UNIVERSIDAD DE VALLADOLID

ESCUELA TÉCNICA SUPERIOR

INGENIEROS DE TELECOMUNICACIÓN

TRABAJO FIN DE MASTER

MASTER UNIVERSITARIO EN INVESTIGACIÓN

EN TECNOLOGÍAS DE LA INFORMACIÓN Y LAS COMUNICACIONES

Scalable RDF compression with MapReduce and HDT

Autor:

D. José Miguel Giménez García

Tutores:

Dr. D. Pablo de la Fuente
Dr. D. Miguel A. Martínez-Prieto
Dr. D. Javier D. Fernández

Valladolid, 11 de septiembre de 2015

TÍTULO:	**Scalable RDF compression with MapReduce and HDT**
AUTOR:	**D. José Miguel Giménez García**
TUTORES:	**Dr. D. Pablo de la Fuente**
	Dr. D. Miguel A. Martínez-Prieto
	Dr. D. Javier D. Fernández
DEPARTAMENTO:	**Departamento de Informática**

Tribunal

PRESIDENTE:	**Dr. D. Carlos Alonso González**
VOCAL:	**Dr. D. Mercedes Martínez**
SECRETARIO:	**Dr. D. Arturo González Escribano**
FECHA:	**11 de septiembre de 2015**
CALIFICACIÓN:	

Resumen del TFM

El uso de RDF para publicar datos semánticos se ha incrementado de forma notable en los últimos años. Hoy los datasets son tan grandes y están tan interconectados que su procesamiento presenta problemas de escalabilidad. HDT es una representación compacta de RDF que pretende minimizar el consumo de espacio a la vez que proporciona capacidades de consulta. No obstante, la generación de HDT a partir de formatos en texto de RDF es una tarea costosa en tiempo y recursos. Este trabajo estudia el uso de MapReduce, un framework para el procesamiento distribuido de grandes cantidades de datos, para la tarea de creación de estructuras HDT a partir de RDF, y analiza las mejoras obtenidas tanto en recursos como en tiempo frente a la creación de dichas estructuras en un proceso mono-nodo.

Palabras clave

Big Data, HDT, MapReduce, RDF, Web Semántica.

Abstract

The usage of RDF to expose semantic data has increased dramatically over the recent years. Nowadays, RDF datasets are so big and interconnected their management have significant scalability problems. HDT is a compact representation of RDF data aiming to minimize space consumption while providing retrieval features. Nonetheless, HDT generation from RDF traditional formats is expensive in terms of resources and processing time. This work introduces the usage of MapReduce, a framework for distributed processing of large data quantities, to serialize huge RDF into HDT, and analyzes the improvements in both time and resources against the prior mono-node processes.

Keywords

Big Data, HDT, MapReduce, RDF, Semantic Web

Agradecimientos

Me gustaría expresar mi agradecimiento a varias personas, sin las cuales este trabajo no hubiera sido posible, o hubiera sido muy diferente.

En primer lugar a mis tutores, Javier D. Fernández, Miguel A. Martínez-Prieto, y Pablo de la Fuente, por su ayuda e infinita pacienca durante la realización del trabajo.

A Javier I. Ramos, por su apoyo constante con el cluster Hadoop, en especial cuando hubo problemas que hacían que el sistema de virtualización fallase.

A Jurgen Umbrich, por prestar el servidor en el que se realizaron las pruebas mono-nodo de `hdt-lib`.

A Mercedes Martínez y Diego Llanos. Trabajos realizados en sus asignaturas sirvieron de inspiración para lo que más tarde se convirtió en parte de esta memoria.

A mí familia, que como siempre han estado dándome su apoyo, estuvieran o no de acuerdo con mis decisiones.

A mis amigos, que siempre me han dado ánimos para continuar.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Motivation

Since the end of the last century, the volume of data generated and stored across information systems has been growing at a dramatic rate. The estimated amount of data of the digital universe was 281 exabytes (281 billion gigabytes) in 2006 [42], and reached 1.8 zettabytes (1.8 trillion gigabytes) in 2011 [41], multiplying its volume by six in five years. In addition, most of these data, ranging from health records to twitter feeds, is not stored in structured format [9].

The so called *Data Deluge* affects a myriad of fields. In scientific research, large amounts of data are collected from simulations, experimental results, and even observation devices such as telescopes: The Murchison Widefield Array (MWA) —a radio telescope based in Western Australia's Mid West— is currently generating 400 megabytes of data per second (33 terabytes per day) [17]. Internet companies like Google, Facebook or Yahoo! have stored data measured in exabytes [111]. Public administrations have released millions of datasets to the public under the Open Government flag [55].

However, management of *Big Data* presents some challenges. The more important among them are related to what are commonly known as *"the three V's of Big Data"*:

- *Variety:* As said before, Big Data are generated by a multitude of sources. The structure of all this data is heterogeneous, and in many cases data are not even structured at all. Because of this, extracting information and establishing relations among data is difficult.
- *Volume:* Probably the most straightforward challenge, the sheer amount of data is a problem by itself. Storage, transmission or querying of information cannot be addressed by classical IT technologies.
- *Velocity:* Data are generated at an increasingly rapid rate. Processing the information quickly enough is an added issue when dealing with Big Data.

The *Semantic Web* [7] is a solution that attempts to deal with the challenge of *Variety*. In short, it is a proposal with the initial purpose of transforming the current web into a *Web of Data*, where semantic content is exposed and interlinked instead of plain documents. To allow a standardized way to represent semantic data, the Resource Description Framework (RDF) [4] is introduced by the W3C. RDF represents data in a labeled directed graph structure, enabling heterogeneous data to be linked in a uniform way, while endowing links with semantic meaning.

The Semantic Web has provided means for publication, exchange and consumption of *Big Semantic Data*. Since its inception, RDF usage has grown exponentially, with datasets spanning hundreds of millions triples. Nonetheless, managing those quantities of data is still a challenge.

While different approaches have been proposed, such as column-oriented databases [109], *Velocity* and *Volume* are still unresolved challenges.

To address the *Velocity* issues that the Semantic Web struggles with, it is necessary a way to transmit and query RDF in an efficient way. HDT is a storage proposal for RDF storage based on succinct data structures [36] to confront scalability problems in the publication, exchange and consumption cycle of Semantic Data. HDT is currently a W3C Member Submission[1]

RDF storage, transmission and querying requirements are alleviated by HDT. However, this comes with a price. Current methods of serializing RDF into HDT need to process the whole RDF dataset, demanding high amounts of memory in order to manage intermediate data structures. That is, scalability issues are moved from RDF consumption to HDT generation. *Volume* is then an issue that RDF serialization into HDT has still to face.

In this scenario, *MapReduce* emerges as a candidate for this process. In short, MapReduce is a framework for distributed processing of large amounts of data over commodity hardware. It is based on two main operations: `Map`, which gathers information items in the form of pairs key-value, and `Reduce`, which groups values with the same key. Originally developed by Google and made public in 2004 [27], its usage took off since then and it is currently widely used by many institutions and companies, such as Yahoo! [108], Facebook [117] or Twitter [75].

1.2 Goals

The main goal for this work is to *address the scalability issues of HDT serialization using the MapReduce framework, reducing hardware requirements to generate an HDT serialization from massive datasets or mashups*. In order to achieve this objective some intermediate or secondary goals are introduced. Those goals are sketched, along with a context overview, in Figure 1.1. They are also described below.

- Perform a study of existing MapReduce based solutions to RDF scalability issues.
- Devise a MapReduce based solution to HDT serialization scalability issues. This solution has to address the generation of the two serialized components of HDT, each one presenting a different challenge:
  - *Dictionary serialization.* Every triple term must be identified, segregated into sections according to its role, and assigned a unique and ascending numerical IDs. Here the challenge is to segregate and sort the terms, considering that terms which are subjects and objects simultaneously belong to a different section than terms that are just subjects or objects. This process is currently being done using temporal in-memory structures, and constitutes the main bottleneck of HDT serialization.
  - *Triples serialization.* Every triple term must be substituted by its ID. Then, each triple must be identified and sorted by subject, predicate and object. While here there are no hardware requirements that make the task unrealizable, speed can be improved by paralleling the operation.
- Develop an evaluation system for HDT generation scalability. This system will have to be applied to the devised solution and provide an unbiased assessment of its scalability, comparing it with previous mono-node versions.

[1] `http://www.w3.org/Submission/2011/03`

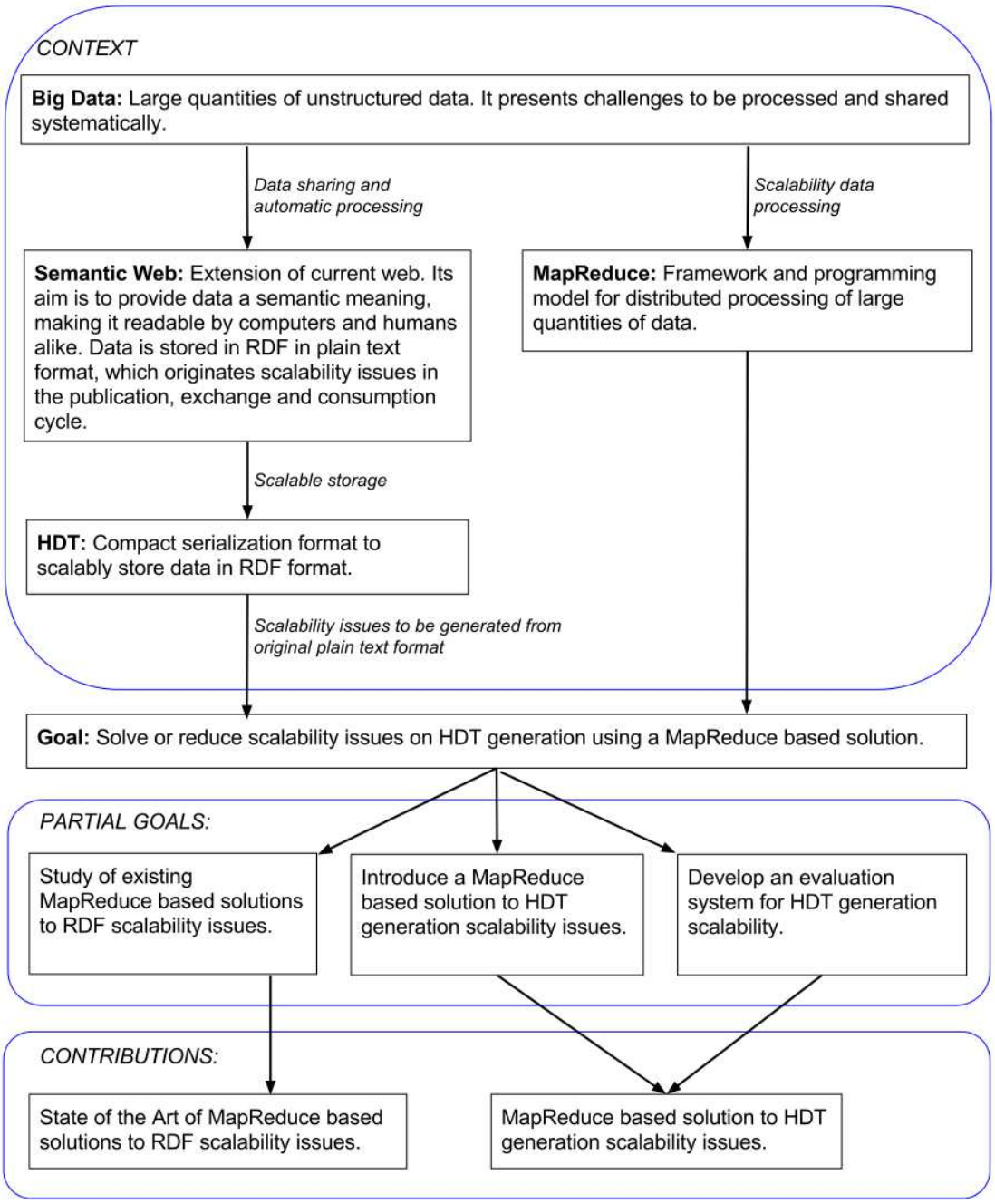


Figure 1.1: Goals overview

1.3 Methodology

This work proposes a novel approach to deal with HDT serialization, which is in itself a solution to deal with RDF scalability issues. This is a multidisciplinary work on the fields of Semantic Web, compression techniques, and parallel computation, so a preliminary study of all fields is very important. This allows to fully understand the subject and to take a grasp of useful techniques, possible issues, and solutions on applying the MapReduce model to deal with RDF and HDT structures. This is visible in the used methodology, which comprises the following steps:

1. Perform a state-of-the-art of Semantic Web, HDT and MapReduce, describing its main characteristics, strengths, weaknesses and current challenges and lines of research.
2. Analyze current MapReduce-based solutions on the Semantic Web regarding scalability issues.
3. Propose and develop a solution to HDT scalability issues. This solution will be based on the MapReduce computational model.
4. Evaluate the solution against the previous mono-node serialization algorithm, providing a fair assessment of the applicability of the devised solution.

1.4 Structure

The current chapter provides an overview of the problem this work addresses, its goals and methodology. The rest of this document is organized as follows:

- Chapter 2 describes the background regarding the Semantic Web, HDT and MapReduce.
- Chapter 3 reviews the most relevant MapReduce solutions to a variety of RDF scalability issues, not only including compression, but also query resolution and inference issues.
- Chapter 4 describes the proposed solution to generate HDT serialization. The evaluation method for this solution is also defined.
- Chapter 5 demonstrates the experiments and their results with real-world and synthetic datasets.
- Chapter 6 gives some conclusions about the work done and delineates some possible improvements and future lines of work.

Chapter 2

Background

This chapter explores the background needed to understand the addressed problem. This background includes the foundations and challenges of the Semantic Web and a description of the MapReduce framework, on which HDT-MR is based upon.

2.1 Semantic Web

This section describes the basics of the Semantic Web: its principles, foundations, and standards. It also comments its current challenges and open fields of research.

2.1.1 Foundations of the Semantic Web

The foundations of the Semantic Web are based on the current World Wide Web. In the WWW the information is published and consumed by a multitude of different actors. Information is published as documents, interconnected by links, and can be viewed as a whole as a directed graph. In this graph, each document is a edge, while each link is a vertex. Then, information is presented in natural language [7], and meaning is inferred by people who read web pages and the labels of hyperlinks [98]. In the current web there are huge quantities of data, but utilization is limited due to inherent problems [24]:

- *Information overload:* Information on the web grows rapidly and without organization, so it is difficult to extract useful information from it.
- *Stovepipe Systems:* Many information system components are built to work only with another components of the same system. Information from these components is not readable by external systems.
- *Poor Content Aggregation:* The Web is full of disparate content which is difficult to aggregate. Scraping is the common solution.

The Semantic Web was first proposed by Tim Berners-Lee [7] in 2001. It expands the concept of organizing information as a directed graph, but instead of linking documents, it links *semantic data*. The links have also semantic labels, in order to describe the relation between the two entities. Its goal is to make data discoverable and consumable not only by people, but also by automatic agents [98]. It is based on the following foundations [7, 98]:

- *Resources:* A resource is intended to represent any idea that can be referred to, whether it is actual data, just a concept, or a reference to a real or fictitious object.

- *Standardized addressing:* All resources on the Web are referred to by URIs [8]. The most familiar URIs are those that address resources that can be addressed and retrieved; these are called URLs, for Uniform Resource Locators.
- *Small set of commands:* The HTTP protocol [38] (the protocol used to send messages back and forth over the Web) uses a small set of commands. These commands are universally understood.
- *Scalability and large networks:* The Web has to operate over a very large network with an enormous number of web sites and to continue to work as the network's size increases. It accomplishes this thanks to two main design features. First, the Web is decentralized. Second, each transaction on the Web contains all the information needed to handle the request.
- *Openness, completeness, and consistency:* The Web is open, meaning that web sites and web resources can be added freely and without central controls. The Web is incomplete, meaning there can be no guarantee that every link will work or that all possible information will be available. It can be inconsistent: Anyone can say anything on a web page, so different web pages can easily contradict each other.

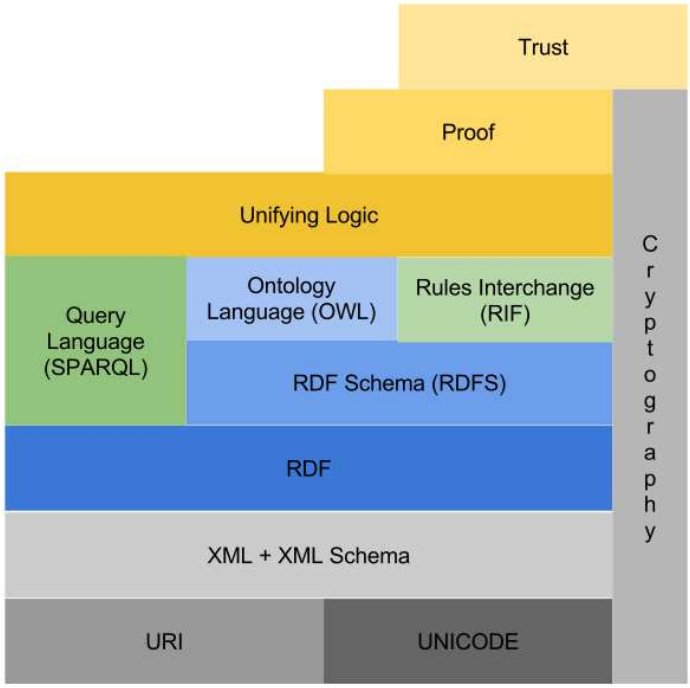


Figure 2.1: Semantic Web Architecture

Semantic Web architecture is usually represented as a layered stack composed of enabling formats and technologies, which can be seen in Figure 2.1. Those elements are briefly outlined below [98] [2]:

- *XML —Extensible Markup Language* [14]: Language framework that lets write structured Web documents with a user-defined vocabulary. Since 1998, it has been used to define nearly all new languages that are used to interchange data over the Web.
- *XML Schema* [34, 114, 11]: A language used to define the structure of specific XML languages.
- *RDF —Resource Description Framework* [74]: A flexible language capable of describing information and meta data. The RDF data model does not rely on XML, but RDF has an XML-based syntax. RDF is covered in section 2.1.1.1
- *RDF Schema* [15]: A framework that provides a means to specify basic vocabularies for specific RDF application languages to use. It provides modeling primitives for organizing Web objects into hierarchies. Key primitives are classes and properties, subclass and sub-property relationships, and domain and range restrictions. RDF Schema is covered in 2.1.1.2.
- *Ontology–Languages*: Expand RDF Schema to allow definition of vocabularies and establish the usage of words and terms in the context of a specific vocabulary. RDF Schema is a framework for constructing ontologies and is used by many more advanced ontology frameworks. OWL [50] is an ontology language designed for the Semantic Web. Section 2.1.1.3 discusses OWL.
- *Logic and Proof*: Logical reasoning is used to establish the consistency and correctness of datasets and to infer conclusions that aren't explicitly stated but are required by or consistent with a known set of data. Proofs trace or explain the steps of logical reasoning. Section 2.1.1.5 covers some issues relating to logic in the Semantic Web.
- *Trust*: A mean of providing authentication of identity and evidence of the trustworthiness of data, services, and agents.

2.1.1.1 RDF

RDF is a data model to describe resources, where a resource can be anything in principle. It proposes a date model based on making statements about the resources. Each statement has then the structure of a triple with the following components [24]:

- *Subject:* The resource that is being described by the ensuing predicate and object. A subject can be either an IRI or a blank node.
- *Predicate:* The relation between the subject and the object. A predicate is an IRI.
- *Object:* Either a resource or value referred to by the predicate. An object can be an IRI, a blank node, or a literal.

An *IRI* (International Resource Identifier) [30] refers unequivocally to a single resource. IRIs are a generalization of an URIs that can contain characters from the Universal Character Set. RDF has rules about how to construct an URI from an IRI so that they can be used conveniently on the Word Wide Web [98]. Blank nodes are local identifiers that are often used to group collections of resources, while a literal is a text string, commonly used for names or descriptions.

Triples are interconnected in a directed labeled graph, whit subjects and objects as vertices, and properties as edges. An example of RDF and its graph-based representation in shown in Figure 2.2. Formally, RDF is described as follows:

Definition 1 (RDF triple) *A tuple* $(s, p, o) \in (I \cup B) \times I \times (I \cup B \cup L)$ *is called an RDF triple, in which "*s*" is the subject, "*p*" the predicate and "*o*" the object.* I *(*RDF IRI references*),* B *(*Blank nodes*), and* L *(*RDF literals*) are infinite, mutually disjoint sets.*

Definition 2 (RDF graph) *An RDF graph* G *is a set of RDF triples. As stated,* (s, p, o) *can be represented as a directed edge-labeled graph* $s \xrightarrow{p} o$*.*

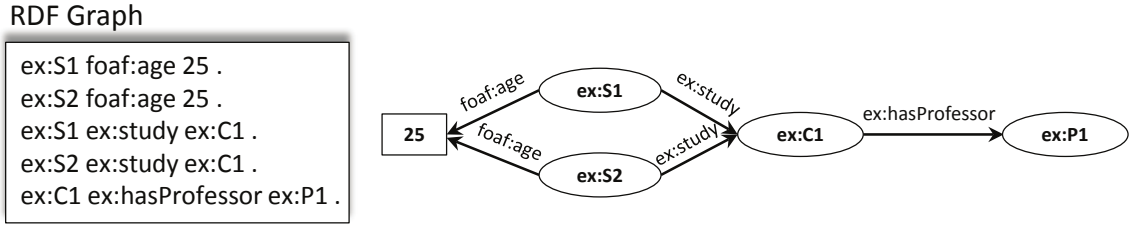


Figure 2.2: An RDF graph example

RDF is more redundant than other ways of storing information —as regular databases— because is schema-less. So, it needs to store property specifications each time they are used. This requires RDF to carry data that might otherwise be redundant [98]. But this trade-off between regularity and flexibility lets RDF do some operations that would be impossible in a conventional database, such as [98]:

- Combine the data with other datasets that do not follow the same data model.
- Add more data that does not fit the table structures.
- Exchange data with any other application that knows how to handle RDF. It can be done over the Web, by email, or any other way by which you can exchange a data file.
- Use an RDF processor that can do logical reasoning to discover unstated relationships in the data.
- Use someone else's ontology to learn more about the relationships between the properties and resources in data.
- Add statements about publications and references that have been defined somewhere else on the Web. All that needs to be done is to refer to the published identifiers (URIs).
- Do all these things using well-defined standards, so that a wide range of software can process the data.

RDF was originally built over XML. RDF/XML is sometimes non-intuitive, as it carries a translation of a list of statements into a hierarchical XML [24]. Because of that, other notations have emerged over time. Notation 3 (N3) and N-triples are the most common non-XML formats. A statement is written in ordinary text in the order `<subject> <predicate> <object>`. N-triples is a line-oriented subset of N3. It was developed to express the desired results of test cases and to be transmittable over the Internet in a MIME format. For simple cases, there is virtually no difference between N-triples and N3 [98].

2.1.1.2 RDF Schema

RDF Schema (or RDFS in short) defines classes and properties, using the RDF data model. Those elements allow to describe vocabularies (i.e. basic ontologies) to structure RDF resources and impose restrictions on what can be stated in an RDF dataset. A continuation of the RDF graph running example with some RDFS statements added is shown on Figure 2.3 The main elements of RDFS are outlined below [2].

RDF Graph

```
ex:Student rdf:type rdfs:Class .
ex:Professor rdf:type rdfs:Class .
ex:Class rdf:type rdfs:Class .

ex:S1 rdf:type ex:Student
ex:S2 rdf:type ex:Student
ex:P1 rdf:type ex:Professor
ex:C1 rdf:type ex:Class

ex:S1 foaf:age 25 .
ex:S2 foaf:age 25 .
ex:S1 ex:study ex:C1 .
ex:S2 ex:study ex:C1 .
ex:C1 ex:hasProfessor ex:P1 .
```

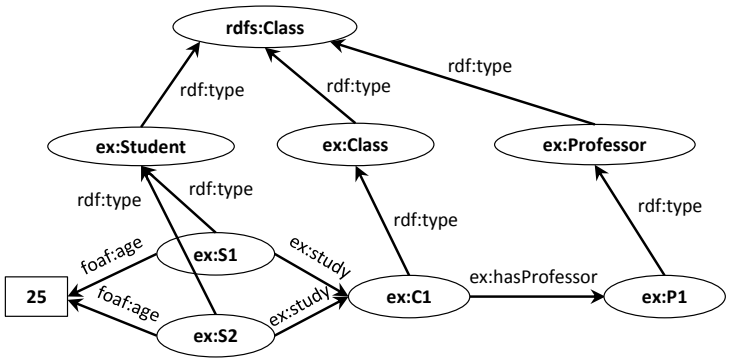


Figure 2.3: An RDFS graph example

Core Classes of RDFS:

- `rdfs:Resource`, the class of all resources.
- `rdfs:Class`: An element that defines a group of related things that share a set of properties [24]. It is the superclass of all classes.
- `rdfs:Literal`, the class of all literals (strings). At present, literals form the only data type of RDF/RDFS.
- `rdf:Property`, the class of all properties.
- `rdf:Statement`, the class of all reified statements.

Core Properties for Defining Relationships:

- `rdf:type`, which relates a resource to its class. The resource is declared to be an instance of that class.
- `rdfs:subClassOf`, which relates a class to one of its superclasses; all instances of a class are instances of its superclass. An element that specifies that a class is a specialization of an existing class. This follows the same model as biological inheritance, where a child class can inherit the properties of a parent class. The idea of specialization is that a subclass adds some unique characteristics to a general concept.
- `rdfs:subPropertyOf`, which relates a property to one of its superproperties.

Core Properties for Restricting Properties:

- `rdfs:domain`, which specifies the domain of a property P, that is, the class of those resources that may appear as subjects in a triple with predicate P. If the domain is not specified, then any resource can be the subject.
- `rdfs:range`, which specifies the range of a property P, that is, the class of those resources that may appear as values in a triple with predicate P.

RDFS also includes utility properties to establish generic relations between resources or provide information to human readers, such as `rdfs:seeAlso`, `rdfs:isDefinedBy`, `rdfs:comment`, and `rdfs:label` [2]

2.1.1.3 Ontologies and OWL

An ontology defines the common words and concepts (*i.e.* the meaning) used to describe and represent an area of knowledge [24]. An ontology language allows to model the vocabulary and meaning of domains of interest: the objects in domains; the relationships among those objects; the properties, functions, and processes involving those objects; and constraints on and rules about those things [24]. While RDF Schema provides some tools to do so, they are limited to a subclass hierarchy and a property hierarchy, with domain and range definitions of these properties [2]. The following requirement should be followed by a complete ontology language [2]:

- *A well-defined syntax:* A necessary condition for machine-processing of information
- *A formal semantics:* Describes the meaning of knowledge precisely. Precisely here means that the semantics does not refer to subjective intuitions, nor is it open to different interpretations.
- *Efficient reasoning support:* Allows to check the consistency of the ontology and the knowledge, check for unintended relationships between classes, and automatically classify instances in classes
- *Sufficient expressive power:* Needed to represent ontological knowledge.
- *Convenience of expression.*

In detail, RDFS lacks expressive power to model an ontology because:

- *Local scope of properties:* rdfs:range defines the range of a property, say eats, for all classes. Thus in RDF Schema we cannot declare range restrictions that apply to some classes only.
- *Disjointness of classes:* Sometimes we wish to say that classes are disjoint. For example, male and female are disjoint. But in RDF Schema we can only state subclass relationships, e.g., female is a subclass of person.
- *Boolean combinations of classes:* Sometimes we wish to build new classes by combining other classes using union, intersection, and complement. For example, we may wish to define the class person to be the disjoint union of the classes male and female. RDF Schema does not allow such definitions.
- *Cardinality restrictions:* Sometimes we wish to place restrictions on how many distinct values a property may or must take. For example, we would like to say that a person has exactly two parents, or that a course is taught by at least one lecturer. Again, such restrictions are impossible to express in RDF Schema.

- *Special characteristics of properties:* Sometimes it is useful to say that a property is transitive (like “greater than”), unique (like “is mother of”), or the inverse of another property (like “eats” and “is eaten by”).

In order to achieve the expressive power that RDFS lacks the W3C developed OWL, an extension of RDF to describe ontologies. In OWL2, there are 3 OWL profiles, based on different description logics, with some trade-off between expressive power and efficiency of reasoning [92]:

- *OWL2-EL:* Tailored for applications that need to create ontologies with very large number of classes and/or properties (as in large science ontologies). It is the profile with more expressive power, allowing classes to be defined with complex descriptions.
- *OWL2-QL:* Aimed at applications that use very large volumes of instance data, and where query answering is the most important reasoning task. Its goal is to allow reasoning to be translated into queries on a database. In order for reasoning to be translated into a query, its expressivity is restricted.
- *OWL2-RL:* Designed for applications that want to describe rules in ontologies. It is essentially a rules language for implementing logic in the form of rules *if-then*.

