Analytique des données massives basée sur des ontologies : application à la plateforme de formation SIDES 3.0 en médecine

Adam Hegel Sanchez Ayte

THÈSE

Pour obtenir le grade de

DOCTEUR DE L'UNIVERSITÉ GRENOBLE ALPES

École doctorale : MSTII - Mathématiques, Sciences et technologies de l'information, Informatique
Spécialité : Mathématiques et Informatique
Unité de recherche : Laboratoire d'Informatique de Grenoble

Analytique des données massives basée sur des ontologies : application à la plateforme de formation SIDES 3.0 en médecine

Large-scale ontology-based data analytics: application to the SIDES 3.0 training platform in Medicine

Présentée par :

Adam Hegel SANCHEZ AYTE

Direction de thèse :

Marie-Christine ROUSSET Professeur des Universités, Université Grenoble Alpes	Directrice de thèse
Fabrice JOUANOT Université Grenoble Alpes	Co-encadrant de thèse

Rapporteurs :

BERND AMANN
Professeur des Universités, SORBONNE UNIVERSITE
FRANÇOIS GOASDOUE
Professeur des Universités, UNIVERSITE DE RENNES

Thèse soutenue publiquement le **19 juin 2023**, devant le jury composé de :

BERND AMANN Professeur des Universités, SORBONNE UNIVERSITE	Rapporteur
FRANÇOIS GOASDOUE Professeur des Universités, UNIVERSITE DE RENNES	Rapporteur
NABIL LAYAIDA Directeur de recherche, INRIA CENTRE GRENOBLE-RHONE-ALPES	Président
MARLENE VILLANOVA-OLIVER Maître de conférences HDR, UNIVERSITE GRENOBLE ALPES	Examinatrice

Invités :

MARIE-CHRISTINE ROUSSET
Professeur des Universités, UNIVERSITE GRENOBLE ALPES
FABRICE JOUANOT
Maître de conférences, UNIVERSITE GRENOBLE ALPES

Large-scale ontology-based data analytics: application to the SIDES 3.0 training platform in Medicine

by

Adam Hegel Sánchez Ayte

A thesis
presented to the University of Grenoble Alpes
in fulfillment of the
thesis requirement for the degree of
Doctor of Philosophy
in
Mathematics and Computer Science

Grenoble, France, 2023

Examining Committee Membership

The following served on the Examining Committee for this thesis.

- Nabil LAYAIDA
  DIRECTEUR DE RECHERCHE, INRIA
  Examinateur, Président
- Bernd AMANN
  PROFESSEUR DES UNIVERSITES, Sorbonne Université
  Rapporteur
- François GOASDOUÉ
  PROFESSEUR DES UNIVERSITES, Université de Rennes 1
  Rapporteur
- Marlène VILLANOVA-OLIVER
  MAITRE DE CONFERENCES, Université Grenoble Alpes
  Examinatrice

Author's Declaration

I hereby declare that I am the sole author of this thesis. This is a true copy of the thesis, including any required final revisions, as accepted by my examiners.

I understand that my thesis may be made electronically available to the public.

Abstract

Modern Big Data technologies are utilized to construct scalable RDF data infrastructure for enabling reasoning at scale. The goal of this thesis is to conceptualize, design, and implement a comprehensive and scalable RDF data management infrastructure, leveraging state-of-the-art Big Data technologies. The core of the research is centered on addressing critical aspects of RDF data management including extraction, storage, processing, and execution of reasoning along with complex analytical queries.

This thesis takes place in the context of the SIDES 3.0 project. The aim is to transform the current French national e-learning platform for medical education, called SIDES, into an intelligent learning environment based on Semantic Web and Big Data technologies.

The contributions have substantial and varied impacts across different areas related with scalable RDF data management infrastructure. The first is the development of an Ontology-Based Data Access (OBDA) methodology to facilitate the evolution of a comprehensive knowledge base, named OntoSIDES. The methodology consists of two steps: the first step is manual, based on expert guidance, and involves the manual construction of the OntoSIDES ontology enriched with rules, while the second step is automatic and involves the automatic population of the ontology using OBDA mappings. The OBDA mappings can also be used to modularize a RDF graph by mapping RDF quads to queries in the datasources. This contribution showcases the feasibility of scaling ontology-based learning management systems in the field of Medicine.

The second contribution is the design and implementation of a Big RDF triplestore called TESS. TESS has a modular architecture that supports massive and reliable data updates, complemented by GPU acceleration for enhanced processing capabilities. The layers of TESS are data distribution, storage, query processing, transactional metadata layer, and reasoning.

The final contribution is a performance evaluation of complex queries and reasoning mechanisms to discern the most effective methods for implementing forward-chaining reasoning at scale in TESS. In this regard, diverse CPU-based implementations are evaluated: serial, parallel, incremental and modular. Additionally, GPU versus CPU performance is assessed in the context of serial forward chaining.

Acknowledgements

With these lines, one of the greatest intellectual adventures of my life comes to a close. The spark was ignited many years ago, reading Tim Berners Lee's "Weaving the Web." It made me understand that the web would be at the heart of a technological revolution where knowledge could finally be accessible to all of society.

I set myself the goal of being part of this revolution and charted a course of self-learning. I never envisioned this journey leading to a doctorate; my only desire was to learn and share as I learned. Looking back on my life, my passion for knowledge equipped me with an immense willpower that enabled me to overcome the material hardships of my upbringing.

A doctoral journey is not walked alone. I owe a great deal to so many people who guided me on which path to follow throughout my life. Within my doctoral studies, I am eternally grateful to my supervisors Marie-Christine Rousset and Fabrice Jouanot.

Marie-Christine welcomed me under her supervision while I was working as a software engineer in GRICAD for the SIDES 3.0 project. She taught me—with extraordinary patience—the essentials I needed to transition into a researcher. Her ability to instantly comprehend the core of a problem and formalize it is truly amazing. On the other hand, Fabrice Jouanot, with his profound knowledge about databases, played a crucial role in my research, guiding me safely while I was building the core architecture of my triplestore. His kindness and readiness to assist students are qualities that I will always admire.

In the SIDES 3.0 project, I am very grateful to Olivier Palombi and Grégory Mathes from UNESS. Olivier is a visionary at the intersection of computer science and medicine. He is always open to new ideas, a thorough researcher, and an incredibly kind person. He was an early promoter of the digitalization of the medical education in France. Grégory was always very comprehensive when it was difficult to find the right balance between my work as a software engineer and my duties as a PhD student.

My experience at GRICAD brought me closer to Christian Lenne, Christophe Cance, Alireza (Adrien) Moussaei and Mohannad Almasri. They are very close friends, always willing to discuss my ideas, and I am grateful for their support throughout my years as a software engineer, along with many others, including Yves Jacques, Lucy Ruffier, Myriam Laurens, Nicolas Gibelin, Pierre-Antoine Bouttier, Glenn Cougoulat, and Violaine Louvet.

Before my PhD, I worked in Rome and Grenoble. In Rome, I am very grateful to my friends at FAO: Johannes Keizer, Valeria Pesce, Antonella Picarella, Giampaolo Rugo, Ahsan Morshed, Erna Klupacs, Teresa Iniesta, Caterina Caracciolo, and Imma Subirats. They helped me settle in Rome during my first years in Europe when I barely spoke English. Johannes, in particular, had faith in my abilities and recommended me for a PhD.

In Grenoble, I met extraordinary people who later became dear friends. While working at INRIA, I met Armen Inants, Tatiana Lesnikova, Nicola Guillouet and Shreyas Saxena.

Jerome Euzenat and Jerome David, also at INRIA, taught me the first steps of becoming a researcher, and their lessons will always be cherished.

Before coming to Europe, I lived 38 years in Peru and I must express my gratitude to my teachers and lifelong friends there. At school, Gloria Medina, Susana Ricalde, Felícita Espino, Alicia Ypanaqué, Manuela Vásquez, Olinda Segovia, Pedro Castañeda, Manuela Medina, and Juanita Guillén, among others, shared their wisdom and companionship with me as I was building the pillars of my academic career. At university, Walter Gomez, Raul Morales and Julio Tello among others with whom I shared memorable moments as mechanical engineering students. I also want to acknowledge my lifelong friends: Juan Alvarez, Efrain Quispe, Grober Nolasco, and Yissella Acuache. Our friendship, formed in the early days of our lives, has endured the test of time.

Miguel Saravia, a dear friend, deserves special mention. He believed in my capabilities at a time when my overqualification made it difficult to find a job. Miguel was one of the first to promote the expansion of Internet connectivity in rural Peru. Luis Fernando Crespo, Andrés Gallego, and Gustavo Gutiérrez, the wise masters of the UNEC, equally deserve acknowledgement. These mentors illuminated my university days, instilling the mantra, "knowledge is for serving". Angel Palacios and Lourdes Puma also played vital roles, their constant inquiries over the years demonstrating their care. Furthermore, a special nod to Luis Huacho, who was an unwavering source of support during my initial years in Europe. As a lifelong friend, his companionship, albeit from a distance, has been steadfast and unfailing.

The presence of Arturo Vilca, my thesis advisor at the National University of Engineering (UNI) in Peru, was also providential throughout this journey. He supported me with timely advice to overcome all the difficulties I encountered.

Finally, I would like to express my deepest gratitude to my family in both Lima and Grenoble. I am profoundly grateful to my father, Feliciano, my mother, Ana, and my siblings, Javier, Jorge, Diana, and Ricardo, who have witnessed firsthand the challenges I have overcome to reach this point. I extend my heartfelt thanks to their partners, Andrea, Milagros, and Blue, for their unwavering support. I would also like to express my gratitude to the family of my wife: Victor, Felícita, Victor Alberto, Jesús, and Angélica, who have always been available whenever needed.

In Grenoble, words fall short to express my appreciation for my beloved wife Rosa (Rosita) and my cherished son Elias, who together make up my whole heart. I met Rosita when I was just 17, and since then, I've been blessed with her unwavering companionship. Elias Galileo, the joy of my existence, completes our family in a way that words cannot articulate. You two have been the bedrock of this extraordinary journey. I publicly apologize for the times when my studies took precedence over our precious shared moments.

To everyone mentioned and those who silently supported me, thank you. This journey has been an embodiment of a shared passion for knowledge and the power of resilience.

Table of Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Context

Nowadays, Big Data is revolutionizing the way we think about data management systems. Technological advances in computing and large-scale storage are allowing these systems to evolve and address increasingly complex challenges in processing and analyzing massive volumes of data. Big Data technologies are at the center of an unprecedented revolution in history, where the processing of vast amounts of information on an industrial scale is leading to important discoveries and advances in fields as important as medicine and artificial intelligence.

In recent years, RDF (Resource Description Framework) has become one of the most popular options for representing complex knowledge. RDF uses graphs to model knowledge, making it easier for humans to represent and more efficient for machines to process. As one of the pillars of the Semantic Web, RDF is commonly used to provide a graph-based representation of ontologies. Data modeled with RDF can be queried using SPARQL, a query language with powerful querying and reasoning capabilities. RDF's ability to represent complex relationships between data and its graph-based structure allows for easy integration and querying of data from various sources, making it a valuable tool for many industries and domains.

As RDF knowledge bases have become increasingly large, new challenges have arisen, many of them related to heavy querying and data consistency. Current systems have shown their limits and do not provide all data management services required to handle RDF data at scale. Among the RDF tasks challenged by the growing size of a dataset, we can find reasoning and the execution of complex (aggregated) queries.

Reasoning at scale involves the efficient processing and updating of large volumes of RDF data, making it critical to detect and correct errors in data modification operations. Optimizing these operations becomes essential to ensure the integrity and consistency of

the data at all times, and to avoid errors in the reasoning processes, which in turn ensures the quality and reliability of the results obtained.

Executing complex (aggregated) queries at scale often requires retrieving large amounts of data and satisfying a large number of conditions before aggregating the query. Failure to satisfy any of these conditions can result in an inconsistent response due to scalability issues.

Big Data technologies have proven to be effective in managing large amounts of RDF data, as evidenced by various studies [1, 5–7]. Many of these works have focused on optimizing data retrieval operations by proposing efficient data partitioning [8] and algorithms to optimize joins plans during a query [9]. By leveraging Big Data technologies, organizations can address the challenges associated with handling RDF data at scale, such as reasoning and executing complex queries. This enables them to access valuable insights, facilitate decision-making, and improve data-driven processes.

This thesis takes place in the context of the SIDES 3.0 project. The aim is to transform the current French national e-learning platform for medical education, called SIDES, into an intelligent learning environment based on Semantic Web and Big Data technologies. Since 2013, SIDES has been used by all French medical schools (34 nationwide) for assessments and diplomas. The new environment aims to replace the complex relational data model of SIDES, which is difficult to query and analyze efficiently, with an ontology called OntoSIDES, which provides students with a clear representation of their learning process and allows them to track their evaluation transparently.

The goal of this thesis is to conceptualize, design, and implement a comprehensive and scalable RDF data management infrastructure, leveraging state-of-the-art Big Data technologies. This infrastructure will address key aspects of RDF data management, including the extraction, storage, and processing of RDF data, as well as the execution of reasoning and complex analytical queries.

1.2 Our Research Focus

This thesis aims to address the challenges associated with creating a scalable and efficient RDF data management infrastructure that can effectively handles extraction, storage, reasoning, and complex analytical queries. In particular, we will focus on leveraging Big Data technologies to improve the reliability and integrity of RDF data management at scale, minimizing the impact on reasoning process performance. This research will include the design and experimentation of scalable infrastructures for large RDF triplestores, resulting in two potential research directions

- **Big data architectures for reasoning at scale**
  Investigate various cutting-edge technologies and techniques that can be used to

support reliable and efficient rule-based reasoning and heavy querying at scale. In particular, this investigation will explore how to integrate them into a modular architecture that is capable of handling the increasing size and complexity of big RDF data sets, as well as a variety of rule-based reasoning tasks.

- **Data modularization**
  Explore the potential connection between the modularization of RDF data and the massive and consistent updating of RDF data to improve the efficiency and reliability of RDF reasoning processes. In particular, we want to study how modularization can efficiently support the inference and querying of data from a subset of the graph, without the need to involve the entire graph in the process.

1.3 Main contributions of the thesis

In summary, the main contributions of this thesis are:

- An OBDA methodology to support the evolution of a big knowledge base called OntoSIDES. This contribution also shows the feasibility of an ontology-based learning management system at scale in Medicine.
- A Big RDF triplestore called TESS with a modular architecture that include support for massive and reliable data updates while also incorporating GPU acceleration for enhanced processing capabilities.
- A comparative performance evaluation for complex queries and reasoning to identify the most optimal ways to implement forward chaining reasoning at scale in TESS during a reasoning process.

The first contribution has been published in the Journal of Artificial Intelligence in Medicine in 2019 [10]. The second contribution and part of the third contribution have been published at the European Semantic Web Conference (ESWC) in 2022 [11].

1.4 Thesis outline

Chapter 2 covers the definition of the relevant notions used in this thesis, particularly those of related with Ontology Based Data Access and Rule-based reasoning.

Chapter 3 presents the methodology and the output of the OBDA-based approach that we have followed for constructing the different versions of the OntoSIDES knowledge base. In addition, it shows a novel use case of OBDA to modularize a RDF graph.

Chapter 4 presents the novel Big RDF triplestore architecture that we propose for solving the scalability issues. Among the novel components are a metadata management layer during transactions and the addition of GPU support to enhance performance.

In Chapter 5, we presented a comparative performance evaluation of complex queries and reasoning. To achieve this, we examined several forward chaining reasoning implementations, including serial, parallel, incremental, modular, and GPU, and compared their performance.

Chapter 6 concludes the thesis and discusses some possible research perspectives.

Chapter 2

Preliminaries

In this chapter, we introduce the main definitions and techniques on which this thesis is based. In Sections 2.1 and 2.2, we summarize the RDF data model and SPARQL that are the building blocks of the Semantic Web. Ontology-Based Data Access is presented in Section 2.3 while rule-based reasoning for data saturation is presented in Section 2.4.

2.1 RDF data model

RDF is a graph data model specified in a W3C recommendation [12], the last version of which is dated 2014. An RDF statement relates two nodes, representing respectively a subject and an object, by a directed edge corresponding to a property. Properties are identified by IRIs, while subjects and objects can be IRIs or blank nodes, and objects can also be literals. An IRI is an acronym for *Internationalized Resource Identifier* and it can provide built-in information about processing type (e.g. http protocol), the authority domain, the access path and, provided parameters. Instead, a literal is value which can be a string, boolean or a number, while a blank represents an unnamed node.

RDF uses namespaces to distinguish one set of IRIs from another, to avoid collissions, and to give them all unique identifiers. However, since namespaces can make IRIs very long, the use of a prefix is recommended instead. A prefix is the the compact representation of a namespace. For example, for a given namespace `http://www.sides-sante.fr/sides#` represented by the prefix `sides`, an IRI such as `http://www.sides-sante.fr/sides#student` can be shortened to `sides:student`. The absence of a prefix indicates a default namespace e.g `:student`.

Figure 2.1 shows an example of RDF graph. The RDF graph shows a representation of a student's answer to a question that has a result. Note that there is a typing edge `rdf:type` between the node `sides:stu34` and the node `sides:student`. The prefix `rdf:` refers to the namespace `http://www.w3.org/1999/02/22-rdf-syntax-ns#`. Note also that

nodes are represented by ellipses and rectangles. The ellipsis contains IRIs and the rectangle contains a literal, a string value.

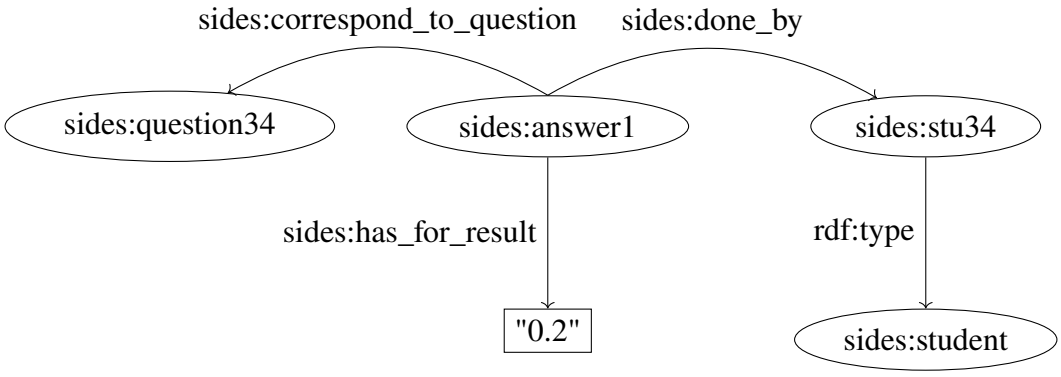

Figure 2.1 A RDF graph

2.1.1 RDF graphs

An RDF graph can be visualized as a diagram like in Figure 2.1 but it is formally defined as a collection of triples. For example, `(sides:answer1, sides:done_by, sides:stu34)` and `(sides:stu34, rdf:type, sides:student)` are two triples corresponding to two edges of the graph depicted in Figure 2.1. They state that `sides:answer1` is related by the property `sides:done_by` to `sides:stu34` that is an instance of the class `sides:student`.

Definition 2.1 (RDF graph)**.** Let I, L, and B be pairwise disjoint sets of IRIs, literals and blank nodes, respectively.

An RDF triple is of the form (s, p, o) where s is the subject, p the property and o the object and $s \in (I \cup B)$, $p \in I$, and $o \in (I \cup L \cup B)$.

An RDF graph is a set of RDF triples.

2.1.2 RDF named graphs and datasets

Named graphs and datasets [13] are a simple extension of the RDF data model allowing to associate IRIs to some RDF graphs. They are very useful for structuring RDF data within an RDF store, and can be exploited by SPARQL [14] to limit the scope of queries to subsets of triples.

Definition 2.2 (RDF named graph)**.** An RDF named graph is a pair consisting of an IRI (the graph name), and an RDF graph.

Named graphs can be represented as quads statements indicating for each triple the named graph it belongs to. In this thesis, we restrict the usage of quads to the representation and the storage of named graphs

Definition 2.3 (RDF quad)**.** A RDF quad (s, p, o, g) is a quadruplet where g is the IRI identifying a named graph, and (s, p, o) is an RDF triple in the RDF graph named by g.

Definition 2.4 (RDF datasets)**.** An RDF dataset is a set of RDF named graphs, and can then be represented as a set of RDF quads.

2.1.3 RDF Schema

RDF Schema (RDFS) is part of the RDF 1.1 specification [12]. It provides two namespaces `rdf:` and `rdfs:` with predefined properties to state relationships between instances, classes and properties:

- `rdf:type` is used to express that a resource identified by an IRI is an instance of a class (also identified by an IRI);
- `rdfs:subClassOf` is used to specify subsumption relationships between classes, i.e., that a class is a subclass of another.
- `rdfs:subPropertyOf` is used to denote that a property is a subproperty (specialization) of another
- `rdfs:domain` relates a property to a class to express that the subjects of the property are instances of the class.
- `rdfs:range` relates a property to a class to express that the objects of the property are instances of the class.
- `rdfs:label` associates a human-readable name to an IRI identifying an RDF resource.

Triples in which the property is an RDFS property `rdfs:subClassOf`, `rdfs:subPropertyOf`, `rdfs:domain` or `rdfs:range` are called *schema triples* (also called *RDFS triple*), as they express semantic constraints on the classes and properties used to describe RDF data of a given domain.

Within an RDF graph, we will distinguish schema triples from data ones. Though they are all denoted as RDF triples, the former ones convey a rule-based semantics and can be used to infer implicit triples from data triples.

For example, from the data triple `(sides:question34, rdf:type,sides:QMA)`, expressing that the IRI `sides:question34` is an instance of the class `sides:QMA` of questions with multiple answers, and the schema triple `(sides:QMA, rdfs:subClassOf, sides:question)` the implicit data triple `(sides:question34, rdf:type, sides:question)` can be inferred.

2.2 SPARQL queries

SPARQL is a query language based on matching graph patterns onto RDF graphs to retrieve data. Graph patterns extend RDF graphs by allowing variables in the subject, property or object positions. Matching is the operation consisting in replacing variables with URIs, blank nodes or literals in order to obtain a set of triples contained in the queried RDF graph. In this thesis, we consider queries covering the full expressive power of SPARQL 1.1 [14].

Definition 2.5 (SPARQL 1.1 graph pattern)**.** :
Given a set V of variables disjoint from $I \cup L \cup B$:

- A basic graph pattern BGP is a set of triple patterns $(s, p, o) \in (I \cup V) \times (I \cup V) \times (I \cup L \cup V)$.

- A SPARQL 1.1 graph pattern is an expression P generated from the following grammar:

$P ::= BGP \mid (BGP\ UNION\ BGP') \mid (BGP\ OPTIONAL\ BGP') \mid P\ FILTER\ R \mid P\ FILTER\ NOT\ EXISTS\ BGP \mid GRAPH\ g\ P$

where BGP and BGP' are basic graph pattern, $g \in V \cup I$ and R is a constraint expression over variables in P.

In this thesis, we consider SELECT and CONSTRUCT queries.

Definition 2.6 (SELECT queries)**.**
By $\overline{x}$ we denote a vector of variables.

- A simple SELECT query is of the form:

`SELECT` $\overline{x}$ `WHERE {` GP `}` where GP is a SPARQL 1.1 graph pattern including variables in $\overline{x}$. When evaluated over an RDF graph G (or a dataset UG) , there are as many answers $\mu(\bar{x})$ as mappings μ allowing to match GP with a subgraph of G (or of RDF graph in UG).

- An aggregate SELECT query is of the form

`SELECT` $\overline{x}$`,` $f(\overline{y})$ `WHERE {` GP `} GROUP BY` $\overline{x}$

where f is an aggregate function and GP a SPARQL 1.1 graph pattern including variables in $\overline{x} \cup \overline{y}$. When evaluated over G, there are as many groups as mappings allowing to match the tuple $\bar{x}$ with tuples of values $\bar{v}$ and as many answers $(\bar{v}, av)$ where av is computed by the aggregate function on the corresponding group.

- A nested SELECT query is a SELECT query for which the WHERE clause is of the form { GP { SQ } } where GP is a SPARQL 1.1 graph pattern and SQ is a (simple or aggregate) SELECT query. The inner SELECT query is called a subquery and is evaluated first. The subquery result variable(s) can then be used in the outer SELECT query.

Example 2.2.1. An example of an aggregate SELECT query is shown in Figure 2.2. The query asks for information about a specific student identified by the IRI sides:stu96154. The information requested is the number of graded responses to the questions associated with a specialty, and the average of the obtained results (scores) since 2015. The graph pattern of the query is shown between lines 3 and 10.

```
1  SELECT ?year (count(?answer) AS ?NumberOfAnswers)
          (AVG(?r) AS ?AverageResult))
3  {
     ?answer sides:done_by sides:stu96154.
5    ?answer sides:has_for_timestamp ?t.
     ?answer sides:has_for_result ?r.
7    ?answer sides:correspond_to_question ?q.
     ?q sides:is_linked_to_the_medical_speciality ?speciality.
9    FILTER (str(year(?t)) > "2014")
   }
11 GROUP BY (year(?t) as ?year
```

Figure 2.2 Example of SPARQL query

CONSTRUCT queries are SPARQL queries that enable ETL[1] data pipelines (to reduce large datasets to workable datasets), graph interoperability (to merge graphs from different sources) and are a key component in several W3C specifications (e.g., SPIN[2], and later SHACL[3]) for supporting rule-based inference.

Definition 2.7 (CONSTRUCT queries)**.** :
- A simple CONSTRUCT query is of the form:

CONSTRUCT { $Template$ } WHERE { GP [{ SQ }] }

where GP is a SPARQL 1.1 graph pattern, $Template$ is a basic graph pattern (possibly containing blank nodes) with variables appearing in GP, and SQ is an optional SELECT subquery.

[1]Extraction, Transformation, Load
[2]https://spinrdf.org/spin.html
[3]https://www.w3.org/TR/shacl-af/#rules

The result of the evaluation over an RDF graph G is the union of graphs obtained by instantiating the variables x in $Template$ with values $\mu(x)$ for each mapping μ satisfying the WHERE clause.

- A Quad CONSTRUCT query is of the form:

CONSTRUCT { $QuadTemplate$ } WHERE { GRAPH GN { GP [{ SQ }] }}

where GP is a SPARQL 1.1 graph pattern, GN is a graph name or a variable and $QuadTemplate$ is a quad pattern (i.e., a quad in which variables can appear in subject, property, object or named graph positions) and SQ is an optional SELECT subquery.

The result of the evaluation over a dataset is the set of quads obtained by instantiating the variables x in $QuadTemplate$ with values $\mu(x)$ for each mapping μ satisfying the WHERE clause.

Example 2.2.2. An example of a CONSTRUCT query is presented in the Figure 2.3. The query returns a single RDF graph where each triple contains information about the number of proposals of answer per question.

```
1  CONSTRUCT  {
   ?question sides:has_for_number_of_proposals ?np
3  }
   WHERE
5  {
     SELECT ?question (COUNT (?p) As ?np)  {
7    ?question sides:has_for_proposal_of_answer ?p
     }
9    GROUP BY ?question
   }
11
```

Figure 2.3 Example of CONSTRUCT query

Example 2.2.3. Figure 2.4 exhibits an example of a Quad CONSTRUCT query. It is similar to the query shown in Figure 2.3 except that it returns quads associating the number of proposals of answer per question to the *named graph* which the question belongs to. The quad pattern is shown in line 2.

```
CONSTRUCT {
  ?question sides:has_for_number_of_proposals ?np ?g
}
WHERE {
  SELECT ?g ?question (COUNT (?p) As ?np){
    GRAPH ?g {?question sides:has_for_proposal_of_answer?p}
  }
  GROUP BY ?g ?question
}
```

Figure 2.4 Example of Quad CONSTRUCT query

Definition 2.8 (SELECT query induced by a CONSTRUCT query)**.**
Given a CONSTRUCT query

CONSTRUCT { $Template$ } WHERE { GP [{ SQ }] }

Its induced SELECT query is:

SELECT $\bar{x}$ WHERE { GP [{ SQ }] }

where $\bar{x}$ is made of all the variables in the graph pattern $Template$.

Example 2.2.4. Figure 2.5 shows the SELECT query induced by the CONSTRUCT query shown in Figure 2.3.

```
SELECT ?question (COUNT (?p) As ?np)  {
?question sides:has_for_proposal_of_answer ?p
}
GROUP BY ?question
```

Figure 2.5 SELECT query induced by the query of Figure 2.3

The computation of the result of a CONSTRUCT query can be decomposed into the evaluation of its induced SELECT query followed by the construction of a RDF graph as the union of the template instances obtained by replacing each variable by its corresponding value in the answer set of the SELECT query.

The computation of a Quad CONSTRUCT query is performed in a very similar way except that the evaluation of its induced SELECT query is followed by the construction of a RDF dataset.

2.3 Ontology-Based Data Access (OBDA)

An OBDA system provides a single access point for query answering across multiples datasources with different schemas. Overall, an OBDA system consists of three components:

an ontology, a set of mappings and the datasource(s). The ontology provides the user with a high-level domain representation of the data to build the input query. The high level representation abstracts away the complex intricacies of the data source(s) model.

The mappings are a declarative way to define a correspondence between an RDF template and a datasource query. The query output provides the bindings to instantiate the corresponding placeholders in the RDF template.

2.3.1 Ontologies

Ontologies are used to provide a unified and abstract view to the user of the specific and low-level terms defining schemas of (possibly heterogeneous) data sources. They allow users to formulate their queries using a high-level vocabulary that is more familiar to them because related to their domain of expertise and not to the names chosen by database managers for the tables used to store the data.

The W3C has standardized several ontology languages. RDFS is the simplest language for describing ontological statements on top of RDF data, which has the advantage to be fully conform to RDF and SPARQL. At the other end of the spectrum, the full ontology language OWL [15] is too expressive to be exploited in OBDA systems. To address this issue, the W3C introduced distinguished tractable subsets of OWL, called profiles. The OWL2 QL profile [16] is based upon the DL-Lite family [17] and was designed specifically for OBDA systems.

Ontological statements can also be expressed as positive first-order rules of the form $body \Rightarrow head$ where the $body$ (also called the $conditions$ or the $premisses$ of the rule) and the $head$ (also called the $conclusion$ of the rule) are conjunctions of atoms (without functions). Datalog [18] is a popular rule-based language in the database community in which the rules are *safe*, i.e., all variables in the head occur in the rule body. Recently, an extension of Datalog, called Datalog+/− [19], has been considered in the knowledge representation community to express *existential rules* in which some variables in the head may be existential variables that do not occur in the rule body.

RDFS and OWL2 QL can both be captured by Datalog and a decidable fragment of Datalog+/− respectively. In practice, Datalog and Datalog+/− rules can be encoded as SPIN rules [20] in which the body is a basic graph pattern.

