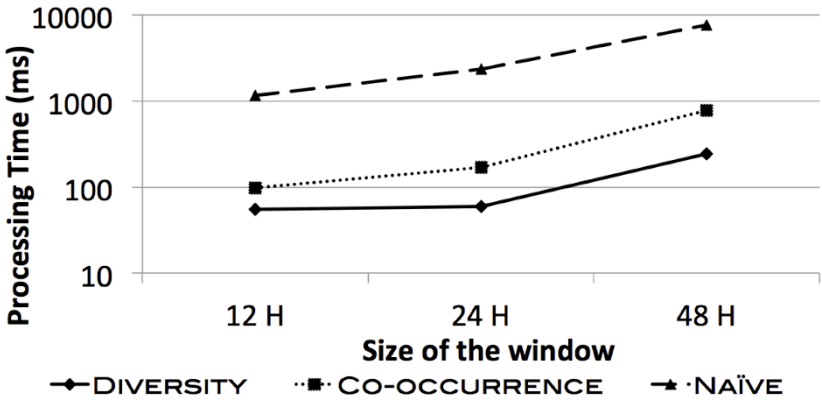


Figure 7.11. *Processing time for different sizes of sliding windows*

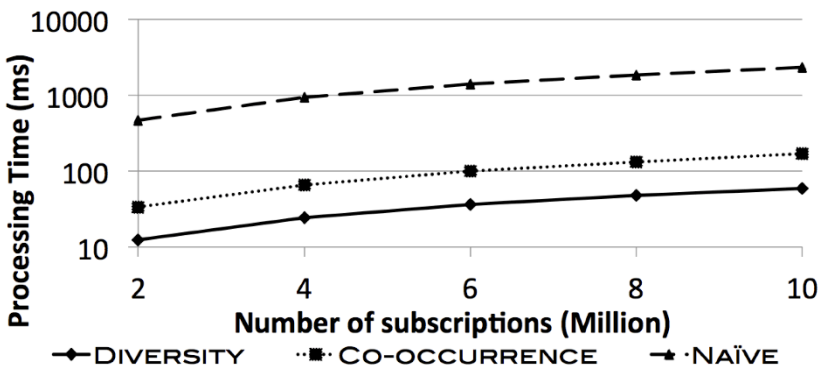


Figure 7.12. *Processing time by varying the number of subscriptions*

7.7.5. *Performance evaluation in a distributed environment*

We now study the effect of distributing our filtering process as proposed in section 7.5.2. Subscription- and item-based models are placed on a collection with 1M, 2M and 4M subscriptions with 10 days of history already processed. We then get the average processing time of items with 100k items. We plot the processing time by varying the distribution of the number of shards (servers).

Figure 7.13 plots the processing time of the subscription-based model for the matching process (dotted lines) and the total time with subscription updates (lines). The filtering process is efficient since it takes around 300 ms per item. However, we can notice that the update time is getting more and more important with the number of shards. In fact, the number of updates is spread all over the shards; since those updates are appended in blocks of data, new reallocation of blocks and distribution of data must be done. This time increases dramatically, which shows that this solution is not scalable.

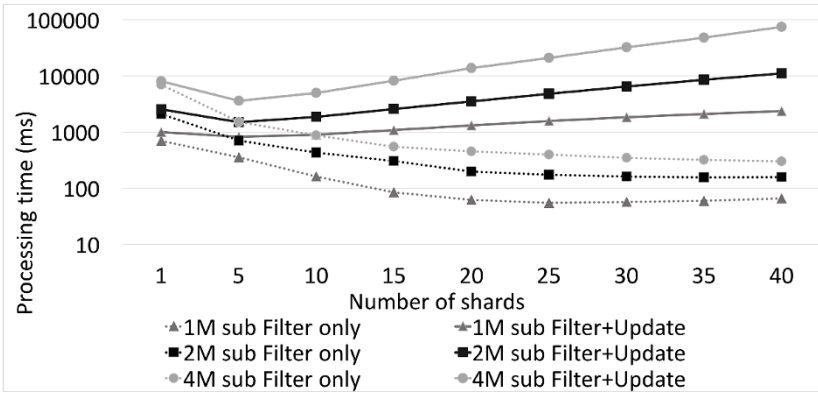


Figure 7.13. *Processing time for subscription-based model by varying the number of shards*