The main elements of OWL are described below [2].

Class Elements allow to declare and define classes with relation to another classes.

- `owl:Class`: Defines a class.
- `owl:disjointWith`: To say that a class is disjoint with the specified class.
- `owl:equivalentClass`: To indicate that a class is equivalent with the specified class.

Property Elements allow to declare and define properties with relation to another properties.

- `owl:DatatypeProperty`: Defines a data type property (a property that relates objects with data type values).
- `owl:ObjectProperty`: Defines a object property (a property that relates objects with objects).
- `owl:equivalentProperty`: To indicate a equivalent class, with the same range and domain.
- `owl:inverseOf`: To say a property is inverse of the specified property (*i.e.* uses its range as domain, and its domain as range).

Property Restrictions allow to place property restrictions on defined classes.

- `owl:Restriction`: Specifies property restrictions on any class. Property restriction must be located between `owl:Restriction` tags inside a class.
- `owl:onProperty`: Indicates the property that will be affected by the restriction.
- `owl:allValuesFrom`: Specifies the class of possible values that the property can take as range.

- `owl:hasValue`: Specifies a fixed value that the property will have.
- `owl:someValuesFrom`: Makes mandatory to have at least one property with a value from the specified class. If the class has more than one of these properties, the rest are not restricted in this way.
- `owl:minCardinality`: Specifies the minimum cartinality of the property.
- `owl:maxCardinality`: Specifies the maximum cartinality of the property.

Special Properties allow to define directly some attributes of property elements.

- `owl:TransitiveProperty` defines a transitive property, such as "has better grade than", "is taller than", or "is ancestor of".
- `owl:SymmetricProperty` defines a symmetric property, such as "has same grade as" or "is sibling of".
- `owl:FunctionalProperty` defines a property that has at most one value for each object, such as "age", "height", or "directSupervisor".
- `owl:InverseFunctionalProperty` defines a property for which two different objects cannot have the same value, for example, the property "isTheSocialSecurityNumberfor" (a social security number is assigned to one person only).

Boolean Combinations allow to specify boolean combinations of classes.

- `owl:complementOf`: Specifies that the class is disjoint with the specified class. It has the same effect as `owl:disjointWith`.
- `owl:unionOf`: Specifies that the class is equal to the union of specified classes.
- `owl:intersectionOf`: Specifies that the class is equal to the intersection of specified classes.

Enumerations allow to define a class by listing all its elements.

- `owl:oneOf`: Is used to list all elements that comprise a class.

Versioning information gives information that has no formal model-theoretic semantics but can be exploited by human readers and programs alike for the purposes of ontology management.

- `owl:priorVersion`: Indicate earlier versions of current ontology.
- `owl:versionInfo`: Contains a string giving information about the current ontology version.
- `owl:backwardCompatibleWith`: Identifies prior versions of the current ontology that are compatible with.
- `owl:incompatibleWith`: Identifies prior versions of the current ontology that are not compatible with.

Instances of OWL classes are declared as in RDF. OWL does not assume that instances with different name or ID are not the same instance (*i.e.* does not adopt the *unique-names assumption*). In order to ensure that resources are considered different from each other, this elements must be used:

- `owl:differentFrom`: Identifies the resource as different from the specified resource.
- `owl:AllDifferent`: Allows to specify a collection. All elements of the collection are considered different from each other.

2.1.1.4 SPARQL

Previous sections described how data are stored. As with traditional databases, it is necessary a way to inquire about the information contained in a dataset. SPAQRQL is the query language for the Semantic Web proposed by the W3C. SPARQL syntax is based on Turtle [5].

SPARQL is essentially a declarative language based on graph-pattern matching with a SQL-like syntax. Graph patterns are built on top of *Triple Patterns* (TPs), *i.e.*, triples in which each of the subject, predicate and object may be a variable. These TPs are grouped within *Basic Graph Patterns* (BGPs), leading to query subgraphs in which TP variables must be bounded. Thus, graph patterns commonly join TPs, although other constructions, such as alternative (union) and optional patterns, can be specified in a query [30]. An SPARQL query example is presented on Figure 2.4. We can formally define TPs and BGPs as follows:

Definition 3 (SPARQL triple pattern) *A tuple from* $(I \cup L \cup V) \times (I \cup V) \times (I \cup L \cup V)$ *is a triple pattern.*

Note that blank nodes act as non-distinguished variables in graph patterns [100].

Definition 4 (SPARQL Basic Graph pattern (BGP)) *A SPARQL Basic Graph Pattern (BGP) is defined as a set of triple patterns. SPARQL FILTERs can restrict a BGP. If* B_1 *is a BGP and* R *is a SPARQL* built-in *condition, then* $(B_1$ `FILTER` $R)$ *is also a BGP.*

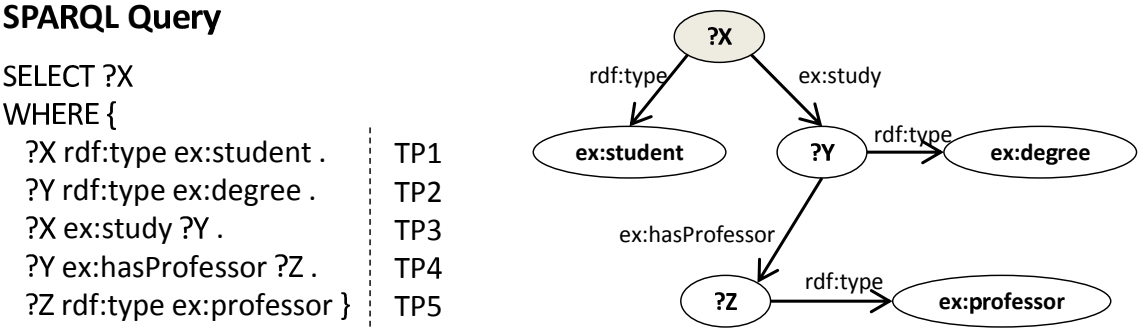


Figure 2.4: A SPARQL query example

Query resolution performance mainly depends on two factors:

1. *Triples retrieval*, which depends on how triples are organized and indexed.
2. *Join of Triples* of different TPs, determined by optimization strategies and algorithms for join resolution [32].

Both concerns are typically addressed within RDF storage systems (RDF stores), which are usually built on top of relational systems or carefully designed from scratch to fit particular RDF peculiarities.

SPARQL allows to enquire information with four different statements:

- *SELECT:* Used to extract values.
- *CONTRUCT:* Used to extract information as RDF text.
- *ASK:* Used to obtain *yes/no* questions.
- *DESCRIBE:* Used to extract arbitrary "useful" information.

A *WHERE* block is added (mandatory except in the case of DESCRIBE) where graph patterns to be matched are included.

2.1.1.5 Reasoning

RDF allows for inference of new knowledge not previously stated on the original data by using entailment rules. An entailment rule can be seen as a left-to-right rules: If the original data comply with the left side, the conclusion is added to the graph. ter Horst [113] describes entailment rules for RDFS and a subset of OWL (which serve as a basis for OWL2 RL definition), which can be seen in Figures 2.5 and 2.6, respectively.

There are two main approaches to perform inference: The first one consist of applying the rules at query time. In that case, the information related to the query is derived using *backward-chaining* reasoning. The second approach computes what is called the *Graph Closure*, applying all the entailment rules using *forward-chaining* reasoning, deriving and storing all the implicit information. Both approaches have advantages and disadvantages. The main advantage of the reasoning at query time is that it doesn't require neither expensive precomputation nor space consumption. Thus, it is more suitable to datasets with dynamic information. However, the computations needed to perform the reasoning at query time are usually too expensive to be used in interactive applications. The computation of the Graph Closure, on the other hand, have the advantage of not needing any additional computation at query time. While this approach is suited for interactive applications, it is not efficient when only a small portion of the derivation is useful at query time.

2.1.1.6 Linked Data

Linked Open Data[1] is a movement that advocates the publication of data under open licenses, promoting reuse of data for free. The project's original and ongoing goal is to leverage the WWW infrastructure to publish and consume RDF data, achieving ubiquitous and seamless data integration over the WWW infrastructure [35]. This is accomplished by identifying existing datasets that are available under open licenses, converting them to RDF according to the Linked Data principles, and publishing them on the Web [12].

In 2006, Berners-Lee [6] enumerated the four principles the Web of Linked Data should follow in order to make possible for different datasets to be published on the current WWW infrastructure and connected together:

1. *Use URIs as names for things.* This allows any entity to be unambiguously referenced.

[1] http://linkeddata.org/

1: s p o (if o is a literal)		⇒ _:n rdf:type rdfs:Literal
2: p rdfs:domain x	& s p o	⇒ s rdf:type x
3: p rdfs:range x	& s p o	⇒ o rdf:type x
4a: s p o		⇒ s rdf:type rdfs:Resource
4b: s p o		⇒ o rdf:type rdfs:Resource
5: p rdfs:subPropertyOf q	& q rdfs:subPropertyOf r	⇒ p rdfs:subPropertyOf r
6: p rdf:type rdf:Property		⇒ p rdfs:subPropertyOf p
7: s p o	& p rdfs:subPropertyOf q	⇒ s q o
8: s rdf:type rdfs:Class		⇒ s rdfs:subClassOf rdfs:Resource
9: s rdf:type x	& x rdfs:subClassOf y	⇒ s rdf:type y
10: s rdf:type rdfs:Class		⇒ s rdfs:subClassOf s
11: x rdfs:subClassOf y	& y rdfs:subClassof z	⇒ x rdfs:subClassOf z
12: p rdf:type rdfs:ContainerMembershipProperty		⇒ p rdfs:subPropertyOf rdfs:member
13: o rdf:type rdfs:Datatype		⇒ o rdfs:subClassOf rdfs:Literal

Figure 2.5: RDFS inference rules [113]

1: p rdf:type owl:FunctionalProperty, u p v , u p w	⇒ v owl:sameAs w
2: p rdf:type owl:InverseFunctionalProperty, v p u, w p u	⇒ v owl:sameAs w
3: p rdf:type owl:SymmetricProperty, v p u	⇒ u p v
4: p rdf:type owl:TransitiveProperty, u p w, w p v	⇒ u p v
5a: u p v	⇒ u owl:sameAs u
5b: u p v	⇒ v owl:sameAs v
6: v owl:sameAs w	⇒ w owl:sameAs v
7: v owl:sameAs w, w owl:sameAs u	⇒ v owl:sameAs u
8a: p owl:inverseOf q, v p w	⇒ w q v
8b: p owl:inverseOf q, v q w	⇒ w p v
9: v rdf:type owl:Class, v owl:sameAs w	⇒ v rdfs:subClassOf w
10: p rdf:type owl:Property, p owl:sameAs q	⇒ p rdfs:subPropertyOf q
11: u p v, u owl:sameAs x, v owl:sameAs y	⇒ x p y
12a: v owl:equivalentClass w	⇒ v rdfs:subClassOf w
12b: v owl:equivalentClass w	⇒ w rdfs:subClassOf v
12c: v rdfs:subClassOf w, w rdfs:subClassOf v	⇒ v rdfs:equivalentClass w
13a: v owl:equivalentProperty w	⇒ v rdfs:subPropertyOf w
13b: v owl:equivalentProperty w	⇒ w rdfs:subPropertyOf v
13c: v rdfs:subPropertyOf w, w rdfs:subPropertyOf v	⇒ v rdfs:equivalentProperty w
14a: v owl:hasValue w, v owl:onProperty p, u p v	⇒ u rdf:type v
14b: v owl:hasValue w, v owl:onProperty p, u rdf:type v	⇒ u p v
15: v owl:someValuesFrom w, v owl:onProperty p, u p x, x rdf:type w	⇒ u rdf:type v
16: v owl:allValuesFrom u, v owl:onProperty p, w rdf:type v, w p x	⇒ x rdf:type u

Figure 2.6: OWL Horst inference rules [113]

2. *Use HTTP URIs so that people can look up those names.* So each entity can be referenced on the WWW.

3. *When someone looks up a URI, provide useful information, using the standards (RDF, SPARQL).* In this way, information can be discovered when following the referenced URI.

4. *Include links to other URIs, so that they can discover more things.* This last rule is necessary to really connect the data into a web.

Statistics show that LOD datasets have increased both in number and in size in recent years [33]. According to LODStats[2] there are more than 89 billion triples on August 2015. Main contributors to the Web of Linked Data currently include the British Broadcasting Corporation (BBC), The US Library of Congress, and the German National Library of Economics. A visual representation of the LOD Cloud as in August 2014 can be seen in Figure 2.7

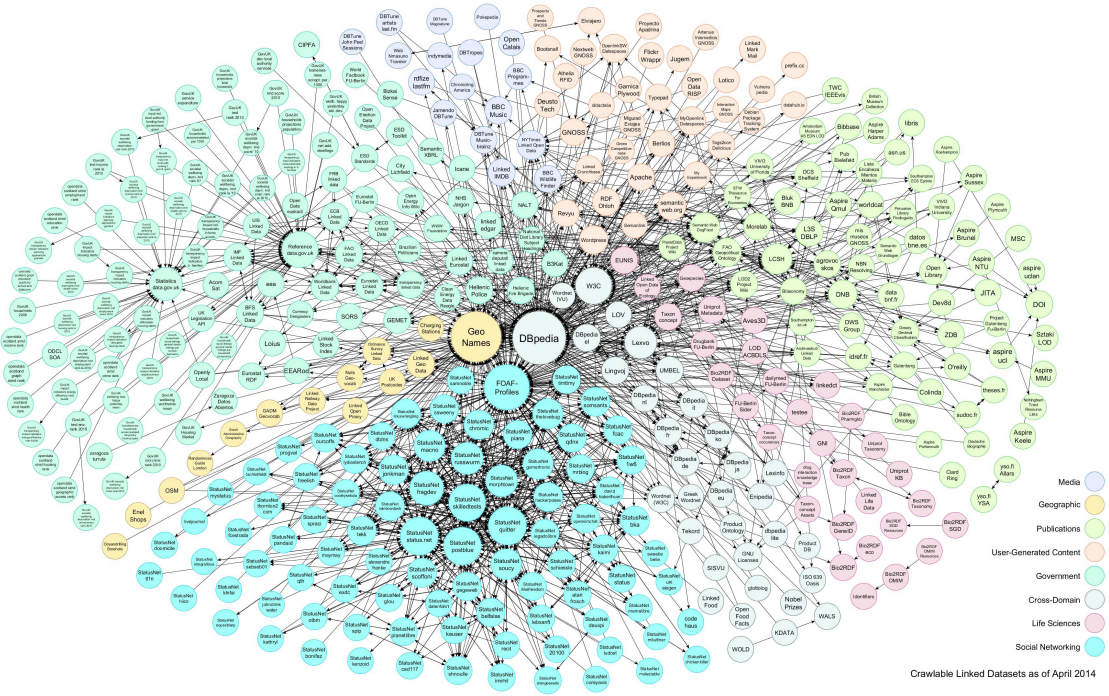


Figure 2.7: LOD Cloud (as of August 2014) [23]

2.1.2 Scalability Challenges

The Semantic Web is a novel technology that allows to represent semantic data in a flexible way, but it is still a novel technology and currently faces some technological challenges that hinder the construction of scalable applications.

In the first place, efficient RDF storage is a current line of research. Standard representations of RDF store the data in plain text. While this reduces the complexity of data serialization and processing, it impacts the final size of RDF datasets. In addition, due to RDF schema-less structure, information about the schema of the data must be included in each dataset, adding a non-trivial amount of space. This is not only relevant because of storage requirements, but also when considering data consumption. This leads to the second issue that the Semantic Web faces:

[2] `http://stats.lod2.eu/`

Large volumes of RDF data are not easily queried. RDF stores usually built on top of relational systems or carefully designed from scratch to fit particular RDF peculiarities [25]. However, RDF stores typically lack of scalability when large volumes of RDF data must be managed [35].

Large-scale reasoning and inference is also a current challenge. RDFS and, specially, OWL rules (see section 2.1.1.5) are complex and frequently generate new triples that are impacted by the same or other rules. This makes online reasoning too much computationally expensive to be performed. The approach adopted by most solutions that support reasoning is the materialization of the inferred triples in a batch process. In fact , there is currently no alternative to materialization that scales to relatively complex logics and very large data sizes [118]. However, materialization is not a good solution when the data are dynamic. Nonetheless, even generating the closure of large datasets is not a trivial task, and doing it efficiently is another line of research [118].

2.2 HDT

HDT [36] is a binary serialization format optimized for RDF storage and transmission. Besides, HDT files can be mapped to a configuration of succinct data structures which allows the inner triples to be searched and browsed efficiently. HDT is a *W3C Member Submission*[3]. The following sections describe its structure and its current mono-node serialization process.

2.2.1 Structure

HDT encodes RDF into three components carefully described to address RDF peculiarities within a *Publication-Interchange-Consumption* workflow. The *Header* (**H**) holds the dataset metadata, including relevant information for discovering and parsing, hence serving as an entry point for consumption. The *Dictionary* (**D**) is a catalogue that encodes all the different terms used in the dataset and maps each of them to a unique identifier: ID. The *Triples* (**T**) component encodes the RDF graph as a graph of IDs, *i.e.* representing tuples of three IDs. Thus, *Dictionary* and *Triples* address the main goal of RDF compactness. Figure 2.8 shows how the *Dictionary* and *Triples* components are configured for a simple RDF graph. Each component is detailed below.

Dictionary. This component organizes the different terms in the graph according to their role in the dataset. Thus, four sections are considered: the section **SO** manages those terms playing both as subject and object, and maps them to the range `[1, |SO|]`, being `|SO|` the number of different terms acting as subject and object. Sections **S** and **O** comprise terms that exclusively play subject and object roles respectively. Both sections are mapped from `|SO|+1`, ranging up to `|SO|+|S|` and `|SO|+|O|` respectively, where `|S|` and `|O|` are the number of exclusive subjects and objects. Finally, section **P** organizes all predicate terms, which are mapped to the range `[1, |P|]`. It is worth noting that no ambiguity is possible once we know the role played by the corresponding ID. Each section of the *Dictionary* is independently encoded to grasp its particular features. This allows important space savings to be achieved by considering that this sort of string dictionaries are highly compressible [86].

Triples. This component encodes the structure of the RDF graph after ID substitution. That is, RDF triples are encoded as groups of three IDs (ID-triples hereinafter): $(\mathtt{id}_s\ \mathtt{id}_p\ \mathtt{id}_o)$, where $\mathtt{id}_s$, $\mathtt{id}_p$, and $\mathtt{id}_o$ are respectively the IDs of the corresponding subject, predicate, and object terms in the *Dictionary*. The *Triples* component organizes all triples into a forest of trees, one per different subject: the subject is the root; the middle level comprises the ordered list of predicates

[3] `http://www.w3.org/Submission/HDT`

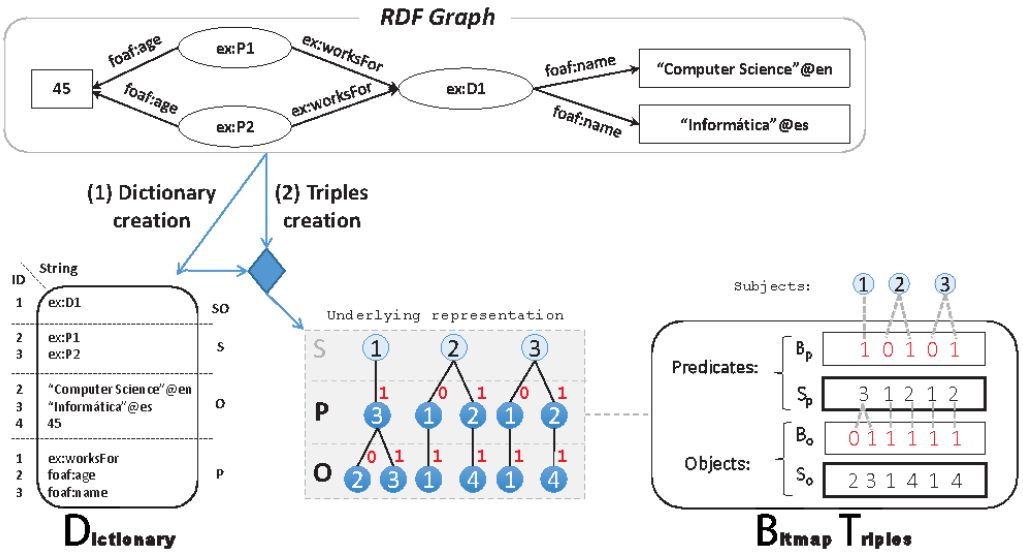


Figure 2.8: HDT Dictionary and Triples configuration for an RDF graph

reachable from the corresponding subject; and the leaves list the object IDs related to each (subject, predicate) pair. This *underlying representation* (illustrated in Figure 2.8) is effectively encoded following the *BitmapTriples* approach [36]. In brief, it comprises *two sequences*: Sp and So, concatenating respectively all predicate IDs in the middle level and all object IDs in the leaves; and *two bitsequences*: Bp and Bo, which are respectively aligned with Sp and So, using a 1-bit to mark the end of each list.

2.2.2 Building HDT

In this section we proceed to summarize how HDT is currently built[4]. Remind that this process is the main scalability bottleneck addressed by our current proposal.

To date, HDT serialization can be seen as a three-stage process:

- **Classifying RDF terms.** This first stage performs a triple-by-triple parsing (from the input dataset file) to classify each RDF term into the corresponding *Dictionary* section. To do so, it keeps a temporal data structure, consisting of three hash tables storing subject-to-ID, predicate-to-ID, and object-to-ID mappings. For each parsed triple, its subject, predicate, and object are searched in the appropriate hash, obtaining the associated ID if present. Terms not found are inserted and assigned an auto-incremental ID. These IDs are used to obtain the temporal ID-triples $(id_s\ id_p\ id_o)$ representation of each parsed triple, storing all them in a temporary ID-triples array. At the end of the file parsing, subject and object hashes are processed to identify terms playing both roles. These are deleted from their original hash tables and inserted into a fourth hash comprising terms in the SO section.

- **Building HDT *Dictionary*.** Each dictionary section is now sorted lexicographically, because prefix-based encoding is a well-suited choice for compressing string dictionaries [85]. Finally, an auxiliary array coordinates the previous temporal ID and the definitive ID after the *Dictionary* sorting.

[4]HDT implementations are available at `http://www.rdfhdt.org/development/`

- **Building HDT *Triples*.** This final stage scans the temporary array storing ID-triples. For each triple, its three IDs are replaced by their definitive IDs in the newly created *Dictionary*. Once updated, ID-triples are sorted by subject, predicate and object IDs to obtain the *BitmapTriples* streams. In practice, it is a straightforward task which scans the array to sequentially extract the predicates and objects into the `Sp` and `So` sequences, and denoting list endings with 1-bits in the bitsequences.

2.2.3 Performance

HDT provides effective RDF decomposition, simple compression notions and basic indexed access in a compact serialization format which provides efficient access to the data. The Triples component is specifically encoded using a succinct data structure that enables indexed access to any triple in the dataset. It provides good performance both in terms of compression rate and query response time. With *Plain-HDT* Space savings of 12-16% are reported with real-world datasets[5], although compression up to 7% is reached [36]. With *HDT-Compress* compression rates are improved to 2-4%, outperforming results of universal compressors by at least 20% [36]. Fernández et al. [36] compare *Plain-HDT* performance with state-of-the-art solutions such as RDF-3X and MonetDB. Tests are performed on the Dbpedia dataset with real-world queries. HDT outperforms both of them in all patterns except `(?S,P,O)`, where RDF-3X obtains the best results, and `(?S,P,?O)`, where HDT obtains the worst results.

2.2.4 Scalability Issues

HDT serialization process, described in section 2.2.2, makes use of several in-memory data structures: four hash tables to store *term-to-ID* mappings during the first stage, an auxiliary array during the second stage for *temporary ID to definitive ID* mappings, and an additional array to store *ID-triples* in the third phase. This is in addition to the actual *Bitmap* serialization, which must be wholly built in-memory before writing to disk.

2.3 MapReduce

MapReduce is a standard framework and programming model for the distributed processing of large amounts of data. Its main goal is to provide efficient parallelization while abstracting the complexities of distributed processing, such as data distribution, load balancing and fault tolerance [27]. MapReduce is not schema-dependent, so it can process unstructured and semi-structured data, although at the price of parsing every item [77]. A MapReduce job is comprised of two main phases: `Map` and `Reduce`. The Map phase reads pairs key-value and *'maps'* relevant information, generating another intermediate pairs key-value. The Reduce phase takes all the values associated with the same key and *'reduces'* them into a smaller set of values [27].

MapReduce was originally developed by Google and published in 2004 [27]. Since then, it has been adopted by many companies that deal with Big Data. Examples of institutions actively contributing to the MapReduce ecosystem include:

- *Google* originally developed MapReduce [27] and many other related technologies: GFS [45] — an underlying file system to be used with MapReduce —, BigTable [20] — a distributed column-oriented database — , and Sawzall [99] — a procedural programming language for analysis of large datasets in MapReduce clusters.

[5] Geonames, Dbtune, Uniprot and Dbpedia

- *Apache* developed Hadoop [10], a popular free implementation of MapReduce. Hadoop is continually under development and includes many other subprojects like HBase — a distributed columnar database inspired in BigTable — or Pig [95] — a procedural data language on top of Hadoop.
- *Yahoo!* has contributed to 80% of the core of Hadoop. In 2010 Hadoop clusters at Yahoo! spanned 25.000 servers, and stored 25 petabytes of application data, with the largest cluster being 3500 servers [108].
- *Facebook* devised Hive, an open-source data warehousing solution built on top of Hadoop [115]. In 2010 Facebook's data warehouse that stored more than 15PB of data and loaded more than 60TB of new data every day [117].
- *Microsoft* created in 2008 their own approach to MapReduce: Cosmos, and SCOPE, a declarative language on top of Cosmos [19]. In recent years, Microsoft has abandoned its own technologies and has integrated Hadoop into their Cloud computing systems[6]

Although MapReduce is not dependent on filesystem architecture, its operation is based on distributed data across many nodes, in order to process chunks of data in the same node it is stored. Google File System (GFS) [45] and Hadoop Distributed File System (HDFS) are the main examples of those file systems. Google MapReduce and Hadoop are commonly deployed on top of them [27, 10].

2.3.1 Distributed FileSystems

To fully understand the MapReduce operation, it is needed to have some knowledge about the underlying distributed filesystem it operates on, which are portrayed in this section.

Distributed file systems serving data to MapReduce are designed to store and deliver large amounts of data on clusters of commodity hardware. They are designed to achieve scalability (*i.e.* scaling computation capacity by simply adding commodity servers) and high aggregate performance, while dealing with fault tolerance. To accomplish these goals these filesystems make use of physical architecture awareness, data replication and placement, constant monitoring, and instant recovery techniques [45, 13, 108].

Those filesystems are based on the premise of storing large quantities of data, which needs to be accessed sequentially by batch processes. Hence, some design principles are the basis of distributed filesystems [45].

- Data are stored on very large files: When working with high volume data sets, each one comprising GB or TB, and millions of objects, it is hard to manage billions of files. As a result, the file system stores the data as large files from hundreds of megabytes to terabytes.
- High sustained bandwidth is more important than low latency: To data-intensive applications a high rate of data transfer is more important that response time for each petition
- Data processing follows a "write once, read many times" pattern, with streaming file access: Once written, data are mostly accessed for data process, and is read often sequentially. Modification of files, whenever it happens, is usually by appending data. Random data modification is almost non-existent.

For these reasons, distributed filesystes are not suitable for all kinds of applications. Some operations are not efficient, such as the following [10]:

[6] `http://azure.microsoft.com/en-us/documentation/services/hdinsight/`

- Low latency data access: Distributed file system are designed to deliver high-throughput data-access. This is done at the expense of latency.

- Lots of small files: The complete namespace needs to be stored in-memory on a single node, which makes the storage of huge number of files unfeasible. This is in addition to under-usage of hard drives due to the block size used to store the files.

- Arbitrary file modifications: Modifications block files to the rest of the clients and require high usage of net bandwidth. Also, random modifications that do not affect end of files are not efficient.

2.3.1.1 Architecture

The architecture of distributed file systems are master/slave in nature. One master (Namenode in HDFS) contains metadata and file placement. Many slaves (chunkservers in GFS, datanodes in HDFS) store files data. Depending on the implementation, there can be "secondary masters" that maintain backups of the master or help with some operations. The general architecture is shown on Figure 2.9.