For example, the following Datalog rule expressing that any answer with 0 discordance w.r.t the expected answers (among the multiple answer options of a question) is graded 1.

`R: hasForNumberOfDiscordance(?answer, 0)` $\Rightarrow$ `hasForResult(?answer, 1)`

can be encoded by the following CONSTRUCT query in which the binary relations `hasForNumberOfDiscordance` and `hasForResult` are encoded by the RDF properties `sides:has_for_number_of_discordance` and `sides:has_for_result`.

```
Q: CONSTRUCT { ?answer sides:has_for_result "1" }
   WHERE {?answer sides:has_for_number_of_discordance "0"}
```

In fact, general SPIN rules extend the expressive power of Datalog+/− rules by allowing RDF 1.1 graph patterns (defined in Definition 2.5) in the body.

In this thesis, we consider ontologies made of RDFS statements enriched with SPIN rules implemented as CONSTRUCT SPARQL queries, which became the main reference for recent W3C specifications on SPARQL-based rules.

More precisely the SPIN rules that we consider do not have blank nodes in their template, and thus can be seen as extending Datalog by allowing RDF 1.1 graph patterns in their body.

Figure 2.3 is an example of such a SPIN rule.

Figure 2.6 shows an example of an RDFS ontology in the form of an RDF graph the nodes of which are classes or properties and the edges are RDFS predefined properties presented in Section 2.1.3.

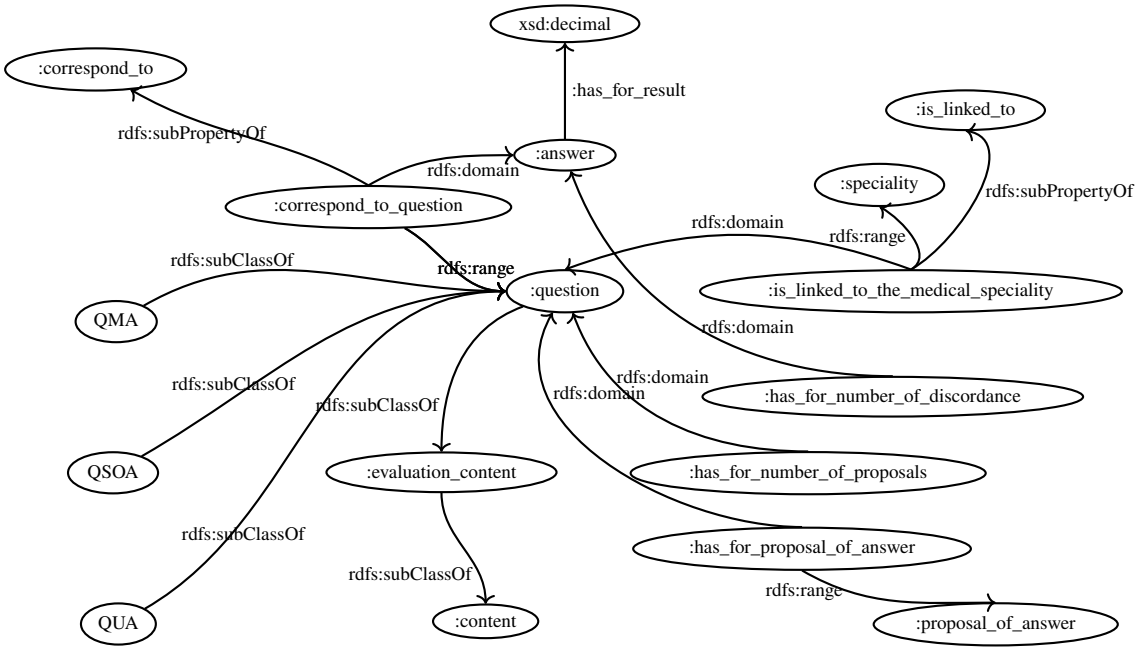

Figure 2.6 An RDFS ontology.

2.3.2 OBDA system

Definition 2.9 (OBDA specification [4])**.** An OBDA specification $\mathcal{P}$ has the form of $\mathcal{P} = (\mathcal{S}, \mathcal{M}, \mathcal{O})$ where $\mathcal{S}$ is the source schema of a source database $\mathcal{D}$, $\mathcal{O}$ is an ontology, and $\mathcal{M}$, a set of mappings $m : \sigma(\vec{x}) \rightsquigarrow \psi(\vec{x})$ from $\mathcal{S}$ to $\mathcal{O}$.

[4]This presentation mainly follows the definition presented in [21]

The set of mappings $\mathcal{M}$ in $\mathcal{P}$ describe how the ontology properties are populated with data from the source $\mathcal{D}$. In a mapping m: $\sigma(\vec{x}) \rightsquigarrow \psi(\vec{x})$, $\sigma(\vec{x})$ is an FOL query over the source schema with output variables $\vec{x}$, and $\psi(\vec{x})$ is a conjunction of RDF triple templates whose only variables are those in $\psi(\vec{x})$.

The pair $(\mathcal{P}, \mathcal{D})$ of an OBDA specification $\mathcal{P}$ and a source database is called an *OBDA system*.

Several languages have been proposed to express mappings. R2RML [22] is a W3C recommendation to express in RDF customized mappings from relational databases to RDF datasets. In our thesis, we have chosen to use Ontop [23] as the OBDA framework software to implement our OBDA system.

Example 2.3.1. The abstract and concrete syntax version of an Ontop mapping are shown in Figure 2.7 and Figure 2.8 respectively. The abstract mapping relates an SQL query (in the left hand side of $\rightsquigarrow$) to a simple RDF template (in the right hand side of $\rightsquigarrow$) on the property `sides:has_for_proposal_of_answer` (part of the ontology shown in the Figure 2.6) that links a question to its answer options (also called its proposals of answers). The concrete syntax version is the machine-readable specification of the mapping to be processed.

```
select c.question_id as question_id,
   c.id as choice_id from public.
   choice c
⇝ sides:q{question_id} sides:
   has_for_proposal_of_answer
sides:prop{choice_id} .
```

Figure 2.7 An abstract Ontop mapping syntax

```
target   sides:q{question_id} sides:
   has_for_proposal_of_answer
sides:prop{choice_id} .
source   select c.question_id as
   question_id, c.id as choice_id
   from public.choice c
```

Figure 2.8 A concrete Ontop mapping syntax

This mapping can be used to populate the target RDF property by evaluating the SQL query over the source database, and by replacing the two placeholders in the RDF template by values returned as answers to the SQL query.

For instance, if the following table is in the source database:

question_id	choice_id
7689	4343
7690	8765

The output of the mapping will be the following set of RDF triples:

```
sides:q7689 sides:has_for_proposal_of_answer sides:prop4343 .
```

```
sides:q7690 sides:has_for_proposal_of_answer sides:prop8765 .
```

Virtual RDF graph versus Materialization approach

Query answering in OBDA systems can be based on Virtual RDF graph or Materialization or both (hybrid).

In Virtual RDF graph, query answering is always performed over the original datasource through a SPARQL/SQL query rewriting based on a set of mappings. The query rewriting keeps the RDF graph virtual because RDF data does not need to be materialized and stored in a triplestore. The Virtual RDF graph is recommended when there is a shortage of memory or disk space. The main drawback is that the query rewriting can be slow in the cases where the query rewriting is complex and difficult to optimize.

Materialization consists in the computing of the output of all mappings and storing it in a triplestore for query answering. Materialization is a valid option when there are no limitations on disk space or memory and fast query response is required. The main drawback is that each time the mappings are changed to reflect changes in the ontology or data schema, the computation of all the mappings must be performed, which is a time-consuming and resource-intensive task.

The hybrid approach is a system in which both approaches complement each other. For example, the same OBDA system that serves to make real-time queries to the data can also be used to generate periodic materializations of the data to perform data analytics tasks.

Ontop

Ontop [23] is an hybrid OBDA framework software with solid theoretical foundations. It supports RDF 1.1, OWL 2 QL/RDFS ontologies and SPARQL 1.1 . A large part of Ontop's success is due to its modular architecture. Such an architecture allows its open source community to extend its support as soon as a new database emerge.

Regarding the mappings, Ontop provides both a simple mapping language and R2RML, the W3C mapping specification. The simple mapping language is an alternative for those users who find it difficult to write mappings directly in R2RML. However, it also provides a translator to convert mappings in Ontop format to R2RML format to enable interoperability.

Ontop's main limitations lie in its query rewriting capabilities, rather than in its mapping materialization. Specifically, there are three primary issues with query rewriting: a) Ontop's translation of complex SPARQL queries into SQL queries can result in performance degradation, b) Ontop doesn't fully support SPARQL queries with RDFS entailments based on property paths, and c) Ontop doesn't fully support negation queries using the FILTER NOT EXISTS operator. As for mapping materialization, Ontop currently lacks the ability to efficiently stream the output of SQL queries for large amounts of data.

In this thesis, we chose to use OBDA for RDF data materialization instead of Virtual RDF graph to have full SPARQL 1.1 support. We built a custom version of Ontop 3.x

where we fixed the following: a) output streaming during mapping materialization and b) some unsupported RDF 1.1 features (e.g `rdf:Seq`) that were not available at the time of starting this thesis.

2.4 Rule-based reasoning

Rule-based reasoning is the process of inferring new knowledge from existing facts (encoded as RDF triples) and rules. Given a set of rules (and possibly a query), there are two main approaches for implementing reasoning.

- Forward reasoning is a saturation process that applies the rules from their conditions to their conclusion to infer facts: at each iteration, each rule whose conditions can be matched to existing (input or previously inferred) facts allows to infer new facts obtained by replacing each variable in the conclusion by the constant to which it is mapped in the matching. The iterative process continues until a fixpoint is reached (i.e., no new fact is inferred).
- Backward reasoning takes as input a target goal to prove (e.g., an atomic query) and consists in applying the rules from their conclusion to their conditions to rewrite each goal into sub-goals until producing sub-goals that can be matched to facts or no rewriting is possible (leading to a failure of reasoning to prove the input goal). Each rewriting step consists in finding a rule such that its conclusion can be matched to a current goal and to replace this goal by sub-goals obtained from each condition of the rule in which the variables in common with the conclusion are replaced by the constants to which they are mapped in the matching.

In this thesis, we consider two types of rules: RDFS entailment rules, which capture the semantics of RDFS ontological statements to infer implicit RDF (data or schema) triples, and domain-specific SPIN rules expressed as CONSTRUCT queries (defined in Section 2.2).

2.4.1 Reasoning on RDFS entailment rules

RDFS entailment rules [24] are a collection of 13 predefined rules to perform logical inference over a RDF graph until a fixpoint is reached.

For real-world datasets, the number of iterations for reaching the fixpoint can be time-consuming and computationally expensive. Different optimizations have been considered to limit the number of iterations such as ordering the execution of rules as shown in [25], or splitting the set of rules into subsets like in [26].

Following [26], we consider RDFS ontologies satisfying the first-order restriction, i.e., in which the predefined RDFS properties do not occur as subjects or objects in RDFS statements. In such cases, the forward reasoning process can be divided in two steps of saturation that can be done in any order. One step of saturation considers only the subset of RDFS rules in Table 2.1, that are called *assertion rules* in [26].

RDFS rule	body $\Rightarrow$ head
`rdfs2`	p rdfs:domain c , $s\ p\ o \Rightarrow s$ rdf:type c
`rdfs3`	p rdfs:range c , $s\ p\ o \Rightarrow o$ rdf:type c
`rdfs7`	p rdfs:subPropertyOf q , $s\ p\ o \Rightarrow s\ q\ o$
`rdfs9`	c rdfs:subClassOf d , s rdf:type $c \Rightarrow s$ rdf:type d

Table 2.1 RDFS assertion rules

The other step of saturation considers the subset of RDFS rules in Table 2.2, that are called *constraint rules* in [26].

RDFS rule	body $\Rightarrow$ head
`rdfs5`	p_1 rdfs:subPropertyOf p_2 , p_2 rdfs:subPropertyOf p_3 $\Rightarrow p_1$ rdfs:subPropertyOf p_3
`rdfs11`	c_1 rdfs:subClassOf c_2 , c_2 rdfs:subClassOf $c_3 \Rightarrow c_1$ rdfs:subClassOf c_3
`ext1`	p rdfs:domain c_1 , c_1 rdfs:subClassOf $c_2 \Rightarrow p$ rdfs:domain c_2
`ext2`	p rdfs:range c_1 , c_1 rdfs:subClassOf $c_2 \Rightarrow p$ rdfs:range c_2
`ext3`	p rdfs:subPropertyOf p_1 , p_1 rdfs:domain $c \Rightarrow p$ rdfs:domain c
`ext4`	p rdfs:subPropertyOf p_1 , p_1 rdfs:range $c \Rightarrow p$ rdfs:range$c$

Table 2.2 RDFS constraint rules

Backward reasoning can be applied for query rewriting but some restrictions must be considered for guaranteeing termination. In this thesis, the queries that we consider correspond to the BGPQ-CQ fragment defined in [26], i.e., queries in which no variable appears in a property or class position. In such cases, backward reasoning can be applied only on the set of assertion rules to obtain the full set of rewritings.

2.4.2 Reasoning on SPIN rules

As indicated in Section 2.3.1, the SPIN rules that we consider are CONSTRUCT queries that do not have blank nodes in their template (i.e., they are not existential rules), but that can have RDF 1.1 graph patterns in their body (i.e., they are more expressive than Datalog rules). For this reason, standard backward reasoning cannot be applied.

Therefore, we focus on forward reasoning for rules encoded as CONSTRUCT queries.

A forward-chaining reasoner can be implemented on top of any RDF triplestore by iterating the triggering of the CONSTRUCT queries and the corresponding update of the RDF graph/dataset until no new triple/quad is added. The termination is guaranteed when the rules are safe, i.e., when no blank nodes appear in the template of the corresponding CONSTRUCT queries.

In this thesis, we will consider a set of CONSTRUCT queries encoding a set of *non recursive* rules: a set of rules is non recursive if and only if its *dependency graph* is acyclic.

The *dependency graph* of a set of rules (possibly encoded as CONSTRUCT queries) is defined as follows:

- The nodes are the identifiers of the rules (or the identifiers of CONSTRUCT queries encoding the rules)
- There is an edge from a node Q to a node Q' if one relation in the head of the rule (i.e., one property in the output of Q) appears in the body of Q'

When the dependency graph is acyclic, the saturation of an input dataset by a forward-chaining reasoning can be computed by applying each CONSTRUCT query only once, by following a *topological ordering* of the dependency graph. A topological ordering is a graph traversal in which each node is visited only after all its dependencies are visited. It allows to take into account that the evaluation of some queries must occur after the update of the dataset by the evaluation of other queries.

Figure 2.9 shows such a forward chaining algorithm that takes as input an RDF dataset and a set CONSTRUCT queries corresponding to non recursive rules, and compute the saturation of the dataset.

Algorithm 1 CONSTRUCT-based forward chaining

Input: Dataset D and n (Quad) CONSTRUCT queries $\{q_1, ..., q_n\}$ ordered by a topological order

Output: Saturated dataset $D^{'}$

$D^{'} \leftarrow D$

for $j \leftarrow 1$ *to* n **do**

  $output \leftarrow Sparql(q_n, D^{'})$

  $D^{'} \leftarrow D^{'} \cup output$ `// update operation`

end

Figure 2.9 CONSTRUCT-based forward chaining algorithm

Based on the dependency graph, the CONSTRUCT queries can also be grouped into *layers* of increasing depth. Since, there is no dependency between queries of a given layer, they can be evaluated independently. This structuration of the queries in layers can thus be exploited to implement a parallel forward-chaining reasoning algorithm that handles sequentially the layers (by ascending order of the depth) and evaluates in parallel the queries of each layer.

2.4.3 Combining forward and backward reasoning

In our thesis, we follow an hybrid reasoning approach made possible by the fact that the two types of rules that we consider (RDFS rules and domain-specific SPIN rules) do not interfere.

We use the forward reasoning algorithm given Figure 2.9 for saturating the datasets by a set of domain-specific SPIN rules (that will be given in Chapter 2).

We encapsulate a restricted backward reasoning on RDFS rules for rewriting queries by exploiting the possibility to express property paths in SPARQL 1.1 .

For each query Q we rewrite it in a query RQ obtained from Q:

- by replacing each triple pattern ?i rdf:type c with ?i rdf:type/rdfs:subClassOf* c (if c is not a leaf class)
- by replacing each triple pattern ?s prop ?o with "?s ?p ?o. ?p rdfs:subPropertyOf* prop" (if prop is not a leaf property)

This rewriting corresponds to the backward chaining exploiting the two assertion rules rdfs7 and rdfs9 of Table 2.1 only.

In theory, such a rewriting process is incomplete since it does not exploit the assertion rules rdfs2 and rdfs3 (see Table 2.1).

In practice, exploiting these rules is useless for RDF ontologies and datasets built following a methodololy ensuring that the following property is satisfied: all the instances are typed such that if there exists triples s prop o, s type c, o type d, prop domain c', prop range d', there exist a subclass path between c and c' and a subclass path between d and d'.

When this property is satisfied, the rules rdfs2 and rdfs3 are redundant with the rule rdfs7.

2.5 Summary

In this chapter, we introduced the core definitions that underlie this thesis. We started by introducing the basic components of the Semantic Web: RDF, RDFS, and SPARQL, along with the definitions of RDF quad, RDF datasets, and quad CONSTRUCT query, which will be used extensively in this thesis. Then, we explained the type of ontologies and mappings we used in our OBDA system. Finally, we presented a hybrid approach for rule-based reasoning based on a combination of forward and backward chaining.

Chapter 3

OBDA architecture and data modularization in OntoSIDES

In this chapter, we describe the OBDA methodology that we have followed for the construction of the OntoSIDES RDF knowledge base built on top of relational database SIDES. SIDES is the operational database of the national e-learning platform used since 2013 by all the French medical schools for student self-assessment and graduation.

For the semi-automatic construction of OntoSIDES we have chosen an expert-based methodology for building manually a lightweight domain ontology enriched with rules, and an OBDA materialization approach for the automatic population of the ontology using mappings.

The materialization step consists in applying the mappings to a particular dump of the SIDES database. During the duration of this thesis, in order to take into account the increasing of the SIDES database over time, we have performed 8 times the materialization step on dumps of increasing size of the SIDES database.

The first SIDES dump that we considered corresponded to the activities over a period of 3 months (May, June and July 2015) of the students registered in Grenoble medical school. The resulting mapping-based OBDA materialization was an RDF graph containing 5,248,288 RDF triples. The result of the materialization computed from the last dump is a huge RDF graph containing approximately 12 billion triples describing training and assessment activities performed by more than 145,000 students over nearly 6 years.

Although OBDA is mainly used to materialize a single RDF graph, it can also be used to scale the materialization of a collection of RDF graphs. This is possible by mapping RDF quad templates to queries in the datasources. In this way, OBDA can be extended to structure a big RDF graph into modules to reduce complexity of data analytics, facilitate maintenance and enable the reuse of components across different datasets and applications.

Section 3.1 describes the OBDA-based construction of successive versions of Onto-

SIDES, while Section 3.2 describes the modularization approach that we have followed for structuring OntoSIDES into modules.

3.1 The OBDA architecture of OntoSIDES

All the versions of OntoSIDES are RDF knowledge bases comprised of the same lightweight *domain ontology* that serves as a pivot high-level vocabulary of the query interface with users, and of an *RDF graph* made of RDF triples relating individual entities to classes and properties of the ontology that are extracted using a same set of *mappings* applied to given dump of the SIDES database.

Figure 3.1 summarizes the OBDA architecture of OntoSIDES.

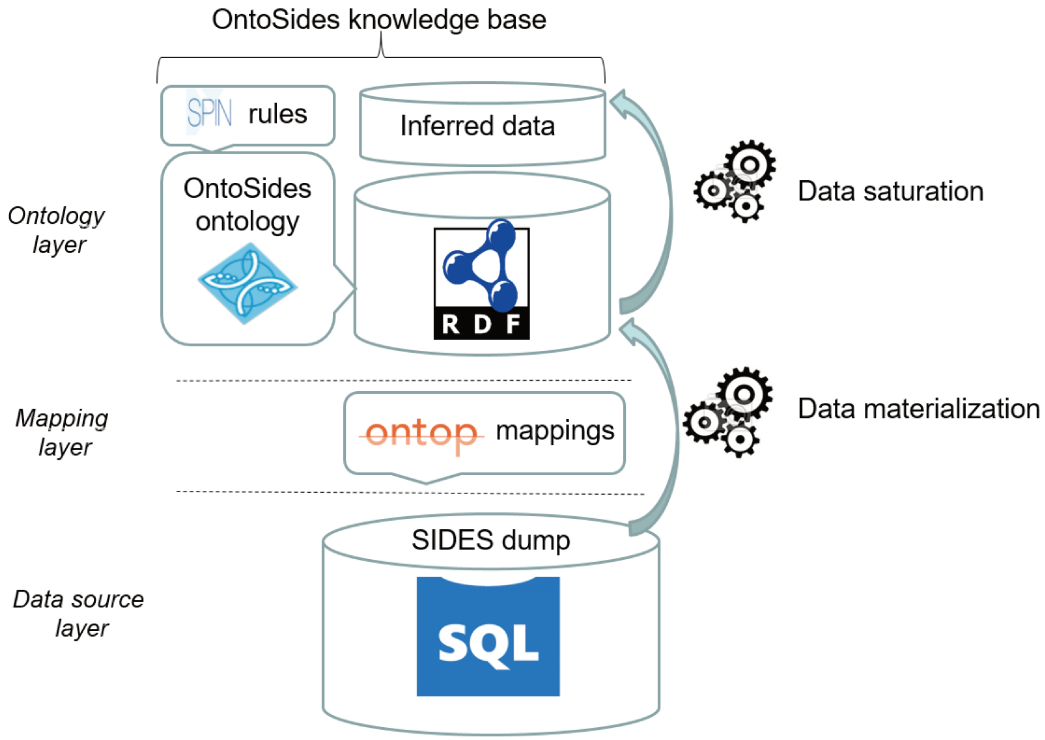

Figure 3.1 OBDA architecture of OntoSIDES

The *data source layer* corresponds to a dump of the SIDES relational database, possibly enriched with materialized views, as described in Section 3.1.2. The *mapping layer* is made of a set of mapping axioms (expressed using the Ontop syntax) that indicate how to transform relational data stored in SIDES into RDF data conform to the target RDFS schema specified in the ontology layer. The *ontology layer* is made of a set of RDFS

statements and of a set of rules (expressed using the SPIN syntax) for defining classes and properties that are meaningful for the target end-users. Within the OntoSIDES knowledge base, we distinguish the OntoSIDES ontology and rules from the OntoSIDES data that is obtained by two distinct processes: *data materialization*, which is the process of exploiting the mappings in order to generate the RDF data corresponding to the instances of the classes and the properties defined in the OntoSIDES ontology ; and *data saturation* which, given the materialized data stored in a triplestore, is the process of making explicit the RDF data that can be logically derived from the SPIN rules.

After summarizing in Section 3.1.1 the OntoSIDES ontology, we describe the materialization process and the saturation process in Section 3.1.2 and Section 3.1.2 respectively.

3.1.1 The OntoSIDES ontology

We have manually built the OntoSIDES ontology [27] with the help of the national coordinator for e-learning in Medicine (Prof. Olivier Palombi[1]), with the support of the TopBraid Composer software suite [28] for editing both the RDFS ontology and the domain-specific rules as SPIN rules.

Such an expert-based construction of the OntoSIDES ontology has been facilitated by the unique position of Prof. Olivier Palombi as a domain expert who has *(i)* a recognized and shared expertise on the organization and requirements of medical studies in France, *(ii)* a detailed technical knowledge on the SIDES database, and *(iii)* a previous experience in building ontologies in the related domain of anatomy [29, 30].

3.1.1.1 The RDFS ontology

The resulting RDFS ontology consists of a taxonomy of 93 classes and a set of 75 properties among which 6 are defined by rules.

Figure 3.2 shows an extract of the OntoSIDES class taxonomy (left) and of properties (right) as displayed by the TopBraid editor.

[1]https://www.univ-grenoble-alpes.fr/universite/organisation/la-gouvernance/la-presidence/olivier-palombi-838094.kjsp

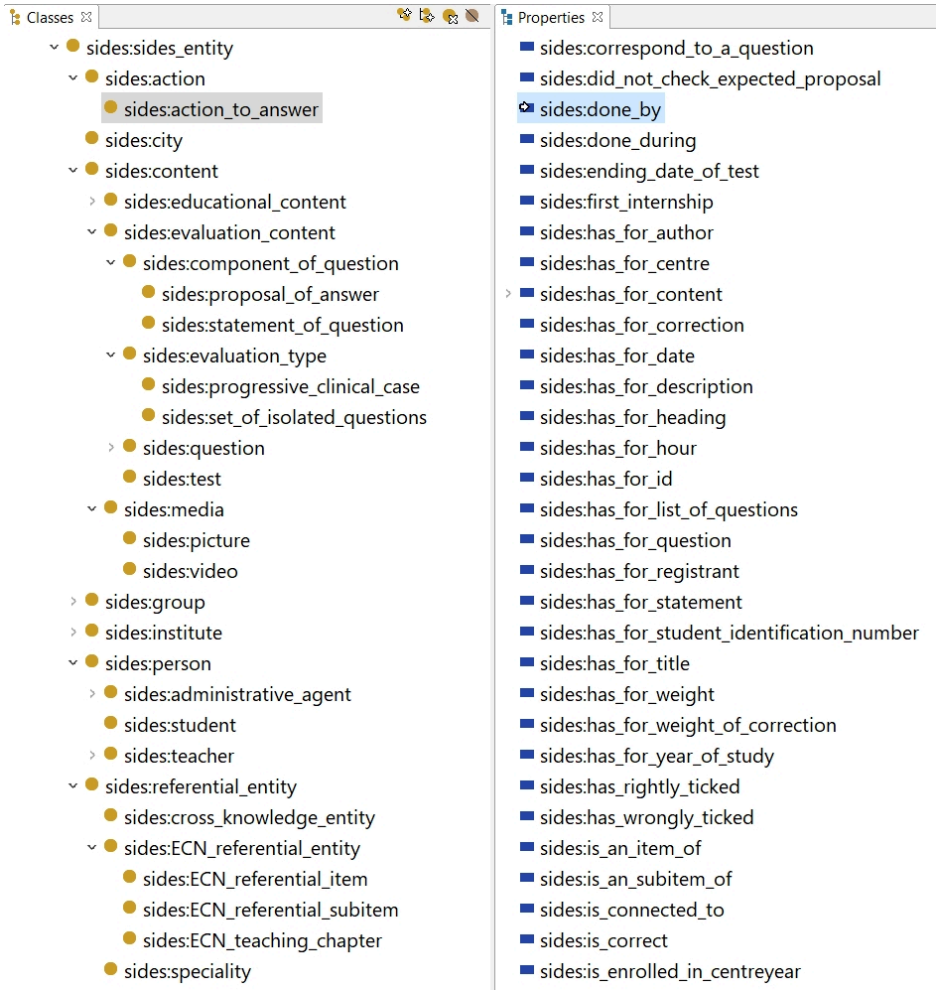

Figure 3.2 Extract of the OntoSIDES ontology visualized in TopBraid

The class taxonomy is built by successive refinements of *7 main classes* which respectively denote:

- the set of possible actions of students when using the SIDES pedagogical online resources (denoted by the class `sides:action` in Figure 3.2),
- the types of pedagogical resources (training or evaluation) available in the SIDES platform (denoted by the class `sides:content` in Figure 3.2),
- the set of reference items of the French educational program in Medicine (denoted by the class `sides:referential_entity` in Figure 3.2), published by the French Ministry of Higher Education in *Bulletin Officiel* [31] and also used in SIDES as metadata,

- the sets of French cities, universities and medical schools (denoted by the classes `sides:city` and `sides:institute` in Figure 3.2) for which there are as many local SIDES platforms that are indexed in the central SIDES database,
- the set of milestones (years, periods of practical internships) to register and validate by students to get their diploma (denoted by the class `sides:group` in Figure 3.2), that are encoded in the SIDES database by specific identifiers,
- the categories of persons (students, academic staff, administrative staff) involved in medical studies (denoted by the class `sides:person` in Figure 3.2), that correspond to specific roles of users in the SIDES database.

The declaration of properties and their signature (using the pre-defined `rdfs:domain` and `rdfs:range`) completes the RDFS ontology by establishing how the instances of different classes can be related or described. For instance, the following RDF statements

```
sides:done_by rdfs:domain sides:action.
sides:done_by rdfs:range sides:student.
```

express that the property `sides:done_by` serves to relate (identifiers of) actions extracted from SIDES log traces with (the identifiers of) the students who performed these specific actions. Some temporal properties such as `sides:starting_date_of_test` and `sides:ending_date_of_test` serve to associate the starting / ending time and date to the training tests taken by students. Another property, `sides:has_for_list_of_questions`, relates an instance of an evaluation to an instance of `rdf:Seq`, an ordered list of questions. The position of each question in such a list of questions is denoted by a numerical ordering property `rdf:_nnn` where `_nnn` is an integer that is given explicitly.

We also use the ontology visualization tool VOWL [32] to display the ontology to expert users. Figure 3.3 shows an extract of the visualization of the OntoSIDES ontology produced by VOWL.