Figure 7.14 shows the processing time of the item-based model for both matching and total times. We can see that the update time is very low (between 50 and 70 ms); in fact, only one item is stored. This time increases since the number of subscriptions to be nested increases. According to the filtering time, it is longer than the subscription-based model (around 3 times more), but the

total time stabilizes after 20 shards for 4M subscriptions. This is due to the fact that the distribution of the process is maximized in connection with the filtering step (Map) and the shuffle step, which requires network communications.

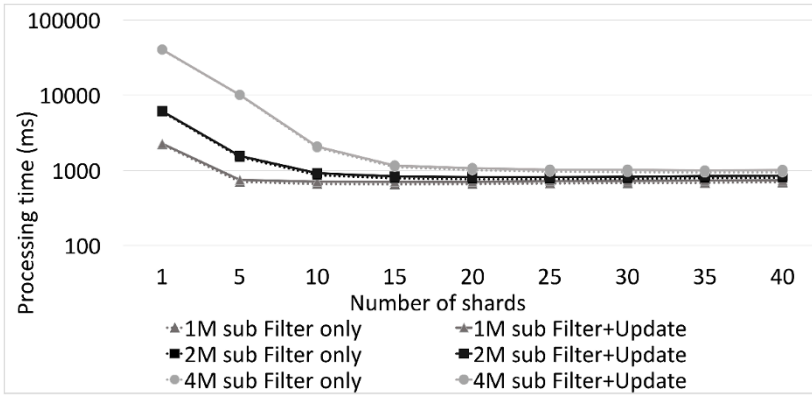


Figure 7.14. *Processing time for item-based model by varying the number of shards*

Despite good response times for the item-based model, it remains 50 times higher than the centralized version. The factorization effect and the shared history management have a huge impact on the processing time, which cannot be done in a distributed context since data cannot be properly interconnected. However, we can see that, in Figure 7.14, the processing time reaches an asymptote (1000 ms) for any number of subscriptions. Consequently, with more subscriptions, the difference in time between the distributed and centralized versions will decrease, and the centralized system will not fit in the main memory with a huge number of subscriptions (see Figure 7.9).

By extrapolating our results, the centralized version (Figure 7.12) should reach the threshold of 1000 ms of the processing time with 400M more subscriptions, which would require more than 40 GB of main memory. In a distributed context, it should require about 120 shards to scale up (Figure 7.14). The point of intersection of the curves can be considered as the threshold of Big Data (volume) in this context.

7.7.6. *Quality of filtering*

In this section, we study the quality of our filtering step with users' behavior. To compute the relevance of our system, we compare the chosen items by users and those obtained by our system. To validate our choice, we compare the quality of filtering when changing the weighting score, the novelty similarity and its threshold. We also compare our real-time filtering with a top-k algorithm [DRO 09b].

To achieve this, we have extracted 10 subscriptions on which we gathered matched items. Then, we asked users to manually filter items according to novelty and diversity of information. Users had to read texts and decide if an item is new or if its information is globally contained in previous items. In order to preserve our context of real-time filtering, items were displayed in sequence in order to filter them in chronological order and histories were shown to users. Here, 60 users performed 106 validations on our subscriptions. These users were academics and PhD students in computer science. Since filtering out by novelty is more trivial than that by diversity, we kept items in the result set only if they were chosen by more than 60% of the users (75% for novelty), giving more weight to diversity.