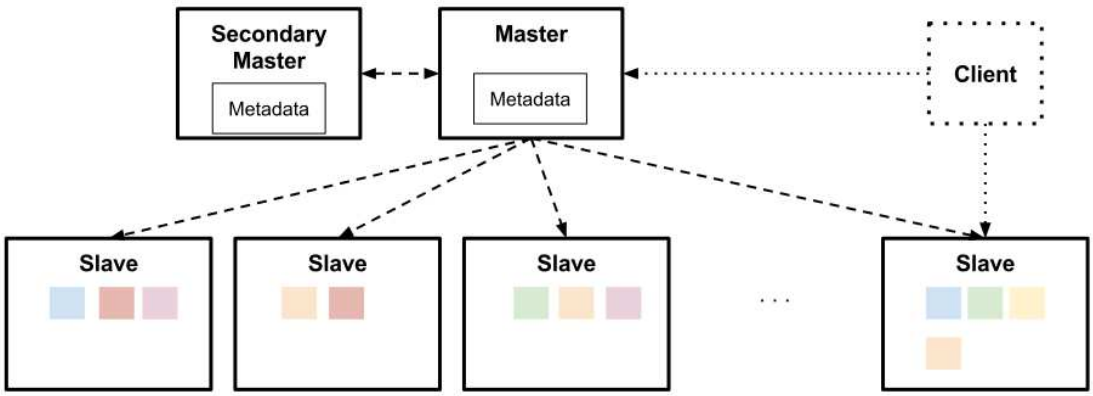


Figure 2.9: Distributed File System Architecture

Files are broken down in blocks and stored across the slaves. Block size is configurable, but is usually of at least 64 MB, and 128 MB are usually the norm nowadays [124]. This large size minimizes the cost of hard drive seeks, and reduces interaction with the master and metadata stored on it [45]. Some studies suggest that even larger sizes improve significantly the performance [77, 66]. Each block is replicated over a number of slaves. The *replication factor* is 3 by default, but is configurable. It is possible to configure different replication factor to specific files, if they are expected to be accessed more or less frequently than the average [45, 108]. If at any time replication of any block falls under its replication factor, it is copied to other nodes from active sources until it reaches the replication factor again.

The master maintains information about file and block namespaces, file-to-block mapping, and the location of each block. Metadata and file-to-block mapping are stored persistently in master hard drives, but the location of blocks is requested to the slaves at cluster startup. Not storing block location simplifies the architecture and avoids dealing with problems of synchronization between master and slaves. It also simplifies fault recovery, since this operation does not need to change form a normal startup [45].

Slaves and master communicate by *heartbeats*: Messages sent by each slave to the master periodically (3 seconds by default in Hadoop). If the master does not receive heartbeat from a slave in a specific timeframe (by default during 10 minutes in Hadoop) it is considered out of service and not used anymore. Heartbeats also carry information about total storage capacity, fraction of storage in use, and the number of data transfers currently in progress. These statistics are used for the master's space allocation and load balancing decisions. The master sends maintenance commands to slaves in reply to heartbeats [108]

When a client needs to access a file, it send a petition to the master with file identification and offset. The master looks for the corresponding block and sends the client the list of slaves that store the block. The block is *leased* to the client for a specified period of time (60 seconds in GFS), although the client can request extensions. The client chooses a slave (typically the nearest one) and all the I/O operations are made through it. That avoids burdening the master with unnecessary data transfer. It is also possible for the client to ask for more than one block in a single request [45, 124, 108].

Multiple clients can read the same file with no additional features, but writings needs to deal with multiple clients trying to modify the same file. Different implementations take distinct approaches. HDFS only allows one client to write in the same file at a time [13, 108], GFS on the other allows multiple clients to append information to the same file. Data are stored in a temporary location in the nodes and is added to the file in the same order as the clients close the file [45].

In a write operation, slaves containing the block are sorted in a serial pipeline by proximity to the client, and one of them is chosen as the primary. The client writes data to the closest slave, and data are transferred through the pipeline. Receiving slaves send acknowledgment to the previous one. When all data has been transferred, primary client closes the file and wait for acknowledgment from other slaves. This acknowledgment contains a checksum that is validated. Master and client are informed of the end of the process and its final state [45, 108]. A write operation diagram is shown in Figure 2.10.

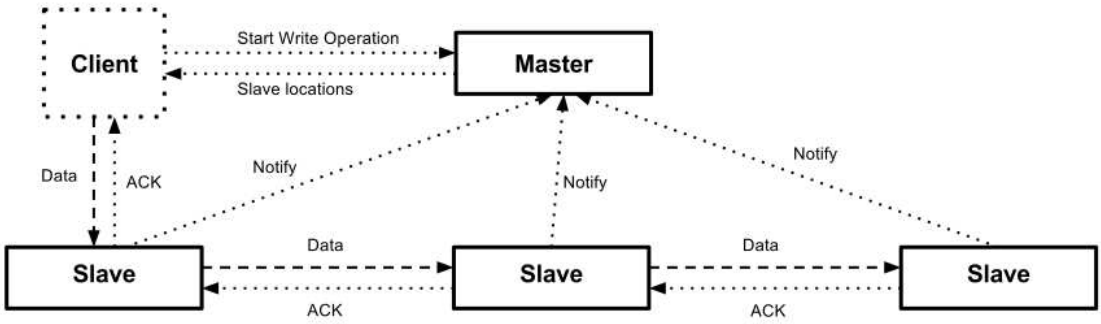


Figure 2.10: Distributed File System Write Dataflow

In the case of file creation, slaves are chosen according to a *replica placement* policy. This policy tries to achieve a compromise between transfer speed and fault tolerance. In HDFS, the default policy is to store the first replica in the same rack as the client, the second replica in another rack and the third replica in the same rack as the second one. If replication factor is greater than 3, following replicas are stored in random nodes over the cluster with this restrictions: No more than one replica is written on the same node, and no more than two replicas are written on the same rack [108].

Deletion follows a similar pattern as write operations, but blocks are not deleted immediately. Instead, a lazy garbage collection mechanism is implemented. Block references are marked in

master and slave nodes; if not used during a specified time (3 days by default) they will by deleted by a background process [45, 13].

Load balancing must be taken into account when considering replica placement. GFS ponders current disk utilization rate of each node at write time [45]. HDFS, in contrast, does not consider it at write time in order to avoid placing new data into the same nodes. New data are frequently more accessed than older one, and placing most of it in a reduced number of slaves could affect performance. HDFS solution consist in running a load balancer as a background process; if a node exceeds a threshold, some of its block are moved to another slave in the cluster [124].

System metadata is stored in the namespace, which is stored by the in memory at runtime. It is periodically stored into disk as a checkpoint, but in order to make it efficiently, the master maintains a write-ahead commit log for changes to the namespace (journal on HDFS). The last checkpoint and the log are merged periodically to make changes persistent [45, 124].

Hadoop implements two other kind of nodes, which help the master in its duties. The Checkpoint Node reads checkpoint and journal, merges them into a single checkpoint, and returns it to the master. The Backup Node contains all metadata except block locations, reads the journal and creates its own checkpoints. If the master fails it can replace it [108].

GFS allows making snapshots by file or directory. When a snapshot is requested, the master revokes all leases, duplicates all the metadata referring to affected data, and points it to the same original blocks. When a client wants to modify a file, affected block are duplicated before any modification occurs [45]. HDFS only allows one snapshot and it can only be made at startup. A static checkpoint is written and slaves are notified to make local snapshots. When a slave needs to modify blocks, it makes a local copy before [108].

2.3.2 MapReduce

MapReduce is designed to abstract the complexities of distributed data process to designers. It aims to perform distributed computations over a high number of computers, dealing with issues like parallelization, data distribution, load balancing, fault tolerance and scalability in a transparent way [27].

In order to achieve these goals, MapReduce works with the principle of *"moving algorithm to data"*. That is, executing data process in the same nodes data are stored, rather than moving data over the machines that make the computations [27].

MapReduce model is based on two main operations, carried out in order: `Map` and `Reduce`. These operations work with pairs key-value. Mappers input pairs and produces intermediate key-value pairs. An intermediate step groups all values with the same key, and Reducers read all grouped values with the same key and processes them, producing a smaller set of values [27]. A schema of `Map` and `Reduce` operations can be seen in Figure 2.11

map:	$(k1, v1) \rightarrow list(k2, v2)$
reduce:	$(k2, list(v2)) \rightarrow list(v2)$

Figure 2.11: Map and Reduce input and output

MapReduce works over data with no schema. This has the advantage of not being schema-dependent, so it can work over unstructured or semi-structured data (like Google's Protocol Buffers, XML, JSON or Apache's Thrift). On the other hand, it needs to parse every item read, which impacts efficiency [77].

MapReduce is I/O intensive, as it needs to write all intermediate data to disk between `Map` and `Reduce` Phases. I/O is the main bottleneck in MapReduce performance. This also impacts in energy efficiency of MapReduce clusters. Many of current research lines try to deal with this problem [77].

2.3.2.1 Architecture

MapReduce clusters have a master/slave architecture. One master (Jobtracker in Hadoop) initializes the process, schedules tasks and keeps bookkeeping information. Many slaves (workers in Google MapReduce, Tasktrackers in Hadoop) execute `Map` and `Reduce` tasks [27]. A diagram of MapReduce architecture can be seen in Figure 2.12.

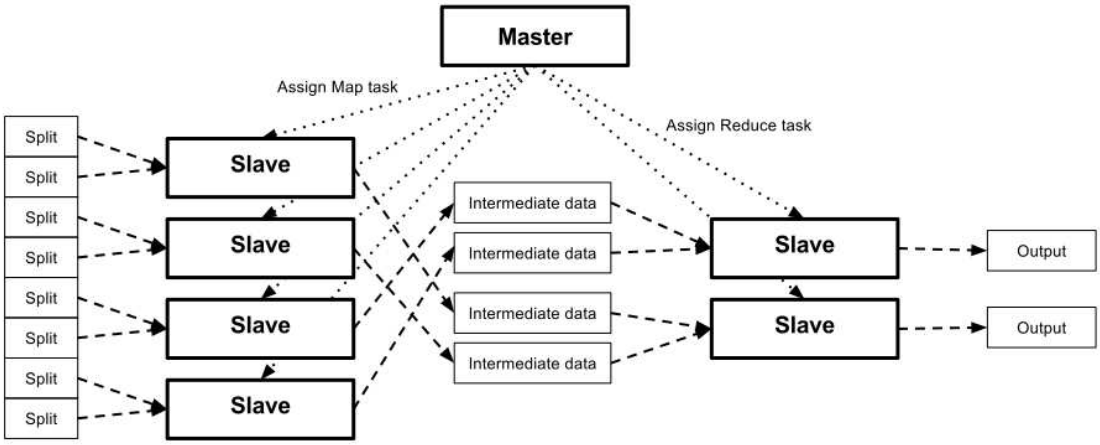


Figure 2.12: MapReduce Architecture Overview

As stated before, a MapReduce job includes two main phases: `Map` and `Reduce`, but more intermediate phases are needed in order to manage data. Below are presented all phases a MapReduce job, and is visually represented in Figure 2.13.

- `Map`: Reads input data and produces intermediate key/value pairs.
- `Sort`: Map output is sorted by key.
- `Combine` (optional): Processes Map output in order to extract only meaningful data, its purpose is to minimize data transferred between Map and Reduce. If a *Combine* function is used, its usage is similar to Reduce, as seen in Figure 2.14.
- `Shuffle`: Data flows between Map and Reduce task. Data received by a Reduce task contains the same key.
- `Merge`: Data from different Map tasks is merged on the Reduce node.
- `Reduce`: Reads pairs key-value and produces a list of values. Each Reduce task works with only one key.

At the beginning of a MapReduce job, input data are divided into input *splits*. Each split is assigned to a `Map` task. When possible, Map tasks are performed in the same nodes data are stored, in order to avoid data transfer between nodes [27, 80]. Note that the same principle is

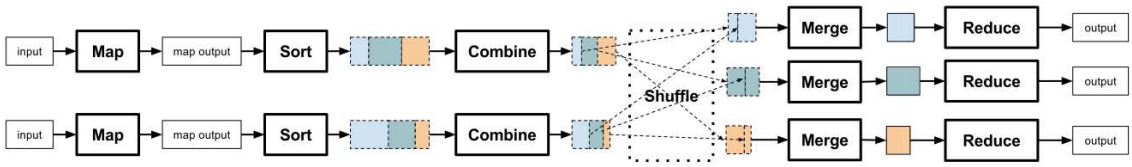


Figure 2.13: Complete MapReduce Dataflow

$$
\begin{aligned}
&\text{map:} && (k1, v1) \rightarrow list(k2, v2)\\
&\text{combine:} && (k2, list(v2)) \rightarrow list(k3, v3)\\
&\text{reduce:} && (k3, list(v3)) \rightarrow list(k3, v3)
\end{aligned}
$$

Figure 2.14: Map, Combine and Reduce inputs and outputs

not possible with `Reduce` task, as they need data generated by multiple Map tasks, each one ran in different nodes, so data transfer is unavoidable [124]. While both input and output are read from and written to distributed filesystems, this is not the case with intermediate values. They are temporary and replication would create unnecessary bandwidth usage [124].

When dividing input data into splits, it is important to take into account splits size. Smaller splits benefit load balancing, specially in heterogeneous environments, because they can be scheduled in different amount to each node. Bigger splits, by contrast, reduce the necessary overhead of scheduling and bookkeeping tasks [66]. The overhead is not only of processing time, but also of memory usage of the master node. This puts a *de facto* lower limit in split size. In addition, using split sizes bigger than block size of underlying distributed system is not recommended, as it could impact in data locality: If a split is bigger than a block, the probabilities of needing data transfer to retrieve part of the split are increased [27, 124].

MapReduce assigns nodes to different jobs using a scheduler. Usage of a FIFO scheduler is common in mono-user environments. In a FIFO scheduler all cluster nodes are used to process a MapReduce Job. When nodes are freed from a job, they can be used to process another job. This is the default scheduler in Hadoop, but in environments where more than one job is needed to run concurrently it is possible to use different schedulers. Hadoop allows to select a Fair Scheduler, that assign equal resources to each job, or a Capacity Scheduler, that uses a more complex scheduling with multiple queues, each of which is guaranteed to possess a certain capacity of the cluster [124, 77].

Slave nodes periodically send heartbeats messages to the master with information about their state and progress of the tasks they are running. Master node uses that information to schedule new tasks, and to identify faulty or straggling nodes (nodes that shows slower-than-expected progress) [27]. If a node does not send a heartbeat during a specified time, or if the heartbeat informs about a failed task, the master marks it as faulty, and no more tasks will be assigned to it. Every task previously assigned, even completed tasks, are rescheduled to be processed by another nodes. If a task is retried for a specified number of times (four by default in Hadoop), it will not be rescheduled. Usually the whole MapReduce job is stopped when that happens, although it is possible to establish a number of "allowed failures" for a job [27, 124]. If a node shows a slower-than-expected progress, its current task will be scheduled to be processed by another node. When the task ends in one of the nodes, the other one will be discarded. This is known as *speculative task scheduling*. Hadoop compares node progress with average progress in the cluster to detect straggling nodes [124].

2.3.3 Challenges and Main Lines of Research

MapReduce is a young technology, with many challenges ahead and open lines of research. The most relevant are presented in this section.

2.3.3.1 Efficiency and Energy Issues

MapReduce is becoming a widespread solution for large-scale data analysis [78, 73]. With an architecture based in replication and constant data traffic and I/O operations, energy consumption is high [78]. This causes an energy waste that must be considered. Some studies that deal with this problem are:

- Covering Set: Keeps only a small fraction of the nodes powered up during periods of low usage. It can save between 9% and 50% of energy consumption [78].
- All-In Strategy: Uses all the nodes in the cluster to run a workload and then powers down the entire cluster. Presents lower effectiveness that Covering Set only when time needed to transition a node to and from a low power state is a relatively large fraction of the overall workload time. In all other cases the benefits of All-In Strategy are significant [73].
- Green HDFS: Proposes data-classification-driven data placement that allows scale-down by guaranteeing substantially long periods (several days) of idleness in a subset of servers in the datacenter. Simulation results show that GreenHDFS is capable of achieving 26% savings [68].

2.3.3.2 Complex Execution Paths

MapReduce reads a single input and produces a single output in a fixed execution path. Many execution plans require more complex path. There are solutions that implement the possibility to construct dataflow graphs with different possible paths [77]:

- Dryad: Execution engine that uses MapReduce but allows more general execution plans. A Dryad application combines computational "vertices" with communication "channels" to form a dataflow graph. Dryad runs the application by executing the vertices of this graph on a set of available computers, communicating as appropriate through files, TCP pipes, and shared-memory FIFOs [63]
- Clustera: Designed for extensibility, it enables the system to be easily extended to handle a wide variety of job types ranging from computationally-intensive, long-running jobs with minimal I/O requirements to complex SQL queries over massive relational tables. Another unique feature of Clustera is the way in which the system architecture exploits modern software building blocks including application servers and relational database systems in order to carry out important performance, scalability, portability and usability benefits. [28]
- Nephele/PACT: Centered around a programming model of so called Parallelization Contracts (PACTs) and the scalable parallel execution engine Nephele. The system as a whole is designed to be as generic as (and compatible to) MapReduce systems, while overcoming several of their major weaknesses: 1) The functions `Map` and `Reduce` alone are not sufficient to express many data processing tasks both naturally and efficiently. 2) MapReduce ties a program to a single fixed execution strategy, which is robust but highly suboptimal for many tasks. 3) MapReduce makes no assumptions about the behavior of the functions. [122, 3]

2.3.3.3 Declarative Languages

Developers working with MapReduce must code their own `Map` and `Reduce` procedures. If optimization is required, they could need to code Sort, Combine or Merge operations. Declarative languages have been developed to abstract queries from program logic. This allows query independence, reuse and optimization. Proposals of declarative languages include:

- Pig: Offers SQL-style high-level data manipulation constructs, which can be assembled in an explicit data flow and interleaved with custom Map- and Reduce-style functions or executables. Pig programs are compiled into sequences of Map-Reduce jobs, and executed in the Hadoop MapReduce environment. [95, 43].
- HiveQL: Queries are expressed in a SQL-like style, and are compiled into MapReduce jobs that are executed using Hadoop. In addition, HiveQL enables users to plug in custom MapReduce scripts into queries. [115, 116].
- SCOPE: Designed for ease of use with no explicit parallelism, while being amenable to efficient parallel execution on large clusters. SCOPE borrows several features from SQL. Data are modeled as sets of rows composed of typed columns. The select statement is retained with inner joins, outer joins, and aggregation allowed. Users can easily define their own functions and implement their own versions of operators: extractors (parsing and constructing rows from a file), processors (row-wise processing), reducers (group-wise processing), and combiners (combining rows from two inputs). SCOPE supports nesting of expressions but also allows a computation to be specified as a series of steps [19].
- DryadLINQ: A DryadLINQ program is a sequential program composed of LINQ expressions performing arbitrary side-effect-free operations on datasets, and can be written and debugged using standard .NET development tools. The DryadLINQ system automatically and transparently translates the data-parallel portions of the program into a distributed execution plan which is passed to the Dryad execution platform [62].

2.3.3.4 Progress Estimation

Hadoop speculative task scheduling compares node progress with the average. This approach can present problems when applied over heterogeneous hardware, in which progress rate can vary from one node to another [77]. Solutions to deal with this issue include:

- Longest Approximate Time to End (LATE): Proposes estimating task finalization through individual progress rate. Improves Hadoop response times by a factor of 2 in heterogeneous cluster environments [128].
- Parallax: Targets environments where queries consist of a sequential paths of MapReduce jobs and is fully implemented in Pig. It handles varying processing speeds and degrees of parallelism during query execution [91]
- ParaTimer: Estimates the progress of queries that translate into directed acyclic graphs of MapReduce jobs, where jobs on different paths can execute concurrently. The essential techniques involve identifying the critical path for the entire query and producing multiple time estimates for different assumptions about future dynamic conditions. [90]
- KAMD: Uses an estimate of the time required to process a single record for each phase of each job and an estimate of the number of records that remain to be processed [89]

2.3.3.5 Multi-user Task Scheduling

In many multi-user production environments, MapReduce jobs often perform similar tasks. MapReduce schedulers don't take this into account, causing duplicate work.

MRShare is a solution for environments where different jobs often perform similar work, and there are many opportunities for sharing. It transforms a batch of queries into a new batch that will be executed more efficiently, by merging jobs into groups and evaluating each group as a single query [70].

2.3.3.6 Global State Information

MapReduce does not store global state information. Complex algorithms that use state information are hard to implement in MapReduce, and require lots of I/O usage and complex computations. There are some advances that include state information in a MapReduce job execution:

- HaLoop: Haloop extends MapReduce with programming support for iterative applications, it also dramatically improves their efficiency by making the task scheduler loop-aware and by adding various caching mechanisms. Compared with Hadoop, on average, HaLoop reduces query runtimes by 1.85 [16]
- Twister: Programming model and the architecture of an enhanced MapReduce runtime that supports iterative MapReduce computations efficiently [31]
- Pregel: Implements a programming model motivated by the Bulk Synchronous Parallel(BSP) model. In this model, each node has each own input and transfers only some messages which are required for next iteration to other nodes [84, 77]

2.3.3.7 Multiple Inputs

MapReduce is originally designed to read a single input and generate a single output. Algorithms that require multiple inputs or outputs are not well supported. Here are presented some solutions to deal with this issue:

- Map-Reduce-Merge: Adds to Map-Reduce a Merge phase that can efficiently merge data already partitioned and sorted (or hashed) by `Map` and `Reduce` modules. This model can express relational algebra operators as well as implement several join algorithms [127].
- Map-Join-Reduce: Proposes a filtering-join-aggregation programming model, a natural extension of MapReduce's filtering-aggregation programming model. It also presents a data processing strategy which performs filtering-join-aggregation tasks in two successive MapReduce jobs. The first job applies filtering logic to all the datasets in parallel, joins the qualified tuples, and pushes the join results to the reducers for partial aggregation. The second job combines all partial aggregation results and produces the final answer. The advantage is that it joins multiple datasets in one go and thus avoids frequent checkpointing and shuffling of intermediate results, a major performance bottleneck in most of the current MapReduce based systems. [64]
- Tuple MapReduce: Allows to bridge the gap between the low-level constructs provided by MapReduce and higher-level needs required by programmers, such as compound records, sorting or joins. It also presents Pangool as an opensource framework implementing Tuple MapReduce [37]

2.3.3.8 Blocking Operators

`Map` and `Reduce` functions are blocking operations in that all tasks should be completed to move forward to the next stage or job. The reason is that MapReduce relies on external merge sort for grouping intermediate results. Merge phase is I/O intensive. This property causes performance degradation and makes it difficult to support online processing [65, 79].

- Incremental MapReduce: Combines MapReduce abstraction with a wide-scale distributed stream processor. Incremental MapReduce operators avoid data re-processing, and the stream processor manages the placement and physical data flow of the operators across the wide area [83].
- MapReduce Online: Proposes a modified MapReduce architecture that allows data to be pipelined between operators. This extends the MapReduce programing model beyond batch processing, and can reduce completion times and improve system utilization for batch jobs as well. It also supports online aggregation, which allows users to see "early returns" from a job as it is being computed, and continuous queries, which enable MapReduce programs to be written for applications such as event monitoring and stream processing. [22]
- Intermediate Hash tables: Proposes a new data analysis platform that employs hash techniques to enable fast in-memory processing, and a frequent key based technique to extend such processing to workloads that require a large key-state space. It improves the progress of map tasks, allows the `Reduce` progress to keep up with the `Map` progress, with up to 3 orders of magnitude reduction of internal data spills, and enables results to be returned continuously during the job. [79]

2.3.3.9 Distributed Databases

MapReduce default framework operates over data files replicated in a distributed file system. These data are schema-free and index-free, so parsing each element is a requirement. If semi-structured data are used, it can be used to check data integrity, but data size may grow as data contains schema information in itself [77]. There are some approaches of distributed databases that try to make data access more efficient:

- HadoopDB: Hybrid solution of parallel DBMS and Hadoop approaches to data analysis. The ability of HadoopDB to directly incorporate Hadoop and open source DBMS software (without code modification) makes HadoopDB particularly flexible and extensible for performing data analysis at the large scales expected of future workloads [1].
- SQL/MapReduce: Hybrid framework that enables to execute user-defined functions in SQL queries across multiple nodes in MapReduce-style [40, 77]
- BigTable: Column-based database. A Bigtable is a sparse, distributed, persistent multidimensional sorted map. The map is indexed by a row key, column key, and a timestamp; each value in the map is an uninterpreted array of bytes. [20]
- Hadoop++: Changes the internal layout of a split — a large horizontal partition of the data — and/or feeds Hadoop with appropriate user-defined functions. It also proposes new index and join techniques: Trojan Index and Trojan Join, to improve runtimes of MapReduce jobs [29].
- Dremel: Combines multi-level execution trees and columnar data layout [87]

- Teradata's Hadoop and PDBMS integration: Present three efforts towards tight integration of Hadoop and Teradata EDW. DirectLoad approach provides fast parallel loading of Hadoop data to Teradata EDW. TeradataInputFormat approach allows MapReduce programs efficient and direct parallel access to Teradata EDW data without external steps of exporting and loading data from Teradata EDW to Hadoop. SQL users can directly access and join Hadoop data with Teradata EDW data from SQL queries via user defined table functions. [126]

- HBase: Hadoop's columnar database that extends the Bigtable model with secondary indexes and filters that reduce data transferred over the network [44]

- RCFile (Record Columnar File): Stores row groups with column-wise data compression to provide efficient storage space utilization. RCFile is read-optimized by avoiding unnecessary column reads during table scans [54].

- Llama: A hybrid data management system which combines the features of row-wise and column-wise database systems. In Llama, columns are formed into correlation groups to provide the basis for the vertical partitioning of tables. It possess a join algorithm to facilitate fast join processing and exploits the map-side join to maximize the parallelism and reduce the shuffling cost [81].

2.3.3.10 Intra-node parallelism

In some studies, MapReduce model is used in intra-node parallelism, such as multi-core environments [101, 66] or GPU computations [18, 53]. There are also specific studies regarding Cell architectures [26]. In those cases, a core substitutes the role of a node, and data are transferred through shared memory. Fault tolerance features are usually discarded [77].

Chapter 3

State of the Art

This chapter describes the current State of the Art on MapReduce-based applications to scalability issues of the Semantic Web (see section 2.1.2).

1. *SPARQL Query Resolution:* The majority of the works deal with efficient querying of RDF graphs. Section 3.1 classifies, analyzes, and compares the most prominent solutions.
2. *RDFS and OWL inference:* MapReduce-based solutions to RDF and OWL reasoning focus on computing a graph closure using forward-chaining entailment. That is, materializing all triples which can be inferred from the original dataset using RDFS and/or OWL rules. This solutions are described in section 3.2.
3. *RDF Compression:* Urbani et al. [121] deal with RDF compression using MapReduce, proposing an algorithm based on dictionary encoding. Their work is portrayed in section 3.3.

3.1 SPARQL Query Resolution

Different MapReduce-based solutions have been proposed for SPARQL resolution on a large scale. Thus, we propose a specific classification which allows us to review these solutions in a coherent way. To do this, we divide the technique according to the complexity that their nodes must assume within the MapReduce cluster. We consider two main families that we refer to as **native** and **hybrid** solutions.

Native solutions. This first family considers all solutions relying exclusively on the MapReduce framework. That is, queries are resolved with a series of MapReduce jobs on low complexity nodes, typically disregarding the help of specific semantic technology in each node of the cluster. Thus, the overall SPARQL resolution performance mainly depends on (a) physical tuning decisions, and (b) processing strategies which reduce the number of required jobs. As for (a), we consider two main decisions to optimize the physical aspects:

- *Data storage* is one the main bottlenecks on the MapReduce framework. Note that data are persistently read from disk and stored again to communicate `Map` and `Reduce` stages. Thus, how the data are organized for storage purposes in HDFS is one of the considerations addressed by the existing solutions.
- *Data indexing* is an immediate improvement for reducing the aforementioned I/O costs. For this purpose, some approaches use a NoSQL database (in general, HBase) on top of HDFS.

Solution Type	Purpose	Solution	Reason	Papers
Native solutions	Simplify automated processing	Single line notations	Each triple is stored in a separate line	[59], [60], [61], [93]
	Reduce storage requirements	Substitution of common prefixes by IDs	Data size reduction	[59], [60], [61]
		Division of data in several files by predicate and object type	Only files with corresponding TPs will be read	
	Improve data processing speed	Storage of all triples with the same subject in a single line	Improve reading speed of queries with large number of results	[104]
		Map-side joins	Reduce data shuffled and transferred between tasks	[48], [106]
		NoSQL solutions	Provide indexed access to triples	[112], [97], [106]
	Reduce number of MapReduce jobs	Greedy algorithm	Optimize Star Queries	[59], [60], [93]
		Multiple Selection algorithm	Optimize Path Queries	
		Early elimination heuristic	Prioritize jobs that completely eliminate variables	[61]
		Clique-based heuristic	Resolve queries with map-side joins	[48]
Hybrid solutions	Allow parallel subgraph processing	Graph Partitioning	Each node receives a significant subgraph	[58], [76]
		Parallel subgraph processing	Each node resolves subgraph joins	

Table 3.1: Classification of MapReduce-based solutions addressing SPARQL resolution.

These approaches leverage database indexes to improve RDF retrieval within individual nodes, therefore improving TP resolution within the `Map` stage. These random access capabilities allow more complex `Map` and `Reduce` stages to be implemented in practice.