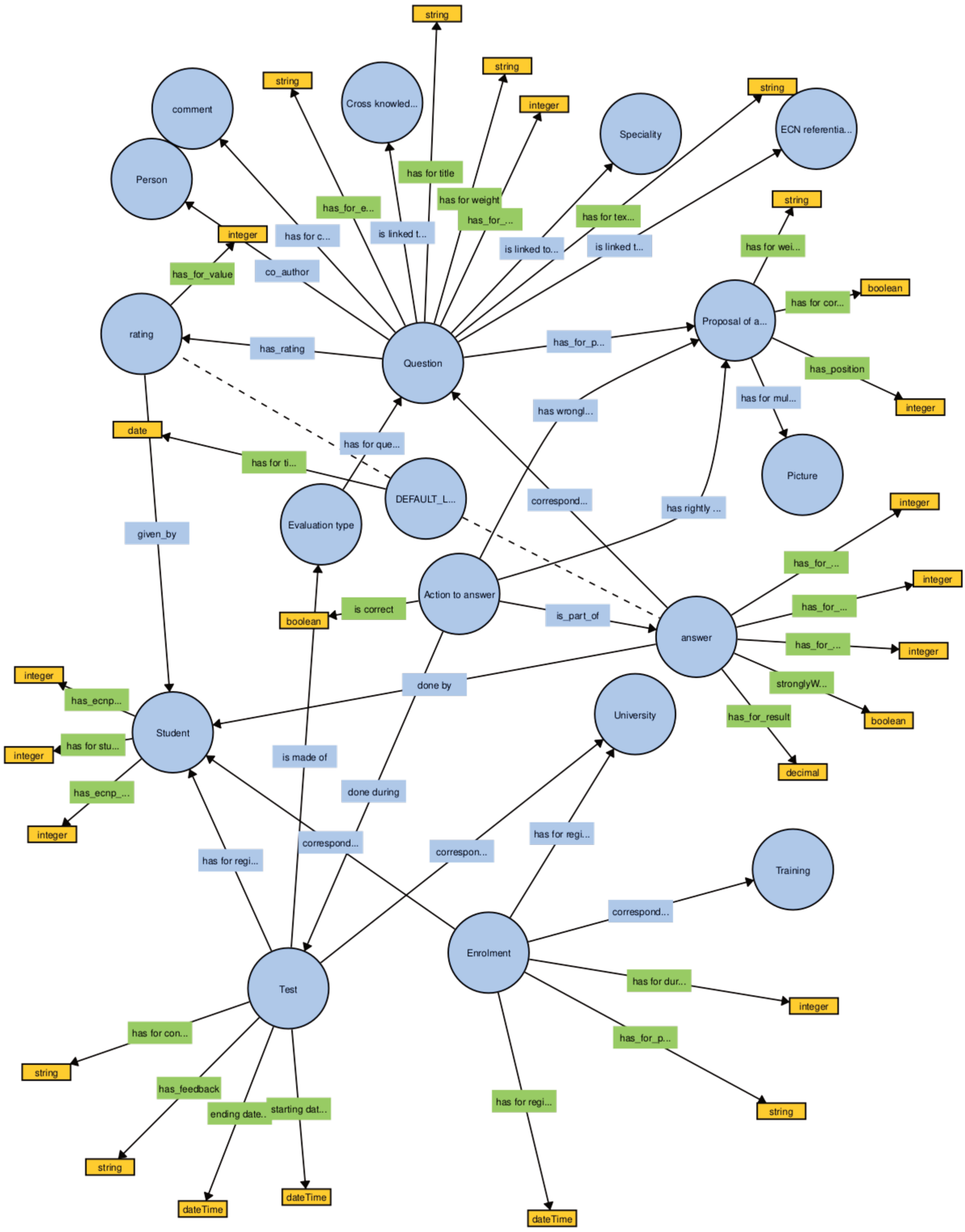

Figure 3.3 Extract of the VOWL visualization of the full OntoSIDES ontology

3.1.1.2 SPIN Rules for defining some properties

With the help of the expert, we wrote 18 SPIN rules to define 6 properties (specified in the RDFS ontology) in function of other properties. They are shown in Figure 3.4.

The property `sides:has_for_result` is the target property that makes explicit the calculation rules of the students'scores to multiple choice questions. The scores are computed based on the recorded (wrong of right) clicking actions done by the students for ticking the answer options within each question.

The scoring techniques for multiple choice questions can be complex and varying depending on the intended pedagogical goal (quantifying a level of knowledge for assessing individual students or for ranking a group of students). They can take into account *differently* several criteria such as the type of the questions (with a unique answer option to be ticked versus the possibility for students to tick several options) the number of answer options for each question, the number of correct (respectively false) answer options chosen by the student, but also the number of correct answer options not chosen by the student, and whether the choice of a false (respectively correct) answer option is eliminatory (respectively mandatory).

Following a rule-based declarative approach to make explicit the score calculation rules is useful because it enables to easily change them and to adjust them based on measuring the impact of such changes for example on minimizing the ties in the resulting ranking.

The student's scoring in OntoSides is defined by a set of 11 different rules (expressed by the CONSTRUCT queries Q8 to Q18 in Figure 3.4) to infer values of the property `sides:has_for_result` for each student's answer to a question. the scores of students' answers to questions.

In contrast with simple scoring techniques, it is not restricted to the computation of the ratio of correct options ticked by the student but it is based on the *number of discordances* that is the sum of the *number of ticked options that are not correct* and of the *number of correct options that have not been ticked*. Zero discordance leads to the maximal score of 1 for the question. On the other hand, *not ticking* a correct option marked as mandatory and ticking a wrong option marked as eliminatory leads to a score equal to 0. In the other cases, the score (between 0 and 1) depends on the number of answer options of the question and on the number of discordances of the student's answer to this question.

For taking into account this complex scoring method, the rule-based definition of the property `sides:has_for_result` relies on the definition of the following intermediary properties:

1. The property `sides:has_for_number_of_proposals` is defined by a single rule expressed by the CONSTRUCT query Q1, which is an aggregate query that counts the number of answer options per question (i.e., the number of values of the materialized

property `sides:has_for_number_of_proposals` grouped by question) and relates each question to this number.

2. The property `sides:has_for_number_of_wrong_tick` relates each student's answer corresponding to a given question to the number of ticked options that are not correct (if this case happens). It is defined by a single rule expressed by the CONSTRUCT query Q2, which is an aggregate query that counts the number of wrong clicking actions within each answer of a student to a question (i.e., the number of values grouped by answer of the materialized property `sides:has_wrongly_ticked` relating a clicking action to a false option).

3. The property `sides:has_for_number_of_missed_right_tick` relates each student's answer corresponding to a given question to the number of correct answer options of the question that has not been ticked (if this case happens). This property is also defined by a single aggregate CONSTRUCT query (namely Q3). However, this query is more complex than the two previous counting queries because it contains a sub-query counting for each answer of a student to a question the number of correct options of this question for which *there does not exist* a right clicking action recorded for this student's answer.

4. The property `sides:has_for_number_of_discordance` relates each student's answer corresponding to a given question to its number of discordance. It is defined by 4 rules expressed by the CONSTRUCT queries Q4, Q5, Q6 and Q7. The query Q4 handles the case where the number of discordances is equal to 0 because there is no wrong tick and no missing right tick. This requires to express negation by absence of the two properties `sides:has_for_number_of_wrong_tick` and `sides:has_for_number_of_missed_right_tick` using the FILTER NOT EXISTS constructor of SPARQL. Since these two properties are defined by rules, the corresponding CONSTRUCT queries Q2 and Q3 must be evaluated beforehand and the dataset must be updated by their results before evaluating the query Q4, and also Q5, Q6 and Q7 that relies on the computation of these two properties too: Q5, for computing the number of discordances of answers for which there exists both wrong ticks and missed right ticks ; Q6 (respectively Q7) for computing the number of discordances of answers for which there is no missed right tick (respectively no wrong tick) but there exists wrong ticks (respectively missed right ticks) the number of which has been computed by the dataset updating resulting from the evaluation of the CONSTRUCT query Q2 (respectively Q3).

5. The property `sides:stronglyWrong` is a boolean property defined by the 3 CONSTRUCT queries Q9, Q10 and Q11 to specify cases where students' answers must be scored to 0 either because an eliminatory wrong option has been ticked by the

student (Q9), or a mandatory correct option has not been ticked by the student (Q10), or the number of discordances is strictly greater than 0 and the question is of type `sides:QUA` (i.e., such that only one answer option has to be ticked by students). These 3 queries have the distinguishing feature that the result is not restricted to a triple pattern but is a graph pattern of size 2 to infer simultaneously *true* as value for property `sides:stronglyWrong` and 0 as value of the property `sides:has_for_result` for answers recognized as strongly wrong. Among these queries, Q9 is a simple conjunctive query, while Q10 contains a FILTER NOT EXISTS constructor to check negation by absence and Q11 contains a FILTER constructor.

Finally, in addition of the 3 queries Q9, Q10 and Q11 mentioned above, the property `sides:has_for_result`, that relates each student's answer (corresponding to a given question) to its score, is defined by 8 other CONSTRUCT queries of varied complexity to cover the remaining different cases depending on the number of discordances and the number of answer options in the question:

- The CONSTRUCT query Q8 is a very simple query with a single triple pattern in its body and in its output to specify that for answers with no discordance the score is equal to 1. Its evaluation requires that the number of discordances has been inferred beforehand and added to the dataset (as the result of the evaluation of the CONSTRUCT queries Q4, Q5, Q6 and Q7).

- The CONSTRUCT queries Q12, Q13, Q14, Q15 and Q17 have in common a FILTER NOT EXISTS condition to handle the cases in which the answers have not been recognized as strongly wrong (thus requiring the previous evaluation of the queries Q9, Q10 and Q11). Each of them assigns a score value (between 0 and 1) depending on the number of discordances and the number of options in the corresponding question. Q12 and Q13 handle the frequent case of questions with 5 answer options (only one discordance leads to a score of 0.5 ; 2 discordances lead to a score of 0.2) while Q14 and Q15 handle the case of questions with 4 answer options (only one discordance leads to a score of 0.425 ; 2 discordances lead to a score of 0.1) and Q17 handles the case of questions with 3 options (only one discordance leads to a score of 0.3).

- The CONSTRUCT queries Q16 and Q18 assign 0 as score for answers with a number of discordances strictly greater than 2 for all cases, and strictly greater than 1 for answers corresponding to questions with 3 options.

Q1

```
CONSTRUCT  {
?question sides:has_for_number_of_proposals ?np}
WHERE {
SELECT ?question (COUNT (?p) As ?np)  {
?question sides:has_for_proposal_of_answer ?p}
GROUP BY ?question}
```

Q2

```
CONSTRUCT {
?answer
sides:has_for_number_of_wrong_tick ?nw }
WHERE {select ?answer (COUNT (?a) As ?nw)
{?a sides:is_part_of  ?answer.
?a sides:has_wrongly_ticked ?p}
GROUP BY ?answer
}
```

Q3

```
CONSTRUCT {
?answer
sides:has_for_number_of_missed_right_tick ?nm}
WHERE {SELECT ?answer (COUNT(?p) As ?nm)
{?answer sides:correspond_to_question ?q.
?q sides:has_for_proposal_of_answer ?p.
?p sides:has_for_correction "true"^^xsd:boolean.
FILTER NOT EXISTS {
?a sides:is_part_of  ?answer.
?a sides:has_rightly_ticked  ?p}
}  GROUP BY ?answer
}
```

Q4

```
CONSTRUCT {
?answer sides:has_for_number_of_discordance
        "0"^^xsd:integer}
WHERE {
?answer a sides:answer.
FILTER NOT EXISTS  {
?answer sides:has_for_number_of_wrong_tick ?nw.
?answer sides:
        has_for_number_of_missed_right_tick ?nm.
}}
```

Q5

```
CONSTRUCT {
?answer
sides:has_for_number_of_discordance ?count}
WHERE {
SELECT  ?answer  (?nw + ?nm as ?count)   {
   ?answer
   sides:has_for_number_of_wrong_tick ?nw.
   ?answer
   sides:has_for_number_of_missed_right_tick ?nm
}
}
```

Q6

```
CONSTRUCT {
?answer
sides:has_for_number_of_discordance ?nw
}  WHERE {
?answer sides:has_for_number_of_wrong_tick ?nw.
FILTER NOT EXISTS {
?answer sides:
        has_for_number_of_missed_right_tick ?nm}
}
```

Q7

```
CONSTRUCT {
?answer
sides:has_for_number_of_discordance ?nm}
WHERE {
?answer sides:
        has_for_number_of_missed_right_tick ?nm.
FILTER NOT EXISTS  {
?answer sides:has_for_number_of_wrong_tick ?nw.
         }
}
```

Q8

```
 CONSTRUCT {
 ?answer sides:has_for_result 1}
 WHERE {
 ?answer
 sides:has_for_number_of_discordance "0"^^xsd:
         integer
 }
```

Q9

```
  CONSTRUCT {
  ?answer sides:has_for_result "0"^^xsd:integer  .
  ?answer sides:stronglyWrong  "true"^^xsd:
          boolean .}
  WHERE {
  ?a sides:is_part_of  ?answer.
  ?a sides:has_wrongly_ticked ?p.
  ?p
  sides:has_for_weight_of_correction
  "Unacceptable"^^xsd:string .
  }
```

Q10

```
CONSTRUCT {
?answer sides:has_for_result "0"^^xsd:integer  .
?answer sides:stronglyWrong  "true"^^xsd:boolean .
         }
WHERE {
?answer sides:correspond_to_question ?q.
?q sides:has_for_proposal_of_answer ?p.
?p sides:has_for_correction "true"^^xsd:boolean .
?p sides:has_for_weight_of_correction "
        Indispensable"^^xsd:string .
FILTER NOT EXISTS {
?a sides:is_part_of  ?answer.
?a sides:has_rightly_ticked  ?p
}}
```

Q11

```
CONSTRUCT {
?answer sides:has_for_result "0"^^xsd:integer  .
?answer sides:stronglyWrong  "true"^^xsd:boolean .
         }
WHERE {
?answer sides:correspond_to_question ?q.
?q rdf:type sides:QUA.
?answer sides:has_for_number_of_discordance ?d.
FILTER (?d > 0)
}
```

Q12

```
CONSTRUCT {
?answer sides:has_for_result 0.5^^xsd:decimal}
WHERE {
?answer sides:has_for_number_of_discordance
        "1"^^xsd:integer .
?answer sides:correspond_to_question ?q.
?q sides:has_for_number_of_proposals "5"^^xsd:
        integer.
FILTER NOT EXISTS {?answer sides:
        stronglyWrong  "true"^^xsd:boolean }
}
```

Q13

```
CONSTRUCT {
?answer sides:has_for_result "0.2"^^xsd:decimal}
WHERE {
?answer sides:has_for_number_of_discordance
        "2"^^xsd:integer .
?answer sides:correspond_to_question ?q.
?q sides:has_for_number_of_proposals "5"^^xsd:
        integer .
FILTER NOT EXISTS {
?answer sides:stronglyWrong  "true"^^xsd:boolean }
}
```

Q14

```
CONSTRUCT {
?answer sides:has_for_result "0.425"^^xsd:decimal}
WHERE {?answer sides:
        has_for_number_of_discordance "1"^^xsd:
        integer .
?answer sides:correspond_to_question ?q.
?q sides:has_for_number_of_proposals "4"^^xsd:
        integer .
FILTER NOT EXISTS {?answer sides:
        stronglyWrong  "true"^^xsd:boolean }}
```

Q15

```
CONSTRUCT {
?answer sides:has_for_result "0.1"^^xsd:decimal  }
WHERE {
?answer sides:has_for_number_of_discordance
        "2"^^xsd:integer .
?answer sides:correspond_to_question ?q.
?q sides:has_for_number_of_proposals "4"^^xsd:
        integer .
FILTER NOT EXISTS {
?answer sides:stronglyWrong  "true"^^xsd:boolean
        }}
```

Q16

```
CONSTRUCT {
?answer sides:has_for_result "0"^^xsd:integer}
WHERE {
?answer sides:correspond_to_question ?q.
?q sides:has_for_number_of_proposals ?np.
?answer sides:has_for_number_of_discordance ?n.
FILTER (?np > 3 && ?np < 6 && ?n > 2).}
```

Q17

```
CONSTRUCT  {
?answer sides:has_for_result "0.3"^^xsd:decimal}
WHERE {
?answer sides:has_for_number_of_discordance
        "1"^^xsd:integer .
?answer sides:correspond_to_question ?q.
?q sides:has_for_number_of_proposals "3"^^xsd:
        integer .
FILTER NOT EXISTS {
?answer sides:stronglyWrong  "true"^^xsd:boolean }
}
```

Q18

```
CONSTRUCT {
?answer sides:has_for_result "0"^^xsd:integer }
WHERE {
?answer sides:has_for_number_of_discordance ?n.
?answer sides:correspond_to_question ?q.
?q sides:has_for_number_of_proposals "3"^^xsd:
        integer.
FILTER (?n > 1)  }
```


Figure 3.4 SPIN rules based definition of properties

3.1.2 Mapping-based data materialization

In this section, we describe our approach for populating classes and properties of a given ontology by extracting data from an existing relational database and by transforming them into RDF triples.

In our setting, the (OntoSIDES) ontology is made of a hierarchy of classes and a hierarchy of properties (defined as a set of RDFS statements) among which some properties are defined by rules. Therefore, the classes and properties that we populate by data extraction are the classes and properties the instances of which cannot be derived by RDFS or domain-specific rules, which are:

- the most specific classes, i.e., the classes that do not have any sub-class

- the properties that do not have any sub-property and that are not in the conclusion of any domain-specific rule.

The existing data source from which we extract data is the operational relational database SIDES that is a large database whose schema contains 400 tables.

As explained in Section 2.3.2, mappings are a declarative way to specify the correspondence between (simple) RDF templates (described using the terms of a target ontology) and SQL queries over the schema of a source database. An RDF template in a mapping is an RDF pattern in which some placeholders indicate how to instantiate this pattern by URIs or literals created from answers returned by the SQL query. Mapping can then be used to populate a target ontology by evaluating for each mapping the SQL query over the source database, and by replacing the placeholders in the RDF template by values returned as answers to the SQL query.

We use Ontop to declare and materialize mappings. Figure 3.5 shows an example of an Ontop mapping declaration given a set of prefixes that must be declared beforehand.

Each mapping consists of three parts: mappingId, the mapping identifier; target, the triple template(s); and source, which contains the SQL query.

mappingId urn:test
target sides:test{id} a sides:test ; sides: has_for_title { title }^^xsd:string .
source select id, title from public.assessment

Figure 3.5 An example of an Ontop mapping

This mapping shows how to populate the class `sides:test` i.e., how to populate the typing property `rdf:type`, shortened as `a`, for the instances of the `sides:test` class, and how to extract their value for the property `sides:has_for_title`. The target is thus an RDF template with two triples. The source is a simple SQL query over the single SIDES table called `public.assessment`.

The template placeholders are enclosed in curly brackets and their names correspond to the same column names in the SQL query. The first placeholder shows how we create automatically URIs involved in the RDF triples from identifiers in the SIDES tables. The second one illustrates the extraction of (typed) literals from values found in the SIDES tables.

Similar simple mappings can be specified for populating the other properties having `sides:test` as domain: `sides:starting_date_of_test`, `sides:ending_date_of_test`.

The reason is that all the useful information that we need to extract concerning the instances of the `sides:test` class are grouped into a single table of SIDES, in particular with a direct correspondences between the column names of the table and the datatype properties of `sides:test`.

This simple case is however not the rule: most of the OntoSIDES classes do not correspond to a single table in SIDES.

We now summarize the methodology that we have applied for creating mappings for each properties that have to be populated by data extraction from the source SIDES. First, we create one mapping for each object property, and we group data type properties with same domain in the same mapping. We then examine the tables in the database to construct SQL queries with the minimum number of joins for these templates.

We have used *materialized views* to reduce the complexity of the SQL queries and the size of the data to be queried. A materialized view is the precomputation of the output of an SQL query. This solution relies on data redundancy, as the materialized view duplicates existing data. This solution is a good alternative in cases where additional storage space is not an issue. In addition, a materialized view has the advantage of preserving the original database schema.

We have created five types of materialized views: The first type is a single-column *lookup* table, which reduces the number of record IDs to process during joins. The second type is an *entity* table, which consolidates attributes of a record that are scattered across multiple tables into one table (denormalization). The third type is a two-column *relationship* table, which stores the relationships between record ids of other tables. The fourth type is a *hybrid*, a combination of the previous types. The fifth type is a *repair* table, which reshapes malformed data. An example of malformed data is the presence of duplicate ticks stored as a result of a user interface failure.

The creation of materialized views is a very complex task because it requires to examine all the objects in a database (e.g. table attributes, primary keys, foreign keys, indexes) to find the optimal queries. To facilitate the task, we have built an Entity Relation (ER) diagram of all relationships and schema information about the SIDES tables. We have used SchemaSpy[2] software to automate the ER diagram extraction.

[2]https://schemaspy.org/

We have completed the construction of the mappings once all the materialized views have been computed. In this context, each template containing an object property is mapped to a materialized view of the type *relationship*. Regarding the templates containing datatype properties, some of them are mapped directly to a materialized view of type *entity* or *hybrid*, and the remaining templates are mapped to the corresponding SQL queries built by joining materialized views of type *lookup*, *entity*, *relationship*, and *repair* against existing SIDES tables.

An example of materialized view of the *hybrid* type is shown in the figure 3.6. The contents of the seven tables `response_table`, `responses_choices`, `assessment_session`, `participant`, `assessment`, `choice`, and `choice_correction` are denormalized into a new table named `ontosides.response` using the SQL command `CREATE MATERIALIZE VIEW`. The correct selection of join columns was facilitated by the ER diagram.

```
CREATE MATERIALIZED VIEW
ontosides.response AS (
SELECT CONCAT(CAST(response_id AS text), CAST(choice_id AS text), LENGTH(choice_id::text)) AS response_id_new, rt.id AS response_id, rc.
      choice_id, p.participant_id, a.id AS assessment_id, ass.begindate, cc.valid, case cc.valid WHEN true THEN false WHEN false THEN
      true END AS invalid
FROM response.response_table rt
INNER JOIN response.responses_choices rc ON rc.response_id = rt.id
INNER JOIN response.assessment_session ass ON ass.id = rt.test_participant_id
INNER JOIN participant p ON p.id = ass.participant_id
INNER JOIN assessment a ON a.id = p.assessment_id
INNER JOIN choice c ON c.id = rc.choice_id
INNER JOIN correction.choice_correction cc ON cc.id = c.correction_id
);
```

Figure 3.6 An example of materialized view

A materialized table such as `ontosides.response` allows the declaration of simple OBDA mappings that are easy to read and verify. For example, Figure 3.7 shows three OBDA mappings using Ontop syntax that declare a simple SELECT query on the materialized view `ontosides.response` as the source.

mappingId urn:relation_answer_action_to_answer
target sides:adr{response_id_new} sides:is_part_of sides:answer{response_id} .
source select response_id_new, response_id from ontosides.response

mappingId urn:relation_answer_student
target sides:answer{response_id} sides:done_by sides:stu{participant_id} .
source select response_id, participant_id from ontosides.response

mappingId urn:relation_has_wrongly_ticked
target sides:adr{response_id_new} sides:has_wrongly_ticked sides:prop{choice_id} .
source select response_id_new, choice_id from ontosides.response r where r.invalid = true

Figure 3.7 Examples of OBDA mappings using a materialized view in the source

We created 76 OBDA mappings to materialize OntoSIDES. The list can be reviewed in Appendix A.1. Over time, the number of mappings did not change. Each time there was a change in the SIDES schema, we just modified the source of the mapping but kept the same target. Table 3.1 shows the size of the materialized data, the size of the database dump and its release date.

Version	materialized data size (billion triples)	database dump size (GB)	dump release date
v1	0.0052	2.1	2015-07-31
v2	0.12	4.51	2016-06-19
v3	5.26	191	2019-01-31
v4	5.53	197	2019-03-20
v5	7.67	264	2020-03-23
v6	9.86	424	2021-03-30
v7	11.06	447	2021-07-01
v8	11.09	485	2021-08-01

Table 3.1 The evolution of OntoSIDES' size over the years

To provide context for the scale of the data generated, the last version of Ontosides data contains almost 219 000 students, 817 millions answers and 2.4 millions questions.

By default, Ontop stores the output of each mapping sequentially in a single file, which creates a bottleneck when materializing a large dataset. To solve this problem, we modified Ontop's source code to stream the output of each mapping into a single file to parallelize the process and improve performance. The solution not only improved performance during materialization but also enabled parallelization during loading the RDF data into a triplestore.

We selected Virtuoso as our preferred triplestore to store the materialized data generated by the mappings. While later research revealed that it may lack reliability for certain types of types of queries, our selection was based on its suitability for our storage requirements and the resources available at the time.

3.1.3 Rule-based data saturation

The data saturation consists in applying a forward reasoning algorithm to the 18 SPIN rules given in Figure 3.4 and to the RDF graph resulting from the data materialization process described in the previous section. This saturation process allows to finalize the population of the properties in the ontology that are defined by rules (and not extracted by mappings).

The dependency graph (as defined in Chapter 2, Section 2.4) built on these 18 CONSTRUCT-based SPIN rules is provided in Figure 3.8.

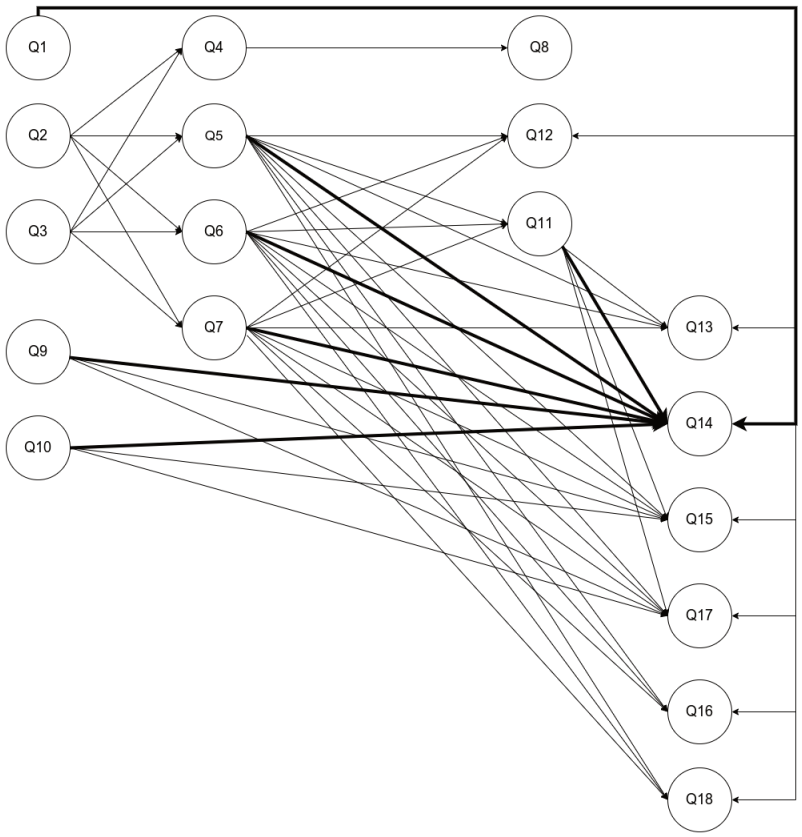

Figure 3.8 The dependency graph built from the 18 CONSTRUCT-based rules

Since it is acyclic, the rules are non recursive and can be structured in 4 layers of increasing depth:

- Layer 1 = {Q1, Q2, Q3, Q9, Q10}
- Layer 2 = {Q4, Q5, Q6, Q7}
- Layer 3 = {Q8, Q12, Q11}
- Layer 4 = {Q13, Q14, Q15, Q16, Q17, Q18}

Since, there is no dependency between queries of a given layer, they can be evaluated in any order within each layer.

Therefore the saturation process can be performed by using the forward-chaining algorithm given in Chapter 2 (Section 2.4, Figure 2.9), in which the CONSTRUCT queries are ordered by layer of increasing depth with any order in each layer.

Table 3.2 shows the size of the sets of inferred triples resulting from of the saturation process applied to the different versions of the materialized data given in Table 3.1.

Version	materialized data size (billion triples)	inferred data (billion triples)
v1	0.0052	0.0027
v2	0.12	0.052
v3	5.26	1.35
v4	5.53	1.44
v5	7.67	1.58
v6	9.86	1.95
v7	11.06	2.21
v8	11.09	2.26

Table 3.2 The evolution of the size of inferred data in OntoSIDES over time

3.2 Data modularization and module extraction

Although extracting modules from ontologies have been largely studied, mainly from a theoretical perspective (e.g. [33]), the extraction of modules from RDF triplestores has been the subject of little research. The most inspiring approaches for our work are the following:

- the SOMET framework [34] is based on a parameterised graph traversal algorithm that is implemented using CONSTRUCT SPARQL queries, and captures in a uniform setting different module extraction approaches ([35], [36], [37] and [38]),

- in [39], modules can be obtained from, or are related to, a set of pre-computed modules of the same ontology,

- in [40], the specification of bounded-level modules allows for controlling the size of extracted modules.

Despite all these works come with implementations, they did not deal with the scalability issues that we had to face with for modularizing big RDF graphs.

A module is a subset of a given input RDF graph that is built for various purposes. In particular, it can be used for:

1. retrieving a subset of interest for a user from a reference RDF triplestore, for example for extracting easily data specific to a given student who would like to delete all the data about her/him that he/she considers as sensitive,

2. modularizing a large single RDF graph to structure it into subgraphs for optimization purposes, for example for optimizing rule-based reasoning.

In this thesis, we propose a unifying approach for handling the two above cases, based on the explicit specification of the properties paths to be extracted either because they correspond to sensitive data for instances of a given class, or because they are involved in the conditions of rules to infer new properties of some classes. In Section 3.2.1, we formalize the notion of modules we consider, and in Section 3.2.2, we describe the methodology that we followed for their effective extraction.

3.2.1 Module specification and semantics

Modules are defined based on property paths rooted at a given instance of a class.

Definition 3.1 (rooted property path)**.**
Given a set of properties of an RDFS ontology, a property path is a sequence, denoted $q_1/q_2, .../q_n$, where each q_i is either a property p of the ontology or the inverse p^- of a property of the ontology.

Given an instance i_0 of a class C in an RDF graph G, a *property path rooted* at i_0, denoted $[i_0 : C](q_1/q_2, .../q_n)$, is a set of triples $\{(i_O, q_1, i_1), ...(i_{n-1}, q_n, i_n)\}$ such that for every $k \in [1..n]$ $i_{k-1} \neq i_k$ and $(i_{k-1}, q_k, i_k) \in G$, where (i_{k-1}, q_k, i_k) denotes the triple (i_k, p, i_{k-1}) if $q_k = p^-$.

Definition 3.2 (Module)**.** Given an RDF data graph G, a *module* rooted at a given instance i_0 of a class C, specified by $[i_0 : C](propertyPath_1, ..., propertyPath_k)$, where for each i, $propertyPath_i$ is a property path, is the subgraph of G that is the union of all the property paths rooted at i_0:

$[i_0 : C](propertyPath_1, ..., propertyPath_k) = \bigcup_{i=1}^{k} PropertyPath_i(i_0, C)$

where $PropertyPath_i(i_0, C)$ is the set of all the property paths $[i_0 : C](propertyPath_i)$ in G.

It is often the case that the modules of interest correspond to the same specification for all the instances of a given class. This is captured by the definition of a module pattern that is a graph pattern built on property path patterns, as defined in Definition 3.3

Definition 3.3 (Module pattern)**.**
Given an RDFS ontology:

- a property path pattern, specified by $[?s_o : C](q_1/q_2, .../q_n)$, where $q_1/q_2, .../q_n$ is a property path, is a graph pattern $\{(?s_0, q_1, ?s_1), ...(?s_{n-1}, q_n, ?s_n)\}$ where all the variables are pairwise distinct and $(?s_{k-1}, q_k, ?s_k)$ denotes the triple pattern $(?s_k, p, ?s_{k-1})$ if $q_k = p^-$.

- a module pattern, specified by $[?s_o : C](propertyPath_1, ..., propertyPath_k)$ is the union of the property path patterns $[?s_o : C](propertyPath_k)$ in which all the variables are pairwise distinct.

Theorem 3.1 is a straightforward consequence of Definition 3.2 and Definition 3.3. It gives a constructive way to extract modules by computing all the projections of a module pattern on the input RDF graph, which is precisely what SPARQL does.