The top-k algorithm [DRO 09b] determines the k most distant items from a set of items satisfying subscriptions to achieve diversity. A result set is initialized with the two most distant items among the items satisfying the subscription and extended with the next most diversifying items. Each subscription has its own value k, which is equal to the history size generated by our approach. Having the same size will allow us to make result sets comparable for quality measurement. Moreover, this algorithm cannot take into account the novelty since it is an asymmetric measure based on time. We must recall that our window-based approach relies on the time assumption, which means that none of the notified items can be removed from the result set, while the top-k algorithm could put an old item in the new snapshot.

Table 7.5 shows the average precision, recall and F-Measure for all the subscriptions compared with the user result set. Different settings on our system have been made to find the most relevant measure for our filtering step: diversity without novelty, the top-k approach result set, different thresholds for the weighted coverage with diversity, changing TDV by the standard TF-IDF with and without novelty and, finally, novelty computed by the Jaccard

distance. We compare especially our novelty measure by weighted coverage (Definition 7.4) with the standard Jaccard similarity for different thresholds, and also the relevance of our term weight TDV versus TF-IDF, either in diversity or novelty. We also study the behavior of the top-k algorithm.

	Diversity only	top-k	Coverage 25%	Coverage 50%	Coverage 75%	TF-IDF	TF-IDF 50%	Jaccard 50%
Precision	0.782	0.711	0.930	0.939	0.944	0.764	0.884	0.916
Recall	0.698	0.634	0.652	0.652	0.610	0.618	0.545	0.652
F-Measure	0.726	0.660	0.732	0.736	0.710	0.626	0.646	0.729

Table 7.5. *Filtering relevance with various techniques, thresholds and metrics*

We can see that a combination of diversity and novelty produces better results than diversity alone, especially for the precision of the result. However, the result set recall decreases when using the novelty, which can be too selective and not diverse enough. As expected, TF-IDF weights cannot have a good impact on measures since items are short, so the TF is low and only IDF is taken into account. With a low precision (0.884) and recall (0.545), it gives the lowest F-Measure of our tests. According to novelty, the effect of the asymmetric measure and lack of weights for terms makes the Jaccard measure less relevant to the precision of the result set. Finally, the top-k technique is not as relevant as our solution since using an interchange algorithm to choose the most diverse items does not rely on a real-time assumption as for user validation. The relevance of our technique with a real-time filtering system using a TDV-weighted coverage measure for novelty with a threshold of 50% gives a good accuracy.

7.8. Conclusion

In this chapter, we presented a Pub/Sub system, which filters by novelty and diversity on the fly. The filtering is based on items already notified to a user. We chose a sliding window based on time to manage the subscriptions history. Our main contributions are (a) the proposition of the TDV to weight terms, combined with (b) a weighted coverage measure for novelty which is asymmetric and adapted to small items, (c) designing an optimized system

which factorizes similarities and distances, and reduces diversity computation costs, (d) a distributed implementation of our filtering process, (e) a distributed and incremental implementation of TDV updates computation and (f) a quality measurement of our propositions with a user validation based on real-time filtering with novelty and diversity.

From our experimental study, we show that novelty and diversity are complementary filters. Moreover, we observe that the filtering rate depends on novelty threshold and on window size, and diversity has less effect for a large window size. The performances of our system are also studied and we obtain an average gain of 97% in the processing time with our optimization for factorizing co-occurrences and computing the density of history. The distributed implementation of the filtering process is efficient with an item-based modeling but with a very high number of subscriptions (about 400M). We compare the quality of our system with different settings and a top-k and show that real-time delivery is a strong constraint which our system guarantees with a TDV-weighted coverage combined with diversity.

For further work, we aim to tune the quality of the diversity measure since cosine and Euclidean do not focus on the same kind of filtering. Another track is to integrate correlations between users in order to integrate collaborative filtering such that the user can filter items by interests thanks to other users.

7.9. Bibliography

[ABB 13] ABBAR S., AMER-YAHIA S., INDYK P. *et al.*, "Real-time recommendation of diverse related articles", *WWW*, pp. 1–12, 2013.