Regardless of this physical tuning, all approaches carry out specific strategies for reducing the number of MapReduce jobs required for general SPARQL resolution. Both physical decisions and these processing strategies are reviewed in Section 3.1.1 within the corresponding approaches.

Hybrid solutions. This second family, in contrast to native solutions, deploys MapReduce clusters on more complex nodes, typically installing mono-node state-of-the-art RDF stores. This decision allows each node to partially resolve SPARQL queries on its stored subgraph, so that MapReduce jobs are only required when individual node results must be joined. In this scenario, *triples distribution* is an important decision to minimize (or even avoid) the number of MapReduce jobs required for query resolution.

Table 3.1 summarizes the most relevant solutions, in the state of the art, according to the above classification. The corresponding papers are reviewed in the next two sections.

In addition to the reviewed papers, it is worth noting that there exist some other proposals

[110, 103, 105, 69] which are deployed on top of additional frameworks running over Hadoop, such as Pig [95] or Hive [115, 116]. These solutions use high-level languages over Hadoop to hide `Map` and `Reduce` task complexities from developers. However, we do not look at them in detail because their main contributions are related to higher level details, while their internal MapReduce configurations respond to the same principles discussed below.

3.1.1 Native solutions

This section reviews the most relevant techniques within the native solutions. Attending to the previous classification, we analyze solutions running on native HDFS storage (Section 3.1.1.1), and NoSQL-based proposals (Section 3.1.1.2).

3.1.1.1 HDFS-based Storage

Before going into detail, it is worth noting that all these solutions use **single line notations** for serializing RDF data in plain files stored in HDFS. Some solutions use straight N-Triples format [49] for storage purposes [93], while others preprocess data and transform them to their own formats. This simple decision simplifies RDF processing, because triples can be individually parsed line-by-line. In contrast, formats like RDF/XML [4] force the whole dataset to be read in order to extract a triple [59, 60, 93, 102, 58].

RDF storage issues are addressed by Rohloff and Schantz [104], Husain et al. [59, 60, 61] and Goasdoué and Kaoudi [48] in order to reduce space requirements and data reading on each job. Rohloff and Schantz [104] transform data from N3 into a plain text representation in which triples with the same subject are stored in a single line. Although it is usually not an effective approach for query processing (in which a potentially small number of triples must be inspected), it is adequate in the MapReduce context because large triple sets must be scanned to answer less-selective queries [104].

Another immediate approach is based on common **RDF prefix substitution** [59]. In this case, all occurrences of RDF terms (within different triples) are replaced by short IDs which reference them in a dictionary structure. It enables spatial savings, but also parsing time because the amount of read data is substantially reduced.

Husain et al. [60] focus on storage requirements and I/O costs by **dividing data into several files**. This decision allows data to be read in a more efficient way, avoiding the reading of the whole dataset in its entirety. This optimization comprises two sequential steps:

1. A predicate-based partitioning is firstly performed. Thus, triples with the same predicate are stored as pairs `(subject,object)` within the same file.

2. Then, these files are further divided into *predicate-type* chunks in order to store together resources of the same type (e.g. `ex:student` or `ex:degree` in the example in Figure 2.2). This partitioning is performed in two steps:

   (a) *Explicit type information* is firstly used for partitioning. That is, `(subject,object)` pairs, from the `rdf:type` predicate file, are divided again into smaller files. Each file stores all subjects of a given type, enabling resources of the same type to be stored together.

   (b) *Implicit type information* is then applied. Thus, each predicate file is divided into as many files as different object types it contains. This division materializes the *predicate-type* split. Note that this is done with the information generated in the previous step. Thus, each chunk contains all the `(subject,object)` pairs for the same predicate and type.

General query resolution performs on an iterative algorithm which looks for the files required for resolving each TP in the query. According to the TP features, the algorithm proceeds as follows:

- If both predicate and object are variables (but the type has been previously identified), the algorithm must process all *predicate-type* files for the retrieved type.
- If both predicate and object are variables (but the type has not been identified), or if only the predicate is variable, then all files must be processed. Thus, the algorithm stops because no savings can be obtained.
- If the predicate is bounded, but the object is variable (but the type has been previously identified), the algorithm only processes the *predicate-type* file for the given predicate and the retrieved type.
- If the predicate is bounded, but the object is variable (but the type has not been identified), all *predicate-type* files must be processed.

For instance, when making split selection for the query of Figure 2.4, the selected chunks would be:

- `TP1`: type file `ex:student`.
- `TP2`: type file `ex:degree`.
- `TP3`: predicate-type file `ex:study-ex:degree`.
- `TP4`: predicate-type file `ex:hasProfessor-ex: professor`.
- `TP5`: type file `ex:professor`.

Goasdoué and Kaoudi [48] focus their attention on another MapReduce issue: the job performance greatly depends on the amounts of intermediate data shuffled and transmitted from `Map` to `Reduce` tasks. Their goal is to partition and place data so that most of the joins can be resolved in the Map phase. Their solution replaces the HDFS replication mechanism by a personalized one, where each triple is also replicated three times, but in three different types of partition: subject partitions, property (predicate) partitions, and object partitions. For a given resource, the subject, property, and object partitions of this resource are placed in the same node. In addition, subject and object partitions are grouped within a node by their property values. The property `rdf:type` is highly skewed, hence its partition is broken down into smaller partitions to avoid performance issues. In fact, property values in RDF are highly skewed in general [71], which can cause property partitions to differ greatly in size. This issue is addressed by defining a threshold: when creating a property partition, if the number of triples reaches the threshold, another partition is created for the same property. It is important to note that this replication method improves the performance, but at the cost of fault-tolerance: HDFS standard replication policy ensures that each item of the data is stored in three different nodes, but with this personalized method, this is not necessarily true.

Husain et al. [59] and Myung et al. [93] focus on **reducing the number of MapReduce jobs** required to resolve a query. Both approaches use algorithms to select variables in the optimal way. Their operational foundations are as a follows. In a first step, all TPs are analyzed. If they do not share any variable, no joins are required and the query is completed in a single job. Note that while a cross product would be required in this case, this issue is not addressed by these works. Otherwise, there are TPs with more than one variable. In this case, variables must be ordered and two main algorithms are considered:

- The *greedy algorithm* promotes variables participating in the highest number of joins.
- The *multiple selection algorithm* promotes variables which are not involved in the same TP because these can be resolved in a single job.

Both algorithms can be combined for variable ordering. Whereas Myung et al. [93] make an explicit differentiation between the algorithms, Husain et al. [59] implement a single algorithm which effectively integrates them. These algorithms are simple, report quick performance, and lead to good results. However, they are not always optimal. In a later work, Husain et al. [60] obtain each possible job combination and select the one reporting the lowest estimate cost. This solution, however, is reported as computationally expensive.

Husain et al. [61] revisit their previous work and develop *Bestplan*, a more complex solution for TPs selection. In this solution, jobs are weighted according to their estimated cost. The problem is then defined as a search algorithm in a weighted graph, where each vertex represents a state of TPs involved in the query, and edges represent a job to make the transition from one state to another. The goal is to find the shortest weighted path between the initial state v_0, where each TP is unresolved, to the final state v_{goal}, where every TP is resolved. However, it is possible (in the worst case) that the complexity of the problem were exponential in the number of joining variables. Only if the number of variables is small enough, is it a feasible solution for generating the graph and finding the shortest path.

For higher numbers of joining variables, the *Relaxed-Bestplan* algorithm is used. It assumes uniform cost for all jobs; *i.e.*, the problem is to find the minimum number of jobs. This concern can also be infeasible, but it is possible to implement a greedy algorithm that finds an upper bound on the maximum number of jobs that performs better than the greedy algorithm defined in [59]. This algorithm is based on an *early elimination heuristic*. That is, jobs that *completely eliminate* variables are selected first, where *complete elimination* means that this variable is resolved in every TP it appears.

On the other hand, Goasdoué and Kaoudi [48] represent the query as a *query graph* where nodes are TPs, and edges model join variables between them (these are labeled with the name of the join variables). Then, the concept *clique subgraph* is proposed from the well-known concept of *clique* in Graph Theory; a clique subgraph G_v is the subset of all nodes which are adjacent to the edge labeled with the variable v (*i.e.: triples that share variable* v). Using this definition, possible queries are divided into the following groups:

- *1-clique query:* where the query graph contains a single clique subgraph. These queries can be resolved in the `Map` stage of a single job, because the join can be computed locally at each node.
- *Central-clique query:* where the query graph contains a single clique subgraph that overlaps with all other clique subgraphs. These queries can be resolved in a single complete job. The query can be decomposed into 1-clique queries that can be resolved in the `Map` phase of a MapReduce job, and then the results of these joins can be joined in the variable of the common clique on the `Reduce` stage.
- *General query*: This is neither 1-clique nor central-clique query. These queries require more than one job to be resolved. A greedy algorithm, referred to as *CliqueSquare*, is used to select join variables. This algorithm decomposes queries into clique subgraphs, evaluates joins on the common variables of each clique, and finally collapses them. All this processing is implemented in a MapReduce job. As cliques are collapsed, each node in the query graph represents a set of TPs.

3.1.1.2 NoSQL Solutions

In order to improve RDF retrieval in each node, some approaches use the NoSQL distributed database *HBase* [112, 97, 106] on top of HDFS. These solutions perform triples replication, in diverse tables, and use some indexing strategies to speed up query performance for TPs with distinct unbound variables. The main drawback of these approaches is that each triple is now stored many times, one for each different table, and this spatial overhead is added to the HDFS replication itself (with a default replication factor of three).

Sun and Jin [112] propose a sextuple indexing similar to the one of Hexastore [123] or RDF3X [94]. It consists of six indexes: *S_PO*, *P_SO*, *O_SP*, *PS_O*, *SO_P*, and *PO_S*, which cover all combinations for unbound variables. Thus, all TPs are resolved with one access to the corresponding index.

Papailiou et al. [97] reduce the number of tables to three, corresponding to indexes *SP_O*, *OS_P*, and *PO_S*. TPs with only one bound resource are resolved with a range query `[<resource>, increment(<resource>)]`. To further improve query performance, all indexes store only the 8-byte MD5Hashes of `{s,p,o}` values; a table containing the reverse MD5Hash to values is kept and used during object retrieval.

Schätzle and Przyjaciel-Zablocki [106] reduce the number of tables even more, using only two, corresponding to indexes *S_PO* and *O_PS*. The *HBase Filter API* is used for TPs which bound `(subject and object)` or `(object and predicate)`. In turn, predicate bounded TPs can be resolved using the *HBase Filter API* on any of the two tables.

Although these solutions rely on HBase for triple retrieving, join operations are still performed via `Map` and `Reduce` operations. Sun and Jin [112] use similar algorithms to that described previously for [59, 93]. Papailiou et al. [97], on the contrary, develop a complex join strategy where joins are performed differently depending on BGP characteristics. The different join strategies are:

- The **map phase join** is the base case and follows the general steps described in section 2.3.2. That is, the mappers read the triples and emit *(key, value)* pairs in which i) the *keys* correspond to the join variable bindings, and ii) the *values* correspond to the bindings for all other variables included in the TPs of the join (if exist) calculated in previous jobs. In turn, the reducers merge, for each key, the corresponding list of values in order to perform the join. Although this strategy is named *map phase join*, the actual join is performed in the `Reduce` phase (the name only refers to when data are extracted from HBase).

- The **reduce phase join** is used when one of the TPs retrieves a very small number of input data compared to the rest. In this case, the `Map` stage is the same as in the *Map phase join*, but only this TP is used as input. It is in the `Reduce` stage where, for each mapped binding, data are retrieved if they match other TPs.

- The **partial input join** is similar to *Reduce phase join*, but allows an arbitrary number of TPs to be selected for extracting their results in the `Map` stage. These selection use information gathered during the bulk data loading to HBase.

- Instead of launching a MapReduce job, the **centralized join** performs the join operation in a single node. This is only an efficient choice when the input data size is small, and the initialization overhead of a MapReduce job is a major factor in query performance.

Furthermore, Schätzle and Przyjaciel-Zablocki [106] develop a join strategy named *Map-Side Index Nested Loop Join* (*MAPSIN join*). This strategy performs the join in the `Map` stage instead of the `Reduce` one. It firstly performs a distributed table scan for the first TP, and retrieves all local

results from each machine. For each possible variable binding combination, the `Map` function is invoked for retrieving compatible bindings with the second TP. The computed multiset of solutions is stored in HDFS. This approach highly reduces the network bandwidth usage, as only compatible data for the second TP needs to be transferred to the nodes which run the `Map` tasks. Note that, when the join is materialized in the `Reduce` stage, all possible bindings are transferred from `Map` to `Reduce` tasks. Joins involving three or more TPs are computed successively. For each additional TP, a `Map` stage is performed after joining the previous TPs. In this case, the `Map` function is invoked for each solution obtained in the previous joins. Finally, in the case of multi-way joins, compatible bindings for all TPs are retrieved from HBase in a single `Map` stage. Finally, for queries involving a high selective TP (retrieving few results), the `Map` function is invoked in one machine for avoiding the MapReduce initialization.

3.1.2 Hybrid Solutions

The approaches reviewed in the previous section strictly rely on the MapReduce framework for resolving SPARQL queries. Some other techniques introduce specific semantic technology in each node of the cluster. In general, this class of solutions deploy an RDF store in each cluster machine and distribute the whole dataset among all nodes. The motivation behind this idea is to manage a significant RDF subgraph in each node in order to minimize inter-node communication costs. In fact, when join resolution can be isolated within a single node, the complete query is resolved in parallel without using MapReduce. These queries are defined as *Parallelizable Without Communication* (PWOC) by Huang et al. [58]. In this case, the final results are just the addition of each node output (note that a cross product would be required in this case, but this is not considered in the original paper). If a query is not PWOC, it is decomposed in parallelizable queries and their result is finally merged with MapReduce.

A straightforward, but smart, data distribution performs hashing by subject, object or subject-object (resources which can appear as subject or object in a triple) [76]. This hash-based partitioning is also used in multiple distributed RDF Stores such as YARS2 [52], Virtuoso Cluster [32], Clustered TDB [96], or CumulusRDF [72]. In the current scenario, hash-partitioning enables TPs sharing a variable resource to be resolved without inter-node communication. This result is especially interesting for star-shaped query resolution, but more complex queries require intermediate results to be combined and this degrades the overall performance [58]. Edge-based partitioning is another effective means of graph distribution. In this case, triples which share subject and object are stored in the same node. However, triples describing the same subject can be stored in different nodes, hindering star-shaped query resolution [58]. Finally, Huang et al. [58] and Lee and Liu [76] perform a vertex-based partitioning. By considering that each triple models a graph edge, this approach distributes subsets of closer edges in the same machines. This partitioning involves the following three steps:

1. The whole graph is vertex-based partitioned in disjoint subsets. This class of partitioning is well-known in Graph Theory, so standard solutions such as the *Metis partitioner* [67], can be applied.

2. Then, triples are assigned to partitions.

3. Partitions are finally expanded through controlled triple replication.

It is worth noting that `rdf:type` generates undesirable connections: every resource is at two hops of any other resource of the same type. These connections make the graph more complex and reduce the quality of graph partitioning significantly. Huang et al. [58] remove triples with

predicate `rdf:type` (along with triples with similar semantics) before partitioning. Highly-connected vertices are also removed because they can damage quality in a similar way. In this case, a threshold is chosen and those vertices with more connections are removed before partitioning.

Huang et al. [58] create partitions including triples with the same subject. In turn, Lee and Liu [76] obtain partitions for three different kinds of groups:

- *Subject-based triple groups*, comprising those triples with the same subject.
- *Object-based triple groups*, comprising those triples with the same object.
- *Subject-object-based triple groups*, comprising those triples with the same subject *or* object.

The overall query performance could be improved if triples replication is allowed. It enables larger subgraphs to be managed and queried, yielding configurable space-time tradeoffs [58]. On the one hand, storage requirements increase because some triples may be stored in many machines. On the other hand, query performance improves because more queries can be locally resolved. Note that the performance gap between completely parallel resolved queries and those requiring at least one join is highly significant. Huang et al. [58] introduce two particular definitions to determine the best triple replication choice:

- *Directed n-hop guarantee:* an *n-hop guarantee* partition comprises all vertices which act as objects in triples whose subject is in an *(n-1)-hop guarantee* partition. The *1-hop guarantee* partitions corresponds to the partition created by the vertex partitioning method previously described.
- *Undirected n-hop guarantee:* this is similar to the previous one, but it includes each vertex linked to a vertex in the *(n-1)-hop guarantee* partition (*i.e.* vertices which are subject of a triple having its object on the *(n-1)-hop guarantee*).

Lee and Liu [76] propose comparable definitions:

- *k-hop forward direction-based expansion* is similar to *directed n-hop guarantee*. It adds triples with the subject acting as the object of a triple in the partition.
- *k-hop reverse direction-based expansion* adds triples with an object which appears as the subject of a triple in the partition.
- *k-hop bidirection-based expansion* is similar to *undirected n-hop guarantee*. Thus, it adds triples with a resource playing any resource role for a triple in the partition.

Example. We illustrate these definitions using the RDF excerpt shown in Figure 2.2. Let us suppose that triples are partitioned by subject, and each one is assigned to a different partition. In this case, the *1-hop guarantee* or any type of *1-hop expansion* would simply obtain the initial partitions without adding any additional triples. This is shown in Figure 3.1(a).

The 2-hop partitions are obtained by including triples "related" to those in the corresponding 1-hop partition. How this relationship is materialized depends on the type of partition. *Directed 2-hop guarantee* and *2-hop forward direction-based expansion* add triples whose subject is object in any triple within the 1-hop partition. In the current example, the triple `(ex:C1, ex:hasProfessor, ex:P1)` is added to partitions 1 and 2, but the partition 3 is unchanged because there are no more triples with their subject included in the 1-hop partition. The resulting partitions are shown in Figure 3.1(b).

The *2-hop reverse direction-based expansion*, illustrated in Figure 3.1(c), add triples whose object is a subject in any triple within the 1-hop partition. For the current example, partitions 1 and 2 remain unchanged whereas the partition 3 adds `(ex:S1, ex:study, ex:C1)` and `(ex:S2, ex:study, ex:C1)`.

The *undirected 2-hop guarantee* and *2-hop bidirection-based expansion* add triples whose subject or object appear, as subject or object, in any triple within the 1-hop partition. In our current example, `(ex:C1, ex:hasProfessor, ex:P1)` is added to partitions 1 and 2 because their subject (`ex:C1`) is already in the partition. In turn, `(ex:S1, ex:study, ex:C1)` and `(ex:S2, ex:study, ex:C1)` are added to the partition 3 because their object (`ex:C1`) is also in the partition. The resulting partitions are illustrated in Figure 3.1(d).

The subsequent 3 and 4-hop partitions are obtained following the same decisions. It can be tested that, in this example, both *undirected 4-hop guarantee* and *4-hop bidirection-based expansion* include the whole graph. □

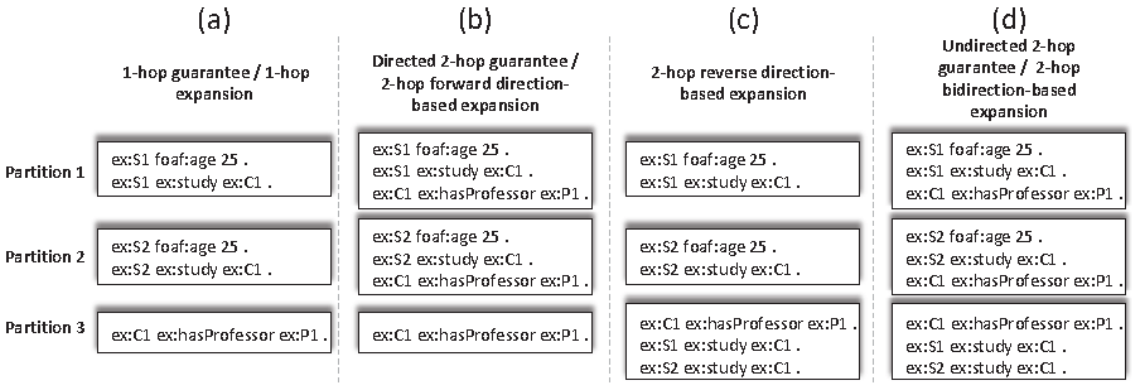


Figure 3.1: Example of different hop partitions

This n-hop review leads to an interesting result: in fully connected graphs, both *undirected k-hop guarantee* and *k-hop bidirection-based expansion* partitions will eventually include the whole graph if n/k is sufficiently increased. However, this is not true for directed guarantees/expansions, as some resources can be connected by the direction which is not considered [58].

To determine if a query is completely resolved in parallel, centrality measures are used. Huang et al. [58] use the concept of *Distance of Farthest Edge* (DoFE). The vertex of a query graph with the smallest DoFE will be considered as the *core*. If the DoFE of the core vertex is less than or equal to the n-hop guarantee, then the query is PWOC. It is worth noting that if directed n-hop guarantee is used, the distance must be measured considering the query as a directed graph; if undirected n-hop guarantee is used, the query can be considered as an undirected graph. Lee and Liu [76] propose similar definitions with the name of *center vertex* and *radius*. Radius can be calculated as *forward radius* (when using forward direction-based expansion), *reverse radius* (when using reverse direction-based expansion), or *bidirectional radius* (when using bidirectional-based expansion). In these cases, the query graph must be considered as directed, inversely directed, and undirected.

If triples replication is used, it is possible that more than one partition could resolve a query. This could lead to duplicate results when resolving PWOC queries. Huang et al. [58] address this issue in the following way: when a partition is created, additional triples with the form `(`v`,<isOwned>,"Yes")` are added, where v corresponds to core vertexes of the partition (*i.e.* not

added as an n-hop guarantee). When resolving a query in parallel, an additional TP with the form (*core*,<isOwned>,"Yes") is added to the query.

3.1.3 Analysis of Results

This section summarizes the experimental results provided by the authors of the most prominent techniques described in the previous sections. It is worth mentioning that any performance comparison would be unfair, as the solutions are tested under different configurations and most of them do not compare to each other. These variations include different node configurations and cluster compositions; the version of Hadoop used in the experiments and also its configuration; and the datasets size. Nevertheless, all of them use the well-known Lehigh University Benchmark[51] (LUBM), obtaining datasets from LUBM(100) to LUBM(30K). This benchmark allows synthetic data, of arbitrary size, to be generated from a university ontology. It also provides a set of 14 queries varying in complexity. Thus, we aim to analyze how solutions face two correlated dimensions: i) **dataset size** and ii) **resolution performance** at incremental query complexity.

3.1.3.1 Native solutions on HDFS

As stated, native solutions running on HDFS make use exclusively of MapReduce infrastructure for SPARQL resolution. On the one hand, RDF is stored using different file configurations within HDFS. On the other hand, SPARQL queries are resolved with successive jobs across the nodes. It is worth noting that all techniques analyzed in this section use multi-way joins for query optimization.

The initial work by Husain et al. [59] proposes a promising specific optimization on the basis of organizing triples in files by certain properties. They perform, though, a reduced evaluation with a cluster of 10 nodes, aimed at testing the feasibility and scalability of the proposal. They report runtime for six queries from LUBM on incremental dataset sizes. The solution scales up to $1,100$ million triples and shows sublinear resolution time *w.r.t.* the number of triples. However, no comparisons are made against any other proposal, so its impact within the state of the art cannot be quantified. Their next solution [60] fills this gap and evaluates the approach (again on a cluster of 10 nodes) against mono-node stores: BigOWLIM[1] and Jena[2] (in-memory and the SDB model on disk). The latest solution by Husain et al. [61] compare their approach against the mono-node RDF3X [94]. In this latter comparison, larger datasets are tested, ranging from LUBM(10K), with 1.1 billion triples, to LUBM (30K), with 3.3 billion triples. In addition to LUBM, a subset of the SP^{2}Bench Performance Benchmark [107] is also used as evaluation queries. These last works reach similar conclusions. As expected, the Jena in-memory model is the fastest choice for simple queries, but it performs poorly at complex ones. Moreover, it runs out of memory on a large scale (more than 100 million triples). Jena SDB works with huge sizes, but it is one order of magnitude slower than HadoopRDF. In general, BigOWLIM is slightly slower in most dataset sizes, and slightly faster in the 1 billion dataset (mostly because of its optimizations and triple pre-fetch). A detailed review shows that BigOWLIM outperforms the MapReduce proposal in simple queries (such as Q12), whereas it is clearly slower in the complex ones (*e.g.* Q2 or Q9). They also evaluate the impact of the number of reducers, showing no significant improvement in the performance with more than 4 reducers. RDF3X performs better for queries with high selectivity and bound objects (*e.g.* Q1), but HadoopRDF outperforms RDF3X for queries with unbound objects, low selectivity, or joins on large amounts of data. Moreover, RDF3X simply cannot execute the two queries with unbound objects (Q2 and Q9) with the LUBM(30K) dataset.

[1] http://www.ontotext.com/owlim/editions

[2] http://jena.apache.org/

Goasdoué and Kaoudi [48] compare their solution with respect to HadoopRDF [61]. Datasets LUBM(10K) and LUBM(20K) are used in the tests. All the queries correspond to 1-clique or central-clique queries, and thus they can be resolved in a single MapReduce job. This allows CliqueSquare to outperforms HadoopRDF in each query by a factor from 28 to 59.

Myung et al. [93] focus their evaluation on testing their scalability with LUBM at incremental sizes. However, the maximum size is only LUBM(100), *i.e.* synthetic-generated triples from 100 universities. In contrast, Husain et al. [59] start their evaluation from 1,000 universities. Therefore, the very limited experimentation framework prevents the extraction of important conclusions. Nonetheless, they also verify the sublinear performance growth of the proposal (*w.r.t.* the input) and the significant improvement using multi-way joins versus two-way joins.

3.1.3.2 Native solutions on NoSQL

As explained before, this class of solutions replaces the plain HDFS storage by a NoSQL database in order to improve the overall data retrieval performance. The following results support this assumption, showing interesting improvements for query resolution.

Sun and Jin [112] make an evaluation using LUBM [51] datasets from 20 to 100 universities. Although no comparisons are made with respect to any other solutions, their results report better performance for growing dataset sizes. This result is due to the impact of MapReduce initialization decreases for larger datasets.

Papailiou et al. [97] compare themselves with the mono-node RDF3X and with the MapReduce-based solution HadoopRDF [61]. The experiments comprise a variable number of nodes for the clusters and a single machine for RDF3X. This machine deploys an identical configuration to that used for the nodes in the clusters. LUBM datasets are generated for 10,000 and 20,000 universities, comprising 1.3 and 2.7 billion triples respectively. Their solution shows the best performance for large and non-selective queries (Q2 and Q9), and outperforms HadoopRDF by far for centralized joins. Nonetheless, it is slightly slower than RDF3X for this case. Regarding scalability, execution times are almost linear *w.r.t.* the input when the number of nodes does not vary within the node, and decreases almost linearly when more nodes are added to the cluster.

Schätzle and Przyjaciel-Zablocki [106] perform an evaluation of their MAPSIN join technique over a cluster with 10 nodes. They also use the SP^{2}Bench in addition to LUBM. For LUBM, datasets from 1,000 to 3,000 universities are generated; for SP^{2}Bench, datasets from 200 million to 1 billion triples are generated. Their results are compared against PigSPARQL [105], another work from some of the same authors that uses Pig to query RDF datasets. Both approaches scale linearly, but MAPSIN on HBase enable an efficient way of Map-Side join because it reduces the necessary data shuffle phase. This allows join times to be reduced from 1.4 to 28 times with respect to the compared technique.

3.1.3.3 Hybrid solutions

The hybrid solutions aim to minimize the number of MapReduce jobs, resolving queries in local nodes and restricting the communication and coordination between nodes just for complex queries (cross-joins between nodes). This is only effective on the basis of a previous smart subgraph partitioning.

Huang et al. [58] establish a fixed dataset of 2,000 universities (around 270 million triples) for their evaluation, and do not compare incremental sizes. They perform on a cluster of 20 nodes, and their proposal is built with the RDF3X [94] triple store working in each single node. First, they compare the performance of RDF3X on a single node against the SHARD [104] native solution, showing that this latter is clearly slower because most joins require a costly complete redistribution

of data (stored in plain files). In contrast, subject-subject joins can be efficiently resolved thanks to the hash partitioning. Next, the performance of Huang et al.'s solution is evaluated against RDF3X on a single node. Once again, the simplest queries (Q1, Q3, Q4, etc.) run faster on a single machine, whereas the hybrid MapReduce solution dramatically improves the performance of complex queries (Q2, Q6, Q9, Q13 and Q14), ranging from 5 to 500 times faster. The large improvement is achieved for large clusters, because chunks are small enough to fit into main memory. In addition, they verify that the 1-hop guarantee is sufficient for most queries, except those with a larger diameter (Q2, Q8 and Q9), in which the 2-hop guarantee achieves the best performance and, in general, supports most SPARQL queries (given the small diameter of the path queries).