Definition 3.4 (Projection of a graph pattern onto an RDF graph)**.** A projection of a graph pattern GP onto an RDF graph G is the image of GP by an homomorphism μ that replaces each variable in the graph pattern GP by a constant (i.e., an URI, a literal or a blank node) such that for each triple pattern tp in GP, $\mu(tp) \in G$.

Theorem 3.1. *Given a module pattern specified by* $[?s_o : C](propertyPath_1, ..., propertyPath_k)$*, and given a RDF graph G and an instance i of the class C in G, the module* $[i : C](propertyPath_1, ..., propertyPath_k)$ *can be obtained as the union of all the projections onto G of the graph pattern obtained by replacing the variable* $?s_0$ *in the module pattern* $[?s_o : C](propertyPath_1, ..., propertyPath_k)$.

We will say that a module $[i : C](propertyPath_1, ..., propertyPath_k)$ is *an instance of the module pattern* $[?s_o : C](propertyPath_1, ..., propertyPath_k)$, and that the module pattern $[?s_o : C](propertyPath_1, ..., propertyPath_k)$ is *a module specification* for the class C.

The following example shows a module specification for the class *student* for defining the sensitive information that each student could ask to delete to protect his/her privacy. Such a specification must be given by an expert user who is helped by the visualization of the ontology (see Figure 3.3) for inspecting the different property paths related to the class of interest.

Example 3.2.1. Suppose that the expert user (here the data protection officer) specifies that the sensitive information about students (thus the information that each student should be enabled to ask to be removed), consists in:

- their rank at the ECN (which is the national grading exam at the end of second cycle),
- the different tests they registered to,
- the questions they have answered to, with the grade they obtained for each of them,
- and the information about their successive enrolments.

Based on the properties in the ontology, this will result in the following module specification:

$[?s_0 : student]($
$has\_ecnp\_rank,$
$has\_for\_registrant^-,$
$done\_by^-/correspond\_to\_question,$
$done\_by^-/has\_for\_result,$
$correspond\_to\_student^-/has\_for\_registration\_place,$
$correspond\_to\_student^-/has\_for\_registration\_date,$

$correspond\_to\_student^{-}/correspond\_to\_training$
)

3.2.2 Module extraction

Theorem 3.1 is the formal basis for showing how to use SPARQL to extract modules. For example, Figure 3.9 provides the SPARQL CONSTRUCT query to extract the student module in Example 3.2.1 for a specific student (*sides*:*student*2023). It is a direct implementation in SPARQL of the module specification.

```
CONSTRUCT {
?s_0 sides:has_for_ecnp_rank ?rank .
?test sides:has_for_registrant ?s_0.
?answer sides:done_by ?s_0 .
?answer sides:correspond_to_question ?question .
?answer sides:has_for_result ?result.
?enrolment sides:correspond_to_student ?s_0.
?enrolment sides:has_for_registration_place ?place.
?enrolment sides:has_for_registration_date ?date.
?enrolment sides:correspond_to_training  ?training.
}
WHERE {
?s_0 sides:has_for_ecnp_rank ?rank .
?test sides:has_for_registrant ?s_0.
?answer sides:done_by ?s_0 .
?answer sides:correspond_to_question ?question .
?answer sides:has_for_result ?result.
?enrolment sides:correspond_to_student ?s_0.
?enrolment sides:has_for_registration_place ?place.
?enrolment sides:has_for_registration_date ?date.
?enrolment sides:correspond_to_training  ?training.
FILTER (?s_0 = sides:student2023)
}
```

Figure 3.9 A CONSTRUCT query to extract a specific module specified in Example 3.2.1

However, in practice, this naive approach does not scale to the size of the large RDF graphs that we consider in this thesis, even if we split the process by evaluating several CONSTRUCT SPARQL queries (one by property path of the module specification) to have templates with less joins in the output of the CONSTRUCT queries.

The scalability issues are related both to the large size of some modules (like modules of students that may contain millions of triples) as well as to the large number of modules (for instance if the target class is the class *answer* having 817 million instances in the last version of OntoSIDES).

For scaling up module extraction, we have chosen to represent modules as *RDF named graphs* identified by the roots of the modules, which can be materialized or computed on the fly. *Module materialization* (like view materialization used in Section 3.1.2) allows to compute some modules only once, and to re-use them for computing other modules by *module combination*.

We now explain our approach for module materialization in Section 3.2.2.1 and for module combination in Section 3.2.2.2.

3.2.2.1 Mapping-based module materialization

We have designed Ontop mappings to materialize modules as sets of RDF quads where the first three elements correspond to triple patterns in the module and the fourth element to its module root.

More precisely, given the specification of a module, each property path pattern is splitted into its triple patterns that are transformed into *quad templates* with four placeholders: the first three placeholders of the quad template correspond to the three variables of the triple template and the fourth placeholder corresponds to the root variable of the module specification. The Ontop mappings are then completed by building the corresponding SQL queries for populating the quad templates. Finally, the mappings are materialized and the resulting RDF quads are stored in the triplestore.

In the setting of OntoSIDES, we have used this mapping-based approach to extract (and store in Virtuoso) the modules of the classes `answer`, `question` and `enrolment` according to their following specification.

Module specification for the class $answer$**:** This specification has been guided both by the need to get the properties appearing in the conditions of rules in the ontology, and by the need to relate the answers to the corresponding questions, the students who entered them, with the corresponding time stamp, and also the tests during which they have been done.

$[?s_0 : answer]($
$done\_by,$
$correspond\_to\_question,$
$is\_part\_of^-/has\_rightly\_ticked,$
$is\_part\_of^-/has\_wrongly\_ticked,$
$is\_part\_of^-/done\_during,$
$has\_for\_timestamp,$
$type)$

Module specification for class *question***:** This specification has been guided by the need to isolate all the information describing questions to facilitate their access for allowing teachers to analyse of their pedagogical content, and the UNESS administrators to find easily the authors and co-authors of each question.

$[?s_0 : question]($
$has\_for\_question^{-}$,
$has\_for\_proposal\_of\_answer/has\_for\_multimedia\_content/type$,
$has\_for\_proposal\_of\_answer/type$,
$has\_for\_proposal\_of\_answer/has\_for\_correction$,
$has\_for\_proposal\_of\_answer/has\_for\_textual\_content$,
$has\_for\_proposal\_of\_answer/has\_for\_weight\_of\_correction$,
$has\_for\_proposal\_of\_answer/has\_position$,
co_author,
has_for_weight,
$is\_linked\_to\_the\_cross\_knowledge\_entity$,
$is\_linked\_to\_the\_medical\_speciality$,
$is\_linked\_to\_ECN\_referential\_entity$,
has_for_title,
$has\_for\_comment$,
$has\_for\_expected\_answer\_text$,
$has\_for\_textual\_content$,
$has\_rating/type$,
$has\_rating/given\_by$,
$has\_rating/has\_for\_timestamp$,
has_rating/has_for_value,
$type$)

Module specification for class *enrolment***:** This corresponds to the need to have a view of all the history of the enrolments of students that can have changed university and training levels over the years.

$[?s_0 : enrolment]($
$correspond\_to\_student^{-}$,
$correspond\_to\_student^{-}/correspond\_to\_training$,
$correspond\_to\_student^{-}/has\_for\_registration\_place$,
$correspond\_to\_student^{-}/has\_for\_duration$,
$correspond\_to\_student^{-}/has\_for\_provenance$,
$correspond\_to\_student^{-}/has\_for\_registration\_date$,
$type$)

For all these 3 module specifications, the mappings to extract the data of the modules

are direct extensions to quad templates of the corresponding mappings used for extracting triples. For example, Figure 3.10 shows 3 mappings to extract RDF quads corresponding to the 3 property paths $done\_by$ and $is\_part\_of^{-}/has\_wrongly\_ticked$ of the modules As we can see, the fourth position of the quad templates is bound to the root variable of the module specification, which in this case is represented by `sides:answer{response_id}`. The `{response_id}` placeholder is instantiated with values from a column named "response_id" that is available in the SQL query mapping source.

mappingId urn:relation_answer_action_to_answer_nquad
target sides:adr{response_id_new} sides:is_part_of sides:answer{response_id} sides:answer{response_id} .
source select response_id_new, response_id from ontosides.response

mappingId urn:relation_answer_student_nquad
target sides:answer{response_id} sides:done_by sides:stu{participant_id} sides:answer{response_id}.
source select response_id, participant_id from ontosides.response

mappingId urn:relation_has_wrongly_ticked_nquad
target sides:adr{response_id_new} sides:has_wrongly_ticked sides:prop{choice_id} sides:answer{response_id}.
source select response_id_new, choice_id, response_id from ontosides.response r where r.invalid = true

Figure 3.10 Example of Ontop mappings to materialize modules using RDF quads templates

It is important to note that using such mappings enables the construction of all the modules rooted in instances of the class $answer$ by evaluating a few SQL queries the number of which does not depend on the number of instances of the class.

3.2.2.2 CONSTRUCT-based module combination

Large modules can be obtained by combining modules already stored in the triplestore as RDF quads. This combination can be expressed by quad CONSTRUCT queries whose evaluation will generate the RDF quads making up the module.

For example, Figure 3.11 shows a Quad CONSTRUCT query that specifies a module of a specific student ($sides$:$student$2023) as a combination of the modules corresponding to his/her answers, the related questions, and his/her enrolments. This module differs from the module specified in Example 3.2.1 since it contains, for each answer, question or enrolment related to the student, all the information in the union of their respective modules. For the experiments that we will report in Chapter 5, we have used this module specification both

for constructing datasets of increasing size (as the union of such modules extracted for an increasing number of students), and for a module-based optimization of reasoning.

Extracting the corresponding data can be done by evaluating this query. The body of the query consists of a basic graph pattern (lines 6-8) and a collection of subqueries (lines 11-13). The basic graph pattern is a set of chained triple patterns connecting the answer, question, enrolment and student variables, one of which is filtered by a value specified in line 9. This value is the URI of the new module (i.e. *sides*:*student*2023). As for the subqueries, each one retrieves data from a named graph whose identifier matches a variable of the basic graph pattern.

```
1          CONSTRUCT {
           ?s ?p ?o ?student .
3          }
           WHERE {SELECT ?s ?p ?o ?student{
5          {
           ?answer sides:done_by ?student .
7          ?answer sides:correspond_to_question ?question .
           ?enrolment sides:correspond_to_student ?student .
9          FILTER (?student = sides:student2023)
           }
11         {SELECT ?s ?p ?o ?answer1 {GRAPH ?answer1 {?s ?p ?o}}}
           {SELECT ?s ?p ?o ?question1 {GRAPH ?question1 {?s ?p ?o}}}
13         {SELECT ?s ?p ?o ?enrolment1 {GRAPH ?enrolment1 {?s ?p ?o}}}
           FILTER (?answer1=?answer && ?question1 = ?question && ?enrolment1
    = ?enrolment)
15         }}
```

Figure 3.11 A Quad CONSTRUCT query to specify a module for a specific student

Quad CONSTRUCT queries are not within the SPARQL 1.1 specification and thus are not supported by the existing triplestores such as Virtuoso or GraphDB. An alternative solution would consist in evaluating the induced SELECT queries and then constructing the output quad triples from the results of the SELECT queries. However, this solution does not work either in practice for big RDF data, mainly due to the multiple GRAPH subqueries involved, and the fact that their big results have then to be processed by multiple joins and multiple filtering conditions.

To solve this issue, we have implemented an application for evaluating efficiently Quad CONSTRUCT queries on top of the TESS Spark-based infrastructure for big RDF triplestores that we have developed and that will be described in the next chapter.

3.3 Summary

In this chapter, we have introduced an OBDA architecture for periodically releasing big RDF graphs, and we have shown how we have used it to build the OntoSIDES knowledge base. We have described the OntoSIDES ontology and the mappings used to materialize RDF data. The ontology contains SPIN rules used to saturate the data.

We have also explained the usefulness of modularizing big RDF graphs, and we have described our methodology for facing the scalability issues by materializing modules as named graphs and by combining them. The combination of modules based on quad CONSTRUCT queries led us to design a novel architecture for big RDF triplestores that is described in the next chapter.

Chapter 4

TESS infrastructure for Big RDF triplestores

Scalability is still an open issue for state-of-art triplestores, when it concerns large RDF datasets constantly growing. Our initial decision to use Virtuoso as the triplestore to hold the RDF data produced by materialization proved to be inadequate. As the size of the data increased, performance problems began to surface in queries, indicating that triplestores are still struggling with the challenge of scalability. We showed in [11] that classic triplestores like Virtuoso and GraphDB fall into completeness, correctness and timeouts issues when dealing with aggregated/complex queries and large outputs.

There are two ways to deal with scalability of management and querying big RDF data: extending traditional Data Base Management Systems (DBMS) or exploiting Big Data technologies. The choice depends on the support for columnar storage: native or hybrid.

Native column storage saves data columns contiguously to speed up queries that require large data retrieval (e.g. analytical queries). In a hybrid column storage, there are two versions of the same data. While a version remains in disk, the other version is kept in RAM memory but in columnar format. Hybrid column storage has been implemented by traditional DBMS like SQL server and Oracle but it does not yield good performance [41] .

Big Data technologies cover a broad spectrum of software tools that can be used and combined for developing data-intensive applications. Spark [42] is one of the most powerful big data technologies based on parallelization optimizations over distributed environments.

Sansa [43] and Bellman [44] are Spark-based engines for scalable processing of large-scale RDF data. A variety of tools and technologies can be combined for handling efficiently the different aspects of distributed RDF data management. The diversity of big data technologies for building so-called *Big RDF frameworks* is summarized in Figure 4.1 extracted from [1].

Big RDF frameworks emulate some DBMS features at scale such as storage, partitioning,

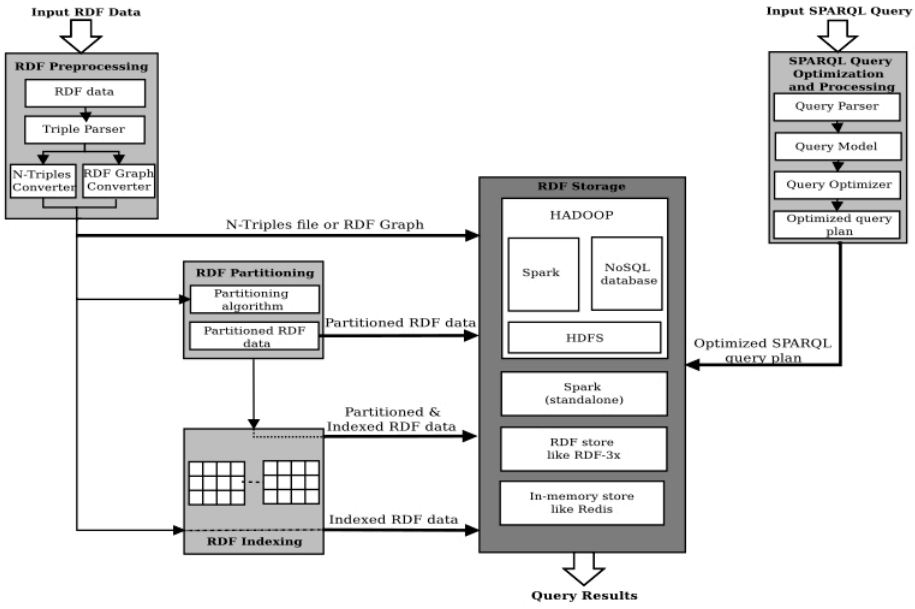

Figure 4.1 A generic Big RDF framework [1]

indexing, query optimization and processing. However, up to our knowledge, no existing big RDF framework supports metadata management, and transactional updating that are two important DBMS features.

In our PhD work, through TESS, we propose a transition from Big RDF frameworks to *Big RDF triplestores* for a full implementation of DBMS operations. This transition is not trivial because such as implementation must interplay with an ecosystem of technologies under constant evolution.

We have designed and implemented a Big RDF triplestore, called TESS, offering DBMS service layer to cover data updates as transactions. By adding this layer, we extend the dimensions considered in [45] for describing triplestores, and we propose a Big RDF triplestore of five layers: Data distribution, Storage, Query processing, Transactional Metadata layer and Reasoning.

The remaining content of this chapter is presented as follows. In Section 4.1, we survey the Big Data technologies according the different layers of a Big RDF triplestore. In Section 4.2, we describe the layers of the TESS architecture. Finally, in Section 4.3 we provide a summary of the topics above discussed.

4.1 State of art

Big Data technologies are not a source of long-term solutions for semantic web applications. Instead, it offers a buoyant ecosystem of technologies where the best tool for the job

yesterday, it is not necessarily the best tool for the job today. For this reason, each time a semantic web application requires to use Big Data technologies, a deep review must be conducted in the state of the art.

To narrow the review, we inspect those technologies that are the most compatible with Spark. We chose Spark because: a) an extense survey about Big RDF frameworks [46] shows its versatility for many high performance scenarios, b) it has been awarded with the 2022 ACM SIGMOD Systems Award [1] as "an innovative, widely-used, open-source, unified data processing system encompassing relational, streaming, and machine-learning workloads", and c) it catches up latest hardware improvements (e.g. GPUs).

Big RDF frameworks based on Spark has been studied in [46]. Overall, the frameworks are aligned with the two main Spark data abstractions: Resilient Distributed Dataset (RDD) and Dataframe. A RDD is an in-memory datastructure that enables data sharing between computation stages. Dataframe is built on top of RDD and it contains additional schema-level information to deal with structured data. SANSA [2] and Bellman are two Spark-based frameworks that have gained some traction among researchers and practitioners. In contrast with SANSA [47], Bellman [44] loads RDF data directly into Dataframes and execute SPARQL queries (even CONSTRUCT queries) translated into Spark-SQL. However, it does not support full SPARQL 1.1 and it is not clear if it supports aggregate queries due to lack of documentation.

The survey in [1] shows how distributed storage and query processing have evolved over time. For SPARQL parallel query processing, MapReduce has been replaced gradually by Spark, whereas for distributed storage and storage format, Hive[3] and HBase[4] has been superseeded by HDFS[5] and Parquet[6]. Although some of the frameworks considered aggregate queries in their performance studies, none of them dealt with FILTER NOT EXISTS queries.

In this section, we review the most reliable parallelization technologies on each layer of Big RDF triplestores.

4.1.1 Data distribution

The distribution of data over a cluster of disk nodes allows to scale storage as data grows. As of today, there are two widely-used alternatives to distribute data: HDFS and Object Storage. The former is deployed on cloud while the latter is deployed on-premise using local infrastructure.

[1] https://sigmod.org/2022-sigmod-awards
[2] https://sansa-stack.net/
[3] https://hive.apache.org/
[4] https://hbase.apache.org/
[5] https://hadoop.apache.org/docs/r1.2.1/hdfs_design.html#Introduction
[6] https://parquet.apache.org/

1. **Hadoop Distributed File System (HDFS)**. It consists of a filesystem where data is stored over a cluster of low-cost hardware. The use of a filesystem allows a direct interaction between an user and typical command line operations for maintenance tasks. In HDFS, data is broken down into blocks. The default size block is 128MB. The blocks are distributed and replicated over the cluster to guarantee fault tolerance and data availability. The default replication factor is 3 and it means that 3 copies of the same block are distributed in the cluster. The replication factor is costly because it transforms a large dataset into one even larger.

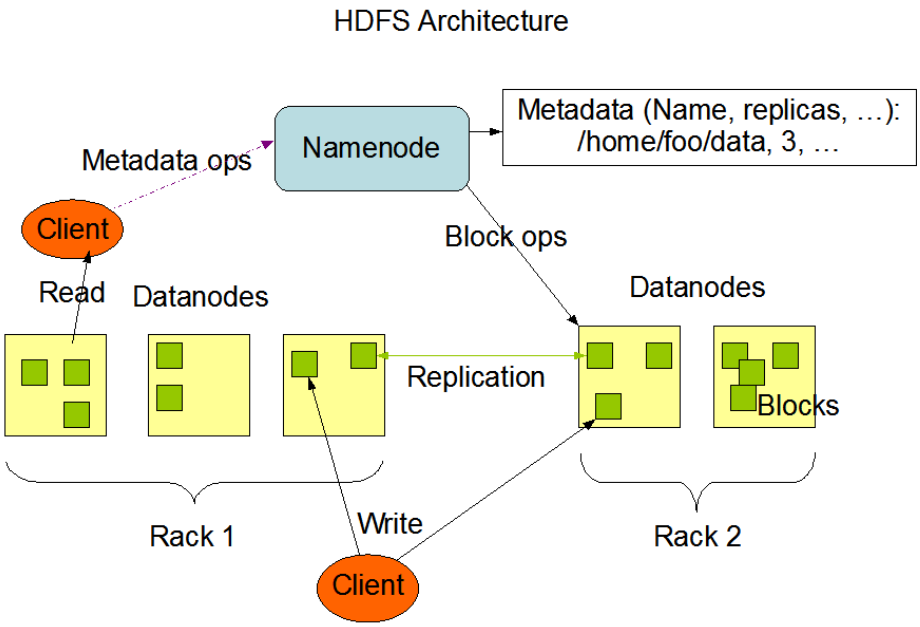

Figure 4.2 HDFS architecture [2]

A HDFS cluster is conformed by a *namenode* (master) and a set of *datanodes* (slaves). The namenode contains all metadata (name, replicas, location, etc) about all the datanodes.

In Figure 4.2, the namenode (light blue color) receives read/write requests from two clients (orange color). The data is splitted in blocks (green color) and, distributed and replicated in 5 datanodes (yellow color) across two racks (curly brackets).

HDFS does not come with a DBMS but there are some open source technologies like Hive or Spark which provides basic DBMS support to handle tables schema metadata.

2. **Object Storage**. It is a key-value store where a flat list of objects is stored in a bucket. Each object contains data and is associated to an identifier, custom attributes and metadata. Object Storages are generally deployed in cloud environments and up to

date, the main providers are Amazon S3, Azure Blob Storage, Google Cloud Storage and Openstack Swift.

The use of an object storage allows a very basic interaction between an user and the remote storage through an REST API. The REST API uses standard HTTP requests to create, fetch, and delete buckets and objects [7].

Object Storage does not come with a DBMS either but there are paid services which provides basic DBMS support. For example, Amazon SimpleDB can be used to support a basic DBMS for storing metadata and for object metadata querying.

The selection of a data distribution framework depends on several requirements. Object Storage can be the right choice for: high throughput, cross-zone data durability, high data availability and cloud-based querying based on simple SPARQL queries as AMADA shows in [48]. Instead, HDFS could be the right option to follow for very intensive I/O data operations on on-premise[8] cluster installations [9] with support for metadata, directories, appending data, low latency and complex SPARQL queries.

Although a data distribution framework already provides parallelization and replication, I/O operations can be accelerated if data in each cluster node is distributed over more than one disk. RAID [10] is a virtualization technique that allows to run an operating system on top an array of disks as if they were a single one. RAID has 3 properties: *mirroring* to secure data by redundancy, *striping* to split data in different disks, and *parity* that uses a calculated value to restore eventually lost data from still available disks.

In RAID, the different disks arrangements are denominated *levels*. The best known levels are 0,1,5,6 and 10. Level 0 enables mirroring, Level 1 enables striping and Level 10 combines both. Levels 5 and 6 add parity.

The selection of the proper RAID level depends on the critical feature to be tackled by the architecture. If the critical feature is resilience to disk failure, then best choice is Level 5 or 6 although the controller is expensive. If the critical feature is I/O acceleration then the best choice is Level 10.

4.1.2 Storage

The storage format refers to how a data table is stored physically: row or columnar. In a row storage, table rows are stored contiguously keeping the order of the attributes values in each row. This type of storage is widely used for read/write single-record operations when

[7] https://docs.aws.amazon.com/AmazonS3/latest/userguide/developing-rest-api.html
[8] It refers to an installation under the infrastructure of the person or organization
[9] https://cloud.google.com/architecture/hadoop/migrating-apache-spark-jobs-to-cloud-dataproc
[10] It stands for "Redundant Array of Inexpensive Disks"

data changes at very fast pace : for example, to select all attributes of a record or to update a particular attribute value as in an Online Transaction Processing (OLTP) database.

In a column storage, table columns are stored contiguously. Column storage is used for read/write batch operations when data changes very slow. A frequent use case is to aggregate some attributes from a large amounts of records as in an Online Analytical Processing (OLAP) database.

ORC and Parquet are column storage formats widely used to handle large volumes of data with Big Data technologies. Whereas ORC is highly optimized to work with Hive, Parquet offers high data compression rate and performs better when works with Spark. In recent years, with the rise of Spark as leading cluster-based query engine, Parquet appears to have become the preferred option for columnar storage formats.

There are three optimization techniques that Spark can apply when works with Parquet: Partitioning pruning, Predicate push down and Min/max statistics.

Partitioning pruning is a technique to reduce input data size before query execution. In partitioned pruning, partitioned data is organized in a hierarchy of directories and subdirectories. The data is contained in each leaf directory. Based on a given filter, it allows to Spark to retrieve data from specific folders and skip the rest.

In Predicate push down, Spark filters query predicates based on metadata stored in the Parquet files. The filtering operations are tunneled into the scan operator which reads data before query execution. In Min/max statistics, data is skipped based on min/max value statistics available in each row group.

4.1.3 Query processing

Query processing can be distributed if it lies on a robust cluster programming model that abstracts away critical networking issues like data replication, input/output operations, fault tolerance, load balancing and data serialization.

- **MapReduce**. It is a programming model to process large datasets presented in [49]. The pipeline is a collection of succesive operations where the shared data between intermediate steps are key/value pairs obtained from a reduce operation to all the values that shared the same key. Although MapReduce can emulate many real world tasks, the main drawback is that its computing intermediate steps requires disk access to read/write.

- **Resilient Distributed Framework (RDD).** It is an unified cluster programming abstraction able to emulate almost any cluster programming model (e.g. MapReduce, Pregel, etc) [50]. RDD enable *composed* computations, an efficient in-memory data sharing across parallel computation stages.

> RDD is an efficient programming model for batch analysis and streaming based on micro-batches. A RDD pipeline is made of transformations and actions. A transformation is a lazy operation that only defines a new RDD but without computing it. Instead, an action triggers a computation which will return a value or will write data in the storage. Although RDD is a very versatile abstraction, it is not recommended for applications that makes asynchronous fine-grained updates over a shared state[50].

MapReduce was implemented by Apache Hadoop and although is still available, its usage have decreased dramatically in the last years due to its inability to share in-memory between intermediate steps. Instead, the Spark implementation of RDD, can emulate MapReduce operations and has become the facto standard for distributed query engines. Spark also offers high level datastructures like Dataframes and Datasets on top of RDD to deal with structured data. The dataframes are the core of Spark-SQL and all their optimizations are available for SPARQL queries if they are translated into SQL.

Regarding querying workload, SPARQL queries can be classified into 2 groups: *CPU Bound* and *I/O Bound*. *CPU bound* refers to queries which execution depends far more on the CPU, whereas the other components of the computer system are almost not used. *I/O bound* refers to queries that executes a large amount of read/write operations from/to disks or peripheral devices. To the best of our knowledge, much research has been done on *CPU Bound* queries (e.g. read queries) rather than *I/O Bound* queries (e.g. update queries), and no research on queries involving both cases (e.g. forward chaining reasoning).

The use of GPUs can improve the performance for *CPU Bound* queries by increasing dramatically the number of parallel operations with thousands of cores. A GPU does not replace a CPU in a hardware system. Instead, it is an additional *device* which allow to run a query plan through different stages, that are executed either on the CPU (host) or the GPU (device).

The addition of a GPU could mean the redesign of a software from the ground up to ensure efficient use of GPU hardware while catching up latest improvements at the same time [51]. Fortunately, Spark can also provide GPU acceleration via RAPIDS, a external library based on cuDF and Apache Arrow, a columnar memory format. In this way, a Spark program can run in a GPU-based cluster without any change.

Regarding *I/O Bound* queries, the performance depends on the implementation of a direct data path between the storage and the GPU memory to increase I/O bandwidth (e.g. GPUDirect Storage). The potential bottlenecks when working with GPUs at scale have been are outlined in [51, 52]. For example, the addition of a GPU could degradate the overall performance when in-memory data have to be spilled into the disk because it does not fit into the GPU global memory, or when there are many intermediate results and the data transference among the host and the GPU device is slow.

Despite the GPU performance in Spark is limited by the amount of data that can be loaded in the GPU memory, the benefits that the use of GPUs brings to CPU bound queries largely pay the eventual shortcoming. Further, the suitable use of partitioning pruning can avoid the bottleneck produced by I/O Bound queries when reducing the size of data to be read.

4.1.4 Transactional metadata layer

In traditional DBMS, a transaction is an unit of work that consists of a sequential execution of SQL operations. There are three basic commands to group SQL statements inside a transaction:

- BEGIN TRANSACTION, to start a transaction.
- END TRANSACTION, to commit (save) updates, and
- ROLLBACK, to undo changes.

A transaction is characterized by four properties to guarantee consistency and correctness in data operations. Those properties are better know by the acronym ACID, which stands for:

- Atomicity, all transaction items are processed or none.
- Consistency, data remains in consistent state after transaction.
- Isolation, no interference between multiple transactions, and
- Durability, changes produced by transactions are preserved even after system failure.

A Big RDF triplestore requires ACID support to ensure consistency and data integrity for update operations. As of today, transactions support for Big Data is still basic and it is not fully featured like in traditional DBMS.

Big Data transactions have not commands to group SQL statements. Instead, a transaction is triggered whenever an operation (INSERT, UPDATE or DELETE) is performed upon a table with ACID support, hereafter known as an ACID table. Once a transaction is initiated, is broken down into its components parts: *actions*. A *commit* is defined as an *action* recorded in a transaction log which in turn, it is stored along with the data.

In this regard, we offer an illustrative example. Suppose that a column is added to an ACID table and then, some data is populated in the new column. This process would result in two commits being added to the transaction log: a) Updating the table's metadata to track

the addition of a new column to the table schema, and b) Adding a file containing the new data to the table. This ensures that the addition of the new column and the insertion of data into it are performed atomically and consistently, providing ACID guarantees for the operation.

Big Data transactions are supported for only a single table instead of many as occurs in traditional DBMS. Regarding isolation levels, traditional databases offer four levels of isolation, namely Read uncommitted, Read committed, Repeatable read, and Serializable. In contrast, Big Data systems prioritize scalability and performance over strict consistency guarantees, and therefore, offer a limited number of isolation levels. The two isolation levels provided by Big Data systems, Serializable and WriteSerializable, are designed to ensure high consistency in transactions, with WriteSerializable providing stronger guarantees than Serializable. This approach applies the stronger isolation level only to write transactions, allowing for improved performance and concurrency for read transactions. Despite the limited number of isolation levels, both Serializable and WriteSerializable provide strong consistency guarantees for write transactions, ensuring that the data is always reliable in Big Data environments.