[ANG 11] ANGEL A., KOUDAS N., "Efficient diversity-aware search", *SIGMOD'11*, ACM, pp. 781–792, Athens, Greece, June 2011.

[BAE 99] BAEZA-YATES R.A., RIBEIRO-NETO B.A., *Modern Information Retrieval*, ACM Press / Addison-Wesley, New York, 1999.

[BAV 10] BAVI V., BEIRNE T., BONE N. *et al.*, Comparison of document similarity metrics, Western Washington University Information Retrieval, 2010.

[BEI 04] BEITZEL S.M., JENSEN E.C., CHOWDHURY A. *et al.*, "Hourly analysis of a very large topically categorized web query log", *SIGIR'04*, pp. 321–328, 2004.

[BOO 74] BOOKSTEIN A., SWANSON D., "Probabilistic Models for Automatic Indexing", *Journal American Society for Information Science*, vol. 25, no. 5, pp. 312–318, 1974.

[CAN 90] CAN F., OZKARAHAN E.A., "Concepts and effectiveness of the cover-coefficient-based clustering methodology for text databases", *TDS*, vol. 15, no. 4, pp. 483–517, 1990.

[CAR 01] CARZANIGA A., ROSENBLUM D.S., WOLF A.L., "Design and evaluation of a wide-area event notification service", *ACM TOCS*, vol. 19, no. 3, pp. 332–383, 2001.

[CAR 10] CARMEL D., ROITMAN H., YOM-TOV E., "On the relationship between novelty and popularity of user-generated content", *CIKM*, pp. 1509–1512, 2010.

[CAR 13] CARLSON J.L., *Redis in Action*, Manning Publications Co., Greenwich, 2013.

[CLA 08] CLARKE C.L., KOLLA M., CORMACK G.V. *et al.*, "Novelty and diversity in information retrieval evaluation", *SIGIR*, ACM, pp. 659–666, 2008.

[DRO 09a] DROSOU M., PITOURA E., "Diversity over Continuous Data", *IEEE Data Engineering Bulletin*, vol. 32, no. 4, pp. 49–56, 2009.

[DRO 09b] DROSOU M., STEFANIDIS K., PITOURA E., "Preference-aware publish/subscribe delivery with diversity", *DEBS*, ACM, pp. 1–12, 2009.

[DRO 12a] DROSOU M., PITOURA E., "DisC diversity: result diversification based on dissimilarity and coverage", *VLDB*, vol. 6, no. 1, pp. 13–24, 2012.

[DRO 12b] DROSOU M., PITOURA E., "Dynamic diversification of continuous data", *EDBT*, ACM, pp. 216–227, 2012.

[EIS 00] EISENHAUER G., BUSTAMANTE F.E., SCHWAN K., "Event services for high performance computing", *HPDC*, pp. 113–120, 2000.

[ELS 08] ELSAYED T., LIN J., OARD D., "Pairwise Document Similarity in Large Collections with MapReduce", *ACL*, pp. 265–268, 2008.

[GAB 04] GABRILOVICH E., DUMAIS S., HORVITZ E., "Newsjunkie: Providing Personalized Newsfeeds via Analysis of Information Novelty", *WWW*, pp. 482–490, 2004.

[GRE 07] GREGORIO J., DE HORA B., Atom: The Atom Publishing Protocol, NewBay Software, 2007.

[HME 11] HMEDEH Z., VOUZOUKIDOU N., TRAVERS N. *et al.*, "Characterizing web syndication behavior and content", *WISE*, pp. 29–42, 2011.

[HME 12] HMEDEH Z., KOURDOUNAKIS H., CHRISTOPHIDES V. *et al.*, "Subscription indexes for web syndication systems", *EDBT*, ACM, 2012.

[HME 15] HMEDEH Z., DU MOUZA C., TRAVERS N., "A real-time filtering by novelty and diversity for publish/subscribe systems", *SSDBM*, pp. 1–4, 2015.