Finally, Lee and Liu [76] yield to a very similar approach. They also install RDF3X in each single node (20 nodes in the cluster), but their performance is only compared against a single-node configuration. In contrast, they perform on incremental sizes (up to 1 billion triples) and study different benchmarks besides LUBM. They also conclude that a 2-hop guarantee is sufficient for all queries (it leads to similar results to even a 4-hop guarantee) and, in each case, this subgraph partitioning is more efficient than the hash-based data distribution used, for instance, in SHARD [104]. The single-node configuration does not scale on most datasets, whereas the scalability of the MapReduce system is assured once the resolution time increases only slightly at incremental dataset sizes.

3.2 Reasonig

MapReduce-based reasoning solutions are based on generating the closure of the graph using forward-chaining materialization. That is, using the available data and the rules of either RDFS or OWL (see Figures 2.5 and 2.6) to derive new triples. Note that in all works OWL Horst is used. Hence, at query time there is no need of additional operations.

In a naive algorithm to compute a **RDFS graph closure**, all triples inferred by the rules are derived by MapReduce jobs. The issues of this solution are the focus of Urbani et al. [120]. They distinguish three main problems:

1. *Load balancing:* In essence, a MapReduce job joins and sends to the same reducer triples matching each inference rule. Those groups are consistently larger than others, which leads to load balancing issues.

2. *Duplicated triples:* Any given rule may derive the same triple using different input triples. Also, different rules may derive the same triple. This leads to duplicated output. In experimental results, the ratio of unique derived triples to duplicates is shown to be at least 1:50.

3. *Recursive derivation:* Derived triples can be also be used to generate new triples. Then, it is needed to chain more jobs and iterate until no more new triples are derived. The performance of the whole process greatly depend on the number of jobs needed to compute the closure.

To deal with those issues, they propose the following solutions:

1. *Loading schema triples in memory:* All the RDFS rules include at least one schema triple (`i.e.`, a triple which has an RDFS term). Given that schema triples are usually a small subset of a RDF dataset, they can be loaded in-memory on each node. With this approach, every node can receive any triple and apply the corresponding rules. Then, perfect load balancing can be achieved.

2. *Data grouping:* For any given rule, triples are grouped by the terms that are also used in the derived triple. Those terms are used as *key* in the MapReduce job. Since the key is used to partition the data, all triples that produce some new triple will be sent to the same reducer. It is then trivial to output that triple only once in the reducer. While this does not eliminate duplicate triples (because triples derived from different rules can still be duplicated), it greatly reduces the amount.

3. *Ordering the application of the RDFS rules:* After analyzing rules and their data dependencies (that is, which rule may be triggered by other rule), they devise an efficient rule ordering to minimize the number of required MapReduce jobs. The whole process is performed using four MapReduce jobs (three of which actually apply the rules and another one before the last job to remove duplicates), shown in Figure 3.2. Observe that rules 12 and 13 can produce triples that hypothetically may be used in previous jobs. In order to affect rule 5, there must be either a superproperty of `rdfs:member` or a subproperty of *p*. In order to affect rule 7 there must be some resources connected by *p*. The first case of rule 5 is ignored, following the advice against "ontology hijacking" from [57]. The other two cases, while theoretically possible, never appear in their experimental evaluation.

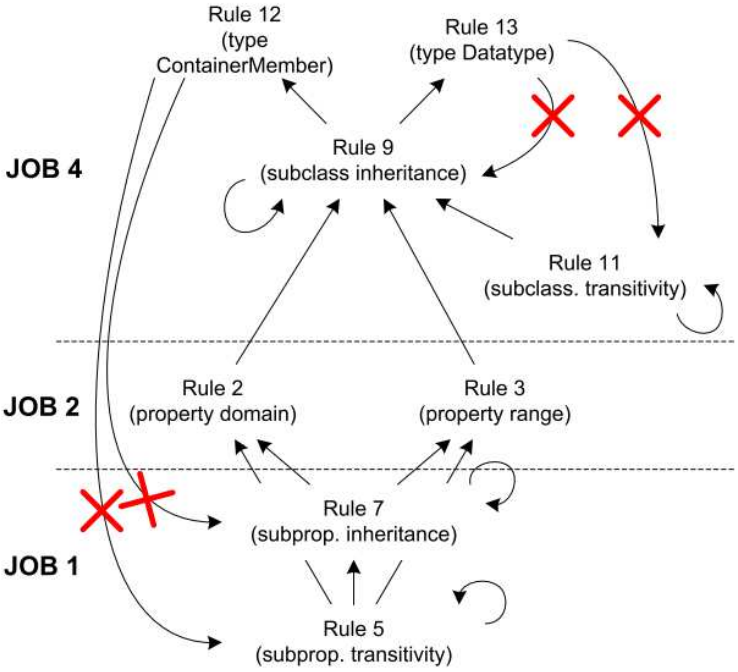


Figure 3.2: RDFS rules ordering [120]

OWL reasoning turns out to be more complex. Urbani et al. [119] identify new challenges introduced by OWL rules that prevent the usage of their previous solution. Those challenges are:

- *Joins between multiple instance triples:* While in RDFS all rules involve at most one instance triple, in OWL a number of rules contain two antecedents that can be matched by instance triples. Thus, the solution of loading schema triples on every node no longer works.

- *Exponential number of derivations:* Rules involving `owl:sameAs` derive an exponential number of triples, which becomes a performance bottleneck. For example, rule 11 derives $2^x \times n$ triples for a given term, where x is the number of synonyms of the term, and n the number of triples where the term appears.

- *Multiple joins per rule:* Some rules require more than one join between two antecedents, which makes impossible to derive them in a single MapReduce job.

- *Unknown number of iterations:* While in RDFS it is possible to identify an execution order with no loops to minimize the execution to a known number of MapReduce jobs, this is not possible in the case of OWL.

They propose three optimizations to perform the OWL closure of a graph. Those optimizations are described below.

- *Limiting duplicates* on rule 4. In this rule the inferred is likely to be used as antecedent again, inducing a chain of terms connected by a transitive relationship. Thus, this rule must be applied iteratively. This can lead to a great number of duplicates because the same triples are inferred again in every iteration. To avoid this issue they define the *distance* between two terms in the chain as the number of hops necessary to reach the second from the first one. Then, on iteration n only triples with distance greater or equal to 2^{n-2}. This strategy completely eliminates duplicates if no chains intersect; if they do intersect, duplicates are still generated, but in a much lesser degree.

- *Building a synonyms table* to avoid materialization of `owl:sameAs` derivations.

- *In-memory redundant join execution* for multiple joins (rules 15 and 16). For those rules the schema triples are loaded in memory on each node. Then, they perform the join between schema and the two other triples of the rules. This means that more joins than necessary will be performed, but as these joins are done with an in-memory data structure do not introduce a significant overhead in the process.

This job is extended by Liu et al. [82] to compute the closure of a graph using **fuzzy logic** where each triple is annotated with a fuzzy degree $n \in (1, n]$, using fuzzy OWL as entailment rule set. The key notion in fuzzy OWL semantics is called the *Best Degree Bound (BDB)* of a triple. The BDB of an derived triple is the largest fuzzy degree that can be derived by applying fuzzy OWL entailment rules, or 0 if no such fuzzy triple can be derived. The goal is, then, derive all possible triples and their BDBs. They focus their work in the following topics:

1. *Duplicates with different fuzzy degrees:* On each derivation step, the same triple with different fuzzy degree may be derived. Thus, an additional step must be included to ensure that only triples with maximal fuzzy degree are kept to the following job.

2. *Shortest path calculation:* In standard OWL inference, rules 4, 10 and 12 are essentially used to compute the transitive closure of an RDF graph. In fuzzy OWL, the dataset can be considered as a weighted graph, with the fuzzy degree as the weight. Thus, calculating the transitive closure is a variation of the all-pairs shortest path calculation problem. They implement an solution based on the Floyd-Warshall algorithm [39], maintaining an in-memory matrix that is iteratively updated on each step.

3. *sameAs rule:* On the one hand, traditional solutions to deal with the semantics of `owl:sameAs` cannot be used with fuzzy OWL withoud missing information. On the other hand, computing the closure by materializing the triples can be a performance bottleneck, as shown by Urbani et al. Urbani and Kotoulas. However, in practice, they verify that `sameAs` triples with a fuzzy degree lower than 1 are relatively few. Hence, they use the same solution as Urbani et al. Urbani and Kotoulas for certain `sameAs` triples, and materialize derived triples by vague `sameAs` triples.

Chen et al. [21] focus their work on reasoning over big biological knowledge networks with user-supplied rulesets. Specifically, they deal with efficiently derivation of **property chains**, which are common within the domain. While a naive algorithm with no improvements needs $n-1$ jobs to compute the chain, where n is the number of triples in the chain (i.e. one job for each derivation), they reduce the number of jobs to $\log n$. This is accomplished by adding a third join condition to perform a derivation: In a chain of property triples, the properties are given ascending IDs: P_0, P_1, P_2, and so on. Then, a triple whose property ID is an odd number P_k, this triple only performs joins with triples with property ID P_{k-1}. The input is divided into $\lceil N/2 \rceil$ groups, where N is the length of the longest property chain, and they perform joins between the triples from the same group in a job. The derived triples will be the new input graph for the next iteration.

Example. To illustrate the process, consider the following chain of property triples: $\{(S_0, P_0, O_0)$, $(O_0, P_1, O_1), (O_1, P_2, O_2), (O_2, P_3, O_3)\}$. The triples are divided into two groups: $G1$ consisting on the first two triples, and $G2$ including the last two. A first job computes the joins on each group, giving the following results: $\{(S_0, P_0 \otimes P_1, O_1), (O_1, P_2 \otimes P_3, O_3)\}$. Those triples are then grouped into a single group and another job generates the final output $\{(S_0, P_0 \otimes P_1 \otimes P_2 \otimes P_3, O_3)\}$. The process is completed with two jobs, instead of the 3 needed jobs (one for each derivation of two consecutive tripes) of a naive algorithm. □

3.3 RDF Compression

To the best of our knowledge, there is only one MapReduce-based solution to compress RDF datasets to date. Urbani et al. [121] perform dictionary encoding. They analyze the main challenges derived from the MapReduce framework:

1. *Data skewness:* RDF data has a high skew. This means that some reducers may receive groups of very different sizes. Given that the overall MapReduce processing time depends on the slowest task (i.e. the task which receives the larger input), it impacts the performance.

2. *Data domain:* In RDF dictionary encoding, the goal is to encode the different terms of a triple. However, the input is given in triples, not ir terms. They argue that a MapReduce algorithm would need to run three jobs, one for each role of the triples.

3. *Global IDs:* A MapReduce process by default has no means to manage global information. In order to assign the same ID to each term, a synchronized access to a global dictionary has to be implemented. This operation would need high network traffic, thus introducing a major bottleneck.

In their approach, they tackle those problems as follows:

1. *Preprocess popular IDs:* A first job randomly samples the input and identifies the most popular terms. Those terms are encoded with a numerical ID. When the job finishes, a dictionary is created for those terms. This dictionary will be distributed among the nodes of the cluster on the next job. Then, when a `Map` task reads a popular term, it will assign the ID retrieved in the dictionary and will send it to a random reducer. Thus, the impact of data skewness of popular terms is canceled.

2. *Split triples into terms:* A second job deconstructs triples and compress each term with a numerical ID. The purpose of this job is to avoid running a different job for each role of the triples. A third job reads the output and reconstructs the statements

3. *Partitioned IDs:* In the second job, the range of IDs is partitioned among the reducers of the job. When a reducer receives a non-popular term it is encoded using a ID within its partition.

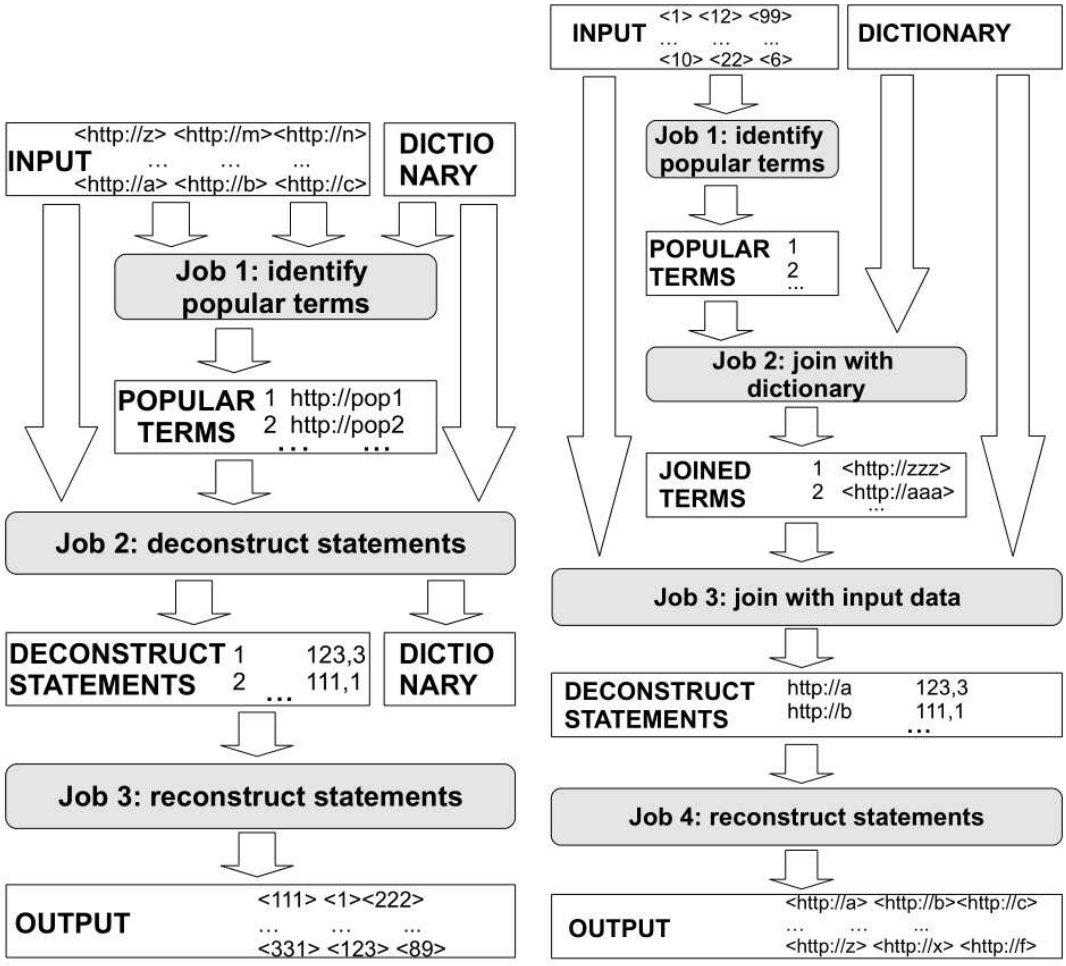
Thus, the entire process is comprised of three MapReduce jobs: The first job identifies and encodes the most popular terms using random sampling. The second job loads the popular terms on every node, which store them in a in-memory cache. Then, it reads and splits the triples into terms, and encodes those terms. The codification can be given by the cache in the case of popular terms or assigned by the reducer in other case. This job stores also the dictionary with the ID of each term. The third job reads the the output of the second job and substitutes the terms by their IDs on the triples. A diagram of the process is shown in Figure 3.3(a).

The decompression process is straightforward process formed by four jobs. The firs job, analogous to the compression process, identifies the popular terms. The second job joins those terms with the dictionary to decode the popular terms. The third job reads the compressed data, the dictionary, and the decoded terms from the previous job, and performs the decoding of the terms. The fourth and final job is also analogous to the encoding process and performs the reconstruction of the triples. The whole process is shown on Figure 3.3(b).

3.4 Discussion

MapReduce is designed to process data in distributed scenarios under the assumption of no inter-communication between `Map` and `Reduce` tasks during their execution. However, RDF data are interweaved because of its graph nature, and triple relationships are spatially arbitrary. For these reasons, multiple MapReduce jobs are usually necessary when dealing with semantic data. Moreover, a plain data storage and organization overloads the processing, and expensive costs must be paid whenever a new job starts. Thus, efficient SPARQL resolution on MapReduce-based solutions is mainly based on optimizing RDF data management and minimizing the number of MapReduce jobs required for query resolution.

We review the most relevant proposals on MapReduce-based solutions to deal with different tasks on semantic data throughout the chapter. For SPARQL query resolution, we establish a categorization in two different groups: (i) native solutions and (ii) hybrid solutions. Native solutions resolve SPARQL queries using MapReduce tasks exclusively, whereas hybrid solutions perform subgraph resolution in each node, and resort to MapReduce to join the results of each subgraph. In native solutions, the main contributions relate to reducing the number of jobs needed to perform joins, and to data organization. Data can be stored in HDFS, where data must be organized in files, or in another solution such as HBase, where triples can be indexed for faster access. In hybrid solutions, the main contributions are related to how data is partitioned in order to obtain optimal subgraphs. For RDFS and OWL inference, reducing the number of jobs is also the primary focus, addressed by grouping and ordering the application of entailment rules.



(a) Compression algorithm (b) Decompression algorithm

Figure 3.3: Compression and decompression algorithms [121]

Dealing with global information is a major achievement on RDF compression, while optimizing load balancing is an important factor in the majority of the solutions.

Although many of the prominent solutions cannot be directly compared, given their different configurations, a detailed analysis of their results draws significant conclusions: (i) MapReduce-based solutions scale almost linearly with respect to incremental data sizes, (ii) SPARQL querying solutions perform worse than classical mono-node solutions with simple queries or small datasets, but (iii) they outperform these solutions when the query complexity or the dataset size increases.

The state-of-the-art approaches also evidence that data must be preprocessed (i) to obtain easily readable notation, (ii) to enable partial reads to be done, and (iii) to reduce storage requirements. In addition, two of the reviewed papers also organize data in such a way that the process can capitalize on data locality and perform joins on `Map` tasks [106, 48]. It highly reduces data shuffling and improves performance. Although this preprocessing step could be computationally expensive, it is a once-only task which improves performance dramatically. In this scenario, binary RDF serialization formats such as RDF/HDT [36] could enhance the overall space/time tradeoffs. Note that these approaches can manage RDF in compressed space, enabling TP resolution at high levels of the memory hierarchy.

Apparently, the more complex the solutions, the better the performance results, but this comes at an important cost. On the one hand, they incur serious storage overheads because of data redundancy: NoSQL solutions can require up to 6 times the space of native solutions, whereas hybrid solutions report up to 4.5 times just for 2-hop partitions. On the other hand, the simpler native solutions are easier to implement in vanilla MapReduce clusters, which make deployment in shared infrastructures or in third party services (such as AWS Elastic MapReduce[3]) an almost straightforward operation. As complexity grows, solutions are harder to implement.

While these works showcase relevant contributions for dealing with RDF scalability issues using MapReduce, the absence of communication between tasks continues to present an important challenge when joins are involved. This can be seen as a general MapReduce issue that motivates different researches. Some proposals add an additional phase to the MapReduce cycle. For instance, Map-Reduce-Merge [127] adds an additional function at the end of the MapReduce cycle in order to support relational algebra primitives without sacrificing its existing generality and simplicity. In turn, Map-Join-Reduce [64] introduces a *filtering-join-aggregation* programming model which is an extension of the MapReduce programming model. Tuple MapReduce [37], though, takes a different approach and proposes a theoretical model that extends MapReduce to improve parallel data processing tasks using compound-records, optional in-reduce ordering, or intersource datatype joins. In addition, there are specific proposals for providing support to iterative programs like Twister [31] or HaLoop [16]. This aims to improve data locality for those tasks accessing to the same data (even in different jobs), while providing some kind of caching of invariant data. Thus, it is expected that all these general-purpose proposals will feedback specific applications, and semantic web applications on MapReduce will be benefited with advances from these lines of research.

[3] `http://aws.amazon.com/elasticmapreduce`

Chapter 4

HDT-MR

This chapter describes HDT-MR, a MapReduce-based scalable solution to serialize RDF in plain text into HDT. As seen in section 2.2.4, current implementations of HDT libraries face scalability issues when serializing huge datasets. This may hinder the adoption and evolution of HDT technologies to deal with the growing data of the semantic web. HDT-MR aims to make possible serialization of large datasets in a scalable way using the MapReduce framework.

4.1 System Design

This section describes the high-level HDT-MR system design. Figure 4.1 illustrates the HDT-MR workflow, consisting in two stages: *(1) Dictionary Encoding* (top) and *(2) Triples Encoding* (bottom), described in the following subsections. The whole process assumes the original RDF dataset is encoded in N-Triples format (one statement per line).

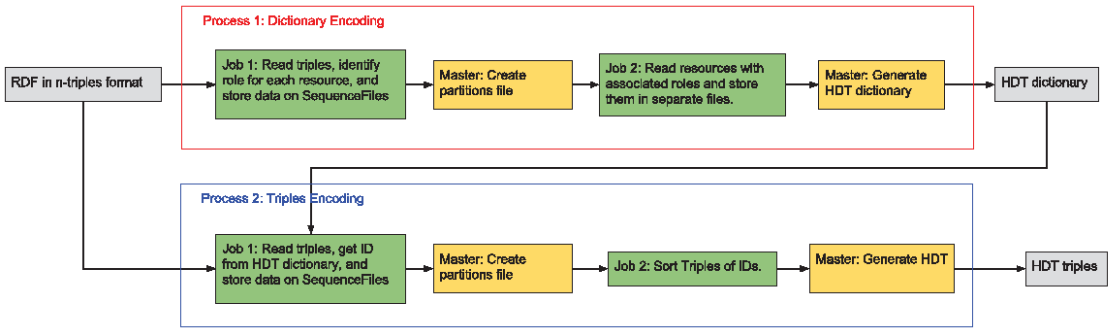


Figure 4.1: HDT-MR workflow.

4.1.1 Process 1: Dictionary Encoding

This first process builds the HDT *Dictionary* from the original N-Triples dataset. It can be seen as a three-task process of (i) identifying the role of each term in the dataset, (ii) obtaining the aforementioned sections (**SO**, **S**, **O**, and **P**) in lexicographic order, and (iii) effectively encoding the *Dictionary* component.

We design HDT-MR to perform these three tasks as two distributed MapReduce jobs and a subsequent local process (performed by the *master* node), as shown in Figure 4.1. The first job performs the role identification, while the second is needed to perform a global sort. Finally,

the *master* effectively encodes the *Dictionary* component. All these sub-processes are further described below.

4.1.1.1 Job 1.1: Roles Detection.

This job parses the input N-Triples file to detect all roles played by RDF terms in the dataset. First, mappers perform a triple-by-triple parsing and output (key,value) pairs of the form `(RDF term, role)`, in which role is S (subject), P (predicate) or O (object), according to the term position in the triple. It is illustrated in Figure 4.2, with two processing nodes performing on the RDF used in Figure 2.8. For instance, `(ex:P1,`S`)`, `(ex:worksFor,`P`)`, and `(ex:D1,`O`)` are the pairs obtained for the triple `(ex:P1, ex:worksFor, ex:D1)`.

These pairs are partitioned and sorted among the reducers, which group the different roles played by a term. Note that RDF terms including roles S and O, result in pairs `(RDF term,` SO`)`. Thus, this job outputs a number of lexicographically ordered lists `(RDF term, roles)`; there will be as many lists as reducers on the cluster. Algorithm 1 shows the pseudo-code of these jobs.

Finally, it is important to mention that a *combiner* function is used at the output of each `Map`. This function is executed on each node node before the `Map` transmits its output to the reducers. In our case, if a mapper emits more than one pair `(RDF term, role)` for a term, all those pairs are grouped into a single one comprising a list of all roles. It allows the bandwidth usage to be decreased by grouping pairs with the same key before transferring them to the reducer.

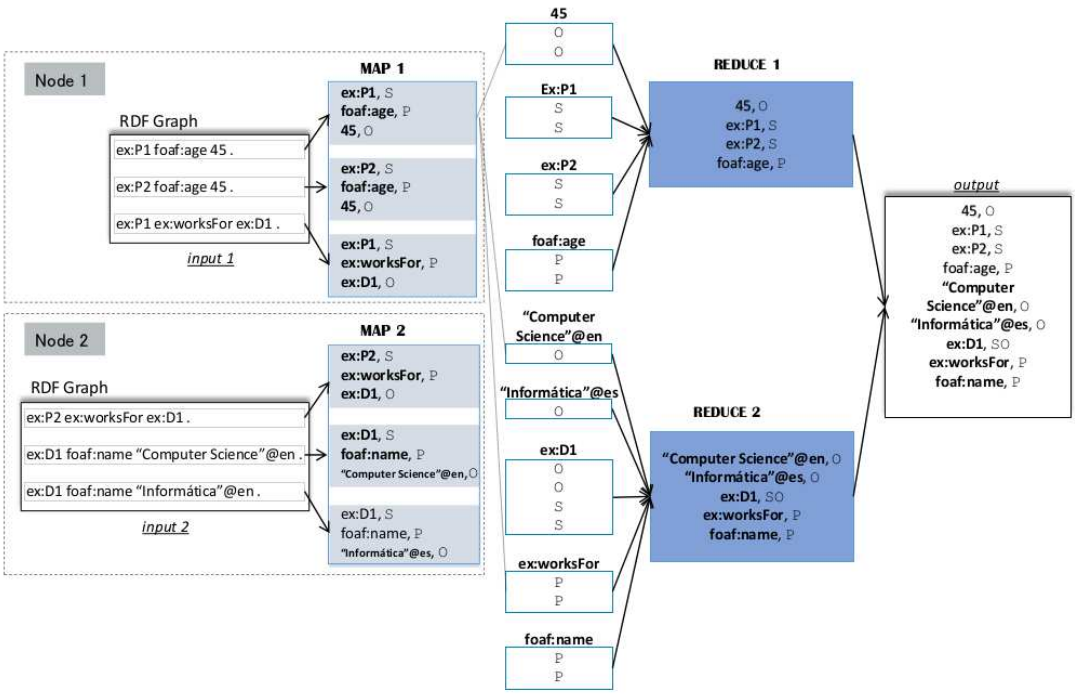


Figure 4.2: Example of Dictionary Encoding: roles detection (Job 1.1).

4.1.1.2 Job 1.2: RDF Terms Sectioning.

The previous job outputs several lists of pairs `(RDF term, roles)`, one per `Reduce` of previous phase, each of them sorted lexicographically. However, the construction of each HDT *Dictionary* section requires a unique sorted list. Note that a simple concatenation of the output lists would not

Algorithm 1 Dictionary Encoding: roles detection (Job 1.1)

```
function MAP(key,value)                          ▷ key: line number (discarded)        ▷ value: triple
    emit(value.subject,"S")
    emit(value.predicate,"P")
    emit(value.object,"O")
end function
function COMBINE/REDUCE(key,values)              ▷ key: RDF term                       ▷ value: roles (S, P, and/or O)
    for role in values do
        if role contains "S" then isSubject ← true
        else if role contains "P" then isPredicate ← true
        else if role contains "O" then isObject ← true
        end if
    end for
    roles ← ""
    if isSubject then append(roles,"S")
    else if isPredicate then append(roles,"P")
    else if isObject then append(roles,"O")
    end if
    emit(key,roles)
end function
```

fulfill this requirement, because the resulting list would not maintain a global order. The reason behind this behavior is that, although the input of each reducer is sorted before processing, the particular input transmitted to each reducer is autonomously decided by the framework in the process called *partitioning*. By default, Hadoop *hashes* the key and assigns it to a given reducer, promoting to obtain partitions of similar sizes. Thus, this distribution does not respect a global order of the input. While this behavior may be changed to assign the reducers a globally sorted input, this is not straightforward.

A naïve approach would be to use a single reducer, but this would result extremely inefficient: the whole data had to be processed by a single machine, losing most of the benefits of distributed computing that MapReduce provides. Another approach is to manually create partition groups. For instance, we could send terms beginning with the letters from *a* to *c* to the first reducer, terms beginning with the letters from *d* to *f* to the second reducer, and so on. However, partitions must be chosen with care, or they could be the root of performance issues: if partitions are of very different size, the job time will be dominated by the slowest reducer (that is, the reducer that receives the largest input). This fact is specially significant for RDF processing because of its skewed features.

HDT-MR relies on the simple but efficient solution of sampling input data to obtain partitions of similar size. To do so, we make use of the *TotalOrderPartitioner* of Hadoop. It is important to note that this partitioning cannot be performed while processing a job, but needs to be completed prior of a job execution. Note also that the input domain of the reducers needs to be different from the input domain of the job to identify and group the RDF terms (that is, the job receives triples, while the reducers receive individual terms and roles).

All these reasons conforms the main motivation to include this second MapReduce job to globally sort the output of the first job. This job takes as input the lists of `(RDF term, roles)` obtained in the precedent job, and uses role values to sort each term in its corresponding list. In this case, identity mappers deliver directly their input (with no processing) to the reducers, which send RDF terms to different outputs depending on their role. Figure 4.3 illustrates this job. As only the term is needed, a pair `(RDF term, null)` is emitted for each RDF term (*nulls* are omitted on the outputs). We obtain as many role-based lists as reducers in the cluster, but these are finally concatenated to obtain four sorted files, one per *Dictionary* section.The pseudo-code for this job is described in Algorithm 2.