There are two ways to implement an ACID table in Big Data systems, either using Hive with ORC ACID storage format or using Spark with Delta Lake[53] metadata layer. In both cases, the transactions metadata are logged along with the data. The main drawback of ORC ACID is that data can only be read by Hive and it is not available for other distributed frameworks like Spark. Conversely, Delta Lake adds metadata management for ACID service to Spark by: a) using a transaction log for keeping track of all the updates made to the ACID table and b) using time travel for loading the ACID table at a given version or timestamp [54].

To the best of our knowledge, ACID tables have not been considered for any Big RDF triplestore so far. The implementation of ACID tables restricts the data layout of a Big RDF triplestores to a single table layout. Although the experiments reported in [55] have shown that single table layout is outperformed by vertical partitioning and property tables, single table layout remains as the dominant layout in real-world deployments (e.g. Virtuoso).

The implementation of non-single layouts is a time consuming task that requires data normalization and query rewriting. In addition, updates are easier in a single table layout because it does not need to be propagated to other tables like in other layouts.

4.1.5 Reasoning

There are no Big Data technologies specifically designed for reasoning processes. Instead, existing technologies are repurposed to execute such processes. Spark-based reasoners have been explored in a limited capacity, as there are only a few examples in the literature [56, 57]. These reasoners typically perform either full or incremental materialization and use Spark

RDD operators to implement RDFS rules, which do not fully leverage the performance optimizations available for Spark Dataframes.

4.2 TESS architecture

TESS is a Big RDF triplestore with a modular architecture. Figure 4.3 shows the five layers of TESS. The modular architecture makes possible to disable some of them, like the transactional metadata layer or the data distribution if they are not useful, for instance if the CONSTRUCT queries are not used for updates or if data is not distributed. Also, modularity allows to add extra layers (e.g. streaming) or replace some of them (e.g. a graph analytics application instead of the reasoner).

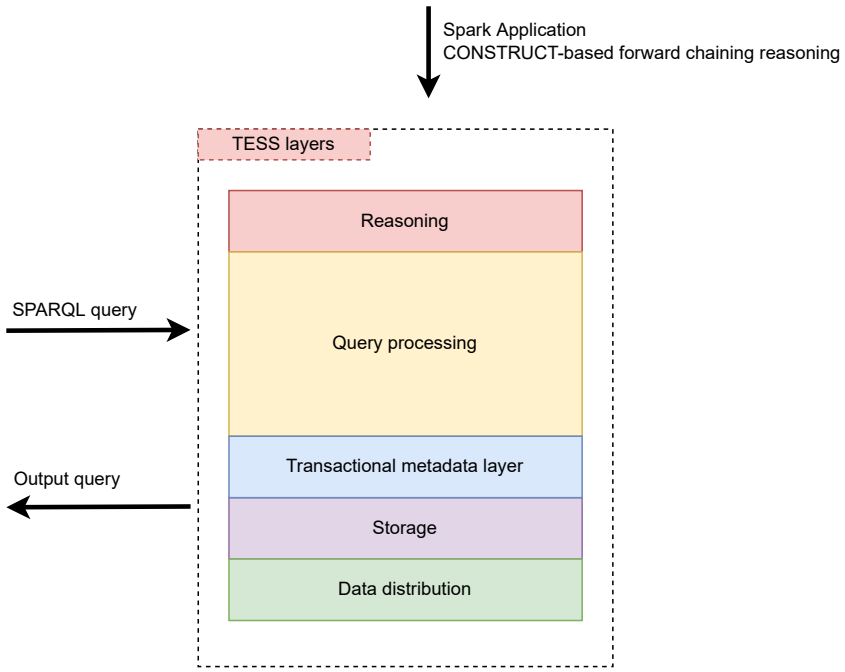

Figure 4.3 TESS triplestore architecture

TESS supports two inputs: A SPARQL query and a Spark Application for CONSTRUCT-based forward-chaining reasoning. Notice in Figure 4.3 that only SPARQL queries have external output. Instead, the outcome of the forward chaining reasoning is meant to be stored in the distributed storage for later querying.

The TESS technology stack is presented in Figure 4.4. It considers selected technologies on the basis of the reviews done in the previous section. At glance, it allows to visualize the distinguished technologies used by TESS: the Rapids GPU accelerator (part of the Query processing layer) and Delta Lake (which manages the Transactional Metadata layer).

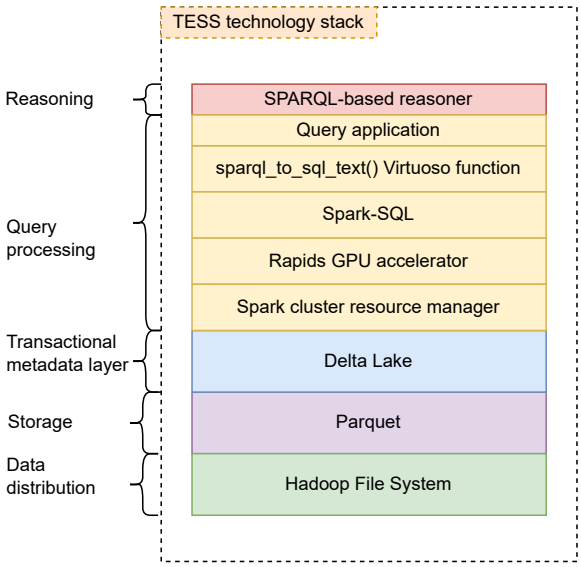

Figure 4.4 TESS triplestore technology stack

In the following, we review each layer of TESS along with its selected technologies.

4.2.1 Data distribution

We deal with a scenario where HDFS can run out of space due to lack of additional hardware to deal with the increasing size of a dataset. Looking for ways to save space, we found that the default HDFS replication scheme is expensive because it has 200% overhead in storage space [58]. After confirming that Big Data players like Facebook had identified the same issue [59], we decided to eliminate replicability by minimizing the number of HDFS nodes as much as possible to maximize available space.

Based on the proposed scenario and the review of previous section, we conclude that the best strategy for data distribution is to deploy HDFS on top of RAID. We believe that working together, data storage could be optimized by reducing the amount of storage that HDFS needs for replication[11]. However, since this deployment is not given by default, it will require an on-premise installation to customize the setup.

We used a *hybrid* approach to distribute data. On this approach, a minimal HDFS conformed by one namenode and one datanode is deployed on top of a collection of n disks merged into one unit using RAID 0. There is not replication to save space and, in case of failure, it uses an external backup service to recover data.

The ingestion of data is a process that lies on an external library for parsing and syntax checking. In general, there are two main Java-based libraries widely used in the semantic

[11] https://docs.cloudera.com/HDPDocuments/HDP3/HDP-3.1.4/data-storage/content/increasing_storage_capacity_with_hdfs_erasure_coding.html

web community: Apache Jena[12] and Eclipse RDF4J[13]. Apache Jena is used by other Spark-based RDF frameworks like Sansa and Bellman, but we discarded it because the parsing of a single RDF string requires the creation of a bunch of auxiliary in-memory objects that end up hitting loading performance [60]. Instead, we developed a Spark library[14] on top of Eclipse RDF4J. The library parses n-triples and quads by column chunks instead of single rows to increase loading speed.

Data is ingested into a directory tree that needs periodical maintenance. An example of maintenance task is the recreation of an outdated data partition without need to recreate the full dataset. HDFS maintenance is performed using the File System (FS) shell. Each FS command bears a close ressemblance with a Linux command and it receives a path URIs as arguments to act over a directory. The format of the URIs is *scheme://authority/path* and allows to access all the subdirectories under that path.

The simple access to the HDFS data through a path URI also facilitates the connection to other Big Data applications. For example, an user can use a popular visualization tool like Tableau[15] to connect to the HDFS, launch some SQL queries, and present the results in statistical dashboard. As another example, an user can use a distributed query engine like Presto [16] to query data accross multiple datasources, one of which is HDFS.

4.2.2 Storage

We conceptualize storage as a customizable layer composed of a directory structure, a storage format and a logical layout. The storage aims to provide swiftest access to that portion of the data that participates in a query.

The directory structure depends on a partitioning operation carried out by Spark at loading time. Partitioning creates as many directories as RDF predicates there are in the data to enable an operation called *partitioning pruning*.

Partitioning pruning is essential because it reduces the input data size for queries. This is crucial for self-join queries that run over multiples copies of a same large table. In that case, partitioning pruning enables the replacement of the full version of a self-join table by a reduced version retrieved from a partition. We use this feature for self-join queries resulting from a SPARQL/SQL translation.

Data is saved in Parquet storage format. At loading time, data is splitted by Spark in default Parquet blocks of 128 MB size each. We kept the default block size in 128 MB for two reasons. The first reason is that the size of a Parquet block should not be larger than

[12] https://jena.apache.org/
[13] https://rdf4j.org/
[14] https://github.com/asanchez75/thesis_experiments_source_code
[15] https://www.tableau.com/
[16] https://prestodb.io/

the size of a HDFS block size that is also 128 MB. A mismatch between both sizes hits performance. If the size of a Parquet block is bigger than a HDFS block, it would have to be stored in many HDFS blocks. This could slow down the retrieval of a Parquet block because it would have to be performed from many HDFS blocks located at different HDFS nodes. The second reason is that this Parquet block size is the minimal recommended value to keep parallelism in very large datasets. A lower value is costly, because it increases both the number of blocks to process and the metadata size to handle in memory. Conversely, a higher value could hit parallellism by processing large blocks sequentially.

The logical layout of the data is based on a single table with a schema of four columns (<s, p, o, g>) to store RDF quads. This table is referred to as the main storage table. Before being stored, RDF data is encoded into long integers using a Spark implementation of the XXhash64 algorithm. The encoding improves the speed of in-memory operations and reduces storage size. The XXhash64 encoding algorithm is fast because it processes data at RAM speed limits [61].

4.2.3 Transactional metadata layer

We use Delta Lake to add ACID properties to the main storage table. Data is updated directly in the main storage table but when the update is very large to fit in memory, an auxiliary ACID table is added as intermediate storage to guarantee data consistency. We do not use the default data spilling into disk provided by Spark when data is larger than available memory. Such data can be corrupted during data spilling.

Figure 4.5 shows an example where the output of a query is saved in a temporary ACID table when the output is very large to be handled totally in memory.

In such as example there are 4 steps. In the first step, the main table containing RDF data is loaded from HDFS storage to RAM memory. At this step, Spark reads all the changes recorded in the transaction log to build a versions history. When reading is finished, Spark loads the latest data version.

Next, the SPARQL query is translated into SQL query and executed on Spark-SQL.

In the third step, the large query output is saved in a temporary ACID table to guarantee data reliability. The temporary ACID table has its own transaction log that we can use in case of error debugging. Finally, in step 4, the temporary data is added to the main ACID table and a new entry is added to the transaction log to record the operation.

In this layer, the ACID properties enforce data reliability as follows:

- **Atomicity.** Data is saved complete as commit or it is not saved. Partial saving is rejected.
- **Consistency.** The ACID table rejects update operations that do not match its data schema. This is called Schema enforcement. The rejection occurs if data to be

added: a) has a bigger number of columns that the target schema of the table. b) contains columns which datatypes are different from the target table. c) the name of the columns does not coincide exactly because the ACID table is case sensitive for column names.

- **Isolation.** Each operation running in parallel works over an in-memory snapshot of the data. To ensure the order of the changes during commit, it uses an optimistic concurrency control algorithm under the assumption that conflicts are rare.
- **Durability.** It is supported on the basis of a transactions log that are read when data is loaded. Delta Lake enables Spark to perform some metadata management to reduce a huge list of transaction logs by performing checkpoint at the current state of a table.

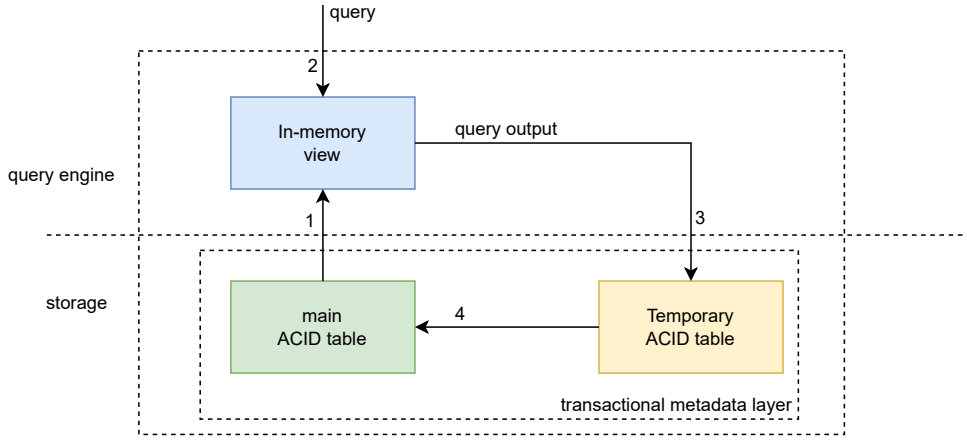

Figure 4.5 TESS transactional metadata layer

4.2.4 Query processing

We select Spark for query processing because: a) it is fault tolerant, b) SPARQL queries can be executed by Spark-SQL if they are translated into SQL, c) it keeps in memory shared data between intermediate steps, d) it supports GPU, and e) it is extendable with Scala programs.

This layer consists of 4 components: a cluster resource manager, a GPU accelerator interface, a Spark-SQL query engine and a SPARQL/SQL translator.

Cluster resource manager. This component allows to distribute the execution of a query over a standalone Spark cluster. It comprises a master and workers nodes, usually as many workers as queries/rules to manage with.

The cluster manager (master) receives Spark applications and *schedules* worker resources to be run among them. A Spark application is organized around jobs, the top level

work unit. By default, Spark jobs within an application are executed serially, but they can also be run in parallel if concurrency is enabled at application level.

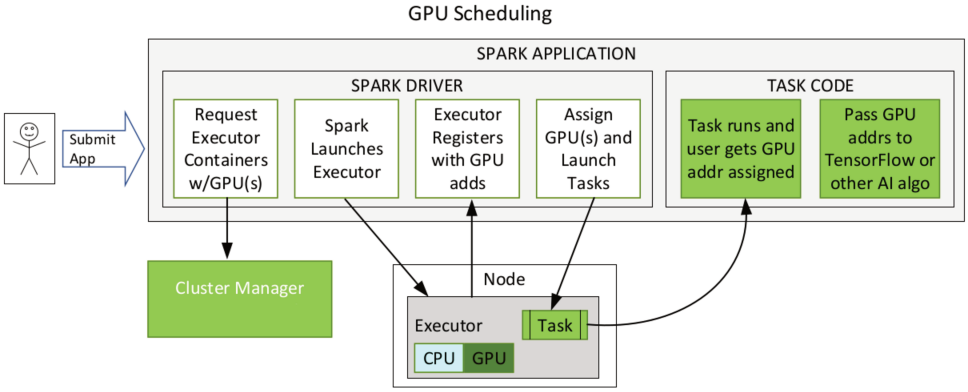

Figure 4.6 GPU scheduling [3]

Figure 4.6 shows the scheduling of GPU resources. It assumes that there is a Spark App to be submitted to a Spark Standalone cluster. It starts from the left. Once a Spark Application is submitted, a Spark driver along with a collection of worker nodes are initialized. Internally, the driver coordinates with the cluster manager the availability of nodes with GPU support. If there are available nodes, they are registered and the driver launches the tasks.

Notice that, as of Spark 3.1.1, a worker is restricted to use only one GPU at a time.

GPU accelerator interface. This component enables access to GPU resources to process data in columnar batches. It picks out those operators from physical query plan that can be GPU accelerated. Next, it generates a GPU physical plan as shown in the example of Figure 4.7.

We use a qualification tool [17] to evaluate the potential GPU performance gain for a given SPARQL query translated into SQL. The operators best suited for GPU optimization querying are: group by, joins, sorts with high cardinality. In case an operator is not GPU supported, the operator fallback to its Spark CPU version.

Spark-SQL. This component processes the self-join queries produced by the SPARQL/SQL translator and select the best join strategy according the size of the tables. For example, if one table is very large and other very small -due to partitioning pruning for example- it will use broadcast join to perform the join operation.

As example, Figure 4.8 shows how a query adapts from a sort-merge join to a broadcast join in execution time. In the first step, Spark-SQL estimates the sizes of the tables. In the second step, it computes the actual size of both tables and finds that one of them is

[17] https://nvidia.github.io/spark-rapids/docs/spark-qualification-tool.html

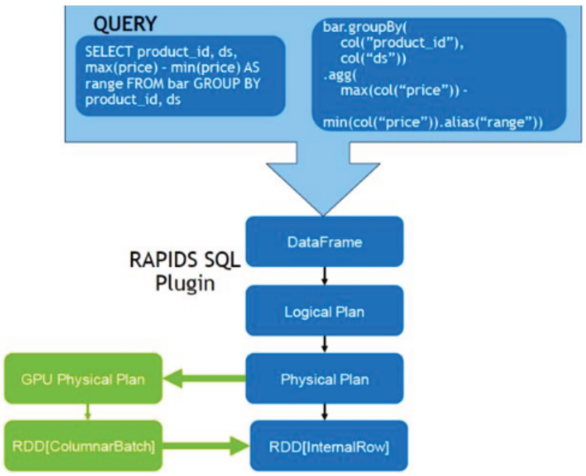

Figure 4.7 GPU columnar data processing [3]

smaller. As result, it chooses to perform broadcast in the third step. During broadcast join, Spark-SQL will send the smaller table to all the nodes of the cluster to perform the join without need of data redistribution (shuffling).

Figure 4.8 Adaptive query execution [4]

SPARQL/SQL translator. This component rewrites SPARQL queries into SQL in order to use Spark SQL, the Spark module for dataframe-based structured data processing.

The translated queries are generated assuming a single layout table, a restriction imposed by the transactional metadata layer that only has ACID support for one single table. We retained sparql_to_sql_text() Virtuoso [62] function to generate queries (self-join) for the single ACID table. The translated queries can be edited manually to adjust some query strategy.

Query Application.

This component is responsible for processing SPARQL queries. First, it translates a query into SQL using the SPARQL/SQL translator. Next, it parses the query and encodes all URIs and filter values using long integers, as described in the 4.2.2 Subsection. It then executes the query using the Spark-SQL component. Finally, the Query Application component retrieves the result from the Spark-SQL component and returns the output to the user after decoding the long integers into strings.

If the query is a (Quad) CONSTRUCT query, an additional step is required. Upon receiving the query, this component extracts the induced SELECT query and its corresponding template, as described in the Subsection 2.8 . The SELECT query is then executed by the Spark-SQL component. Next, the template is instantiated multiple times, equal to the number of rows in the query output provided by the Spark-SQL component. Finally, the Query Application component returns the union of all instantiated templates to the user.

In TESS, we faced the challenge of providing unlimited output size for CONSTRUCT queries, a problem that had previously defeated Virtuoso. The issue with Virtuoso was that it loaded output rows into a vector of fixed and monolithic size, limited to 1 million rows. To overcome this, we replaced the vector with a column data structure that is parallelizable and can be distributed in chunks across the cluster. This approach enabled TESS to handle CONSTRUCT queries of any output size by exploiting hardware capabilities at maximum.

4.2.5 Reasoning

This layer executes serial and parallel forward chaining reasoning for rules encoded as CONSTRUCT queries. The serial forward chaining is an implementation of the algorithm presented in Figure 2.9 that executes the rules ordered by the topological order of the dependency graph. The parallel forward chaining is an implementation of the algorithm depicted below in Figure 4.9. The algorithm iterates over a group of incrementally deepening layers, computing the rules that compose each layer in parallel and in any order.

We implemented a Spark reasoning application [18] to automate the forward chaining reasoning described above. This program uses the Query Application presented above to execute rules encoded as CONSTRUCT queries. After each rule computation, the program stores the data in the HDFS whether the transaction has been validated in the transactional metadata layer.

Forward chaining reasoning requires to ensure consistency and data integrity for update operations during each iteration. We support ACID properties for each iteration as follows:

- The complete output is saved or none (atomicity),

[18] https://github.com/asanchez75/thesis_experiments_source_code

Algorithm 2 Parallel

Input: Dataset D, N layers of Q_N CONSTRUCT queries each
Output: Saturated dataset $D^{'}$

$D^{'} \leftarrow D$
for $i \leftarrow 1$ *to* N **do**
    $m \leftarrow Q_N$
    for $j \leftarrow 1$ to m **do in parallel**
    $output \leftarrow Sparql(q_j, D^{'})$
    $D^{'} \leftarrow D^{'} \cup output$ `// update operation`
end

Figure 4.9 CONSTRUCT-based parallel forward chaining algorithm

- The output is saved only if data schema is validated (consistency),
- Parallel rules do not interfere each other (isolation) and
- once output is saved, it is safely persistent (durability).

Figure 4.10 shows how forward chaining works when it is computed in parallel. The queries of the same layer are computed in parallel and the output is saved in temporary ACID table to avoid memory issues. Next, all the outputs are stored in main ACID table.

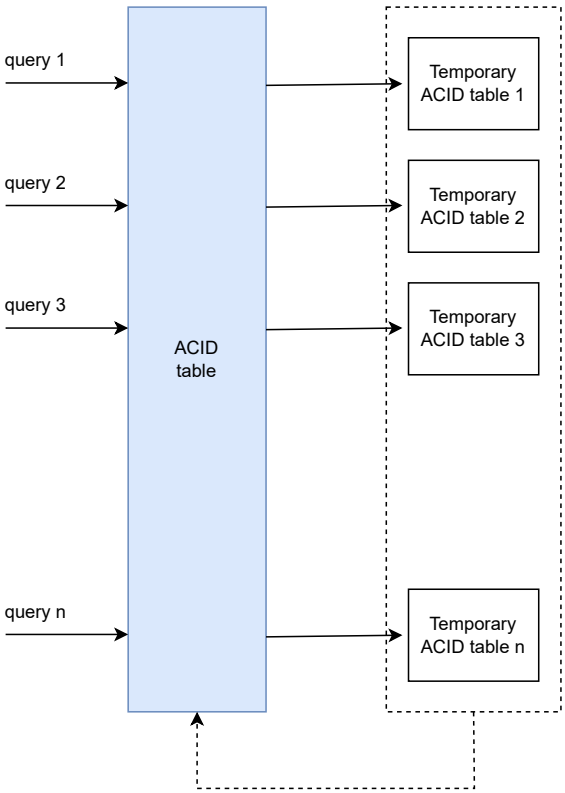

Figure 4.10 TESS parallel forward chaining

4.3 Summary

The performance of a Big RDF triplestore depends on how their components interplay together. Although the selection of the components could be overwhelming due to the amount of criteria to consider, the initial choice of Spark as essential component narrows the scope of the selection.

The hardware for Big Data evolves faster than software. To catch up latest hardware improvements, the software must be upgraded or extended as soon as possible. Spark has adapted quite well to these new challenges and shows GPU support as its latest achievement.

Spark reaches high performance if input data is reduced to that of portion of the data that participates in a query. Data partitioning improves performance by reducing input data size. Keeping in mind that input data size must be as minimal as possible, also prevents slow data transference between nodes and RAM memory issues.

TESS is distinguished from other Spark-based frameworks because it provides ACID support for update queries and forward-chaining reasoning based on rules encoded as CONSTRUCT queries. However, it should be noted that TESS is not a true ACID-compliant and transactional DBMS service. It is designed to ensure data consistency in some internal

operations rather than at the user level.

TESS is based on a modular architecture that supports metadata management for log-based transactions that track data updates.

Chapter 5

Comparative performance evaluation of complex queries and reasoning

We use TESS in conjuction with state-of-the-art triplestores such as Virtuoso and GraphDB to evaluate the performance of complex queries and forward chaining reasoning. The aim is to study how CONSTRUCT query performance is affected by the growing size of the input RDF datasets.

In the absence of appropriate benchmarks (for CONSTRUCT queries or for SELECT queries on large knowledge graphs), we decided to conduct our performance evaluation on the OntoSIDES knowledge graph, the size of which (12 billion triples) is comparable to that of Wikidata (14 billion triples as of 2020) and DBpedia (21 billion triples as of 2021).

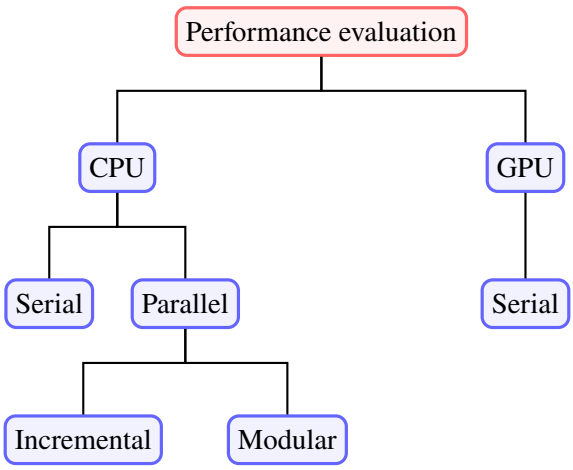

Figure 5.1 Performance evaluation

For the evaluation, we consider a collection of SPARQL CONSTRUCT queries that cover a variety of SPARQL 1.1 features. The performance of forward chaining reasoning is

evaluated in the context of several implementations, depending on the type of hardware (cpu or gpu), parallel rules execution, incremental data updates, and data modularization. For data modularization, we replace the OntoSIDES knowledge graph with a union of named graphs so that forward chaining can be done in parallel on independent groups of named graphs, called *modules* as presented in Section 3.2.

Figure 5.1 shows a tree of comparative performance evaluations of forward chaining implementations:

- Serial versus Parallel. The performance of the implementations of the serial and parallel algorithms presented in the subsection 4.2.5 are compared.
- Parallel versus Incremental. The performance of the implementation of the parallel algorithm of subsection 4.2.5 is compared against a modified version handling incremental data updates.
- Parallel versus Modular. The performance of the implementation of the parallel algorithm of subsection 4.2.5 is compared against an adapted version that takes advantage of data structuration in modules.
- CPU Serial versus GPU Serial. The performance of the implementation of the serial forward chaining algorithm presented in subsection 4.2.5 is compared when running on either CPU or GPU.

In this chapter, after presenting in Section 5.1 the experimental protocol that we have followed, we report in Section 5.2 the experimental results we obtained for assessing the performances of both complex (SELECT and CONSTRUCT) queries evaluation and the serial algorithm of forward chaining reasoning based on the iteration of CONSTRUCT queries. The results show that Virtuoso and GraphDB do not scale to large RDF datasets, in contrast to TESS, which outperforms them on all the datasets. Then, Section 5.3, Section 5.4 and Section 5.5 are respectively dedicated to the comparisons of the runtime performance on TESS between the serial and parallel, parallel and incremental, parallel and modular variants of the forward chaining algorithm. Finally, in Section 5.6, we report on experimental results for comparing the performances of TESS when using CPU or GPU for running the serial algorithm of forward chaining reasoning based on the iteration of CONSTRUCT queries.

5.1 Experimental protocol

We first describe in Section 5.1.1 the queries that we have used in our experiments and we explain in Section 5.1.2 how we have built the different RDF datasets of increasing size over

which the queries have been evaluated. In Section 5.1.3, we define the different measures that we use for evaluating the performance for evaluating queries in isolation as well as for the whole process of forward-chaining reasoning.

5.1.1 CONSTRUCT queries used as rules in OntoSides

As explained in Chapter 3 (Section 3.1.1.2), 18 CONSTRUCT queries are used to specify rules for defining some properties in function of other properties. They are recalled in Figure 5.2.

These 18 queries cover a variety of SPARQL 1.1 features that are rarely encountered in the existing benchmarks used for experimentally evaluating SPARQL queries.

- Only 6 of them (Q5,Q8,Q9,Q11,Q16,Q18) are *simple conjunctive queries* possibly with FILTER conditions.

- The other queries are *complex* queries that can be categorized as follows:

- 3 aggregate queries (Q1, Q2 and Q3) used to count some useful numbers.
- 10 queries with negative conditions expressed using the FILTER NOT EXISTS constructor, among which
  - 7 queries (Q6, Q7, Q12, Q13, Q14, Q15, Q17) have single negative conditions (corresponding to FILTER NOT EXISTS expressions restricted to a single triple patterns)
  - 3 queries (Q3, Q4 and Q10) have multiple negative conditions with graph patterns of size 2 in their FILTER NOT EXISTS expressions.

Q3 combines GROUP BY and FILTER NOT EXISTS constructors and thus appears in the two corresponding rows in Table 5.1.