[HME 16] HMEDEH Z., KOURDOUNAKIS H., CHRISTOPHIDES V. *et al.*, "Content-based publish/subscribe system for web syndication", *JCST*, vol. 31, no. 2, pp. 357–378, 2016.

[KEI 12] KEIKHA M., CRESTANI F., CROFT W.B., "Diversity in blog feed retrieval", *CIKM*, pp. 525–534, 2012.

[LAL 15] LALAOUI O., Efficient TDV computation in no SQL environment, Master's thesis, Vertigo team - CEDRIC Laboratory, Conservatoire national des arts et Métiers, Paris, 2015.

[MIN 11] MINACK E., SIBERSKI W., NEJDL W., "Incremental diversification for very large sets: a streaming-based approach", *SIGIR*, ACM, pp. 585–594, 2011.

[PAN 12] PANIGRAHI D., DAS SARMA A., AGGARWAL G. *et al.*, "Online selection of diverse results", *WSDM*, ACM, pp. 263–272, 2012.

[PRI 08] PRIPUŽIĆ K., ŽARKO I.P., ABERER K., "Top-k/w publish/subscribe: finding k most relevant publications in sliding time window w", *DEBS*, ACM, pp. 127–138, 2008.

[ROW 01] ROWSTRON A.I.T., KERMARREC A.-M., CASTRO M. *et al.*, "SCRIBE: the design of a large-scale event notification infrastructure", *NGC*, pp. 30–43, 2001.

[RSS 03] RSS, RSS 2.0: really simple syndication, Berkman Center for Internet and Society at Harvard Law School. http://www.rssboard.org/rss-specification, 2003.

[SAL 75] SALTON G., WONG A., YANG C.S., "A vector space model for automatic indexing", *ACM Journal*, vol. 18, no. 11, pp. 613–620, 1975.

[SMY 01] SMYTH B., MCCLAVE P., "Similarity vs. Diversity", *CBR*, pp. 347–361, 2001.

[TRA 14] TRAVERS N., HMEDEH Z., VOUZOUKIDOU N. *et al.*, "RSS feeds behavior analysis, structure and vocabulary", *IJWIS*, vol. 10, no. 3, pp. 291–320, 2014.

[WAL 77] WALKER A., "An efficient method for generating discrete random variables with general distributions", *TOMS*, vol. 3, pp. 253–256, 1977.

[WIL 85] WILLETT P., "An algorithm for the calculation of exact term discrimination values", *Information Processing & Management*, vol. 21, no. 3, pp. 225–232, 1985.

[YU 09] YU C., LAKSHMANAN L., AMER-YAHIA S., "It takes variety to make a world: diversification in recommender systems", *EDBT*, ACM, pp. 368–378, 2009.

[ZHA 02] ZHANG Y., CALLAN J., MINKA T., "Novelty and redundancy detection in adaptive filtering", *SIGIR*, ACM, pp. 81–88, 2002.

List of Authors

Manel ACHICHI
LIRMM, CNRS
University of Montpellier
France

Bernd AMANN
LIP6, CNRS
Sorbonne University
Paris
France

Zohra BELLAHSENE
LIRMM, CNRS
University of Montpellier
France

Mohamed BEN ELLEFI
LIRMM, CNRS
University of Montpellier
France

Arnaud CASTELLTORT
LIRMM, CNRS
University of Montpellier
France

Olivier CURÉ
LIGM, CNRS
University of Marne-la-Vallée
France

Bruno DEFUDE
SAMOVAR, CNRS
Telecom SudParis
Évry
France

Cédric DU MOUZA
CEDRIC, CNAM
Paris
France

Dino IENCO
TETIS, IRSTEA, CNRS
Montpellier
France

Vijay INGALALLI
LIRMM, CNRS
University of Montpellier
France

NoSQL Data Models: Trends and Challenges, First Edition. Edited by Olivier Pivert.