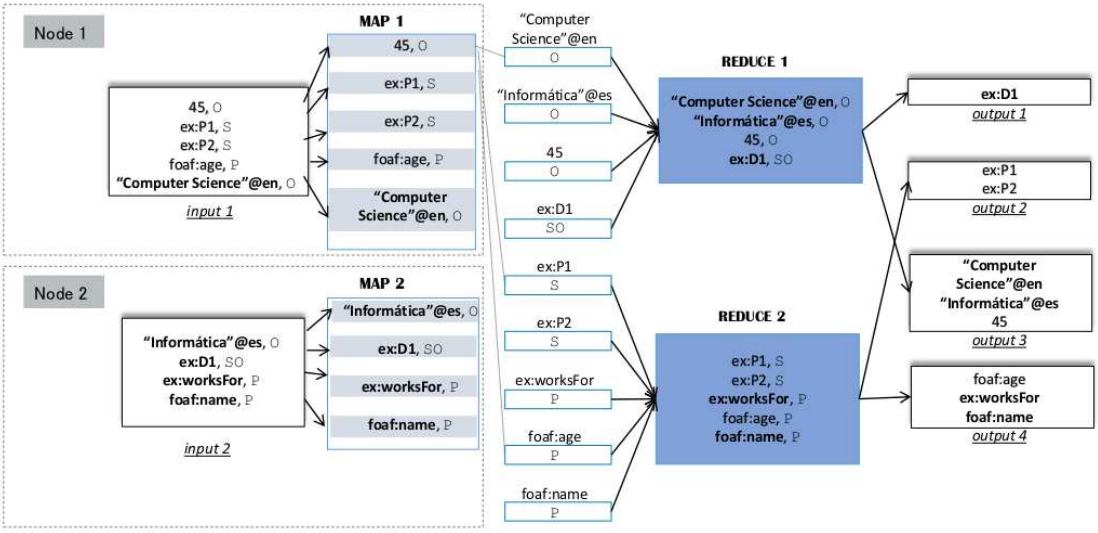


Figure 4.3: Example of Dictionary Encoding: RDF terms sectioning (Job 1.2).

Algorithm 2 Dictionary Encoding: RDF terms sectioning (Job 1.2)

```
function REDUCE(key,value)          ▷ key: RDF term          ▷ value: roles (S, P, and/or O)
    for resource in values do
        if resource contains "S" then isSubject ← true
        else if resource contains "P" then isPredicate ← true
        else if resource contains "O" then isObject ← true
        end if
    end for
    output ← ""
    if isSubject & isObject then emit_to_SO(key, null)
    else if isSubject then emit_to_S(key, null)
    else if isPredicate then emit_to_P(key, null)
    else if isObject then emit_to_O(key, null)
    end if
end function
```

4.1.1.3 Local sub-process 1.3: HDT Dictionary Encoding

This final stage performs locally in the *master* node, encoding dictionaries for the four sections obtained from the MapReduce jobs. It means that each section is read line-per-line, and each term is differentially encoded to obtain a Front-Coding dictionary [85], providing term-ID mappings. It is a simple process with no scalability issues.

4.1.2 Process 2: Triples Encoding

This second process parses the original N-Triples dataset to obtain, in this case, the HDT *Triples* component. The main tasks for such *Triples* encoding are (i) replacing RDF terms by their ID in the *Dictionary*, and (ii) getting the ID-triples encoding sorted by subject, predicate and object IDs. As in the previous process, HDT-MR accomplishes these tasks by two MapReduce jobs and a final local process (see the global overview in Figure 4.1), further described below.

4.1.2.1 Job 2.1: ID-triples serialization

This first job replaces each term by its ID. To do so, HDT-MR first transmits and loads the –already compressed and functional– *Dictionary* (encoded in the previous stage) in all nodes of the

Algorithm 3 Triples Encoding: ID-triples serialization (Job 2.1)

function MAP(key,value) ▷ key: line number (discarded) ▷ value: triple
$emit(\{dictionary.id(value.subject, dictionary.id(value.predicate), dictionary.id(value.object)\}, null)$
end function

cluster. Then, mappers parse N-Triples and replace each term by its ID in the *Dictionary*. Identity reducers simply sort incoming data and output a list of pairs `(ID-triple, null)`. We can see this process in action in Figure 4.4, where the terms of each triple are replaced by the IDs given in the previous example (note that *nulls* are omitted on the outputs). The output of this job is a set of lexicographically ordered lists of ID-Triples; there will be as many lists as reducers on the cluster. The pseudo-code of this job is illustrated in Algorithm 3 .

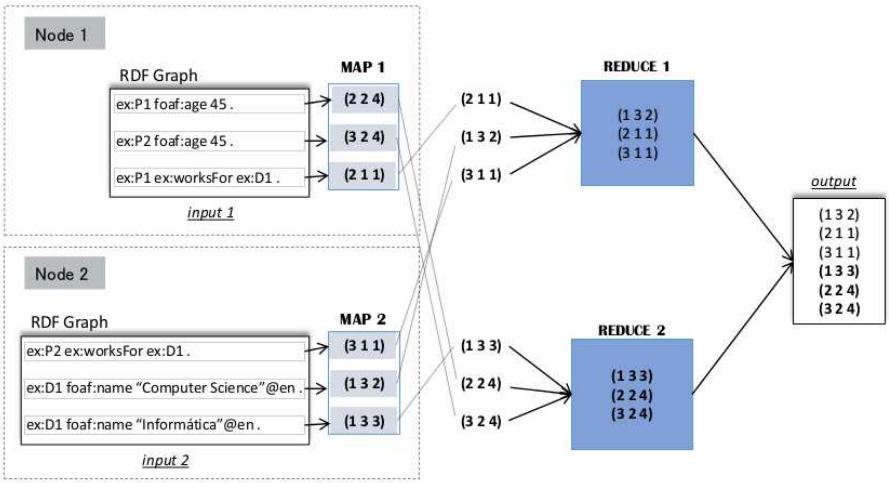


Figure 4.4: Example of Triples Encoding: ID-triples Serialization (Job 2.1).

4.1.2.2 Job 2.2: ID-triples Sorting

Similarly to the first process, Triples Encoding requires of a second job to sort the outputs. Based on the same premises, HDT-MR makes use of Hadoop *TotalOrderPartitioner* to sample the output data from the first job, creating partitions of a similar size as input for the second job. Then, this job reads the ID-triples representation generated and sorts it by subject, predicate and object ID. This is a very simple job that uses identity mappers and reducers. As in the previous job, ID-triples are contained in the key and the value is set to *null*. In fact, all the logic is performed by the framework in the partitioning phase between `Map` and `Reduce`, generating similar size partitions of globally sorted data. Figure 4.5 continues with the running example and shows the actions performed by this job after receiving the output of the previous job (note again that *nulls* are omitted on the outputs).

4.1.2.3 Local sub-process 2.3: HDT Triples Encoding

This final stage encodes the ID-triples list (generated by the previous job) as HDT *BitmapTriples* [36]. It is performed locally in the *master* node as in the original HDT construction. That is, it sequentially reads the sorted ID-triples to build the sequences `Sp` and `So`, and the aligned bitsequences `Bp` and `Bo`, with no scalability issues.

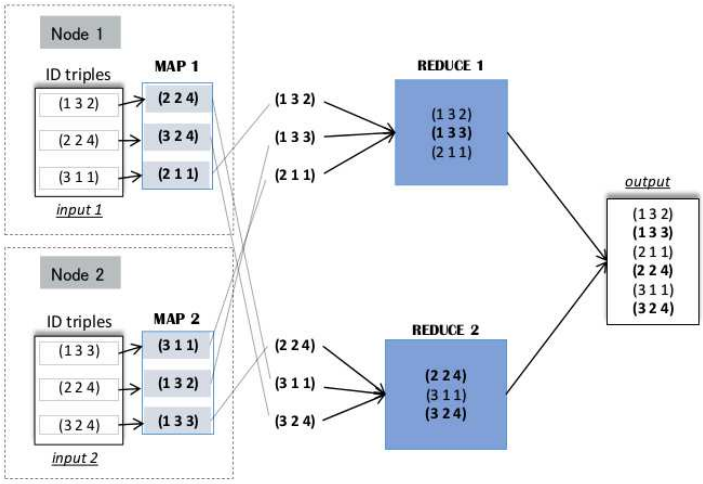


Figure 4.5: Example of Triples Encoding: ID-triples Sorting (Job 2.2)

4.2 Implementation and configuration details

We have developed a proof-of-concept `HDT-MR` prototype (under the Hadoop framework: version 1.2.1) which uses the existing HDT-Java library[1] (RC-2). This section describes technical details of the implementation and configuration of both the prototype and the Hadoop cluster.

HDT-MR is deployed on a virtualized Hadoop cluster consisting on a potent *master* and 10 *slave* nodes running on a more memory-limited configuration. This infrastructure tries to simulate a computational cluster in which further nodes may be plugged to process huge RDF datasets. See table 4.1

MACHINE	CONFIGURATION
Master	Intel Xeon X5675 @ 3.07 GHz (4 cores), 48GB RAM. Ubuntu 12.04.2
Slaves	Intel Xeon X5675 @ 3.07 GHz (4 cores), 8GB RAM. Debian 7.7

Table 4.1: Cluster configuration.

The underlying physical system is comprised by 10 computation nodes divided into two Intel Modular Server chassis. Storage is conducted on a pool of 12 disks on a RAID 10 plus two replacement disks, connected through Serial Attached SCSI. It is important to note that the virtualization scheme for the cluster has not been stable over time, so development and evaluation had to accommodate.

It is worth noting that `HDT-MR` uses `lzo` to compress the datasets before storing them in HDFS. This format allows for compressed data to be split among the tasks, and provides storage and reading speed improvements [88]. In order to be splitted the compressed files need to be indexed after storing the data in HDFS. This is performed using the *Hadoop-LZO* library[2] version `0.4.17`.

HDT-MR operation is managed by the `HDTBuilderDriver` class, which runs the desired jobs or local processes in order. Note that every job and local process reads its input from, and writes its output to, HDFS. This makes possible to launch a sub-set of HDT-MR operations, store intermediate outputs, and continue the process in the future. HDT-MR parameters are read from a configuration file, but can be overridden by command line options, using the

[1] `http://code.google.com/p/hdt-java/`
[2] `https://github.com/twitter/hadoop-lzo`

`HDTBuilderConfiguration` class with the help of *JCommander* library[3] version `1.30`. A complete description of HDT-MR parameters can be seen on appendix A, and the current configuration files (used on the evaluation tests) are available on appendix B.

4.2.1 Job 1.1: Roles Detection

This job is charged with the function of parsing the triples and identify the roles associated with each term. It makes use of the `DictionarySamplerMapper` class for the Mapper and `DictionarySamplerReducer` class for both the Combiner and the Reducer. This job writes to HDFS a series of compressed `SequenceFiles` with terms and their associated roles. A `SequenceFile` is a Hadooop file format consisting of binary key-value pairs designed for MapReduce I/O operations. [124]. A simplified class diagram for this job can be seen in Figure 4.6

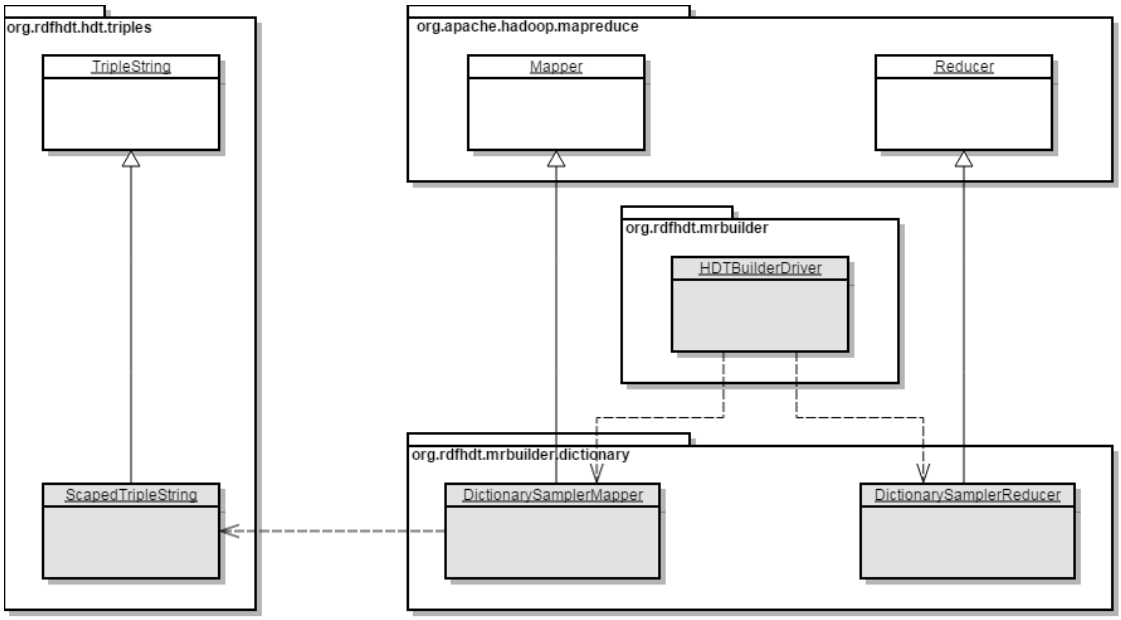


Figure 4.6: Class Diagram: Job 1.1: Roles Detection

4.2.2 Job 1.2: RDF Terms Sectioning

This job divides the terms into the future sections of the HDT Dictionary and counts the number of terms in each section. It uses identity Mappers (a default implemantation of a Mapper in Hadoop, which outputs its input without any processing) and the `DictionaryReducer` class for the Reducers. This class simply reads terms and a list of roles and reads the list sequentially. To assign each term to its section, it maintains a list of flags (one for each different section) . When each role is read, the corresponding flag is set to *true*. Finally, after traversing the list or roles, the term is outputted to the corresponding path for each flag set to *true*. The class makes use of a `MultipleOutputs` object to write to different HDFS paths. In addition, this job maintains a global *counter* for each section. Each time that a term is written to a section, its value is incremented by one. When the job finishes, the values of the counters are written to HDFS.

[3] `http://jcommander.org`

In order to perform the data sampling using the *TotalOrderPartitioner*, Hadoop provides three different sampling methods through the class *InputSampler*. All of them sample a `SequenceFile` stored in HDFS.

- `IntervalSampler`: Samples *s* splits at regular intervals. It accepts as parameters the frequency with which records will be sampled and the maximum number of splits to be sampled.
- `RandomSampler`: Samples from random points in the input. It has the following parameters: The probability with which a key will be chosen, the total number of samples to obtain from all selected splits, and the maximum number of splits to examine.
- `SplitSampler`: Samples the first *n* records from *s* splits. Its parameters are the following: The total number of samples to obtain, and the maximum number of splits to examine.

HDT-MR uses the `IntervalSampler`, which gave better results in initial tests. The frequency used is `0.000001`, while no maximum number of splits is set, meaning that all splits are sampled.

A simplified class diagram for this job can be seen in Figure 4.7

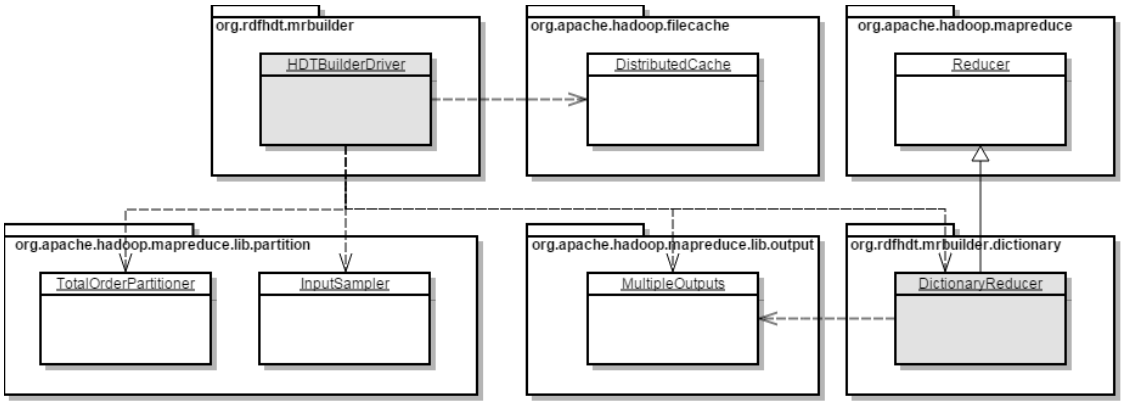


Figure 4.7: Class Diagram: Job 1.2: RDF Terms Sectioning

4.2.3 Local sub-process 1.3: HDT Dictionary Encoding

The final process reads the sorted files on each path sequentially and builds a Dictionary section for each one. Each section is encoded in *Plain Front-Coding* [85], which is based on the *Front-Coding* technique [125]. This technique achieves compression in lexicographically sorted dictionaries by differentially encoding a string (a term in our case) with respect the previous one. Each string is encoded using an integer indicating the number of characters that match the previous string plus the rest of the string. To avoid performance issues due to backtracking an excessive number of terms the dictionary is divided into blocks. Each block is encoding separately, explicitly storing the first string.

A `TransientDictionarySection` object is created for each section, with a default block size of 16 terms. It is worth noting that in java arrays are limited to 2^30 entries, due to using a signed integer as index. For this reason, a more complex structure is needed to store the IDs. We use a two-dimensional array, where the internal arrays store up to 200,000 blocks. In order to optimize compression, the number of terms of the section (stored in HDFS by the first job) is read

and used to determine the number ob bits needed to encode the term ID. When the four sections have been created they are written to HDFS.

4.2.4 Job 2.1: ID-triples serialization

The first job of the second process translates the triples of the dataset to ID-triples. This is performed using the `TriplesSPOMapper` class. This class loads the dictionary and parses the input triples one by one, replacing each term by its corresponding ID. The triples are stored in `TripleSPOWritable` outputted. This job also maintains a global counter for the triples. Each time that a triple is read by a Mapper, its value is incremented by one. When the job finishes, the value of the counter is written to HDFS. This job uses identity Reducers. That is, reducers that output the ID-triples in the same order they read them.

In order to group the triples and send them to the Reducers, it has been necessary to implement the `TripleSPOComparator` class. This class is used by the Hadoop framework to compare ID-triples terms, in the order `S-P-O` to determine if they are equal, and sort order if not. This order is also used by the Reducers to read their input by order.

A simplified class diagram for this job can be seen in Figure 4.8

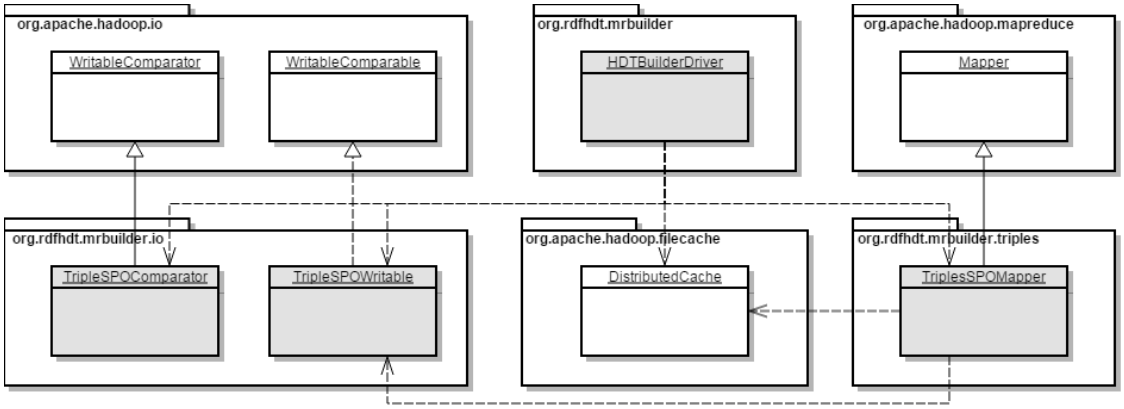


Figure 4.8: Class Diagram: Job 2.1: ID-triples serialization

4.2.5 Job 2.2: ID-triples Sorting

This job reads the ID-triples written by the previous job and sorts them globally using identity Mappers and Reducers. To sample the data this job also uses the `IntervalSampler` with a frequency of `0.000001`. The class `TripleSPOComparator` is used by the *TotalOrderPartitioner* to group and sort the ID-triples.

4.2.6 Local sub-process 2.3: HDT Triples Encoding

The final local sub-process compresses the triples using BitmapTriples encoding [36]. Remember that this encoding is comprised by two sequences of predicate and object IDs: `Sp` and `So`, and two aligned bitsequences, `Bp` and `Bo`, using a 1-bit tomark the end of each list.

The class `TransientBitMapTriples` creates two `TransientSequenceLog64` objects to store the sequences, and two `TransientBitmap375` objects to store the bitsequences. Each time it reads an ID-triple it performs the following steps:

1. The subject ID is compared with the subject ID of the previous triple. In case the new ID increases, both the predicate and object ID are stored in their respective sequences, and `true` is appended to each bitsequence, meaning that the subject has changed for the current triple.

2. In the case of the subject ID not increasing, the predicates ID are compared. If the new ID increases, both the predicate and object ID are stored in their respective sequences. The predicate bitsequence is appended a `false`, meaning that the triple is related to the same subject as the previous one, and `true` is appended to the objects bitsequence, representing the change of predicate.

3. If predicate IDs are equal only the object ID is stored in its sequence, and a `false` appended to the objects bitsequence, meaning that it relates to the same predicate. Note that no comparison is needed; if the ID-triples are lexicographically sorted, an ID-triple which does not fulfill the previous conditions will necessarily increase only the object ID.

Initial experiments showed that storing the data in memory for this process was the main bottleneck for HDT-MR, causing the process to fail. To avoid memory limitations, the `TransientSequenceLog64` and `TransientBitmap375` write their content to HDFS each time a threshold is reached (currently 2^20 items). When all ID-triples are processed, the sequences and bitsequences are appended in the appropriate order `Bp-Bo-Sp-So`.

A simplified class diagram for this job can be seen in Figure 4.8

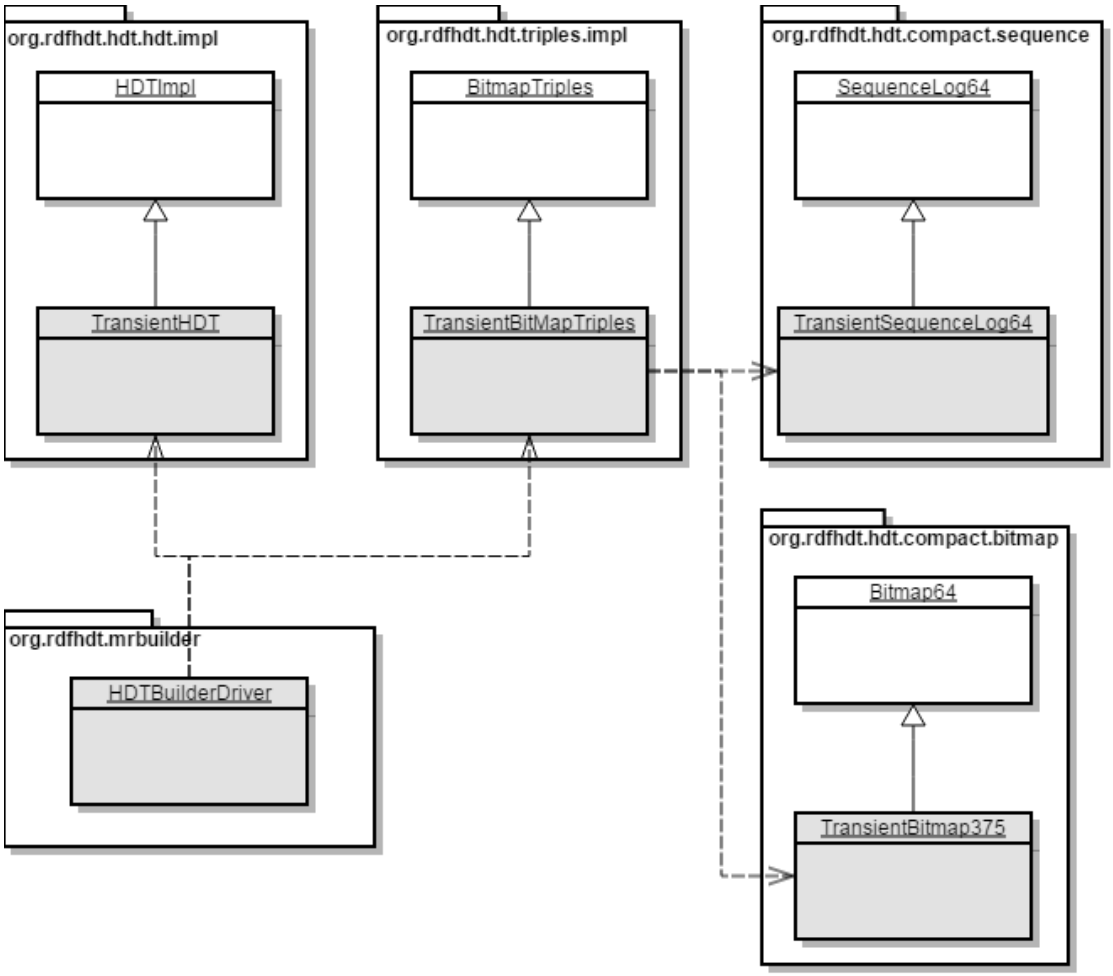


Figure 4.9: Class Diagram: Local sub-process 2.3: HDT Triples Encoding

Chapter 5

Experiments and Results

This chapter evaluates the performance of HDT-MR, the proposed MapReduce-based HDT construction. We compare it to the traditional single-node approach to evaluate scalability. It is worth noting that HDT-MR serialization times provided in this chapter are higher than number reported in [47]. The reason is that the network and disk bandwidths have been limited to improve stability issues on the underlying physical system.

The **experimental setup** is designed as follows. We use the HDT-MR configuration stated on section 4.2 using a potent *master* and 10 *slave* nodes running on a more memory-limited configuration. For the single-node tests we use a powerful computational configuration. For a fair comparison, the amount of main memory in the single node is the same as the total memory available for the full cluster of Hadoop. Table 5.1 summarizes configurations information.

MACHINE	CONFIGURATION
Single Node	Intel Xeon E5-2650v2 @ 2.60GHz (32 cores), 128GB RAM. Debian 7.8
Master	Intel Xeon X5675 @ 3.07 GHz (4 cores), 48GB RAM. Ubuntu 12.04.2
Slaves	Intel Xeon X5675 @ 3.07 GHz (4 cores), 8GB RAM. Debian 7.7

Table 5.1: Experimental setup configuration.

Regarding **datasets**, we consider a varied configuration of **datasets** comprising real-world and synthetic ones. All of them are statistically described in Table 5.2. Among the real-world ones, we choose them based on their volume and variety, but also attending to their previous uses for benchmarking. *Ike*[1] comprises weather measurements from the Ike hurricane; *LinkedGeoData*[2] is a large geo-spatial dataset derived from *Open Street Map*; and DBPedia 3.8[3] is the well-known knowledge base extracted from Wikipedia. We also consider the combination of these real-world datasets in different *mashups* comprising data from the three data sources. On the other hand, we use the LUBM [51] data generator to obtain synthetic datasets. We build "small datasets" from 1,000 (0.13 billion triples) to 8,000 universities (1.07 billion triples). From the latter, we build datasets of incremental size (4,000 universities: 0.55 billion triples) up to 72,000 universities (9.59 billion triples). Data have been preprocessed in order to be stored in N-Triples notation and delete duplicates. This preprocessing also sorts the triples lexicographically.