Category	Queries
Simple	Q5,Q8,Q9
Aggregated	Q1,Q2,Q3
FILTER on terms (FRT)	Q11,Q16,Q18
FILTER NOT EXISTS on graph patterns (FGP)	Q3,Q4,Q6, Q7,Q10,Q12, Q13,Q14,Q15, Q17

Table 5.1 Query classification by category

The size of the graph pattern (i.e., the number of triple patterns) in the query bodies is also likely to impact the complexity of the query evaluation. It is distributed as follows:

Q1

```
CONSTRUCT  {
?question sides:has_for_number_of_proposals ?np}
WHERE {
SELECT ?question (COUNT (?p) As ?np)  {
?question sides:has_for_proposal_of_answer ?p}
GROUP BY ?question}
```

Q2

```
CONSTRUCT {
?answer
sides:has_for_number_of_wrong_tick ?nw }
WHERE {select ?answer (COUNT (?a) As ?nw)
{?a sides:is_part_of  ?answer.
?a sides:has_wrongly_ticked ?p}
GROUP BY ?answer
}
```

Q3

```
CONSTRUCT {
?answer
sides:has_for_number_of_missed_right_tick ?nm}
WHERE {SELECT ?answer (COUNT(?p) As ?nm)
{?answer sides:correspond_to_question ?q.
?q sides:has_for_proposal_of_answer ?p.
?p sides:has_for_correction "true"^^xsd:boolean.
FILTER NOT EXISTS {
?a sides:is_part_of  ?answer.
?a sides:has_rightly_ticked  ?p}
} GROUP BY ?answer
}
```

Q4

```
CONSTRUCT {
?answer sides:has_for_number_of_discordance
        "0"^^xsd:integer}
WHERE {
?answer a sides:answer.
FILTER NOT EXISTS  {
?answer sides:has_for_number_of_wrong_tick ?nw.
?answer sides:
        has_for_number_of_missed_right_tick ?nm.
}}
```

Q5

```
CONSTRUCT {
?answer
sides:has_for_number_of_discordance ?count}
WHERE {
SELECT  ?answer  (?nw + ?nm as ?count)   {
  ?answer
  sides:has_for_number_of_wrong_tick ?nw.
  ?answer
  sides:has_for_number_of_missed_right_tick ?nm
}
}
```

Q6

```
CONSTRUCT {
?answer
sides:has_for_number_of_discordance ?nw
} WHERE {
?answer sides:has_for_number_of_wrong_tick ?nw.
FILTER NOT EXISTS {
?answer sides:
        has_for_number_of_missed_right_tick ?nm}
}
```

Q7

```
CONSTRUCT {
?answer
sides:has_for_number_of_discordance ?nm}
WHERE {
?answer sides:
        has_for_number_of_missed_right_tick ?nm.
FILTER NOT EXISTS  {
?answer sides:has_for_number_of_wrong_tick ?nw.
         }
}
```

Q8

```
CONSTRUCT {
?answer sides:has_for_result 1}
WHERE {
?answer
sides:has_for_number_of_discordance "0"^^xsd:
        integer
}
```

Q9

```
CONSTRUCT {
?answer sides:has_for_result "0"^^xsd:integer  .
?answer sides:stronglyWrong  "true"^^xsd:
        boolean .}
WHERE {
?a sides:is_part_of  ?answer.
?a sides:has_wrongly_ticked ?p.
?p
sides:has_for_weight_of_correction
"Unacceptable"^^xsd:string .
}
```

Q10

```
CONSTRUCT {
?answer sides:has_for_result "0"^^xsd:integer  .
?answer sides:stronglyWrong  "true"^^xsd:boolean .
         }
WHERE {
?answer sides:correspond_to_question ?q.
?q sides:has_for_proposal_of_answer ?p.
?p sides:has_for_correction "true"^^xsd:boolean .
?p sides:has_for_weight_of_correction "
        Indispensable"^^xsd:string .
FILTER NOT EXISTS {
?a sides:is_part_of  ?answer.
?a sides:has_rightly_ticked  ?p
}}
```

Q11

```
CONSTRUCT {
?answer sides:has_for_result "0"^^xsd:integer  .
?answer sides:stronglyWrong  "true"^^xsd:boolean .
         }
WHERE {
?answer sides:correspond_to_question ?q.
?q rdf:type sides:QUA.
?answer sides:has_for_number_of_discordance ?d.
FILTER (?d > 0)
}
```

Q12

```
CONSTRUCT {
?answer sides:has_for_result 0.5^^xsd:decimal}
WHERE {
?answer sides:has_for_number_of_discordance
        "1"^^xsd:integer .
?answer sides:correspond_to_question ?q.
?q sides:has_for_number_of_proposals "5"^^xsd:
        integer.
FILTER NOT EXISTS {?answer sides:
        stronglyWrong  "true"^^xsd:boolean }
}
```

Q13

```
CONSTRUCT {
?answer sides:has_for_result "0.2"^^xsd:decimal}
WHERE {
?answer sides:has_for_number_of_discordance
        "2"^^xsd:integer .
?answer sides:correspond_to_question ?q.
?q sides:has_for_number_of_proposals "5"^^xsd:
        integer .
FILTER NOT EXISTS {
?answer sides:stronglyWrong  "true"^^xsd:boolean }
}
```

Q14

```
CONSTRUCT {
?answer sides:has_for_result "0.425"^^xsd:decimal}
WHERE {?answer sides:
        has_for_number_of_discordance "1"^^xsd:
        integer .
?answer sides:correspond_to_question ?q.
?q sides:has_for_number_of_proposals "4"^^xsd:
        integer .
FILTER NOT EXISTS {?answer sides:
        stronglyWrong  "true"^^xsd:boolean }}
```

Q15

```
CONSTRUCT {
?answer sides:has_for_result "0.1"^^xsd:decimal  }
WHERE {
?answer sides:has_for_number_of_discordance
        "2"^^xsd:integer .
?answer sides:correspond_to_question ?q.
?q sides:has_for_number_of_proposals "4"^^xsd:
        integer .
FILTER NOT EXISTS {
?answer sides:stronglyWrong  "true"^^xsd:boolean
        }}
```

Q16

```
CONSTRUCT {
?answer sides:has_for_result "0"^^xsd:integer}
WHERE {
?answer sides:correspond_to_question ?q.
?q sides:has_for_number_of_proposals ?np.
?answer sides:has_for_number_of_discordance ?n.
FILTER (?np > 3 && ?np < 6 && ?n > 2).}
```

Q17

```
CONSTRUCT  {
?answer sides:has_for_result "0.3"^^xsd:decimal}
WHERE {
?answer sides:has_for_number_of_discordance
        "1"^^xsd:integer .
?answer sides:correspond_to_question ?q.
?q sides:has_for_number_of_proposals "3"^^xsd:
        integer .
FILTER NOT EXISTS {
?answer sides:stronglyWrong  "true"^^xsd:boolean }
}
```

Q18

```
CONSTRUCT {
?answer sides:has_for_result "0"^^xsd:integer }
WHERE {
?answer sides:has_for_number_of_discordance ?n.
?answer sides:correspond_to_question ?q.
?q sides:has_for_number_of_proposals "3"^^xsd:
        integer.
FILTER (?n > 1)  }
```


Figure 5.2 18 CONSTRUCT queries over OntoSIDES knowledge graph

- Q1, Q4, Q7, Q6 and Q8 have a graph pattern reduced to a single triple pattern
- Q2 and Q5 have a graph pattern with 2 triple patterns
- Q3, Q9, Q10, Q11, Q12, Q13, Q14, Q15, Q16 and Q17 have a graph pattern with 3 triple patterns
- Q10 has a graph pattern with 4 triple patterns

Finally, the size of the output template may also impact the performances. As summarized in Table 5.2, in the considered CONSTRUCT queries, the output template can contain 1 or 2 triple patterns.

Template size	Queries
1	Q1,Q2,Q3,Q4,Q5, Q6,Q7,Q8,Q12,Q13, Q14,Q15,Q16,Q17, Q18
2	Q9,Q10,Q11

Table 5.2 Query classification by output template size

When used for implementing forward-chaining reasoning, the 18 CONSTRUCT queries (encoding rules with negation) must be ordered to take into account that the evaluation of some queries must occur after the update of the dataset by the evaluation of other queries. Such an ordering has been determined in Chapter 3 (Section 3.1.3), where it has been shown that, based on their dependency graph (shown in in Figure 3.8, Chapter 3, Section 3.1.3), the 18 CONSTRUCT-based SPIN rules can be structured in 4 layers of increasing depth:

- Layer 1 = {Q1, Q2, Q3, Q9, Q10}
- Layer 2 = {Q4, Q5, Q6, Q7}
- Layer 3 = {Q8, Q12, Q11}
- Layer 4 = {Q13, Q14, Q15, Q16, Q17, Q18}

Since, there is no dependency between queries of a given layer, they can be evaluated independently within each layer. This structuration of the queries in layers can thus be exploited to implement a parallel forward-chaining reasoning algorithm that handles sequentially the layers (by ascending order of the depth) and evaluates in parallel the queries of each layer.

5.1.2 RDF datasets

Since the goal of the experiments is to measure the impact of the dataset size on the performances of query evaluation and reasoning, we have to build datasets of increasing

size on which the execution of the 18 queries described in the previous section can be compared meaningfully.

To achieve this, we utilized the concept of modularization introduced in Chapter 3 and structured the OntoSIDES knowledge graph (prior to saturation) as a collection of named graphs, where each named graph corresponds to a specific student's IRI. Each student's named graph contains RDF descriptions of all their answers, as well as the corresponding questions and enrollments obtained through the combination of their corresponding modules.

We have created 10 nested datasets obtained by grouping increasing numbers of students' named graphs (from 880 students' named graphs for the D1 dataset to 8845 students' named graphs for the D10 dataset). Since the obtained datasets contain RDF quads, we removed the fourth column to have RDF triples. Figure 5.3 shows the size of the resulting 10 datasets extracted from OntoSIDES.

Dataset	Students	Size (millions triples)
D_1	880	121
D_2	1760	194
D_3	2640	273
D_4	3520	380
D_5	4400	497
D_6	5280	633
D_7	6160	791
D_8	7040	977
D_9	7920	1209
D_{10}	8845	1604

Table 5.3 Ontosides datasets

By construction, each extracted dataset contains the required data for each of the 18 CONSTRUCT queries to produce a sound and complete result for the computation of the inferred properties on meaningful fragments of the full OntoSIDES knowledge graph.

In addition, since they are nested (i.e., D1 $\subset$ D2 $\subset$ D3 .... $\subset$ D10), query evaluation and reasoning over these 10 datasets provide (output and time performance) results that are monotonic. The goal of the experiments is to study their variations in a more fine-grained manner.

5.1.3 Performance measures.

For each CONSTRUCT query, in addition to measuring its execution time that we denote

the *construct execution time*, we will also measure:

- the *body execution time*, the time to evaluate its induced SELECT query (see Definition 2.8 in Chapter 2)

- the *template execution time*, the time to instantiate the template. Since triplestores do not provide the *template execution time*, we will compute it as the difference between the *construct execution time* and the *body execution time*

- the *construct storing time*, the update time needed to add the output of a CONSTRUCT query to the triplestore

- the *inference time*, the sum of the *construct execution time* and the *construct storing time*, which estimates the cost of a CONSTRUCT query used as an update rule.

Given the set of the 18 CONSTRUCT queries in Fig. 3.4 used as rules, and their structuration in 4 layers grouping queries that can be evaluated independently, we will also evaluate the performance of both *serial* and *parallel* implementations of CONSTRUCT-based forward-chaining reasoning.

The serial versus parallel implementations of CONSTRUCT-based forward-chaining reasoning differ in the sequential versus parallel execution of the CONSTRUCT queries within each layer, handled by increasing depth. The serial implementation is thus decomposed in 18 inference steps, whereas the parallel implementation is decomposed in 4 inference steps.

We will measure and compare:

- the *serial forward-chaining reasoning time*, as the sum of *inference times* of all the queries applied sequentially in the order induced by the different layers,

- the *parallel forward-chaining reasoning time*, as the sum of the parallel execution and update times for each of the 4 reasoning layers.

5.1.4 Hardware and software setup

The server used in our experiments has the following characteristics:

- Processor: 32 cores, Intel(R) Xeon(R) Gold 6144 CPU @ 3.50Ghz.

- Disk: 7 disks, 2 Terabytes size each.

- Memory: 566 Gigabytes RAM.

For our experiments, we used Spark 3.1.1, Delta Lake 0.8.0 and HDFS 2.9.2. Each dataset from Table 5.3 was stored in the HDFS as an ACID table.

All the software used in the experiments was containerized to simplify the startup and to package all dependencies and associated configuration. In this respect, we used Docker version 19.03.8 for containerization.

5.2 Serial forward-chaining reasoning performance

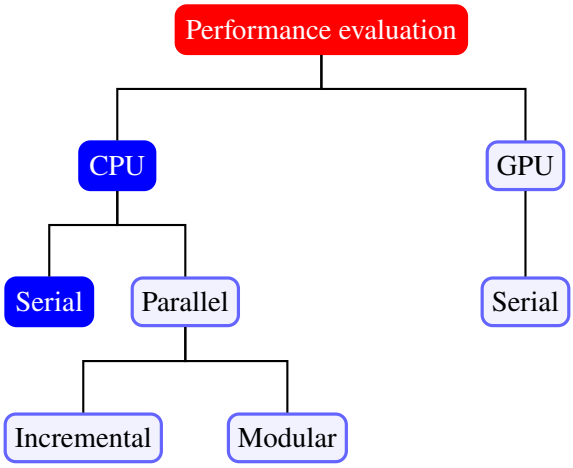

Figure 5.3 Serial performance comparison

In this section, we address serial performance comparison as shown in Fig. 5.3. We perform two experiments.

In the first experiment, we conduct a performance evaluation between Virtuoso, GraphDB and TESS. The results show the limitations of Virtuoso and GraphDB. We chose Virtuoso and GraphDB because they have different SPARQL querying processing. Virtuoso is a column-store triplestore where SPARQL queries are translated into SQL to be executed. GraphDB is a native triplestore where SPARQL queries are executed directly on data. Blazegraph, a self-described "Big Data" triplestore was not considered because it was outperformed by Virtuoso in 3 of 4 tasks in Mocha 2018 [63] (RDF data ingestion, data storage, versioning).

In the second experiment, we examine in detail the performance of TESS during serial forward chaining reasoning. First of all, we show the performance contribution of the body and template computation to the overall construct computation in terms of execution time. Finally, we go further and break down the TESS performance into the performance of the queries to determine individual performance contributions.

Software setup

For both experiments, TESS was configured to run on top of a network of 5 Docker containers: 2 containers for the Hadoop namenode and datanode. 2 containers for the Spark Standalone cluster (1 for the master node, 1 for the worker node) and 1 container for sending the Spark application to the Spark standalone cluster in client mode. We allocated 128GB RAM and 32 CPU cores to each container member of the Spark standalone cluster. We reduce TESS to its minimal configuration with only one worker node to get a fair comparison with other triplestores.

In addition, for the first experiment, we used Virtuoso 07.20.3229 Community Edition and GraphDB 9.0.0 Enterprise Edition. Both were configured for optimal parallelization and memory usage according their online documentation. Each triplestore ran on top of a Docker container configured for 32 CPU cores and 128GB of RAM.

Finally, each dataset from the Table 5.3 was stored in an dockerized instance of Virtuoso, GraphDB and TESS.

First experiment: Limitations of Virtuoso and GraphDB

We have two results, shown in Figure 5.4 and Figure 5.5.

Figure 5.4 shows how the *forward-chaining reasoning time* (y axis) evolves in function of the sizes (x axis) of the 10 datasets reported in Table 5.3:

- for GraphDB, the CONSTRUCT-based forward-chaining rules reasoning can be completed in a reasonable time for D1, D2 and D3 datasets only.

- Instead, TESS completes the reasoning for all the datasets in linear time.

- Virtuoso does not show up at all because the output of each of the 18 CONSTRUCT queries was greater than 1 million triples which is the maximum limit for a CONSTRUCT query output in Virtuoso.

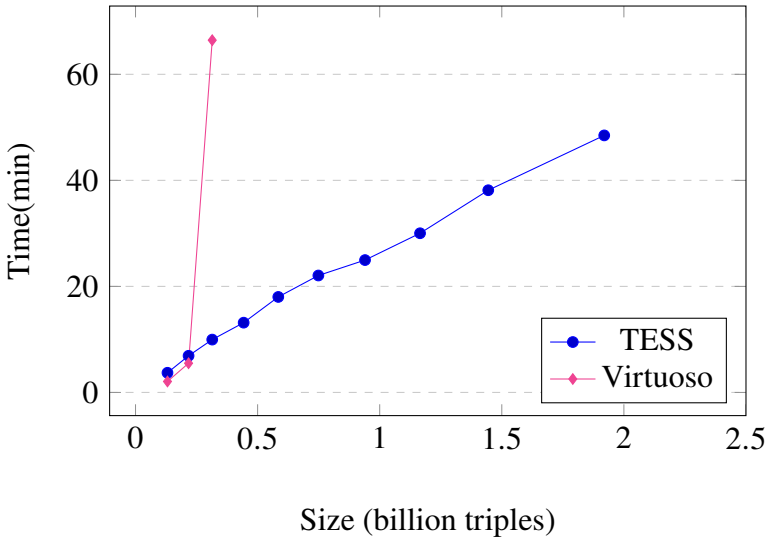

Figure 5.4 Forward-chaining completeness

In Figure 5.5, the y axis corresponds to the *sum of the execution times of the SELECT queries induced by the 18 CONSTRUCT queries*. Virtuoso does not suffer of the above limitations on output size for SELECT queries. However, we have discovered that for the datasets greater than D4 (380 million triples), Virtuoso does not compute the correct answers for *aggregate* queries like Q3 (and others aggregated queries outside the strict setting of our experiment). On the contrary, TESS outputs the correct answers for all the

datasets. We have used PostgreSQL as reference to validate the correctness of the results after transforming the SELECT queries into SQL queries.

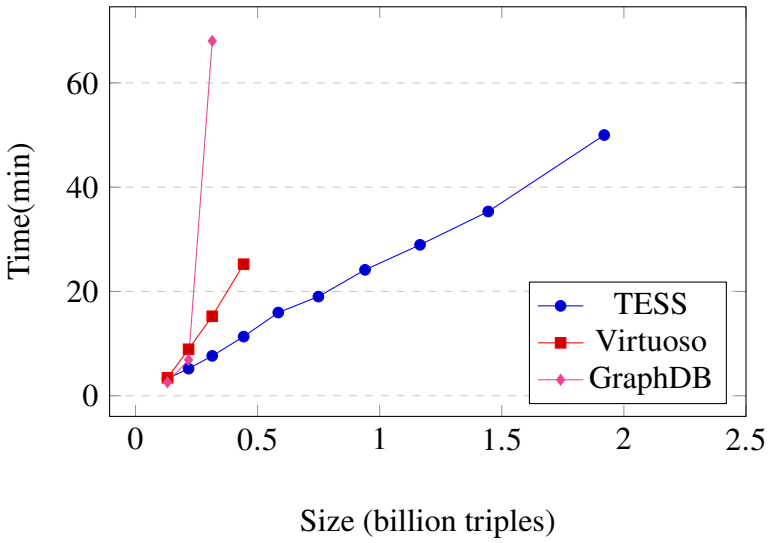

Figure 5.5 Correctness of the induced SELECT queries

These results show the limitation of Virtuoso for outputting CONSTRUCT results of more than 1 million triples, and to compute correct answers to *aggregate* SELECT queries over datasets of size greater than 380 millions triples. They also show the limitation of GraphDB to compute SELECT or CONSTRUCT queries in a reasonable time over datasets of size greater than 275 millions triples.

Second experiment: Evaluation of serial forward-chaining reasoning on TESS

Figure 5.6 shows that TESS completes the forward-chaining reasoning for all datasets in reasonable time and that the time grows linearly w.r.t the size of the input datasets. (see blue curve CONSTRUCT with square shaped dots). It also makes explicit how the *construct execution time* is split into the *body execution time* (see black curve with triangle shaped dots) and the *template execution time* (see red curve with circle shaped dots). We observe that the impact of *body execution time* is much greater than the *template execution time* for CONSTRUCT-based forward-chaining reasoning.

Figure 5.7 shows the individual performance of each of the 18 queries. We observe that for most of the queries, the coefficient of the linear progression of time in function of dataset size is very small (for Q1, Q14, Q15, Q17 and Q18) or small (for Q5, Q6, Q7, Q9, Q11, Q12, Q13 and Q16). The same figure shows that the difference between *construct execution time* and *body execution time* may be important when the graph output of the CONSTRUCT queries is not restricted to a single triple pattern, like in the queries Q9, Q10 and Q11.

In Figure 5.8, we focus on the 5 most expensive queries, namely Q2, Q3,Q4, Q8 and

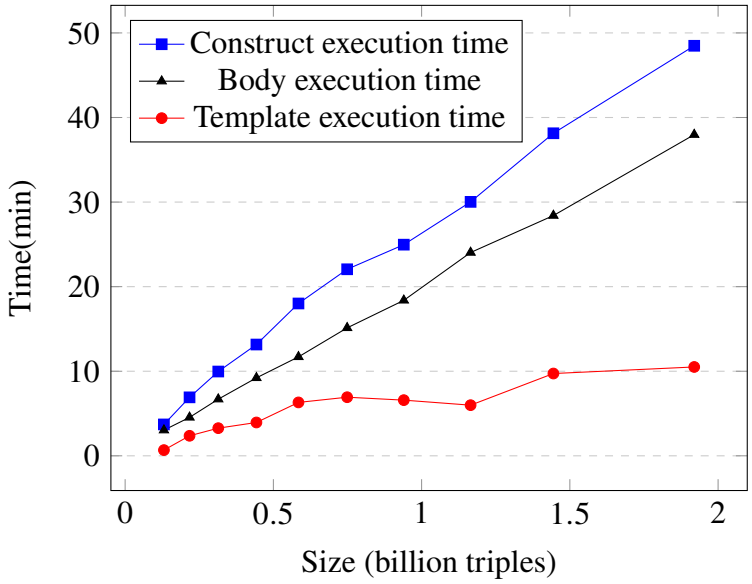

Figure 5.6 Evaluation of serial forward-chaining reasoning time. Best viewed in color.

Q10, and we show the correlation between the query output size and the construct execution time. In this Figure, the center of each circle represents the construct execution time (y axis) of a query for a given dataset size (x axis), and the radius of each circle represents the size of the query output size.

For Q2,Q3,Q4, the query cost can be explained both by the complexity of graph patterns in their body and the size of their output. For Q8, the cost is due to the size of its output since its graph pattern is very simple: a single triple pattern with a single variable. Yet, its execution time is close to the execution time of Q2 (whose body has an aggregate subquery) or of Q3 and Q4 (whose body has FILTER NOT EXISTS clauses).

Figure 5.8 also shows that for queries like Q10 with a template size greater than 1, the CONSTRUCT performance can be costly despite a small query output size. We have analyzed that its high cost is due to join operations between tables of very different size (with a ratio of 1/453), and the fact that the query plan computed by Spark SQL did not choose the most efficient type of joins.

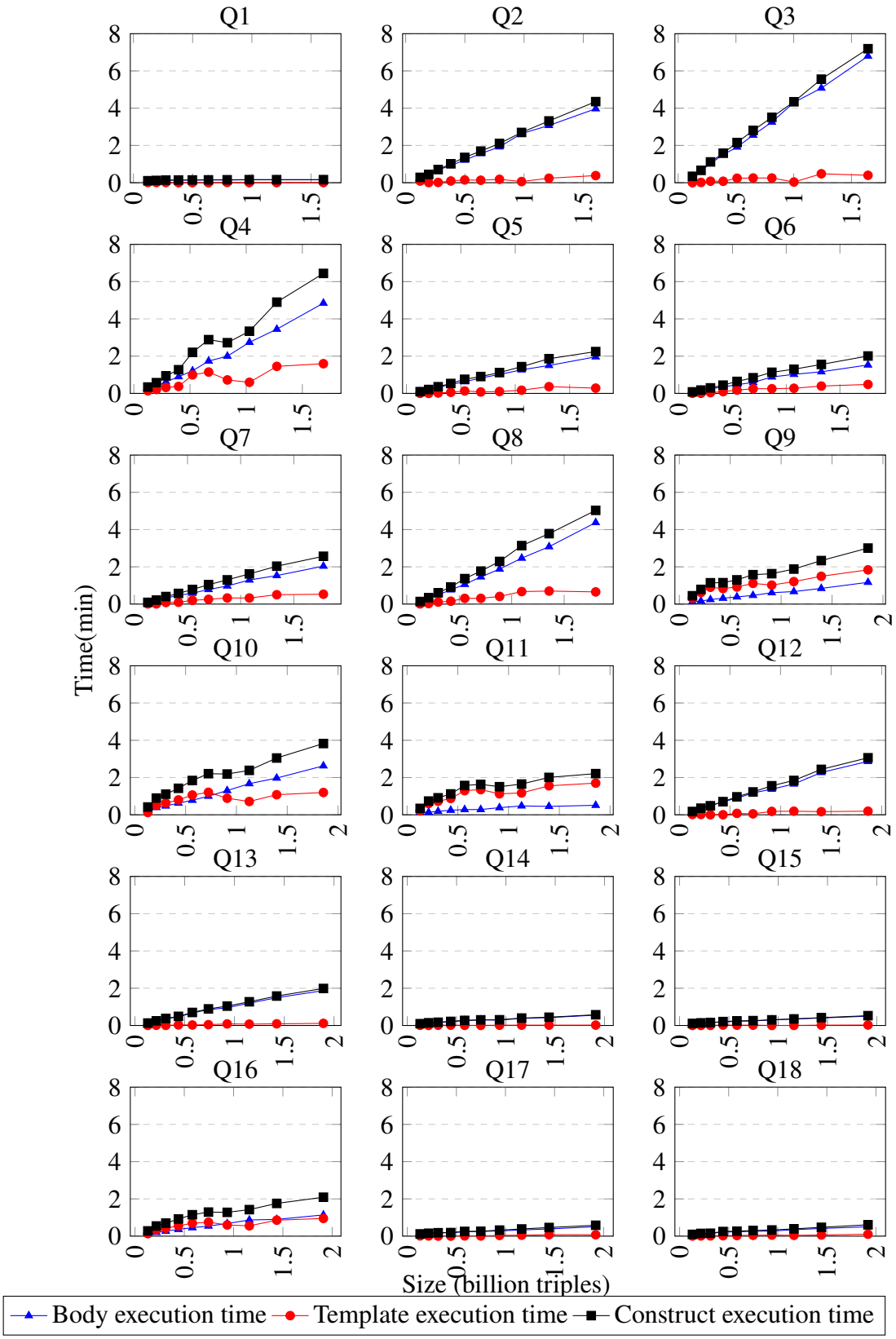

Figure 5.7 CONSTRUCT queries performance. Best viewed in color.

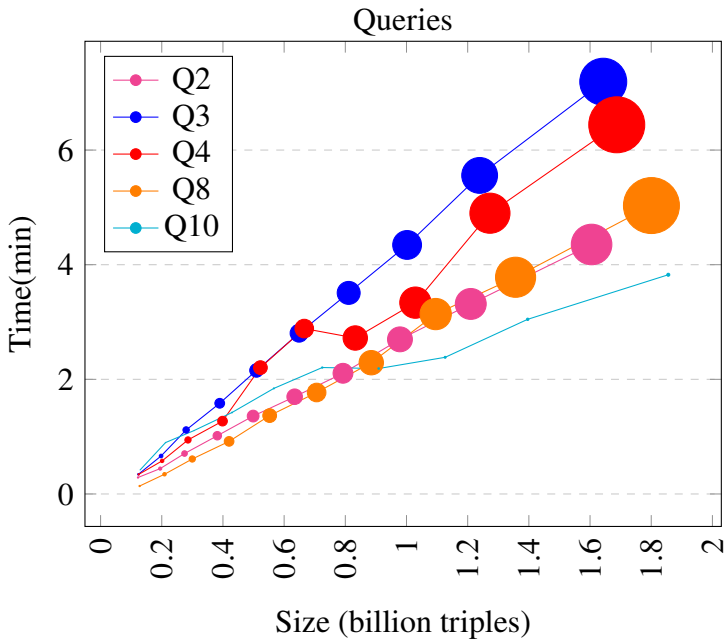

Figure 5.8 The 5 most expensive queries performance. Best viewed in color.

5.3 Serial vs Parallel forward chaining performance

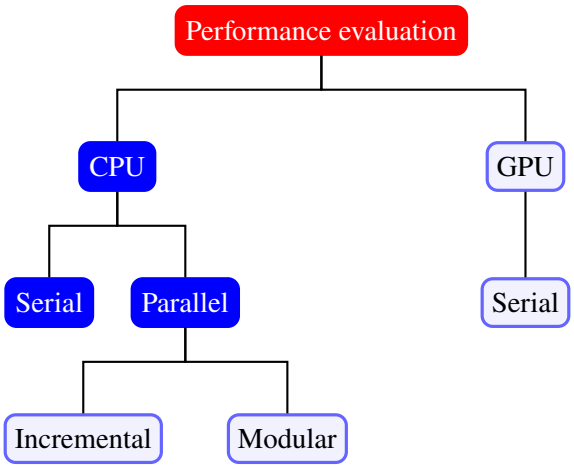

Figure 5.9 Serial vs Parallel performance comparison

In this part, as depicted in Fig. 5.9, we conduct a performance comparison between the serial implementation of forward chaining reasoning considered in the previous section and the implementation of the parallel chaining reasoning algorithm seen in Subsection 4.2.5.

For the experiment with the parallel algorithm, TESS runs on top of a network of 10 Docker containers: 2 containers for the Hadoop namenode and datanode. 7 containers

for the Spark Standalone cluster (1 for the master node, 6 for the worker nodes) and 1 container for sending the Spark Application to the Spark Standalone Cluster in client mode. We deployed 6 worker nodes because it is the maximum number of queries in a layer of reasoning. The Spark Application executes a query per worker node. We reduce the number of workers from 6 to 1 for setting up the experiment with the serial algorithm for comparison purposes. Furthermore, we assign 64GB RAM and 4 CPU cores to each container member of the Spark standalone cluster.

Figure 5.10 shows how the TESS implementation of the parallel forward-chaining algorithm outperforms the serial algorithm for all the datasets. The execution time seems to grow linearly but with a much smaller coefficient than for the serial case.

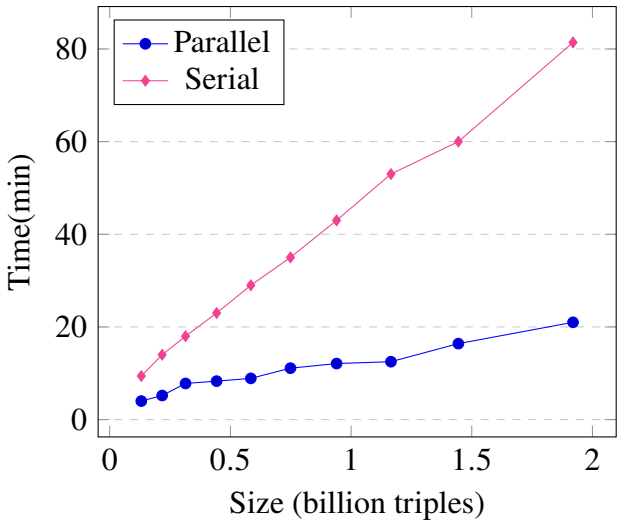

Figure 5.10 Parallel vs serial performance. Best viewed in color.

5.4 Parallel vs Incremental forward chaining performance

In this part, we focus on the comparison highlighted in Fig. 5.11 between the parallel forward chaining reasoning considered in the last section and its incremental variant that may be implemented when the input dataset G is updated regularly with sets Δ of new triples.

In Section 5.4.1, we first describe our approach for designing and implementing incremental forward-chaining reasoning. In Section 5.4.2, we explain the experimental protocol that we have followed to obtain the results summarized in Section 5.4.3.

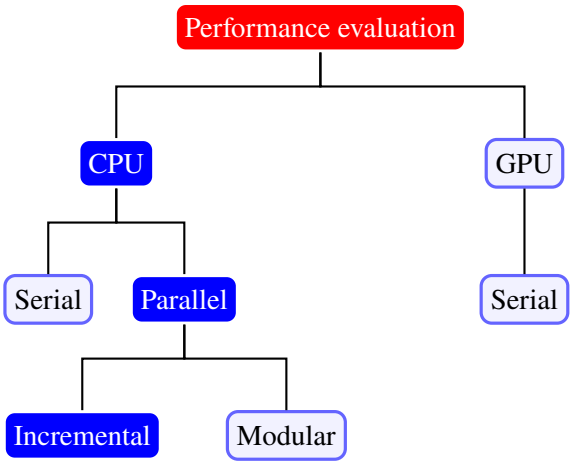

Figure 5.11 Parallel vs Incremental performance comparison

5.4.1 Incremental forward-chaining reasoning

For avoiding to apply the (parallel or serial) forward chaining reasoning to the whole updated dataset $G \cup \Delta$ and thus to redo many inferences for computing $sat(G \cup \Delta)$ at each new Δ , it may be worthwhile to compute $sat(G)$ once and to update it with the set $sat_{\Delta}(G)$ of the new triples inferred using some triples of Δ.