Anne LAURENT
LIRMM, CNRS
University of Montpellier
France

Dominique LAURENT
ETIS, CNRS
Cergy-Pontoise University
France

Hubert NAACKE
LIP6, CNRS
Sorbonne University
Paris
France

Kim NGUYỄN
LRI, CNRS
Paris-Sud University
Orsay
France

Olivier PIVERT
IRISA
University of Rennes
Lannion
France

Pascal PONCELET
LIRMM, CNRS
University of Montpellier
France

Rami SELLAMI
CETIC
Charleroi
Belgium

Olfa SLAMA
IRISA
University of Rennes
Lannion
France

Virginie THION
IRISA
University of Rennes
Lannion
France

Konstantin TODOROV
LIRMM, CNRS
University of Montpellier
France

Nicolas TRAVERS
CEDRIC, CNAM
Paris
France

Index

NoSQL Data Models: Trends and Challenges, First Edition. Edited by Olivier Pivert.

H, I, J

K, L, M

N, O, P

Q, R, S

T, U, V

W, X

Other titles from

in

Computer Engineering

2018

ANDRO Mathieu
Digital Libraries and Crowdsourcing
(Digital Tools and Uses Set – Volume 5)

ARNALDI Bruno, GUITTON Pascal, MOREAU Guillaume
Virtual Reality and Augmented Reality: Myths and Realities

HOMAYOUNI S. Mahdi, FONTES Dalila B.M.M.
Metaheuristics for Maritime Operations
(Optimization Heuristics Set – Volume 1)

JEANSOULIN Robert
JavaScript and Open Data

SEDKAOUI Soraya
Data Analytics and Big Data

SZONIECKY Samuel
Ecosystems Knowledge: Modeling and Analysis Method for Information and Communication
(Digital Tools and Uses Set – Volume 6)

2017

BENMAMMAR Badr
Concurrent, Real-Time and Distributed Programming in Java

HÉLIODORE Frédéric, NAKIB Amir, ISMAIL Boussaad, OUCHRAA Salma, SCHMITT Laurent
Metaheuristics for Intelligent Electrical Networks
(Metaheuristics Set – Volume 10)

MA Haiping, SIMON Dan
Evolutionary Computation with Biogeography-based Optimization
(Metaheuristics Set – Volume 8)

PÉTROWSKI Alain, BEN-HAMIDA Sana
Evolutionary Algorithms
(Metaheuristics Set – Volume 9)

PAI G A Vijayalakshmi
Metaheuristics for Portfolio Optimization
(Metaheuristics Set – Volume 11)

2016

BLUM Christian, FESTA Paola
Metaheuristics for String Problems in Bio-informatics
(Metaheuristics Set – Volume 6)

DEROUSSI Laurent
Metaheuristics for Logistics
(Metaheuristics Set – Volume 4)

DHAENENS Clarisse and JOURDAN Laetitia
Metaheuristics for Big Data
(Metaheuristics Set – Volume 5)

LABADIE Nacima, PRINS Christian, PRODHON Caroline
Metaheuristics for Vehicle Routing Problems
(Metaheuristics Set – Volume 3)

LEROY Laure
Eyestrain Reduction in Stereoscopy

LUTTON Evelyne, PERROT Nathalie, TONDA Albert
Evolutionary Algorithms for Food Science and Technology
(Metaheuristics Set – Volume 7)

MAGOULÈS Frédéric, ZHAO Hai-Xiang
Data Mining and Machine Learning in Building Energy Analysis

RIGO Michel
Advanced Graph Theory and Combinatorics

2015

BARBIER Franck, RECOUSSINE Jean-Luc
COBOL Software Modernization: From Principles to Implementation with the BLU AGE® Method

CHEN Ken
Performance Evaluation by Simulation and Analysis with Applications to Computer Networks

CLERC Maurice
Guided Randomness in Optimization
(Metaheuristics Set – Volume 1)