First, we evaluate the performance for real-world datasets. Figure 5.1 compares serialization times for the datasets and the different mashups. As can be seen, HDT-Java reports an excellent performance on real-world datasets. This is an expected result because HDT-Java runs the whole process in main-memory while HDT-MR relies on I/O operations. However, HDT-Java crashes

[1] `http://wiki.knoesis.org/index.php/LinkedSensorData`
[2] `http://linkedgeodata.org/Datasets`, as for 2013-07-01
[3] `http://wiki.dbpedia.org/Downloads38`

						Size (GB)				
DATASET	TRIPLES	|SO|	|S|	|O|	|P|	NT	NT+lzo	HDT	Dict.	HDT+gz
LinkedGeoData	0.27BN	41.5M	10.4M	80.3M	18.3K	38.5	4.4	6.4	5.1	1.9
DBPedia	0.43BN	22.0M	2.8M	86.9M	58.3K	61.6	8.6	6.4	4.8	2.7
Ike	0.51BN	114.5M	0	145.1K	10	100.3	4.9	4.8	1.3	0.6
LGD+DBP	0.70BN	63.5M	13.2M	167.0M	76.6K	100.1	13.0	12.6	9.8	3.7
LGD+Ike	0.79BN	156.0M	10.4M	80.4M	18.3K	138.8	9.3	10.37	6.4	1.7
DBP+Ike	0.95BN	136.5M	27.8M	87.0M	58.3K	161.8	13.5	10.45	6.1	3.0
LGD+DbP+Ike	1.22BN	178.0M	13.2M	167.2M	76.6K	200.3	18.0	17.1	11.2	4.6
LUBM-1000	0.13BN	5.0M	16.7M	11.2M	18	18.0	1.3	0.7	0.3	0.2
LUBM-2000	0.27BN	10.0M	33.5M	22.3M	18	36.2	2.7	1.5	0.6	0.5
LUBM-3000	0.40BN	14.9M	50.2M	33.5M	18	54.4	4.0	2.3	0.8	0.8
LUBM-4000	0.53BN	19.9M	67.0M	44.7M	18	72.7	5.3	3.1	1.1	1.0
LUBM-5000	0.67BN	24.9M	83.7M	55.8M	18	90.9	6.6	3.9	1.4	1.3
LUBM-6000	0.80BN	29.9M	100.5M	67.0M	18	109.1	8.0	4.7	1.6	1.6
LUBM-7000	0.93BN	34.9M	117.2M	78.2M	18	127.3	9.3	5.5	1.9	1.9
LUBM-8000	1.07BN	39.8M	134.0M	89.3M	18	145.5	10.6	6.3	2.2	2.2
LUBM-12000	1.60BN	59.8M	200.9M	133.9M	18	218.8	15.9	9.6	3.3	2.9
LUBM-16000	2.14BN	79.7M	267.8M	178.6M	18	292.4	21.2	12.8	4.4	3.8
LUBM-20000	2.67BN	99.6M	334.8M	223.2M	18	366.0	26.6	16.3	5.5	5.5
LUBM-24000	3.20BN	119.5M	401.7M	267.8M	18	439.6	31.9	19.6	6.6	6.6
LUBM-28000	3.74BN	139.5M	468.7M	312.4M	18	513.2	37.2	22.9	7.7	7.7
LUBM-32000	4.27BN	159.4M	535.7M	357.1M	18	586.8	42.5	26.1	8.8	8.8
LUBM-36000	4.81BN	179.3M	602.7M	401.8M	18	660.5	47.8	30.0	10.0	9.4
LUBM-40000	5.32BN	198.4M	666.7M	444.5M	18	730.9	52.9	33.2	11.1	10.4
LUBM-44000	5.85BN	218.3M	733.7M	489.2M	18	804.6	58.2	36.7	12.2	12.2
LUBM-48000	6.38BN	238.3M	800.7M	533.8M	18	877.8	63.6	40.3	13.3	13.3
LUBM-52000	6.92BN	258.2M	934.6M	578.4M	18	951.5	68.9	43.6	14.4	14.4
LUBM-56000	7.45BN	278.1M	1,001.6M	623.1M	18	1,024.5	74.2	47.3	15.5	15.5
LUBM-60000	7.99BN	298.0M	1,068.5M	667.8M	18	1,097.6	79.5	50.7	16.7	16.1
LUBM-64000	8.52BN	318.0M	1,135.5M	712.4M	18	1,170.8	84.8	53.9	17.8	17.1
LUBM-68000	9.05BN	337.9M	1,202.4M	757.0M	18	1,244.4	90.2	57.6	18.9	18.9
LUBM-72000	9.59BN	357.8M	1269.4M	801.7M	18	1,318.0	95.5	61.3	20.0	20.0

Table 5.2: Statistical dataset description

for the *mashups* because the 128 GB of available RAM are insufficient to process such scale in the single node.

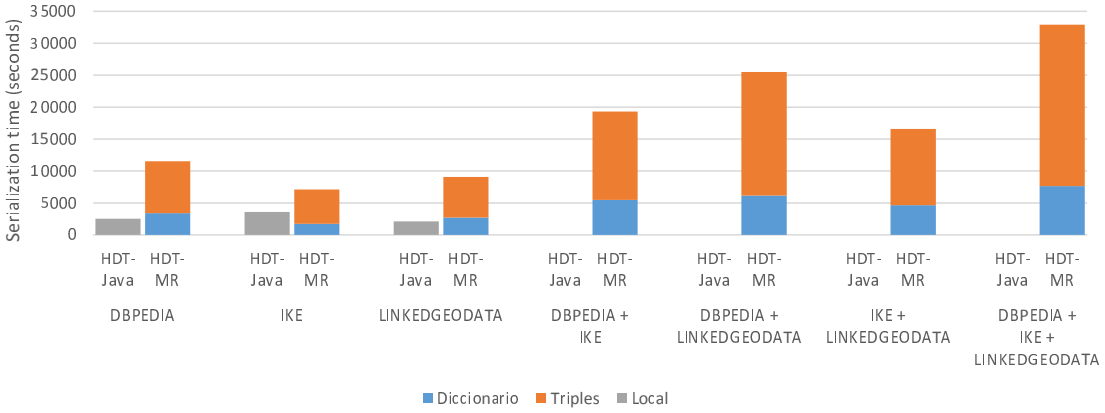


Figure 5.1: Serialization times: Real-World Datasets

Second, we perform the evaluation for the LUBM datasets: HDT-Java is again the best choice for the smallest datasets, but the difference decreases with the dataset size. Figure 5.2 compares serialization times for HDT-Java and HDT-MR. HDT-Java fails to process datasets from *LUBM-8000* (1.07 billion triples) because of memory requirements. This is the target scenario for HDT-MR, which scales to the *LUBM-72000* without issues. Serialization times for HDT-MR can be seen in Figure 5.3. As can be seen in both figures, serialization times increase linearly with the dataset size, and triples encoding remains the most expensive stage.

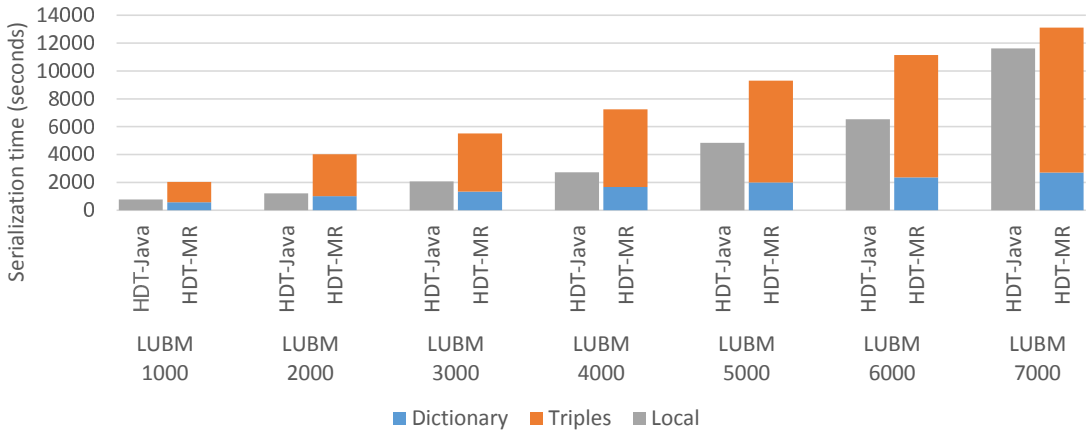


Figure 5.2: Serialization times: LUBM (1)

In addition, we have also performed tests against the synthetic datasets obtained using SP2B [107]. Those datasets are generated directly in N-Triples, so no preprocessing is necessary. This evaluation has shown a fundamental behavior of HDT-MR regarding data order. `Map` tasks need to cache different parts of the dictionary too frequently, with a high impact on performance. To show behavior we compare SP2B against LUBM. We use LUMB datasets from 1,000 to 5,000 universities, and we generate SP2B datasets with a corresponding number of triples. The statistical description of the datasets are shown in Table 5.3. Figure 5.4 shows serialization times for those datasets.

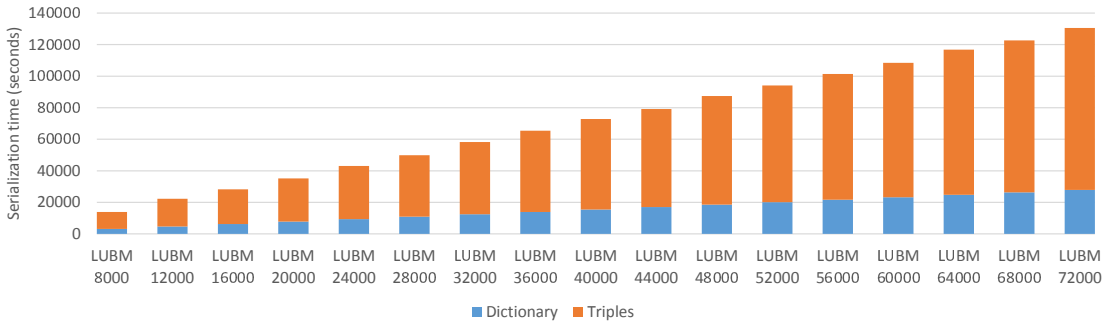


Figure 5.3: Serialization times: LUBM (2)

						Size (GB)				
DATASET	TRIPLES	|SO|	|S|	|O|	|P|	NT	NT+lzo	HDT	Dict.	HDT+gz
LUBM-1000	133.6M	5.0M	16.7M	11.2M	18	18.0	1.3	0.7	0.3	0.2
SP2B-134	133.6M	13.9M	10.1M	5.0M	87	13.9	2.6	2.3	1.8	0.8
LUBM-2000	267.0M	10.0M	33.5M	22.3M	18	36.2	2.7	1.5	0.6	0.5
SP2B-267	267.0M	29.6M	19.6M	10.0M	87	27.6	5.2	4.5	3.4	1.6
LUBM-3000	400.5M	14.9M	50.2M	33.5M	18	54.4	4.0	2.3	0.8	0.8
SP2B-401	400.5M	46.0M	29.0M	15.0	87	41.2	7.7	6.7	5.0	2.4
LUBM-4000	534.2M	19.9M	67.0M	44.7M	18	72.7	5.3	3.1	1.1	1.0
SP2B-534	534.2M	62.3M	38.4M	20.0	87	54.8	10.3	8.9	6.6	3.1
LUBM-5000	667.6M	24.9M	83.7M	55.8M	18	90.9	6.6	3.9	1.4	1.3
SP2B-668	667.6M	78.5M	47.8M	25.0M	87	68.5	12.8	11.1	8.2	4.0

Table 5.3: Statistical dataset description of SP2B and LUBM

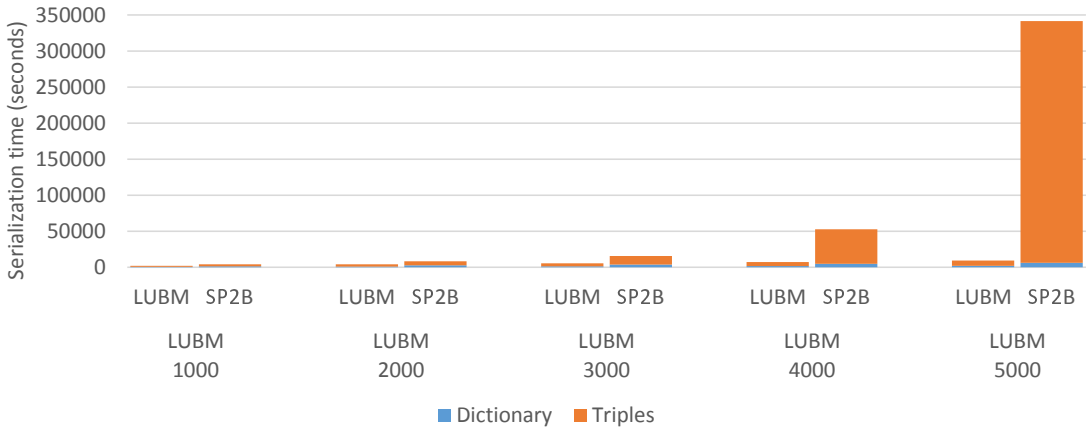


Figure 5.4: Serialization times: LUBM vs SP2B

While RDF compression numbers are not the main purpose of this work, they are worth to mention. On the one hand, previous literature does not report HDT serialization results for such large datasets. HDT always reports smaller sizes than the original datasets compressed with `lzo`, with the exception of the mashup of *LinkedGeoData* and *DBPedia*. For instance, HDT serializes *LUBM-40000* using 19.7 GB less than `NT+lzo`. The difference increases when compressed with `gzip`. For *LUBM-40000*, `HDT+gz` uses 42.5 GB less than `NT+lzo`. In practice, it means that `HDT+gz` uses 5 times less space than `NT+lzo`. These numbers are summarized in tables 5.2 and 5.3. Finally, it is worth remembering that HDT-MR obtains the same HDT serialization than a mono-node solution, hence achieving the same compression ratio and enabling the same query functionality.

Chapter 6

Conclusions and Future Work

6.1 Conclussions

HDT is currently gaining traction, positioning itself as a baseline for RDF compression. Latest practical applications exploit the HDT built-in indexes for RDF retrieval with no prior decompression, making HDT evolve to a self-contained RDF store. However, HDT achievements are at the price of moving scalability issues from consumer to publishers. Serializing RDF into HDT is not a simple task, given that the whole dataset must be exhaustively processed in memory to obtain the Dictionary and Triples components. Current HDT implementations demand not negligible amounts of memory, so the HDT serialization lacks of scalability for huge datasets (i.e. those having hundreds of millions or billions of triples). Although these datasets are currently uncommon, semantic publication efforts on emerging data-intensive areas (such as biology or astronomy) or integrating several sources into heterogeneous mashups (as RDF excels at linking data from diverse datasets) are starting to face this challenge.

This work improves the HDT workflow by introducing MapReduce as the computation model for large HDT serialization. MapReduce is a framework for the distributed processing of large amounts of data, and it can be considered as *de facto* standard for Big Data processing. Our MapReduce-based approach, HDT-MR, reduces scalability issues arising to HDT generation, enabling larger datasets to be serialized for end-user consumption.

We have performed evaluations against the previous mono-node solution, scaling up to more than 1 TB of data (20 times larger than the largest dataset serialized by the original HDT). Results show that HDT-MR is able to scale up to more than 9 billion triples, while the mono-node solution fails to process datasets larger than 1 billion triples. Thus, HDT-MR greatly reduces hardware requirements for processing Big Semantic Data.

Nonetheless, evaluations have also identified a dependence of data sorting. Triples Encoding process of HDT-MR does not scale linearly when processing unsorted data data. This issue needs to be addressed in future work.

As part of the work, we also review MapReduce and its applications to Semantic Web scalability issues, drawing conclussions about its appropriateness, current challenges, and different approaches to deal with them. A comprehensive discussion and conclusions about the state of the art can be found in section 3.4.

6.2 Future Work

While HDT-MR is a contribution to improve serialization of large datasets, there is still more work to be done. Firstly, more experimentation is needed to perform a fine-grained assessment

of the possible bottlenecks and limitations. Secondly, some improvements of HDT-MR are worth exploring. The more immediate are the following:

- The dependence on data sorting needs to be addressed. This can be solved by moving the replacing of terms by ID from the `Map` phase to the `Reduce` phase. The reducers receive the data sorted by the framework, removing the necessity to cache different parts of the dictionary frequently.
- The current version of HDT libraries serialize the dictionary using *Plain Front-Coding*. While this compression technique achieves goods results in terms of triples retrieval, the compression rate can be improved. This is specially true when dealing with datasets with high proportion of literals. HDT++ [56] is a recent implementation of HDT that can be integrated into HDT-MR to improve its performance.
- HDT-MR distributes the complete dictionary among each node of the cluster. A smart distribution of data among nodes, instead of relying in vanilla Hadoop mechanisms would allow to partition the dictionary in smaller chunks, highly improving its performance.
- Each one of the two processes of HDT-MR include an step where data sampling is performed. We have chosen one of the Hadoop default sampling techniques, which gave as the better results in preliminary tests. However, more work be done on this topic, including more experimentation or the development of a custom sampling method.

Additional future work include exploring the recent evolution in MapReduce and related technologies and how to apply them to enhance HDT-MR, such as upgrading to Hadoop 2.0 or explore a migration to Apache Spark. Hadoop 2.0 is the most recent version of Hadoop and includes many performance improvements. Spark[1] is also a distributed computing framework, but instead of relying on disk-based operations it is based on in-memory data processing.

6.3 Contributions and Publications

The review of the on MapReduce-based solutions on scalability issues is a contribution of this work. Specifically, the state of the art of SPARQL query resolution using MapReduce on section 3.1 have been published on the *Open Journal of Semantic Web* on the following article:

> José M. Giménez-García, Javier D. Fernández, and Miguel A. Martínez-Prieto. Mapreduce-based solutions for scalable sparql querying. *Open Journal of Semantic Web (OJSW)*, 1(1):1–18, 2014. ISSN 2199-336X

HDT-MR and its evaluation, discussed in chapters 4 and 5 is the main contribution of this work. The content of those chapters has served as the basis for the following publication on the *Extended Semantic Web Conference*:

> José M. Giménez-García, Javier D. Fernández, and Miguel A. Martínez-Prieto. HDT-MR: A scalable solution for rdf compression with HDT and mapreduce. In Fabien Gandon, Marta Sabou, Harald Sack, Claudia d'Amato, Philippe Cudré-Mauroux, and Antoine Zimmermann, editors, *The Semantic Web. Latest Advances and New Domains*, volume 9088 of *Lecture Notes in Computer Science*, pages 253–268. Springer International Publishing, 2015. ISBN 978-3-319-18817-1. doi: 10.1007/978-3-319-18818-8_16

[1] `http://spark.apache.org`

Bibliography

[1] Azza Abouzeid, Kamil Bajda-Pawlikowski, Daniel J. Abadi, Alexander Rasin, and Avi Silberschatz. Hadoopdb: An architectural hybrid of mapreduce and DBMS technologies for analytical workloads. *PVLDB*, 2(1):922–933, 2009.

[2] Grigoris Antoniou and Frank Van Harmelen. *A Semantic Web Primer, Second Edition*. 2008. ISBN 0262012103.

[3] D Battré, S Ewen, F Hueske, O Kao, V Markl, and D Warneke. Nephele / PACTs : A Programming Model and Execution Framework for Web-Scale Analytical Processing Categories and Subject Descriptors. In *Proc. 1st ACM Symp. Cloud Comput.*, pages 119–130. ACM, 2010. ISBN 9781450300360.

[4] Dave Beckett, editor. *RDF/XML Syntax Specification (Revised)*. W3C Recommendation, 2004.

[5] David Beckett and Tim Berners-Lee. *Turtle-terse RDF triple language*. W3C Team Submission, 2008.

[6] Tim Berners-Lee. Design issues: Linked data, 2006.

[7] Tim Berners-Lee, James Hendler, Ora Lassila, and By Tim Berners-lee. The Semantic Web. *Sci. Am.*, 284(5):34–43, 2001. ISSN 00368733. doi: 10.1038/scientificamerican0501-34.

[8] Tim Berners-Lee, Roy Fielding, and Larry Masinter. Uniform resource identifier (uri): Generic syntax. Technical report, 2004.

[9] E Bertino, P Bernstein, D Agrawal, S Davidson, U Dayal, M Franklin, J Gehrke, L Haas, A Halevy, J Han, and Others. Challenges and Opportunities with Big Data. 2011.

[10] A Bialecki, M Cafarella, D Cutting, and O O'MALLEY. Hadoop: a framework for running applications on large clusters built of commodity hardware. *Wiki http//lucene.apache.org/hadoop*, 11, 2005.

[11] Paul Biron, Ashok Malhotra, World Wide Web Consortium, et al. Xml schema part 2: Datatypes. *World Wide Web Consortium Recommendation REC-xmlschema-2-20041028*, 2004.

[12] Christian Bizer. The emerging web of linked data. *IEEE Intelligent Systems*, 24(5):87–92, 2009. doi: 10.1109/MIS.2009.102.

[13] D Borthakur, Portability Across, and Heterogeneous Hardware. The Hadoop distributed file system: Architecture and design. *Hadoop Proj. Website*, 11:21, 2007.

[14] Tim Bray, Jean Paoli, C Michael Sperberg-McQueen, Eve Maler, and François Yergeau. Extensible markup language (xml). *World Wide Web Consortium Recommendation REC-xml-19980210. http://www. w3. org/TR/1998/REC-xml-19980210*, 16, 1998.

[15] Dan Brickley and Ramanathan V Guha. *RDF Schema 1.1*. W3C Recommendation, 2014.

[16] Yingyi Bu, Bill Howe, Magdalena Balazinska, and Michael D. Ernst. Haloop: Efficient iterative data processing on large clusters. *PVLDB*, 3(1):285–296, 2010.

[17] Michelle Butler, Richard Mount, and Mike Hildreth. Snowmass 2013 Computing Frontier Storage and Data Management. *arXiv Prepr. arXiv1311.4580*, 2013.

[18] Bryan Catanzaro, Narayanan Sundaram, Kurt Keutzer, and Cory Hall. A MapReduce Framework for Programming Graphics Processors. In *Work. Softw. Tools MultiCore Syst.*, 2008.

[19] Ronnie Chaiken, Bob Jenkins, Per-Åke Larson, Bill Ramsey, Darren Shakib, Simon Weaver, and Jingren Zhou. SCOPE: easy and efficient parallel processing of massive data sets. *PVLDB*, 1(2):1265–1276, 2008.

[20] Fay Chang, Jeffrey Dean, Sanjay Ghemawat, Wilson C Hsieh, Deborah A Wallach, Mike Burrows, Tushar Chandra, Andrew Fikes, and Robert E Gruber. Bigtable: A distributed storage system for structured data. *ACM Trans. Comput. Syst.*, 26(2):4, 2008. ISSN 07342071. doi: 10.1145/1365815.1365816.

[21] Huajun Chen, Xi Chen, Peiqin Gu, Zhaohui Wu, and Tong Yu. Owl reasoning framework over big biological knowledge network. *BioMed research international*, 2014. doi: 10.1155/2014/272915.

[22] Tyson Condie, Neil Conway, Peter Alvaro, Joseph M Hellerstein, Khaled Elmeleegy, and Russell Sears. MapReduce Online. In *Proc. 7th USENIX Conf. Networked Syst. Des. Implement.*, page 21, 2010.

[23] Richard Cyganiak and Anja Jentzsch. Linking open data cloud diagram, 2014. URL `http://lod-cloud.net`. [Online; accessed: August-2015].

[24] Michael C. Daconta, Leo Obrst, and Kevin T. Smith. *The Semantic Web: a guide to the future of XML, Web services, and knowledge management*. Wiley, 2003. ISBN 0471432571.

[25] C David, C Olivier, and B Guillaume. A survey of RDF storage approaches. *ARIMA Journal.*, 15:11–35, 2012.

[26] Marc de Kruijf and Karthikeyan Sankaralingam. Mapreduce for the cell broadband engine architecture. *IBM Journal of Research and Development*, 53(5):10–11, 2009.

[27] Jeffrey Dean and Sanjay Ghemawat. MapReduce : Simplified Data Processing on Large Clusters. *Commun. ACM*, 51(1):107–113, 2008. ISSN 00010782. doi: 10.1145/1327452.1327492.

[28] David J. DeWitt, Erik Paulson, Eric Robinson, Jeffrey F. Naughton, Joshua Royalty, Srinath Shankar, and Andrew Krioukov. Clustera: an integrated computation and data management system. *PVLDB*, 1(1):28–41, 2008.

[29] Jens Dittrich, Jorge-Arnulfo Quiané-Ruiz, Alekh Jindal, Yagiz Kargin, Vinay Setty, and Jörg Schad. Hadoop ++ : Making a Yellow Elephant Run Like a Cheetah (Without It Even Noticing). *PVLDB*, 3(1):515–529, 2010.

[30] M Durst and M Suignard. Rfc 3987, internationalized resource identifiers (iris), 2005.

[31] Jaliya Ekanayake, Hui Li, Bingjing Zhang, Thilina Gunarathne, Seung-Hee Bae, Judy Qiu, and Geoffrey Fox. Twister: a runtime for iterative MapReduce. In *Proceedings of the 19th ACM International Symposium on High Performance Distributed Computing*, pages 810–818. ACM, 2010.

[32] Orri Erling and Ivan Mikhailov. Towards web scale RDF. *Proc. 4th Int. Work. Scalable Semant. Web Knowl. Base Syst.*, 2008.

[33] Ivan Ermilov, Michael Martin, Jens Lehmann, and Sören Auer. Linked open data statistics: Collection and exploitation. In Pavel Klinov and Dmitry Mouromtsev, editors, *Knowledge Engineering and the Semantic Web*, volume 394 of *Communications in Computer and Information Science*, pages 242–249. Springer Berlin Heidelberg, 2013. ISBN 978-3-642-41359-9. doi: 10.1007/978-3-642-41360-5_19.

[34] David C Fallside and Priscilla Walmsley. *XML schema part 0: primer second edition*, volume 16. W3C recommendation, 2004.

[35] Javier D Fernández, Mario Arias, Miguel A Martínez-prieto, Claudio Gutiérrez, Javier D Fern, Claudio Guti, and Miguel A Mart. Management of Big Semantic Data. *Big Data Computing*, pages 131–167, 2013.

[36] Javier D. Fernández, Miguel A. Martínez-Prieto, Claudio Gutiérrez, Axel Polleres, and Mario Arias. Binary rdf representation for publication and exchange (hdt). *Web Semantics: Science, Services and Agents on the World Wide Web*, 19:22–41, 2013. ISSN 1570-8268.

[37] Pedro Ferrera, Ivan de Prado, Eric Palacios, Jose Luis Fernandez-Marquez, Giovanna Di Marzo Serugendo, and G Di Marzo. Tuple MapReduce: Beyond Classic MapReduce. In *12th IEEE International Conference on Data Mining*, pages 260–269. IEEE, Ieee, December 2012. ISBN 978-1-4673-4649-8. doi: 10.1109/ICDM.2012.141.

[38] Roy Fielding, Jim Gettys, Jeffrey Mogul, Henrik Frystyk, Larry Masinter, Paul Leach, and Tim Berners-Lee. Hypertext transfer protocol–http/1.1. Technical report, 1999.

[39] Robert W Floyd. Algorithm 97: Shortest path. *Communications of the ACM*, 5(6):345, 1962.

[40] Eric Friedman, Peter Pawlowski, and John Cieslewicz. SQL/MapReduce: A practical approach to self-describing, polymorphic, and parallelizable user-defined functions. *PVLDB*, 2(2):1402–1413, 2009.

[41] J Gantz and D Reinsel. Extracting value from chaos. *White Pap. IDC*, 2011.

[42] J F Gantz and C Chute. The diverse and exploding digital universe: An updated forecast of worldwide information growth through 2011. IDC, 2008.

[43] Alan F Gates, Olga Natkovich, Shubham Chopra, Pradeep Kamath, Shravan M Narayanamurthy, Christopher Olston, Benjamin Reed, Santhosh Srinivasan, and Utkarsh Srivastava. Building a high-level dataflow system on top of MapReduce: the Pig experience. *PVLDB*, 2(2):1414–1425, 2009.

[44] Lars George. *HBase - The Definitive Guide: Random Access to Your Planet-Size Data*. O'Reilly, 2011. ISBN 978-1-449-39610-7.

[45] Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung. The google file system. In *Proceedings of the 19th ACM Symposium on Operating Systems Principles*, pages 29–43, 2003. doi: 10.1145/945445.945450.

[46] José M. Giménez-García, Javier D. Fernández, and Miguel A. Martínez-Prieto. Mapreduce-based solutions for scalable sparql querying. *Open Journal of Semantic Web (OJSW)*, 1(1): 1–18, 2014. ISSN 2199-336X.

[47] José M. Giménez-García, Javier D. Fernández, and Miguel A. Martínez-Prieto. HDT-MR: A scalable solution for rdf compression with HDT and mapreduce. In Fabien Gandon, Marta Sabou, Harald Sack, Claudia d'Amato, Philippe Cudré-Mauroux, and Antoine Zimmermann, editors, *The Semantic Web. Latest Advances and New Domains*, volume 9088 of *Lecture Notes in Computer Science*, pages 253–268. Springer International Publishing, 2015. ISBN 978-3-319-18817-1. doi: 10.1007/978-3-319-18818-8_16.

[48] F Goasdoué and Z Kaoudi. CliqueSquare: efficient Hadoop-based RDF query processing. *Journées de Bases de Données Avancées*, pages 1–28, 2013.

[49] Jan Grant and Dave Beckett, editors. *RDF Test Cases*. W3C Recommendation, 2004.

[50] W3C Owl Working Group. *OWL 2 Web Ontology Language Document Overview*. W3C Recommendation, 2009.

[51] Yuanbo Guo, Zhengxiang Pan, and Jeff Heflin. LUBM: A benchmark for OWL knowledge base systems. *Web Semant. Sci. Serv. Agents World Wide Web*, 3(2):158–182, 2005.