Without any knowledge about Δ, the completeness of $sat_{\Delta}(G)$ requires to consider in turn each condition within each rule for checking whether it can be matched with facts in Δ and if this is the case to evaluate the other conditions over $G \cup \Delta$, and to update Δ accordingly.

In our case, we can exploit the knowledge that we have on the update process of the OntoSides knowledge graph: each update consists in adding to Δ *new students's answers to questions*. These new answers can be done by students that are already identified in G or by students with new identifiers on which Δ may contain additional information. Similarly, the answers added to Δ can correspond to existing questions (already identified and described in G) or novel questions that are thus described in Δ.

Based on this knowledge and the fact that the CONSTRUCT queries encoding rules have been specified to infer properties for new questions (query Q1) or new answers, we have designed a specific incremental forward-chaining algorithm that can be summarized as follows:

- The CONSTRUCT queries Q1, Q2, Q4, Q5, Q6 , Q7 and Q8 can be evaluated over Δ only, with the guarantee to provide the same new results as if they were evaluated over $G \cup \Delta$. The reason is that all the triple patterns in their body involve the output variable ?question, ?answer or variables related to it through functional properties

(such as the variable ?a related to the output variable ?answer by the property *is part of* which is functional, in Q2).

- The other queries (Q3, Q9, Q10, Q11, Q12, Q13, Q14, Q15, Q16, Q17, Q18) can be splitted *at compile time* into two sub-queries such that evaluating the first one against Δ only (and the other one against $G \cup \Delta$) provides the same new results as if the original queries were evaluated over $G \cup \Delta$. The splitting is done by grouping in the first sub-query the triple patterns involving the output variable ?answer, and the remaining triple patterns in the second sub-query.

For implementing this algorithm using SPARQL, the 18 CONSTRUCT queries can be reformulated by exploiting the GRAPH operator of SPARQL 1.1 that allows to restrict the evaluation of some patterns in the query against *named graphs*. For our purpose, the considered named graphs are Δ and G. Despite that in SPARQL 1.1 the GRAPH operator cannot apply to unions of two named graphs, by slight abuse of notation, we will also consider $G \cup \Delta$ as a possible parameter of the GRAPH operator in the reformulations of the queries for clarity purpose.

The result of the corresponding reformulation of the 18 CONSTRUCT queries is provided in Figures 5.12 and 5.13.

Q1

```
CONSTRUCT
{?question  sides:has_for_number_of_proposals ?
        np}
 WHERE { select ?question (COUNT (?p) As ?np)
 {{GRAPH <delta> {?question sides:
        has_for_proposal_of_answer?p }}
 }
 GROUP BY ?question
 }
```

Q2

```
CONSTRUCT
{?answer sides:has_for_number_of_wrong_tick ?
        nw }
 WHERE {
 SELECT ?answer (COUNT (?a) As ?nw)
 {{GRAPH <delta> {?a sides:is_part_of ?answer.
 ?a sides:has_wrongly_ticked ?p}}}
 GROUP BY ?answer
 }
```

Q3

```
CONSTRUCT
{?answer
sides:has_for_number_of_missed_right_tick ?nm}
WHERE {SELECT ?answer (COUNT(?p) As ?nm)
{{
{GRAPH <delta> {
?answer sides:correspond_to_question ?q.
FILTER NOT EXISTS {
?a sides:is_part_of  ?answer.
?a sides:has_rightly_ticked  ?p}
}
}
{GRAPH <g ∪ delta> {
?q sides:has_for_proposal_of_answer ?p.
?p sides:has_for_correction "true"^^xsd:boolean. }
}
}
GROUP BY ?answer
}
```

Q4

```
CONSTRUCT
 {?answer sides:has_for_number_of_discordance
        "0"^^xsd:integer}
 WHERE {GRAPH <delta> {
 ?answer a sides:answer.
 FILTER NOT EXISTS  {
 ?answer sides:has_for_number_of_wrong_tick ?
        nw
 ?answer sides:
        has_for_number_of_missed_right_tick ?nm}
 }}
```

Q5

```
CONSTRUCT
 {?answer sides:has_for_number_of_discordance ?
        count}
 WHERE {
 SELECT  ?answer  (?nw + ?nm as ?count)
 {{GRAPH <delta>{
 ?answer sides:has_for_number_of_wrong_tick ?
        nw.
 ?answer sides:
        has_for_number_of_missed_right_tick ?nm
 }
 }}}
```

Q6

```
CONSTRUCT
 {?answer sides:has_for_number_of_discordance ?
        nw}
 WHERE {{GRAPH <delta> {?answer sides:
        has_for_number_of_wrong_tick ?nw.
 FILTER NOT EXISTS {?answer sides:
        has_for_number_of_missed_right_tick ?nm}
 }}}
```

Q7

```
CONSTRUCT
{?answer sides:has_for_number_of_discordance
?nm}
WHERE {{GRAPH <delta> {?answer sides:
        has_for_number_of_missed_right_tick ?nm.
FILTER NOT EXISTS {?answer sides:
        has_for_number_of_wrong_tick ?nw. }
}}}
```

Q8

```
CONSTRUCT
{?answer sides:has_for_result 1}
WHERE {{GRAPH <delta> {?answer sides:
        has_for_number_of_discordance "0"^^xsd:
        integer
}}}
```

Q9

```
CONSTRUCT
{?answer sides:has_for_result "0"^^xsd:integer  .
?answer sides:stronglyWrong  "true"^^xsd:boolean
        .}
WHERE  {{GRAPH <delta> {
?a sides:is_part_of  ?answer.
?a sides:has_wrongly_ticked ?p. }}
{GRAPH <g ∪ delta> {?p
sides:has_for_weight_of_correction
"Unacceptable"^^xsd:string}}
}
```

Figure 5.12 Reformulation of the CONSTRUCT queries (from Q1 to Q9) for incremental reasoning.

Note that when $g \cup delta$ appears in the reformulated queries, it is actually a shorcut to abbreviate a much longer query expression that uses the SPARQL UNION operator. As an example, the reformulation conform to SPARQL of the query Q3 in Figure 5.12 is provided in Figure 5.14.

Q10

```
CONSTRUCT
{?answer sides:has_for_result "0"^^xsd:integer  .
?answer sides:stronglyWrong  "true"^^xsd:boolean .
}
WHERE {
{GRAPH <delta> {
?answer sides:correspond_to_question ?q.
FILTER NOT EXISTS {
?a sides:is_part_of  ?answer.
?a sides:has_rightly_ticked  ?p
}}}
{GRAPH <g ∪ delta> {
?q sides:has_for_proposal_of_answer ?p.
?p sides:has_for_correction "true"^^xsd:boolean .
?p sides:has_for_weight_of_correction "
        Indispensable"^^xsd:string .}}
}
```

Q11

```
CONSTRUCT
 {?answer sides:has_for_result "0"^^xsd:integer  .
?answer sides:stronglyWrong  "true"^^xsd:boolean .
}
WHERE {
{GRAPH <delta> {?answer sides:
        correspond_to_question ?q.
?answer sides:has_for_number_of_discordance ?d
        .}}
{GRAPH <g ∪ delta> {?q rdf:type sides:QUA.}}
FILTER (?d > 0)
}
```

Q12

```
CONSTRUCT
{?answer sides:has_for_result 0.5^^xsd:decimal}
WHERE {
{GRAPH <delta>{
?answer sides:has_for_number_of_discordance
        "1"^^xsd:integer .
?answer sides:correspond_to_question ?q.
FILTER NOT EXISTS {?answer sides:
        stronglyWrong  "true"^^xsd:boolean }
}}
{GRAPH <g ∪ delta> {
?q sides:has_for_number_of_proposals "5"^^xsd:
        integer.}}
}
```

Q13

```
CONSTRUCT
{?answer sides:has_for_result "0.2"^^xsd:decimal}
WHERE {{GRAPH <delta> {
?answer sides:has_for_number_of_discordance
        "2"^^xsd:integer .
?answer sides:correspond_to_question ?q.
FILTER NOT EXISTS {
?answer sides:stronglyWrong  "true"^^xsd:boolean }
}}
{GRAPH <g ∪ delta> {
?q sides:has_for_number_of_proposals "5"^^xsd:
        integer .}}
}
```

Q14

```
CONSTRUCT
{?answer sides:has_for_result "0.425"^^xsd:decimal
        }
WHERE {
{GRAPH <delta> {
?answer sides:has_for_number_of_discordance
        "1"^^xsd:integer .
?answer sides:correspond_to_question ?q.
FILTER NOT EXISTS {
?answer sides:stronglyWrong  "true"^^xsd:boolean }
}}
{GRAPH <g ∪ delta> {
?q sides:has_for_number_of_proposals "4"^^xsd:
        integer . }}
}
```

Q15

```
CONSTRUCT
{  ?answer sides:has_for_result "0.1"^^xsd:decimal
}
WHERE
{{GRAPH <delta> {
?answer sides:has_for_number_of_discordance
        "2"^^xsd:integer .
?answer sides:correspond_to_question ?q.
FILTER NOT EXISTS {
?answer sides:stronglyWrong  "true"^^xsd:boolean
}}}
{GRAPH <g ∪ delta> {
?q sides:has_for_number_of_proposals "4"^^xsd:
        integer .}}
}
```

Q16

```
CONSTRUCT
{?answer sides:has_for_result "0"^^xsd:integer}
WHERE
{{GRAPH <delta> {?answer sides:
        correspond_to_question ?q.
?answer sides:has_for_number_of_discordance ?n.
        }}
{GRAPH <g ∪ delta> {?q sides:
        has_for_number_of_proposals ?np.}}
FILTER (?np > 3 && ?np < 6 && ?n > 2).
}
```

Q17

```
CONSTRUCT
{?answer sides:has_for_result "0.3"^^xsd:decimal}
WHERE {
{GRAPH <delta> {
?answer sides:has_for_number_of_discordance
        "1"^^xsd:integer .
?answer sides:correspond_to_question ?q.
FILTER NOT EXISTS {
?answer sides:stronglyWrong  "true"^^xsd:boolean }
}}
{GRAPH <g ∪ delta> {?q sides:
        has_for_number_of_proposals "3"^^xsd:
        integer . }}
}
```

Q18

```
CONSTRUCT
 {?answer sides:has_for_result "0"^^xsd:integer  }
 WHERE
 {{GRAPH <delta> {?answer sides:
        has_for_number_of_discordance ?n.
 ?answer sides:correspond_to_question ?q. }}
 {GRAPH <g ∪ delta> {?q sides:
        has_for_number_of_proposals "3"^^xsd:
        integer. }}
 FILTER (?n > 1)
 }
```

Figure 5.13 Reformulation of the CONSTRUCT queries (from Q10 to Q18) for incremental reasoning.

```
CONSTRUCT {
?answer sides:has_for_number_of_missed_right_tick ?nm.
}
WHERE
{
SELECT ?answer (COUNT(?p) AS ?nm)
{{GRAPH <delta> {
  ?answer sides:correspond_to_question ?q.
FILTER NOT EXISTS {
  ?a sides:is_part_of  ?answer.
  ?a sides:has_rightly_ticked  ?p.
}
}}
{
{GRAPH <delta> {
  ?q sides:has_for_proposal_of_answer ?p.
  ?p sides:has_for_correction "true"^^xsd:boolean
}}
UNION
{GRAPH <g> {
  ?q sides:has_for_proposal_of_answer ?p.
  ?p sides:has_for_correction "true"^^xsd:boolean
}}
}
}
GROUP BY ?answer
}
```

Figure 5.14 Reformulation of Q3 conform to SPARQL

Algorithm 3 corresponds to the parallel version of the incremental forward-chaining reasoning algorithm described above.

Algorithm 3 Incremental parallel algorithm

Input: G, update Δ , the reformulated queries organized in 4 layers

Output: The set $sat_\Delta(G)$ of the *new* triples inferred from $G \cup \Delta$ using Δ

$sat_\Delta(G) \leftarrow \emptyset$

for *each Layer from Layer1 to Layer4* **do**

  for each $q \in Layer$ **do in parallel**

  $output \leftarrow Sparql(q)$

  $sat_\Delta(G) \leftarrow sat_\Delta(G) \cup output$ `// update operation`

end

By construction, we have: $sat(G \cup \Delta) = sat(G) \cup sat_\Delta(G)$

5.4.2 Experimental protocol

The goal of the experiment is to determine whether incremental forward chaining can outperform parallel forward chaining reasoning for a given delta size. For performance measures and hardware setup we follow those of Subsection 5.1.3 and Subsection 5.1.4.

For this experiment, TESS keeps the same configuration used in the experiment with the parallel algorithm in Section 5.3, and the implementation of the forward chaining reasoning has been adapted accordingly to include the reformulated queries.

Datasets and incremental deltas

We have created 9 datasets as shown in Table 5.4 in blue color. G is the original dataset D_1 from Table 5.3. Each dataset $D^{'}$ is the union of G and an incremental delta Δ. The deltas were computed from the difference between each dataset from Table 5.3 and G.

For example, $D^{'}_2 = G \cup \Delta_1$ and $D^{'}_9 = G \cup \Delta_8$.

In addition, we define r as a measure of the size variation of Δ regarding G:

$$r = \frac{\Delta}{G}$$

The Table 5.4 also shows in red color, the corresponding datasets used in the parallel forward chaining reasoning for the performance comparison.

Incremental				**Parallel**	
Datasets	G size	Δ size	r	Datasets	size
$D^{'}_2$	121	73	0.60	D_2	194
$D^{'}_3$	121	152	1.26	D_3	273
$D^{'}_4$	121	259	2.14	D_4	380
$D^{'}_5$	121	376	3.10	D_5	497
$D^{'}_6$	121	512	4.23	D_6	633
$D^{'}_7$	121	670	5.54	D_7	791
$D^{'}_8$	121	856	7.07	D_8	977
$D^{'}_9$	121	1088	8.99	D_9	1209
$D^{'}_{10}$	121	1483	12.26	D_{10}	1604

Table 5.4 Datasets and incremental deltas (size in millions triples). Best viewed in color.

Optimization of the SQL translations of the SPARQL queries with UNION

When we use the Virtuoso function sparql_to_text() to translate the reformulation of the CONSTRUCT queries that contain a UNION (such as the SPARQL-compliant reformulation of the query Q3 provided in Figure 5.14), the returned SQL queries are long and contain many joins. For example, Figure 5.15 shows the translation returned by the Virtuoso function sparql_to_text() of the reformulation with UNION of the query Q3 provided in Figure 5.14. This results in SQL queries for which the evaluation has poor performance even in Spark.

For better performance, we do not use query reformulations based on UNION operators. Instead, we apply a two-step post-processing to the 11 queries that use $G \cup \Delta$ as a named graph in Figure 5.12 and Figure 5.13.

The first step starts by taking replacing "$G \cup \Delta$" with "g_union_delta" in each query so that it can be translated into a SQL query using the Virtuoso function sparql_to_text(). Each resulting SQL query contains a collection of joins between copies of the same table "rdf_quad". The table "rdf_quad" is the 4-column table $< s, p, o, g >$ introduced in Subsection 4.2.2 as the main storage table. Note that in each SQL translated query, if a copy of the table "rdf_quad" is denoted by an alias such as table_alias, a reference to its g column in the query is given by table_alias.g.

In the second step, we use the output of the previous step as input. Here, the post-processing consists in replacing all ocurrences in each SQL query of the condition "table_alias.g = g_union_delta" with the condition "table_alias.g IN ('g', 'delta')" for all WHERE clauses.

By doing so, the very complex translation of query Q3 shown in Figure 5.14 is replaced by the optimised translated query shown in Figure 5.16.

```
SELECT s_0_27.answer AS answer,
    COUNT (
      s_0_27.p) AS nm
 FROM (
    SELECT s_1_4_t1-u28-u29.s AS answer,
      s_1_8_t3-c5-u28-u29.s AS p
    FROM rdf_quad AS s_1_4_t1-u28-u29
      INNER JOIN rdf_quad AS s_1_8_t2-c5-u28-u29
      ON (
        s_1_4_t1-u28-u29.o = s_1_8_t2-c5-u28-u29.s)
      INNER JOIN rdf_quad AS s_1_8_t3-c5-u28-u29
      ON (
        s_1_8_t3-c5-u28-u29.s = s_1_8_t2-c5-u28-u29.o)
    WHERE
      s_1_4_t1-u28-u29.g = 'delta'
      AND
      s_1_4_t1-u28-u29.p = 'http://www.side-sante.fr/sides#
       correspond_to_question'
      AND
      s_1_8_t2-c5-u28-u29.g = 'delta'
      AND
      s_1_8_t2-c5-u28-u29.p = 'http://www.side-sante.fr/sides#
       has_for_proposal_of_answer'
      AND
      s_1_8_t3-c5-u28-u29.g = 'delta'
      AND
      s_1_8_t3-c5-u28-u29.p = 'http://www.side-sante.fr/sides#has_for_correction'
      AND
      s_1_8_t3-c5-u28-u29.o = 1
      AND
      not ( EXISTS ( (
        SELECT 1
          FROM rdf_quad AS s_1_18_t6-c3
            INNER JOIN rdf_quad AS s_1_18_t7-c3
            ON (
              s_1_18_t7-c3.s = s_1_18_t6-c3.s)
          WHERE
            s_1_18_t6-c3.g = 'delta'
            AND
            s_1_18_t6-c3.p = 'http :// www.side-sante.fr/sides#is_part_of'
            AND
            s_1_18_t7-c3.g = 'delta'
            AND
            s_1_18_t7-c3.p = 'http :// www.side-sante.fr/sides#has_rightly_ticked'
            AND
            s_1_18_t7-c3.o = s_1_8_t2-c5-u28-u29.o
            AND
            s_1_18_t6-c3.o = s_1_4_t1-u28-u29.s

        )))
UNION ALL SELECT s_1_4_t1.s AS answer,
  s_1_12_t5-c11.s AS p
FROM rdf_quad AS s_1_4_t1
  INNER JOIN rdf_quad AS s_1_12_t4-c11
  ON (
    s_1_4_t1.o = s_1_12_t4-c11.s)
  INNER JOIN rdf_quad AS s_1_12_t5-c11
  ON (
    s_1_12_t5-c11.s = s_1_12_t4-c11.o)
WHERE
  s_1_4_t1.g = 'delta'
  AND
  s_1_4_t1.p = 'http :// www.side-sante.fr/sides#correspond_to_question'
  AND
  s_1_12_t4-c11.g = 'g'
  AND
  s_1_12_t4-c11.p = 'http :// www.side-sante.fr/sides#has_for_proposal_of_answer'
  AND
  s_1_12_t5-c11.g = 'g'
  AND
  s_1_12_t5-c11.p = 'http :// www.side-sante.fr/sides#has_for_correction'
  AND
  s_1_12_t5-c11.o = 1
  AND
  not EXISTS (
    SELECT 1
      FROM rdf_quad AS s_1_18_t6-c9
        INNER JOIN rdf_quad AS s_1_18_t7-c9
        ON (
          s_1_18_t7-c9.s = s_1_18_t6-c9.s)
      WHERE
        s_1_18_t6-c9.g = 'delta'
        AND
        s_1_18_t6-c9.p = 'http :// www.side-sante.fr/sides#is_part_of'
        AND
        s_1_18_t7-c9.g = 'delta'
        AND
        s_1_18_t7-c9.p = 'http :// www.side-sante.fr/sides#has_rightly_ticked'
        AND
        s_1_18_t7-c9.o = s_1_12_t4-c11.o
        AND
        s_1_18_t6-c9.o = s_1_4_t1.s

    ) AS s_0_27
  GROUP BY s_0_27.answer

) AS s_1_23
))
```

Figure 5.15 SQL translation of the reformulated query Q3 shown in Figure 5.14 . Displayed in two columns because it is a very long query.

```
SELECT  s_1_1_t0.s AS answer, COUNT ( s_1_5_t2.s) AS nm
FROM rdf_quad AS s_1_1_t0
  INNER JOIN rdf_quad AS s_1_5_t1
  ON (
    s_1_5_t1.s = s_1_1_t0.o)
  INNER JOIN rdf_quad AS s_1_5_t2
  ON (
    s_1_5_t2.s = s_1_5_t1.o)
WHERE
  s_1_1_t0.g =  'delta'
  AND
  s_1_1_t0.p =  'http :// www.side-sante.fr/sides#correspond_to_question'
  AND
  s_1_5_t1.g in ('g','delta')
  AND
  s_1_5_t1.p =  'http :// www.side-sante.fr/sides#has_for_proposal_of_answer'
  AND
  s_1_5_t2.g in ('g','delta')
  AND
  s_1_5_t2.p =  'http :// www.side-sante.fr/sides#has_for_correction'
  AND
  s_1_5_t2.o =  1
  AND
  not EXISTS (
     SELECT 1
      FROM rdf_quad AS s_1_9_t3
        INNER JOIN rdf_quad AS s_1_9_t4
        ON (
          s_1_9_t4.s = s_1_9_t3.s)
      WHERE
        s_1_9_t3.g =  'delta'
        AND
        s_1_9_t3.p =  'http :// www.side-sante.fr/sides#is_part_of'
        AND
        s_1_9_t4.g =  'delta'
        AND
        s_1_9_t4.p =  'http :// www.side-sante.fr/sides#has_rightly_ticked'
        AND
        s_1_9_t4.o = s_1_5_t1.o
        AND
        s_1_9_t3.o = s_1_1_t0.s

     )
  GROUP BY s_1_1_t0.s
```

Figure 5.16 Optimised SQL translation using SQL IN operators in red color

5.4.3 Evaluation results

Fig.5.17 shows that the incremental parallel reasoning outperforms parallel reasoning from D1 to D7. The x axis at the bottom shows the datasets used in this experiment and the x axis at the top shows the corresponding datasets used in the previous parallel forward chaining experiment. As we can see, beyond size D7, incremental reasoning performs similarly to simple parallel reasoning. We believe that this similar performance is transient, that beyond D10, is reasonable to assume that the performance of incremental parallel reasoning will falls behind parallel reasoning as the delta size continues to grow.

The results also show that our implementation works for an interval of ratio r

$$0.6 < r < 5.54$$

This is much better than expected if we assume that a set of incoming triples is considered a Δ if and only if its size is less or equal to the size of G such that

$$r \leq 1$$

However, in response to the question about why there is a drop in performance for $r > 5.54$, we found that a possible explanation lies in the addition of inferred data. For the addition of data, we used Spark's *union* operations to put together $sat(G)$ and $sat_{\Delta}(G)$ in the implementation of the incremental algorithm. We found that *union* operations are not optimised in Spark for large amounts of data [1].

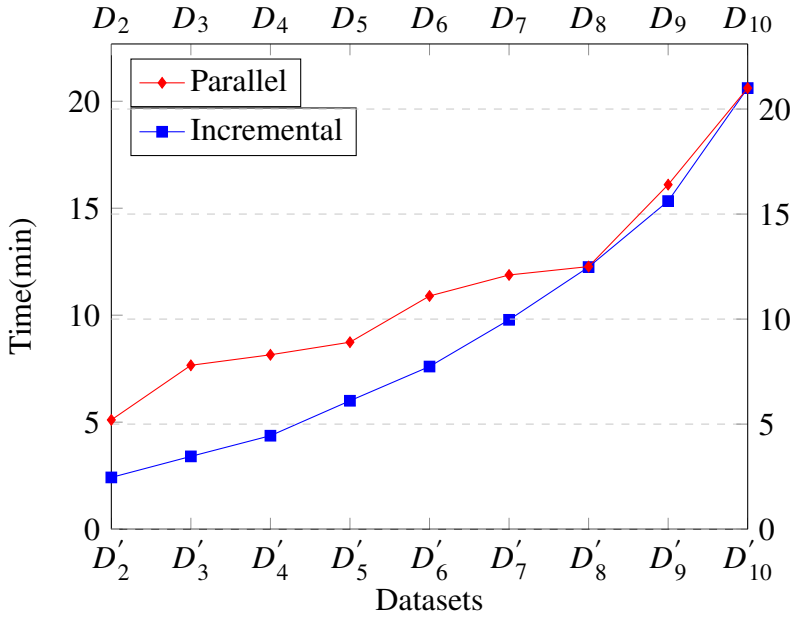

Figure 5.17 Incremental vs Parallel forward chaining reasoning time

5.5 Parallel vs Modular forward chaining performance

In this section, we focus on the comparison shown in Fig. 5.18 between the parallel forward chaining reasoning considered in the section 5.3 and its modular variant that may be implemented when the input dataset G is a union of named graphs.

In Section 5.5, we first describe our approach for designing and implementing modular forward-chaining reasoning. In Section 5.5.2, we explain the experimental protocol that we have followed to obtain the results summarized in Section 5.5.3.

[1] https://www.databricks.com/dataaisummit/session/goodbye-hell-unions-spark-sql

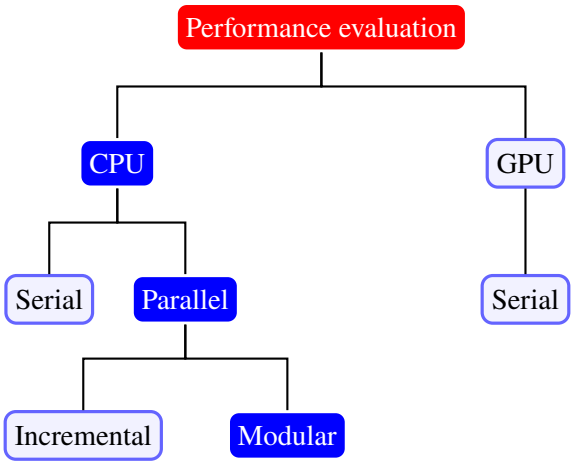

Figure 5.18 Parallel vs Modular performance comparison

5.5.1 Modular forward-chaining reasoning

Modular reasoning exploits the process described in Chapter 3 of building G as a union of named graphs g where each named graph groups the data of one student.

A first approach for the modular implementation of forward chaining reasoning consists in executing the 18 original CONSTRUCT queries encoding rules on each named graph (serially or in parallel). However, this solution would cause a bottleneck due to the large number of queries to be executed, since for a dataset with n named graphs, $18 \times n$ CONSTRUCT queries would have to be evaluated.

We tackle the problem by reformulating the 18 CONSTRUCT queries into 18 CONSTRUCT queries that output quadruplets (quads) and that contain a GRAPH operator applied to a graph variable $?g$ denoting named graphs corresponding to each student data. We have defined the following steps to reformulate the queries:

a. enclosing the body of the query with the operator `GRAPH` and the variable `?g`,

b. adds the variable `?g` to the projection of the query,

c. extending each triple in the query template to a quad by adding the variable `?g`,

d. adding the variable `?g` to the operator `GROUP BY` where it applies.

For example, the CONSTRUCT query in Figure 5.19 is reformulated as a CONSTRUCT quad in Figure 5.20.

As CONSTRUCT quads are not supported in the current SPARQL 1.1 specification, we implemented our solution at the application level.

```
CONSTRUCT {
?answer sides:has_for_number_of_missed_right_tick ?nm}
WHERE {
SELECT ?answer (COUNT(?p) As ?nm)
{?answer sides:correspond_to_question ?q.
?q sides:has_for_proposal_of_answer ?p.
?p sides:has_for_correction "true"^^xsd:boolean.
FILTER NOT EXISTS {?a sides:is_part_of ?answer.
?a sides:has_rightly_ticked  ?p}
}
GROUP BY ?answer
}
```

Figure 5.19 Query Q3

```
CONSTRUCT {
?answer sides:has_for_number_of_missed_right_tick ?nm ?g}
WHERE {
  SELECT ?g ?answer (COUNT(?p) As ?nm)  { GRAPH ?g {
  ?answer sides:correspond_to_question ?q.
  ?q sides:has_for_proposal_of_answer ?p.
  ?p sides:has_for_correction "true"^^xsd:boolean.
  FILTER NOT EXISTS {
    ?a sides:is_part_of  ?answer.
    ?a sides:has_rightly_ticked  ?p
  } }}
GROUP BY ?g ?answer
}
```

Figure 5.20 CONSTRUCT QUAD reformulation for query Q′3

5.5.2 Experimental protocol

The goal of the experiment is to determine whether modular forward chaining can outperform parallel forward chaining reasoning. For performance measures and hardware setup we follow those of Subsection 5.1.3 and Subsection 5.1.4.

For this experiment, TESS keeps the same configuration used in the experiment with the parallel algorithm in Section 5.3, and the forward chaining implementation has been adapted accordingly to include the reformulated queries.

Datasets

We built 10 datasets shown in Table 5.5 in blue color. Each dataset is the union of named graphs of students. The forward chaining reasoning can run in each student named graph as it was in the whole dataset because each named graph contains all the data that corresponds to one student.

Note that the size of some datasets can be very large because when a large number of student named graphs are joined together, it turns out that the same triple can correspond to several named graphs.

The same table also shows in red color, the corresponding datasets used in the parallel forward chaining reasoning for the performance comparison.

	Modular (quads)		Parallel (triples)	
Students named graphs	Dataset	Size	Dataset	Size
880	$\boldsymbol{D''_1}$	0.33	D_1	0.12
1760	$\boldsymbol{D''_2}$	0.97	D_2	0.19
2640	$\boldsymbol{D''_3}$	1.90	D_3	0.27
3520	D''_4	3.26	$\boldsymbol{D_4}$	0.38
4400	D''_5	5.19	$\boldsymbol{D_5}$	0.49
5280	D''_6	7.86	$\boldsymbol{D_6}$	0.63
6160	D''_7	11.60	$\boldsymbol{D_7}$	0.8
7040	D''_8	16.95	$\boldsymbol{D_8}$	0.98
7920	D''_9	25.20	$\boldsymbol{D_9}$	1.2
8845	D''_{10}	43.81	$\boldsymbol{D_{10}}$	1.6

Table 5.5 Size of module-based datasets (size in billions triples/quads)

Queries

The 18 reformulated queries are presented in Figure 5.21 and Figure 5.22.

Join optimization technique

The reformulated queries could increase the number of joins when translated into SQL. To mitigate potential bottlenecks, we implemented *bucketing*, a Spark join optimisation technique to minimise the performance hit. Based on the values of a column, bucketing pre-calculates a predefined number of buckets into which the data is shuffled and sorted. Bucketing is performed on one or more columns before the query is executed.

For this experiment, for the given 4-column table (<s,p,o,g>) where the data is stored, we used bucketing over column g to gain performance when such a table is self-joined in the SQL translation query.

5.5.3 Evaluation results

The results are shown in Fig.5.23 and Fig.5.24. The x axis at the bottom shows the datasets used in this experiment and the x axis at the top show the corresponding datasets used in the previous parallel forward chaining experiment.

Fig.5.23 shows that modular forward chaining is comparable to parallel forward chaining when the bucketing optimisation technique is applied. This can be seen in the significant performance improvement showed between modular (dashed line) and modular with bucketing (blue color with squares). This is expected because once SPARQL queries are translated into SQL queries over self-joined tables, the joins involving the column G is involved benefit greatly from the precomputation of buckets.