DURAND Nicolas, GIANAZZA David, GOTTELAND Jean-Baptiste,
ALLIOT Jean-Marc
Metaheuristics for Air Traffic Management
(Metaheuristics Set – Volume 2)

MAGOULÈS Frédéric, ROUX François-Xavier, HOUZEAUX Guillaume
Parallel Scientific Computing

MUNEESAWANG Paisarn, YAMMEN Suchart
Visual Inspection Technology in the Hard Disk Drive Industry

2014

BOULANGER Jean-Louis
Formal Methods Applied to Industrial Complex Systems

BOULANGER Jean-Louis
Formal Methods Applied to Complex Systems:
Implementation of the B Method

GARDI Frédéric, BENOIST Thierry, DARLAY Julien, ESTELLON Bertrand, MEGEL Romain
Mathematical Programming Solver based on Local Search

KRICHEN Saoussen, CHAOUACHI Jouhaina
Graph-related Optimization and Decision Support Systems

LARRIEU Nicolas, VARET Antoine
Rapid Prototyping of Software for Avionics Systems: Model-oriented Approaches for Complex Systems Certification

OUSSALAH Mourad Chabane
Software Architecture 1
Software Architecture 2

PASCHOS Vangelis Th
Combinatorial Optimization – 3-volume series, 2nd Edition
Concepts of Combinatorial Optimization – Volume 1, 2nd Edition
Problems and New Approaches – Volume 2, 2nd Edition
Applications of Combinatorial Optimization – Volume 3, 2nd Edition

QUESNEL Flavien
Scheduling of Large-scale Virtualized Infrastructures: Toward Cooperative Management

RIGO Michel
Formal Languages, Automata and Numeration Systems 1: Introduction to Combinatorics on Words
Formal Languages, Automata and Numeration Systems 2: Applications to Recognizability and Decidability

SAINT-DIZIER Patrick
Musical Rhetoric: Foundations and Annotation Schemes

TOUATI Sid, DE DINECHIN Benoit
Advanced Backend Optimization

2013

ANDRÉ Etienne, SOULAT Romain
The Inverse Method: Parametric Verification of Real-time Embedded Systems

BOULANGER Jean-Louis
Safety Management for Software-based Equipment

DELAHAYE Daniel, PUECHMOREL Stéphane
Modeling and Optimization of Air Traffic

FRANCOPOULO Gil
LMF — Lexical Markup Framework

GHÉDIRA Khaled
Constraint Satisfaction Problems

ROCHANGE Christine, UHRIG Sascha, SAINRAT Pascal
Time-Predictable Architectures

WAHBI Mohamed
Algorithms and Ordering Heuristics for Distributed Constraint Satisfaction Problems

ZELM Martin *et al.*
Enterprise Interoperability

2012

ARBOLEDA Hugo, ROYER Jean-Claude
Model-Driven and Software Product Line Engineering

BLANCHET Gérard, DUPOUY Bertrand
Computer Architecture

BOULANGER Jean-Louis
Industrial Use of Formal Methods: Formal Verification

BOULANGER Jean-Louis
Formal Method: Industrial Use from Model to the Code

CALVARY Gaëlle, DELOT Thierry, SÈDES Florence, TIGLI Jean-Yves
Computer Science and Ambient Intelligence

MAHOUT Vincent
Assembly Language Programming: ARM Cortex-M3 2.0: Organization, Innovation and Territory

MARLET Renaud
Program Specialization

SOTO Maria, SEVAUX Marc, ROSSI André, LAURENT Johann
Memory Allocation Problems in Embedded Systems: Optimization Methods

2011

BICHOT Charles-Edmond, SIARRY Patrick
Graph Partitioning

BOULANGER Jean-Louis
Static Analysis of Software: The Abstract Interpretation

CAFERRA Ricardo
Logic for Computer Science and Artificial Intelligence

HOMES Bernard
Fundamentals of Software Testing

KORDON Fabrice, HADDAD Serge, PAUTET Laurent, PETRUCCI Laure
Distributed Systems: Design and Algorithms