[52] Andreas Harth, Jürgen Umbrich, Aidan Hogan, and Stefan Decker. YARS2: A federated repository for querying graph structured data from the web. In *The Semantic Web, 6th International Semantic Web Conference.*, pages 211–224, 2007. doi: 10.1007/978-3-540-76298-0_16.

[53] Bingsheng He, W Fang, Qiong Luo, Naga K Govindaraju, and Tuyong Wang. Mars: a MapReduce framework on graphics processors. In *17th International Conference on Parallel Architecture and Compilation Techniques*, pages 260–269. ACM, 2008. ISBN 9781605582825.

[54] Yongqiang He, Rubao Lee, Yin Huai, Zheng Shao, Namit Jain, Xiaodong Zhang, and Zhiwei Xu. Rcfile: A fast and space-efficient data placement structure in MapReduce-based warehouse systems. In *Proceedings of the 27th International Conference on Data Engineering*, pages 1199–1208. IEEE, IEEE, April 2011. ISBN 978-1-4244-8959-6. doi: 10.1109/ICDE.2011.5767933.

[55] Jim Hendler. Broad data: Exploring the emerging web of data. *Big Data*, 1(1):18–20, 2013.

[56] Antonio Hernández-Illera, Miguel A. Martínez-Prieto, and Javier D. Fernández. Serializing rdf in compressed space. In *Data Compression Conference (DCC)*, pages 363–372, 2015.

[57] Aidan Hogan, Andreas Harth, and Axel Polleres. Scalable authoritative owl reasoning for the web. *IGI Global*, 5(2):49–90, 2009.

[58] Jiewen Huang, D J Abadi, Kun Ren, and Daniel J Abadi. Scalable SPARQL Querying of Large RDF Graphs. *PVLDB*, 4(11):1123–1134, 2011.

[59] MF F Husain, Pankil Doshi, Latifur Khan, and B Thuraisingham. Storage and retrieval of large RDF graph using Hadoop and MapReduce. pages 680–686, 2009. ISBN 978-3-642-10664-4. doi: 10.1007/978-3-642-10665-1.

[60] MF F Husain, Latifur Khan, M Kantarcioglu, and Bhavani Thuraisingham. Data intensive query processing for large RDF graphs using cloud computing tools. pages 1–10. Ieee, July 2010. ISBN 978-1-4244-8207-8. doi: 10.1109/CLOUD.2010.36.

[61] Mohammad Husain, James McGlothlin, Mohammad M. Masud, Latifur Khan, and Bhavani M. Thuraisingham. Heuristics-Based Query Processing for Large RDF Graphs Using Cloud Computing. *IEEE Trans. Knowl. Data Eng.*, 23(9):1312–1327, September 2011. ISSN 1041-4347. doi: 10.1109/TKDE.2011.103.

[62] M Isard and Y Yu. Distributed data-parallel computing using a high-level programming language. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 987–994. ACM, 2009. ISBN 9781605585512.

[63] Michael Isard, Andrew Birrell, Dennis Fetterly, M Budiu, and Y Yu. Dryad: distributed data-parallel programs from sequential building blocks. volume 41, pages 59–72. ACM, 2007. ISBN 9781595936363.

[64] David Jiang, Anthony K. H. Tung, and Gang Chen. Map-join-reduce: Toward scalable and efficient data analysis on large clusters. *IEEE Trans. Knowl. Data Eng.*, 23(9):1299–1311, September 2011. ISSN 1041-4347. doi: 10.1109/TKDE.2010.248.

[65] Dawei Jiang, B C Ooi, L Shi, and S Wu. The performance of MapReduce: An in-depth study. *PVLDB*, 3(1-2):472–483, 2010.

[66] W Jiang, Vignesh T Ravi, and Gagan Agrawal. A MapReduce system with an alternate API for multi-core environments. In *Proc. 2010 10th IEEE/ACM Int. Conf. Clust. Cloud Grid Comput.*, pages 84–93. IEEE Computer Society, Ieee, May 2010. ISBN 978-1-4244-6987-1. doi: 10.1109/CCGRID.2010.10.

[67] G Karypis and V Kumar. Metis-unstructured graph partitioning and sparse matrix ordering system, version 2.0. 1995.

[68] R T Kaushik and M Bhandarkar. GreenHDFS: Towards an Energy-Conserving Storage-Efficient, Hybrid Hadoop Compute Cluster. In *Proc. USENIX Annu. Tech. Conf.*, 2010.

[69] Hyeongsik Kim, Padmashree Ravindra, and Kemafor Anyanwu. Optimizing RDF(S) Queries on Cloud Platforms. In *22nd International World Wide Web Conference*, pages 261–264, 2013. ISBN 9781450320382.

[70] George Kollios, Nick Koudas, T Nykiel, M Potamias, and C Mishra. MRShare: Sharing across multiple queries in MapReduce. *PVLDB*, 3(1-2):494–505, 2010.

[71] S Kotoulas, E Oren, and F Van Harmelen. Mind the Data Skew: Distributed Inferencing by Speeddating in Elastic Regions. In *Proceedings of the 19th International Conference on World Wide Web*, pages 531–540, 2010. ISBN 978-1-60558-799-8.

[72] G Ladwig and Andreas Harth. CumulusRDF: Linked data management on nested key-value stores. *7th Int. Work. Scalable Semant. Web Knowl. Base Syst. (SSWS 2011)*, pages 30–42, 2011.

[73] W Lang and J M Patel. Energy management for MapReduce clusters. *PVLDB*, 3(1-2): 129–139, 2010.

[74] Ora Lassila and Ralph R Swick. *Resource description framework (RDF) model and syntax specification*. W3C Recommendation, 1999.

[75] George Lee, Jimmy Lin, Chuang Liu, Andrew Lorek, and Dmitriy Ryaboy. The unified logging infrastructure for data analytics at Twitter. *PVLDB*, 5(12):1771–1780, 2012.

[76] Kisung Lee and L Liu. Scaling Queries over Big RDF Graphs with Semantic Hash Partitioning. *PVLDB*, 6(14):1894–1905, 2013.

[77] Kyong-ha H K.-h. Lee, Y.-j. Yoon-joon J Lee, H Choi, Y D Chung, and B Moon. Parallel Data Processing with MapReduce: a Survey. *ACM SIGMOD Record*, 40(4):11–20, 2012.

[78] J Leverich and C Kozyrakis. On the energy (in) efficiency of Hadoop clusters. *ACM SIGOPS Oper. Syst. Rev.*, 44(1):61–65, 2010.

[79] Boduo Li, Edward Mazur, Yanlei Diao, Andrew McGregor, and Prashant Shenoy. A platform for scalable one-pass analytics using MapReduce. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 985–996, New York, New York, USA, 2011. ACM, ACM Press. ISBN 9781450306614. doi: 10.1145/1989323.1989426.

[80] Jimmy Lin and Chris Dyer. Data-intensive text processing with MapReduce. *Synth. Lect. Hum. Lang. Technol.*, 3(1):1–177, January 2010. ISSN 1947-4040. doi: 10.2200/S00274ED1V01Y201006HLT007.

[81] Yuting Lin, Chun Chen, D Agrawal, B C Ooi, and S Wu. Llama: leveraging columnar storage for scalable join processing in the MapReduce framework. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 961–972. ACM, 2011. ISBN 9781450306614.

[82] Chang Liu, Guilin Qi, Haofen Wang, and Yong Yu. Large scale fuzzy pd* reasoning using mapreduce. In *The Semantic Web - ISWC 2011 - 10th International Semantic Web Conference*, pages 405–420. Springer, 2011. ISBN 978-3-642-25072-9.

[83] Dionysios Logothetis and Kenneth Yocum. Ad-hoc data processing in the cloud. *PVLDB*, 1(2):1472–1475, 2008.

[84] G Malewicz, M H Austern, A J C Bik, J C Dehnert, I Horn, N Leiser, and G Czajkowski. Pregel: a system for large-scale graph processing. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 135–146. ACM, 2010. ISBN 978-1-4503-0032-2.

[85] M. A. Martínez-Prieto, N. Brisaboa, R. Cánovas, F. Claude, and G. Navarro. Practical compressed string dictionaries. 2015. To appear.

[86] M.A. Martínez-Prieto, J.D. Fernández, and R. Cánovas. Querying RDF dictionaries in compressed space. *SIGAPP Appl. Comput. Rev.*, 12(2):64–77, 2012.

[87] S Melnik, A Gubarev, J J Long, G Romer, S Shivakumar, M Tolton, and T Vassilakis. Dremel: interactive analysis of web-scale datasets. *Commun. ACM*, 54(6):114–123, 2011.

[88] N. Mirajkar, S. Bhujbal, and A. Deshmukh. Perform wordcount Map-Reduce Job in Single Node Apache Hadoop cluster and compress data using Lempel-Ziv-Oberhumer (LZO) algorithm. *CoRR*, abs/1307.1517, 2013. arXiv:1307.1517.

[89] K Morton and A Friesen. KAMD: A Progress Estimator for MapReduce Pipelines. Technical report, Computer Science and Engineering Department, University of Washington, 2010.

[90] K Morton, M Balazinska, and D Grossman. ParaTimer: a progress indicator for MapReduce DAGs. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 507–518. ACM, 2010. ISBN 978-1-4503-0032-2.

[91] Kristi Morton, Abram Friesen, Magdalena Balazinska, and Dan Grossman. Estimating the progress of MapReduce pipelines. In *Proceedings of the 26th International Conference on Data Engineering*, pages 681–684. IEEE, IEEE, 2010. ISBN 978-1-4244-5445-7. doi: 10.1109/ICDE.2010.5447919.

[92] Boris Motik, Bernardo Cuenca Grau, Ian Horrocks, Zhe Wu, Achille Fokoue, and Carsten Lutz. *OWL 2 Web Ontology Language: Profiles*, volume 27. W3C recommendation, 2009.

[93] Jaeseok Myung, Jongheum Yeon, and Sang-Goo Lee. SPARQL basic graph pattern processing with iterative MapReduce. *Proc. 2010 Work. Massive Data Anal. Cloud - MDAC '10*, pages 1–6, 2010. doi: 10.1145/1779599.1779605.

[94] T Neumann and G Weikum. The RDF-3X Engine for Scalable Management of RDF data. *VLDB J.*, 19(1):91–113, 2010. ISSN 1066-8888.

[95] Christopher Olston, Benjamin Reed, U Srivastava, Ravi Kumar, and Andrew Tomkins. Pig latin: a not-so-foreign language for data processing. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 1099–1110. ACM, 2008. ISBN 9781605581026.

[96] Alisdair Owens, Andy Seaborne, and Nick Gibbins. Clustered TDB: a clustered triple store for Jena. 2008.

[97] Nikolaos Papailiou, Ioannis Konstantinou, Dimitrios Tsoumakos, and Nectarios Koziris. H2RDF: Adaptive Query Processing on RDF Data in the Cloud. pages 397–400, 2012. ISBN 9781450312301.

[98] Thomas B Passin. *Explorer's guide to the semantic web*. Manning Greenwich, 2004. ISBN 1932394206.

[99] Rob Pike, Sean Dorward, Robert Griesemer, and Sean Quinlan. Interpreting the data: Parallel analysis with Sawzall. *Scientific Programming*, 13(4):277–298, 2005.

[100] E Prud'hommeaux and A Seaborne. *SPARQL Query Language for RDF*. W3C Recommendation, 2008.

[101] C Ranger, R Raghuraman, A Penmetsa, G Bradski, and C Kozyrakis. Evaluating MapReduce for multi-core and multiprocessor systems. In *13st International Conference on High-Performance Computer Architecture*, pages 13–24. IEEE, 2007. ISBN 1-4244-0804-0.

[102] Padmashree Ravindra, Vikas V. Deshpande, and Kemafor Anyanwu. Towards scalable RDF graph analytics on MapReduce. *Proc. 2010 Work. Massive Data Anal. Cloud - MDAC '10*, pages 1–6, 2010. doi: 10.1145/1779599.1779604.

[103] Padmashree Ravindra, Hyeongsik Kim, and Kemafor Anyanwu. An Intermediate Algebra for Optimizing RDF Graph Pattern Matching on MapReduce. pages 46–61, 2011. ISBN 978-3-642-21063-1.

[104] Kurt Rohloff and RE E Richard E Schantz. High-performance, massively scalable distributed systems using the MapReduce software framework: The SHARD triple-store. In *SPLASH Workshop on Programming Support Innovations for Emerging Distributed Applications*, page 4, 2010. ISBN 9781450305440.

[105] Alexander Schätzle. PigSPARQL: mapping SPARQL to Pig Latin. In *Proceedings of the International Workshop on Semantic Web Information Management*, page 4, 2011. ISBN 9781450306515.

[106] Alexander Schätzle and M Przyjaciel-Zablocki. Cascading Map-Side Joins over HBase for Scalable Join Processing. *CoRR*, 2012.

[107] Michael Schmidt, Thomas Hornung, Georg Lausen, and Christoph Pinkel. Sp2bench: a sparql performance benchmark. In *Data Engineering, 2009. ICDE'09. IEEE 25th International Conference on*, pages 222–233. IEEE, 2009.

[108] Konstantin Shvachko, Hairong Kuang, Sanjay Radia, and Robert Chansler. The Hadoop distributed file system. In *IEEE 26th Symposium on Mass Storage Systems and Technologies*, pages 1–10. IEEE, Ieee, May 2010. ISBN 978-1-4244-7152-2. doi: 10.1109/MSST.2010.5496972.

[109] Lefteris Sidirourgos, Romulo Goncalves, Martin Kersten, Niels Nes, and Stefan Manegold. Column-store support for RDF data management: not all swans are white. *PVLDB*, 1(2): 1553–1563, 2008.

[110] Radhika Sridhar, Padmashree Ravindra, and Kemafor Anyanwu. RAPID: Enabling scalable ad-hoc analytics on the semantic web. In *The Semantic Web - ISWC 2009, 8th International Semantic Web Conference*, pages 703–718, 2009. ISBN 978-3-642-04929-3.

[111] Theoretical Statistics and Physical Sciences. *MASSIVE DATA*. 2013. ISBN 9780309287784.

[112] Jianling Sun and Qiang Jin. Scalable RDF Store Based on HBase and MapReduce. *Proc. 3rd Int. Conf. Adv. Comput. Theory Eng.*, pages 633–636, 2010.

[113] Herman J. ter Horst. Completeness, decidability and complexity of entailment for RDF Schema and a semantic extension involving the OWL vocabulary. *Web Semant. Sci. Serv. Agents World Wide Web*, 3(2-3):79–115, October 2005. ISSN 15708268. doi: 10.1016/j.websem.2005.06.001.

[114] Henry S Thompson, David Beech, M Maloney, et al. Xml schema part 1: Structures second edition, 2004.

[115] Ashish Thusoo, Joydeep Sen Sarma, Namit Jain, Zheng Shao, Prasad Chakka, Suresh Anthony, Hao Liu, Pete Wyckoff, and Raghotham Murthy. Hive: a warehousing solution over a MapReduce framework. *PVLDB*, 2(2):1626–1629, 2009.

[116] Ashish Thusoo, Joydeep Sen Sarma, Namit Jain, Zheng Shao, Prasad Chakka, Ning Zhang, Suresh Antony, Hao Liu, and Raghotham Murthy. Hive-a petabyte scale data warehouse using Hadoop. In *Proceedings of the 26th International Conference on Data Engineering*, pages 996–1005. IEEE, Ieee, 2010. ISBN 978-1-4244-5445-7. doi: 10.1109/ICDE.2010.5447738.

[117] Ashish Thusoo, Zheng Shao, Suresh Anthony, Dhruba Borthakur, Namit Jain, Joydeep Sen Sarma, Raghotham Murthy, and Hao Liu. Data warehousing and analytics infrastructure at facebook. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 1013–1020, New York, New York, USA, 2010. ACM, ACM Press. ISBN 978-1-4503-0032-2. doi: 10.1145/1807167.1807278.

[118] Jacopo Urbani and Frank Van Harmelen. QueryPIE: Backward reasoning for OWL Horst over very large knowledge bases. In *The Semantic Web - ISWC 2011 - 10th International Semantic Web Conference*, pages 730–745, 2011. ISBN 978-3-642-25072-9.

[119] Jacopo Urbani and Spyros Kotoulas. OWL reasoning with WebPIE: calculating the closure of 100 billion triples. In *The Semantic Web: Research and Applications, 7th Extended Semantic Web Conference*, pages 213–227, 2010. ISBN 978-3-642-13485-2.

[120] Jacopo Urbani, Spyros Kotoulas, Eyal Oren, and F Van Harmelen. Scalable distributed reasoning using MapReduce. In *The Semantic Web - ISWC 2009, 8th International Semantic Web Conference*, volume 48 of *Lecture Notes in Computer Science*, pages 623–638. Springer, 2009. ISBN 978-3-642-04929-3.

[121] Jacopo Urbani, Jason Maassen, and Henri Bal. Massive Semantic Web data compression with MapReduce. In *Proceedings of the 19th ACM International Symposium on High Performance Distributed Computing*, pages 795–802, New York, New York, USA, 2010. ACM Press. ISBN 978-1-60558-942-8. doi: 10.1145/1851476.1851591.

[122] D Warneke and O Kao. Nephele: efficient parallel data processing in the cloud. In *Proceedings of the 2nd Workshop on Many-Task Computing on Grids and Supercomputers*, page 8. ACM, 2009.

[123] Cathrin Weiss, Panagiotis Karras, and Abraham Bernstein. Hexastore: Sextuple Indexing for Semantic Web Data Management. *PVLDB*, 1(1):1008–1019, 2008. ISSN 2150-8097.

[124] Tom White. *Hadoop: The Definitive Guide*. Definitive Guide Series. O'Reilly, 2012. ISBN 978-1-449-31152-0.

[125] Ian H Witten, Alistair Moffat, and Timothy C Bell. *Managing gigabytes: compressing and indexing documents and images*. Morgan Kaufmann, 1999.

[126] Yu Xu, Pekka Kostamaa, and Like Gao. Integrating Hadoop and parallel DBMs. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 969–974, New York, New York, USA, 2010. ACM, ACM Press. ISBN 978-1-4503-0032-2. doi: 10.1145/1807167.1807272.

[127] H.-c. Hung-chih Yang, Ali Dasdan, R.-l. Ruey-lung L Hsiao, and D Stott Parker. Map-reduce-merge: simplified relational data processing on large clusters. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 1029–1040. ACM, 2007. ISBN 978-1-59593-686-8.

[128] M Zaharia, A Konwinski, A D Joseph, R Katz, and I Stoica. Improving MapReduce performance in heterogeneous environments. In *8th USENIX Symposium on Operating Systems Design and Implementation*, pages 29–42, 2008. ISBN 978-1-931971-65-2.

Appendix A

HDT-MR parameters

HDT-MR behavior can be controlled by configuration file or command line options. The configuration file used by default is `HDTMRBuilder.xml`, but other file can be used by using the options `-c, --conf`. The following are the possible parameters that can be included in a configuration file.

- `global.bucket`: *Amazon Web Services bucket.* If bucket is specified, HDT-MR will use *Amazon S3* storage service for input and output. It can be overridden using the options `-a, --awsbucket`.
- `global.path.base`: *Root directory for the process.* If a value is specified, it will be used as the root directory in HDFS. Other directories will be located in the base directory. It can be overridden using the options `-b, --basedir`.
- `global.path.input`: *Path to input files. Relative to basedir.* If a directory is specified, it is used as input path for HDT-MR. The default value is `input`. It can be overridden using the options `-i, --input`.
- `hdt.build`: *Whether to build HDT or not.* Boolean parameter. Is set to true HDT is built. The default value is `true`. It can be overridden using the options `-bh, --buildhdt`.
- `hdt.dictionary.build`: *Whether to build HDT dictionary or not.* Boolean parameter. Is set to true the dictionary is built. The default value is `true`. It can be overridden using the options `-bd, --builddictionary`.
- `hdt.dictionary.file`: *Name of hdt dictionary file.* If a value is specified, it is used for the HDT dictionary file name. The default value is `dictionary.hdt`. It can be overridden using the options `-fd, --filedictionary`.
- `hdt.file`: *Name of hdt file.* If a value is specified, it is used for the HDT file name. The default value is `output.hdt`. It can be overridden using the options `-fh, --namehdtfile`.
- `hdt-lib.baseUri`: *Base URI.* If a URI is specified, it is used for the triples of the *Header*. The default values is `http://rdfhdt.org/HDTMR`. It can be overridden using the options `-bu, --baseURI`.
- `hdt-lib.configFile`: *Conversion config file.* If a file is specified, it is forwarded to `hdt-lib` to be used as configuration file. It can be overridden using the options `-hc, --hdtconf`.

- `hdtl-lib.options`: *HDT Conversion options (override those of config file).* If options are specified, they are forwarded to `hdt-lib` to be used as options. It can be overridden using the options `-o, --options`.

- `hdt-lib.quiet`: *Do not show progress of the conversion.* Boolean paramenter. If used, the progress conversion of triples by `hdt-java` is not shown. The default value is `false`. It can be overridden using the options `-q, --quiet`.

- `job.dictionary.name`: *Name of dictionary job.* If a value is identified, it is used for the name of the dictionary job. The default value is `DictionaryJob`. It can be overridden using the options `-nd, --namedictionaryjob`.

- `job.dictionary.path.output`: *Path to dictionary job output files. Relative to basedir.* If a directory is specified, it is used as output path for the dictionary job and/or input path for the triples job. The default value is `dictionary`. It can be overridden using the options `-od, --outputdictionary`.

- `job.dictionary.path.output.delete`: *Delete dictionary job output path before running job.* Boolean parameter. Is set to true, the dictionary job output directory is deleted before running the dictionary job. A job fails if its output directory already exists. The default value is `false`. It can be overridden using the options `-dd, --deleteoutputdictionary`.

- `job.dictionary.path.sample`: *Path to dictionary job sample files. Relative to basedir.* If a directory is specified, it is used as output path for the dictionary sampling job and/or input path for the dictionary job. The default value is `dictionary_samples` It can be overridden using the options `-sd, --samplesdictionary`.

- `job.dictionary.path.sample.delete`: *Delete dictionary job sample path before running job.* Boolean parameter. Is set to true, the dictionary sampling job output directory is deleted before running the dictionary sampling job. A job fails if its output directory already exists. The default value is `false`. It can be overridden using the options `-dsd, --deletesampledictionary`.

- `job.dictionary.reducers`: *Number of reducers for dictionary job.* If a value is specified, it is used for the number of reducers in the dictionary job. The default value is `1`. It can be overridden using the options `-Rd, --reducersdictionary`

- `job.dictionary.run`: *Whether to run dictionary job or not.* Boolean parameter. If set to `true`, the dictionary job is launched. The default value is `true`. It can be overridden using the options `-rd, --rundictionary`.

- `job.dictionary.sample.probability`: *Probability of using each element for sampling in dictionary job.* If a value is specified, it is used as frequency by the `InputSampler`. The default value is `0.001`. It can be overridden using the options `-p, --sampleprobability`.

- `job.dictionary.sample.reducers`: *Number of reducers for dictionary input sampling job.* If a value is specified, it is used for the number of reducers in the dictionary sampling job. The default value is `1`. It can be overridden using the options `-Rds, --reducersdictionarysampling`.

- `job.dictionary.sample.run`: *Whether to run dictionary input sampling job or not.* Boolean parameter. If set to `true`, the dictionary sampling job is launched. The default value is `true`. It can be overridden with `-rds, --rundictionarysampling`.
- `job.triples.name`: *Name of triples job.* If a value is identified, it is used for the name of the dictionary job. The default value is `TriplesJob`. It can be overridden using the options `-nt, --nametriplesjob`.
- `job.triples.path.output`: *Path to triples job output files. Relative to basedir.* If a directory is specified, it is used as output path for the triples job. The default value is `triples`. It can be overridden using the options `-ot, --outputtriples`.
- `job.triples.path.output.delete`: *Delete triples job output path before running job.* Boolean parameter. Is set to true, the triples job output directory is deleted before running the triples job. A job fails if its output directory already exists. The default value is `false`. It can be overridden using the options `-dt, --deleteoutputtriples`.
- `job.triples.path.sample`: *Path to triples job sample files. Relative to basedir.* If a directory is specified, it is used as output path for the triples sampling job and/or input path for the triples job. The default value is `triples_samples` It can be overridden using the options `-st, --samplestriples`
- `job.triples.path.sample.delete`: *Delete triples job sample path before running job.* Boolean parameter. Is set to true, the triples sampling job output directory is deleted before running the triples sampling job. A job fails if its output directory already exists. The default value is `false`. It can be overridden using the options `-dst, --deletesampletriples`.
- `job.triples.reducers`: *Number of reducers for triples job.* If a value is specified, it is used for the number of reducers in the triples job. The default value is `1`. It can be overridden using the options `-Rt, --reducerstriples`.
- `job.triples.run`: *Whether to run triples job or not.* Boolean parameter. If set to `true`, the triples job is launched. The default value is `true`. It can be overridden using the options `-rt, --runtriples`.
- `job.triples.sample.probability`: *Probability of using each element for sampling in triples job.* If a value is specified, it is used as frequency by the `InputSampler`. The default value is `0.001`. It can be overridden with `-p, --sampleprobability`.
- `job.triples.sample.reducers`: *Number of reducers for triples input sampling job.* If a value is specified, it is used for the number of reducers in the triples sampling job. The default value is `1`. It can be overridden with `-Rts, --reducerstriplessampling`.
- `job.triples.sample.run`: *Whether to run triples input sampling job or not.* Boolean parameter. If set to `true`, the triples sampling job is launched. The default value is `true`. It can be overridden using the options `-rts, --runtriplessampling`.

Appendix B

HDT-MR configuration files

The following is the content of the configuration files used to perform the evaluation of LUBM datasets on chapter 5.

B.1 Dictionary Encoding

```
<configuration>

  <property>
    <name>job.dictionary.run</name>
    <value>true</value>
  </property>

  <property>
    <name>job.dictionary.sample.run</name>
    <value>true</value>
  </property>

  <property>
    <name>job.dictionary.sample.reducers</name>
    <value>10</value>
  </property>

  <property>
    <name>hdt.dictionary.build</name>
    <value>true</value>
  </property>

  <property>
    <name>job.triples.run</name>
    <value>false</value>
  </property>

  <property>
```

```
  <name>job.triples.sample.run</name>
  <value>false</value>
</property>

<property>
  <name>hdt.build</name>
  <value>false</value>
</property>

<property>
  <name>global.path.base</name>
  <value>.</value>
  <description>Root directory</description>
</property>

<property>
  <name>global.path.input</name>
  <value>lubm</value>
  <description>input path</description>
</property>

<property>
  <name>job.dictionary.path.output</name>
  <value>dictionary</value>
  <description>
    Dictionary output path / Triples input path
  </description>
</property>

<property>
  <name>job.dictionary.path.output.delete</name>
  <value>true</value>
  <description>
    Whether to delete dictionary output path
  </description>
</property>

<property>
  <name>job.dictionary.path.sample</name>
  <value>dictionary_sample</value>
  <description>Dictionary samples path</description>
</property>

<property>
  <name>job.dictionary.path.sample.delete</name>
  <value>true</value>
  <description>
    Whether to delete dictionary samples path
```

```
    </description>
  </property>

  <property>
    <name>job.dictionary.reducers</name>
    <value>10</value>
    <description>Number of reducers used by jobs</description>
  </property>

  <property>
    <name>job.dictionary.sample.probability</name>
    <value>0.000001</value>
    <description>Sampler Probability</description>
  </property>

</configuration>
```

B.2 Triples Encoding

```
<configuration>

  <property>
    <name>job.dictionary.run</name>
    <value>false</value>
  </property>

  <property>
    <name>job.dictionary.sample.run</name>
    <value>false</value>
  </property>

  <property>
    <name>hdt.dictionary.build</name>
    <value>false</value>
  </property>

  <property>
    <name>job.triples.run</name>
    <value>true</value>
  </property>

  <property>
    <name>job.triples.sample.run</name>
    <value>true</value>
  </property>
```

```
<property>
  <name>hdt.build</name>
  <value>true</value>
</property>

<property>
  <name>global.path.base</name>
  <value>.</value>
  <description>Root directory</description>
</property>

<property>
  <name>global.path.input</name>
  <value>lubm</value>
  <description>input path</description>
</property>

<property>
  <name>job.dictionary.path.output</name>
  <value>dictionary</value>
  <description>
    Dictionary output path / Triples input path
  </description>
</property>

<property>
  <name>job.triples.path.output.delete</name>
  <value>true</value>
  <description>
    Whether to delete triples output path
  </description>
</property>

<property>
  <name>job.triples.path.sample</name>
  <value>triples_sample</value>
  <description>Tripls samples path</description>
</property>

<property>
  <name>job.triples.path.sample.delete</name>
  <value>true</value>
  <description>
    Whether to delete tripls samples path
  </description>
</property>

<property>
```

```
    <name>job.triples.reducers</name>
    <value>10</value>
    <description>Number of reducers used by jobs</description>
  </property>

  <property>
    <name>job.triples.sample.probability</name>
    <value>0.000001</value>
    <description>Sampler Probability</description>
  </property>

</configuration>
```