KORDON Fabrice, HADDAD Serge, PAUTET Laurent, PETRUCCI Laure
Models and Analysis in Distributed Systems

LORCA Xavier
Tree-based Graph Partitioning Constraint

TRUCHET Charlotte, ASSAYAG Gerard
Constraint Programming in Music

VICAT-BLANC PRIMET Pascale *et al.*
Computing Networks: From Cluster to Cloud Computing

2010

AUDIBERT Pierre
Mathematics for Informatics and Computer Science

BABAU Jean-Philippe *et al.*
Model Driven Engineering for Distributed Real-Time Embedded Systems
2009

BOULANGER Jean-Louis
Safety of Computer Architectures

MONMARCHE Nicolas *et al.*
Artificial Ants

PANETTO Hervé, BOUDJLIDA Nacer
Interoperability for Enterprise Software and Applications 2010

SIGAUD Olivier *et al*.
Markov Decision Processes in Artificial Intelligence

SOLNON Christine
Ant Colony Optimization and Constraint Programming

AUBRUN Christophe, SIMON Daniel, SONG Ye-Qiong *et al.*
Co-design Approaches for Dependable Networked Control Systems

2009

FOURNIER Jean-Claude
Graph Theory and Applications

GUEDON Jeanpierre
The Mojette Transform / Theory and Applications

JARD Claude, ROUX Olivier
Communicating Embedded Systems / Software and Design

LECOUTRE Christophe
Constraint Networks / Targeting Simplicity for Techniques and Algorithms

2008

BANÂTRE Michel, MARRÓN Pedro José, OLLERO Hannibal, WOLITZ Adam
Cooperating Embedded Systems and Wireless Sensor Networks

MERZ Stephan, NAVET Nicolas
Modeling and Verification of Real-time Systems

PASCHOS Vangelis Th
Combinatorial Optimization and Theoretical Computer Science: Interfaces and Perspectives

WALDNER Jean-Baptiste
Nanocomputers and Swarm Intelligence

2007

BENHAMOU Frédéric, JUSSIEN Narendra, O'SULLIVAN Barry
Trends in Constraint Programming

JUSSIEN Narendra
A TO Z OF SUDOKU

2006

BABAU Jean-Philippe *et al.*
From MDD Concepts to Experiments and Illustrations – DRES 2006

HABRIAS Henri, FRAPPIER Marc
Software Specification Methods

MURAT Cecile, PASCHOS Vangelis Th
Probabilistic Combinatorial Optimization on Graphs

PANETTO Hervé, BOUDJLIDA Nacer
Interoperability for Enterprise Software and Applications 2006 / IFAC-IFIP I-ESA'2006

2005

GÉRARD Sébastien *et al.*
Model Driven Engineering for Distributed Real Time Embedded Systems

PANETTO Hervé
Interoperability of Enterprise Software and Applications 2005

WILEY END USER LICENSE AGREEMENT

Go to www.wiley.com/go/eula to access Wiley’s ebook EULA.

DATABASES AND BIG DATA SET

Coordinated by Dominique Laurent and Anne Laurent

The topic of NoSQL databases has emerged recently in the face of the challenge regarding Big Data – namely, the ever-increasing volume of data to be handled. It is now recognized that relational databases are not appropriate in this context, and thus new database models and techniques are necessary.

This book presents recent studies in the field, covering the following basic areas: semantic data management, graph databases and Big Data management in Cloud environments. The authors report on research regarding the evolution of basic concepts such as data models, query languages and new challenges related to issues of implementation.

Olivier Pivert is Full Professor of Computer Science at the École Nationale Supérieure des Sciences Appliquées et de Technologie (University of Rennes, France), and a member of the Institut de Recherche en Informatique et Systèmes Aléatoires (IRISA), where he leads the research team Shaman. He has published over 300 papers on database-related topics.

iSTE
www.iste.co.uk
WILEY
9 781786 303646