Q1

```
CONSTRUCT
{?question
sides:has_for_number_of_proposals ?np ?g}
WHERE {
SELECT ?g ?question (COUNT (?p) As ?np)
{GRAPH ?g {?question sides:
        has_for_proposal_of_answer?p}}
GROUP BY ?g ?question
}
```

Q2

```
CONSTRUCT
{?answer
sides:has_for_number_of_wrong_tick ?nw ?g}
WHERE {
SELECT ?g ?answer (COUNT (?a) As ?nw)
{GRAPH ?g {?a sides:is_part_of ?answer.
?a sides:has_wrongly_ticked ?p}}
GROUP BY ?g ?answer
}
```

Q3

```
CONSTRUCT
{?answer
sides:has_for_number_of_missed_right_tick ?nm
        ?g}
WHERE {
SELECT ?g ?answer (COUNT(?p) As ?nm)  {
        GRAPH ?g {
  ?answer sides:correspond_to_question ?q.
  ?q sides:has_for_proposal_of_answer ?p.
  ?p sides:has_for_correction 1.
  FILTER NOT EXISTS {
    ?a sides:is_part_of  ?answer.
    ?a sides:has_rightly_ticked  ?p
  } }}
GROUP BY ?g ?answer
}
```

Q4

```
CONSTRUCT
{?answer
sides:has_for_number_of_discordance "0"^^xsd:
        integer ?g}
WHERE {
SELECT ?g ?answer  {GRAPH ?g {?answer a
        sides:answer.
FILTER NOT EXISTS  {
?answer sides:has_for_number_of_wrong_tick ?nw
?answer sides:
        has_for_number_of_missed_right_tick ?nm}
}
}
}
```

Q5

```
CONSTRUCT
{?answer
sides:has_for_number_of_discordance
?count ?g}
WHERE {
SELECT ?g ?answer  (?nw + ?nm) as ?count
{GRAPH ?g {
?answer sides:has_for_number_of_wrong_tick ?nw.
?answer sides:
        has_for_number_of_missed_right_tick ?nm
}}
}
```

Q6

```
CONSTRUCT
{?answer
sides:has_for_number_of_discordance ?nw ?g}
WHERE {
SELECT ?g ?answer ?nw {GRAPH ?g {?answer
        sides:has_for_number_of_wrong_tick ?nw.
FILTER NOT EXISTS {?answer sides:
        has_for_number_of_missed_right_tick ?nm
        }}}
}
```

Q7

```
CONSTRUCT {?answer
sides:has_for_number_of_discordance ?nm ?g}
WHERE {
SELECT ?g ?answer ?nm {GRAPH ?g {?answer
        sides:
        has_for_number_of_missed_right_tick ?nm.
FILTER NOT EXISTS  {?answer sides:
        has_for_number_of_wrong_tick ?nw. }}}
}
```

Q8

```
CONSTRUCT
{?answer sides:has_for_result 1 ?g}
WHERE {
SELECT ?g ?answer {GRAPH ?g {
?answer sides:has_for_number_of_discordance 0
}}
}
```

Q9

```
CONSTRUCT
{?answer sides:has_for_result "0"^^xsd:integer  ?g .
?answer sides:stronglyWrong  "true"^^xsd:boolean
        ?g .}
WHERE {
SELECT ?g ?answer {GRAPH ?g {
?a sides:is_part_of  ?answer.
?a sides:has_wrongly_ticked ?p.
?p sides:has_for_weight_of_correction "
        Unacceptable"^^xsd:string.
}}
}
```

Figure 5.21 18 CONSTRUCT queries for modular reasoning. Showing queries from Q1 to Q9.

Q10

```
CONSTRUCT
{?answer sides:has_for_result "0"^^xsd:integer ?g .
  ?answer sides:stronglyWrong "true"^^xsd:
        boolean ?g. }
WHERE {
SELECT ?g ?answer {GRAPH ?g {
?answer sides:correspond_to_question ?q.
?q sides:has_for_proposal_of_answer ?p.
?p sides:has_for_correction "true"^^xsd:boolean .
?p sides:has_for_weight_of_correction "
        Indispensable"^^xsd:string .
FILTER NOT EXISTS {
  ?a sides:is_part_of ?answer.
 ?a sides:has_rightly_ticked ?p
  }
 }}
}
```

Q11

```
CONSTRUCT
{?answer sides:has_for_result "0"^^xsd:integer ?g .
?answer sides:stronglyWrong "true"^^xsd:boolean
        ?g . }
WHERE {
SELECT ?g ?answer {GRAPH ?g {
?answer sides:correspond_to_question ?q.
?q rdf:type sides:QUA.
?answer sides:has_for_number_of_discordance ?d.
FILTER (?d > 0)
}}
}
```

Q12

```
CONSTRUCT
{?answer sides:has_for_result "0.5"^^xsd:decimal
        ?g}
WHERE {
SELECT ?g ?answer {GRAPH ?g {
?answer sides:has_for_number_of_discordance
        "1"^^xsd:integer .
?answer sides:correspond_to_question ?q.
?q sides:has_for_number_of_proposals "5"^^xsd:
        integer.
FILTER NOT EXISTS {?answer sides:
        stronglyWrong "true"^^xsd:boolean }}}
}
```

Q13

```
CONSTRUCT
{?answer sides:has_for_result "0.2"^^xsd:decimal
        ?g}
WHERE {
SELECT ?g ?answer {GRAPH ?g {
  ?answer sides:has_for_number_of_discordance
        "2"^^xsd:integer .
  ?answer sides:correspond_to_question ?q.
  ?q sides:has_for_number_of_proposals "5"^^xsd:
        integer .
FILTER NOT EXISTS {?answer sides:
        stronglyWrong "true"^^xsd:boolean }}}
}
```

Q14

```
CONSTRUCT
{?answer sides:has_for_result "0.425"^^xsd:decimal
        ?g}
WHERE {
SELECT ?g ?answer {GRAPH ?g {
?answer sides:has_for_number_of_discordance
        "1"^^xsd:integer .
?answer sides:correspond_to_question ?q.
?q sides:has_for_number_of_proposals "4"^^xsd:
        integer .
FILTER NOT EXISTS {?answer sides:
        stronglyWrong "true"^^xsd:boolean }}}
}
```

Q15

```
CONSTRUCT
{?answer sides:has_for_result "0.1"^^xsd:decimal
        ?g}
WHERE {
SELECT ?g ?answer {GRAPH ?g {
?answer sides:has_for_number_of_discordance
        "2"^^xsd:integer .
?answer sides:correspond_to_question ?q.
?q sides:has_for_number_of_proposals "4"^^xsd:
        integer.
FILTER NOT EXISTS {?answer sides:
        stronglyWrong "true"^^xsd:boolean }}}
}
```

Q16

```
CONSTRUCT
{?answer sides:has_for_result "0"^^xsd:integer ?g}
WHERE {
SELECT ?g ?answer {GRAPH ?g {
?answer sides:correspond_to_question ?q.
?q sides:has_for_number_of_proposals ?np.
?answer sides:has_for_number_of_discordance ?n.
FILTER (?np > 3 && ?np <6 && ?n > 2).}}
}
```

Q17

```
CONSTRUCT
{?answer sides:has_for_result "0.3"^^xsd:decimal
        ?g}
WHERE {
SELECT ?g ?answer {GRAPH ?g {
?answer sides:has_for_number_of_discordance
        "1"^^xsd:integer .
?answer sides:correspond_to_question ?q.
?q sides:has_for_number_of_proposals "3"^^xsd:
        integer.
FILTER NOT EXISTS {?answer sides:
        stronglyWrong "true"^^xsd:boolean}}}
}
```

Q18

```
CONSTRUCT
{?answer sides:has_for_result "0"^^xsd:integer ?g}
WHERE {
SELECT ?g ?answer {GRAPH ?g {
?answer sides:has_for_number_of_discordance ?n.
?answer sides:correspond_to_question ?q.
?q sides:has_for_number_of_proposals "3"^^xsd:
        integer.
FILTER (?n > 1)}}
}
```

Figure 5.22 18 CONSTRUCT queries for modular reasoning. Showing queries from Q10 to Q18.

Fig.5.23 also shows that modular reasoning with bucketing has similar performance to parallel from D1 to D5. However, modular is outperformed by parallel for datasets larger than D5.

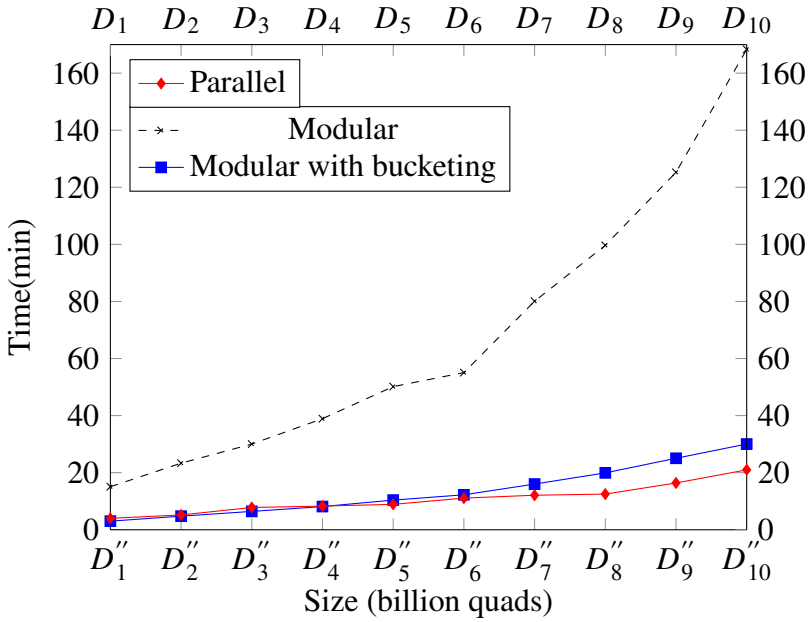

Figure 5.23 Modular-based forward-chaining reasoning time

As a possible explanation we analyse whether performance is affected by data distribution. First of all, we calculated the total number of triples per each student named graph. Next, we use the numerical values obtained to create a histogram for each dataset.

We found that datasets D1,D2,D3,D4 and D5 have few named graphs with small number of triples and a larger number of named graphs with a large number of triples.

On the other hand, we found that D6,D7,D8,D9 and D10 have a large number of named graphs with a small number of triples each and a smaller number of named graphs with a large number of triples each.

We believe that bucketing does not work as expected from D6 to D10 due to the large number of named graphs with small number of triples. A possible explanation is that prior the experiment, all the datasets were created by grouping named graphs that were sorted in ascending order of size. The large number of named graphs with small number of triples generates tons of self-joined queries with high latency due to memory limitations.

Figure 5.24 shows a comparison between datasets of comparable size taken from the Table 5.5. The modular datasets are D''_1, D''_2 and D''_3 in bold blue, and the datasets used in the parallel forward chaining experiment are in bold red (D_4, D_5, D_6, D_7, D_8, D_9 and D_{10}). We observe that, given datasets of comparable size, modular forward chaining (with bucketing) outperforms parallel forward chaining. One possible explanation is that the

number of named graphs with a large number of triples is larger than the number of named graphs with small number of triples in these modular datasets. This reduces the number of queries and therefore, makes the query execution faster.

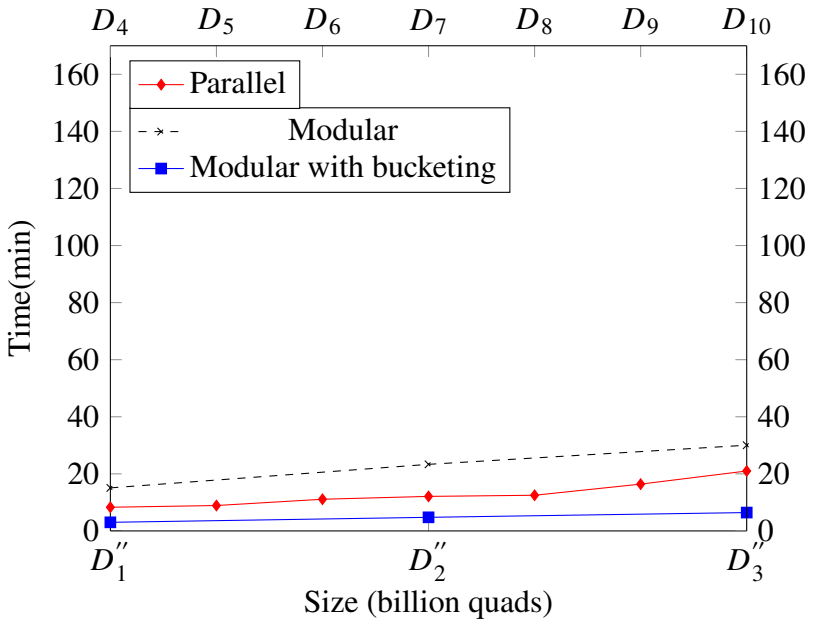

Figure 5.24 Modular-based forward-chaining reasoning time

5.6 CPU vs GPU serial forward chaining performance

In recent years, the usage of GPUs has dramatically reduced the computation time for Big Data pipelines by massively increasing the number of parallel tasks . We argue that SPARQL query performance can achieve a similar benefit when GPU support is enabled for a triplestore.

To investigate the impact of GPUs on SPARQL query performance, we extended the TESS architecture to include GPU acceleration, a hardware enhancement. In this section, we address CPU versus GPU serial forward chaining performance comparison as shown in Fig. 5.25.

5.6.1 Experimental protocol

We conducted two experiments to study how increasing data size affects the performance of CPU and GPU forward chaining reasoning. In the first, we evaluate the performance contribution of individual queries. during *parallel forward chaining reasoning* over datasets that are smaller than 2 billion triples. In the latter, we evaluate the *serial forward chaining*

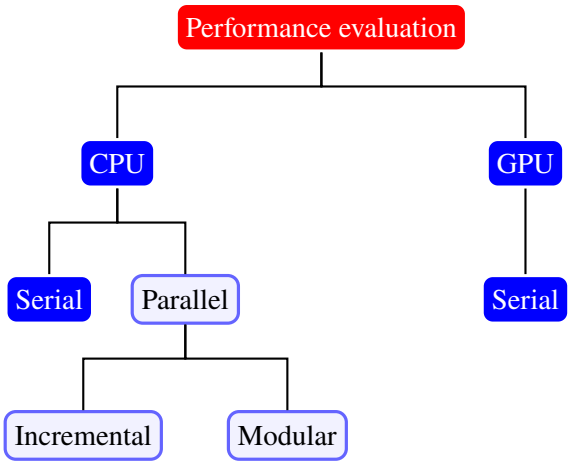

Figure 5.25 CPU-based vs GPU-based serial performance comparison

reasoning performance on a very large dataset of 12 billion triples. In both cases, TESS is deployed on a 4-node Spark cluster, where each node has 50G of CPU RAM memory and 64G of GPU memory. As for the forward chaining implementation, we have reused its serial and parallel versions used in the experiments in Section 5.3 without any modifications.

Datasets

For the first experiment, we selected only 3 datasets: the small size D1, the medium size D5 and the large size D10 from the Table 5.3 For the second experiment, we use the full Ontosides dataset with 12 billion triples.

Hardware setup

We used the same hardware for both experiments except that we did not enable GPU acceleration for the CPU forward chaining reasoning. We used Amazon Web Services (AWS) to host TESS on a EC2 P3 (p3.8xlarge) instance server with 4 Tesla V100 GPUs, 64G for GPU memory, 244G for CPU memory and 32 vCPUs.

The table 5.6 shows the Spark parameter settings used to configure TESS for optimal CPU/GPU parallelization and memory usage according to its online documentation[2].

[2] https://nvidia.github.io/spark-rapids/docs/configs.html

Hardware		Spark parameters for TESS	Values
GPU	CPU	driver-memory	40g
		executor-memory	50g
		executor-cores	7
		num-executors	4
		spark.rapids.memory.pinnedPool.size	32G
		spark.rapids.memory.gpu.pool	ARENA
		spark.rapids.memory.gpu.pooling.enabled	true
		spark.executor.resource.gpu.amount	1
		spark.task.resource.gpu.amount	0.25
		spark.rapids.sql.concurrentGpuTasks	1
		spark.locality.wait	0s
		spark.sql.files.maxPartitionBytes	512m
		spark.plugins	com.nvidia.spark.SQLPlugin
		spark.rapids.memory.gpu.maxAllocFraction	0.7
		spark.rapids.memory.gpu.allocFraction	0.5
		spark.rapids.sql.enabled	true
		spark.sql.shuffle.partitions	200
		spark.rapids.shuffle.enabled	false
		spark.rapids.memory.gpu.unspill.enabled	true

Table 5.6 Spark parameters for TESS

In terms of disk storage, we stored the data in an Amazon EBS volume with a 1T size disk with 125 MB/s throughput and 3000 IOPS. An IOP is a unit of measure that represents input/output operations per second.

For cache storage, we stored cache data in an Amazon EBS volume of size 2T disk at 800 MB/s and 12000 IOPS.

In both experiments, for performance measures and queries we follow those of Subsection 5.1.3 and Figure 3.4 respectively.

Evaluation results

The first experiment, parallel forward chaining reasoning is performed on three datasets of the Table 5.3: D1, D5 and D10 whose size is less than 2 billion triples. Figure 5.26 shows the *construct execution time* performance of each query that makes up the forward chaining. It clearly shows that queries executed via CPU outperform those executed via GPU. This is particularly evident for queries Q3 and Q10 where the coefficient for GPU performance grows linearly but with a much larger value than in CPU case.

A possible explanation lies on the query plan. A query plan in Spark can be optimised in runtime based on statistics that may not have been available when the query was originally

planned. We believe that once the data is read into in-memory columnar format, the query plan analyser evaluates that some parts of the plan may run faster on the CPU than on the GPU due to the relatively small size of the data to be processed. The query plan is then modified accordingly, but there is a bottleneck in the process of converting the data back from GPU columnar format to the CPU row format.

In the second experiment, serial forward chaining reasoning is performed on the large dataset Ontosides with 12 billion triples. The results differ from the previous experiment because a significant performance gain is achieved by using the GPU. We observe in Fig.5.27 that the *construct execution time* performance of all the queries is improved when GPU acceleration is enabled (red bars). We hypothesise that once the data is loaded into in-memory columnar format, the query plan analyser does not implement a fallback to row format because CPU processing cannot outperform GPU processing when a dataset is very large. We also observe that the performance boost is very significant for queries Q2, Q3 and Q10. One explanation could be that Q2, Q3 and Q10 are queries that are best suited for GPU acceleration, since Q2 and Q3 are group by operations with high cardinality, while Q10 contains joins operations with high cardinality.

Finally, Figure 5.28 shows how the GPU outperforms the CPU serial forward chaining performance by a factor of 1.5x on the very large dataset.

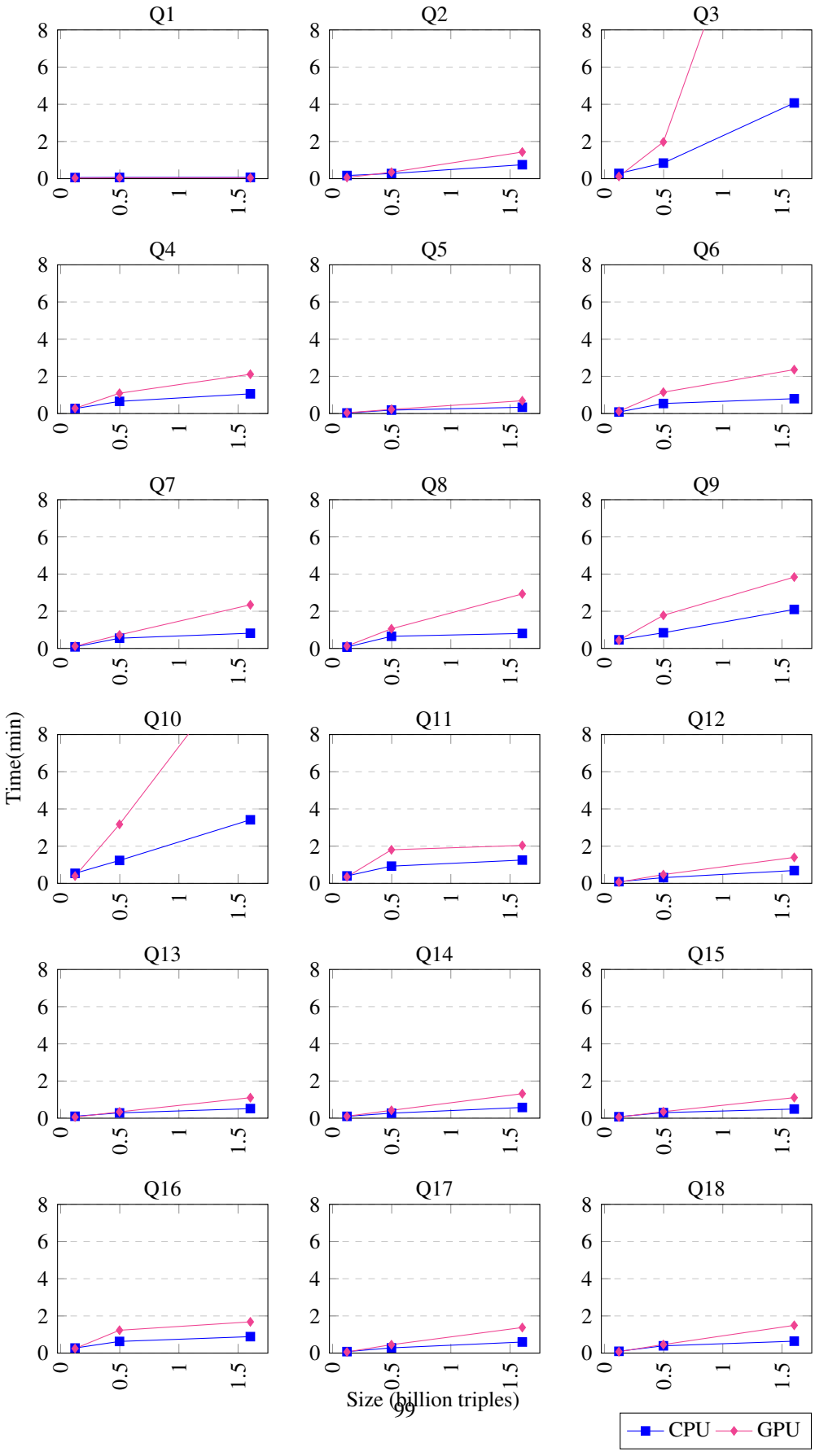

Figure 5.26 CPU vs GPU comparison of the CONSTRUCT queries performance over datasets D1, D5, and D10

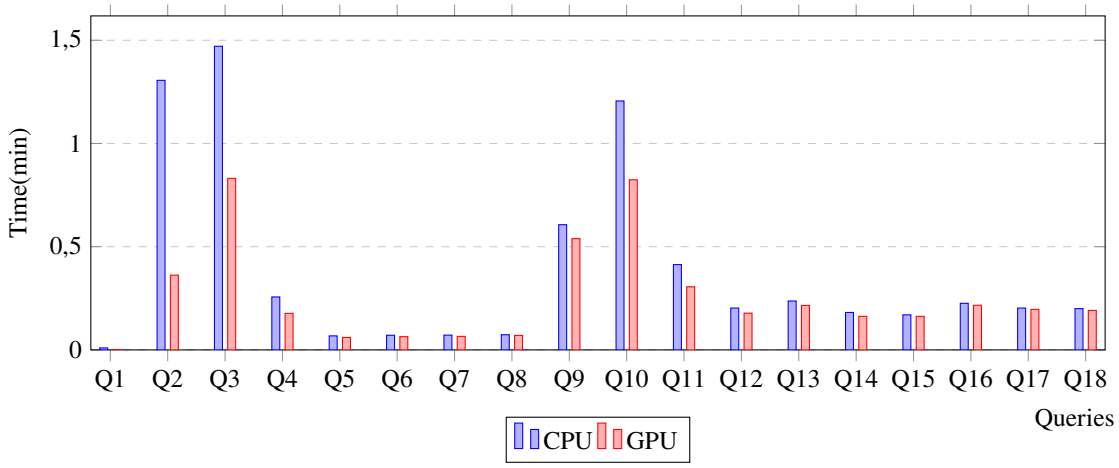

Figure 5.27 CPU vs GPU comparison of the CONSTRUCT query performance on the very large dataset

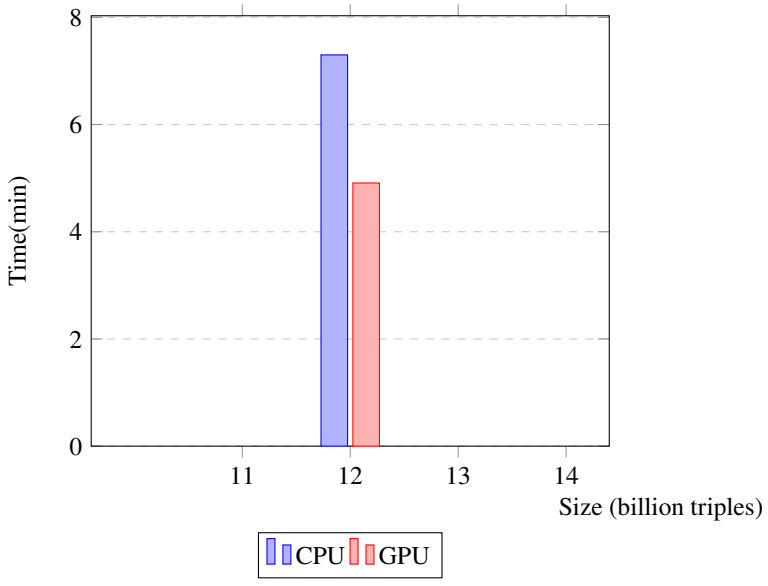

Figure 5.28 CPU vs GPU serial forward-chaining reasoning time applied to the very large dataset

5.7 Summary

We have evaluated several forward chaining reasoning implementations: serial, parallel and incremental and modular on datasets of increasing size. The results show that a) parallel outperforms serial, b) incremental can outperform parallel for delta sizes below a threshold, c) the information about how deltas are built is a valuable resource for reformulating queries, d) modular has a comparable performance to parallel when join columns are bucketed.

TESS has also been evaluated under GPU acceleration. The results show that: a) for datasets smaller than 2 billion triples, CPU forward chaining is a better choice than GPU. b) the use of GPU is recommended for very large datasets with *group by* and join operations with high cardinality.

We have identified some missing features in the SPARQL specification regarding the use of both UNION operation on named graphs and CONSTRUCT quads. In this respect, we have filled the gap by implementing these features at the application level.

Chapter 6

Conclusion

6.1 Summary of the contributions

The objective of this thesis is to design and implement a Big Data infrastructure to support the extraction, storage, reasoning and on-demand analytics queries of large amounts of educational data produced by SIDES. To achieve this goal, we followed two directions: Big Data architectures for reasoning at scale and Data modularization. We have discovered that both directions go hand in hand because they both aim to handle large volumes of data efficiently. While one approach finds the solution in scaling the infrastructure, the other approach finds it in optimizing computation by reducing the amount of data to be processed.

Chapters 1 and 2 of this thesis presented the introduction and the preliminary notions to be used throughout the thesis. We discussed Big Data and RDF knowledge graphs and we explored the potential challenges that Big Data technologies face in terms of reasoning and executing heavy and complex queries. Additionally, we provided formal definitions of some of the concepts related to Semantic Web, Ontology-based Data Access, and Rule-based Reasoning used in this thesis.

In Chapter 3 we presented the first contribution of this thesis: an OBDA methodology to support the evolution of a big knowledge base called OntoSIDES. The methodology consists of two steps: the first step is manual, based on expert guidance, and involves the manual construction of the OntoSIDES ontology enriched with rules, while the second step is automatic and involves the automatic population of the ontology using OBDA mappings. In this chapter, we also showed that OBDA mappings can be adapted to scale the materialization of a collection of RDF graphs by mapping RDF quad templates to queries in the data sources. Consequently, OBDA can be extended to organize a large RDF graph into modular components, effectively reducing the complexity of data analytics.

In Chapter 4, we presented the second contribution of this thesis: a big RDF triplestore called TESS with a modular architecture that includes support for massive and reliable data

updates, while also incorporating GPU acceleration for enhanced processing capabilities. We explored the features of the most reliable parallelization technologies on each layer of big RDF triplestores, including data distribution, storage, query processing, transactional metadata layer, and reasoning. As a result of this exploration, we described the selected components of TESS layer by layer, highlighting its unique features and advantages.

In Chapter 5, we presented the third contribution of this thesis: a comparative performance evaluation for complex queries and reasoning, aimed at identifying the most optimal approaches to implement forward chaining reasoning at scale in TESS. We described the experimental protocol employed to conduct our experiments. The experimental results were presented as follows: first, we presented the experimental results obtained for assessing the performance of complex query evaluations and the serial algorithm of forward chaining reasoning based on the iteration of CONSTRUCT queries. The results demonstrate that Virtuoso and GraphDB do not scale well with large RDF datasets, in contrast to TESS, which outperforms both systems across all datasets. Subsequently, we reported the results derived from comparing the runtime performance of TESS between the serial, parallel, parallel and incremental, and parallel and modular variants of the forward chaining algorithm. Additionally, we discussed the experimental outcomes from comparing TESS's performance when using either a CPU or GPU for executing the serial algorithm of forward chaining reasoning based on the iteration of CONSTRUCT queries.

6.2 Perspectives

There are several directions in which our research can potentially be extended but we focus on three main perspectives:

Mass updating of data in triplestores

In our research, the transactional metadata layer is exclusively responsible for ensuring the data consistency during mass updatings in triplestores, without user intervention. We believe that transactions support can be extended at user level by adding declarative statements to the SPARQL specification. With these declarative statements, a set of SPARQL queries could be grouped as a transaction in the same way that PostgreSQL does to guarantee ACID properties. SPARQL transactions could be a clear way to declare a set of CONSTRUCT queries used for rule-based reasoning. The theoretical implications and the practical implementation of this type of transaction have not yet been explored.

Modularization of RDF data at scale

The potential of OBDA systems to modularize RDF data is not limited to having a SQL/NoSQL database as a data source. In fact, an OBDA system can also be used to modularize an existing RDF graph by using SPARQL queries in the mapping sources. The query output can populate quad templates. We have started to use TESS as a data source

for an OBDA system and the results are promising. However, the use of complex SPARQL queries as a mapping source is limited due to the lack of materialized RDF views. A materialized RDF view could store the pre-computed output of a SPARQL complex query. The execution of a mapping could then perform a simple query over this materialized RDF view. We could use a named graph to store the precomputed output, but the named graph could become outdated very quickly if its data is frequently updated. One line of research would be to represent a named graph as a materialized RDF view that is automatically recomputed whenever data changes.

Development of a new generation of triplestores based on GPU technologies

Hardware technology such as GPUs evolves so quickly that it is difficult for data management to keep up with the latest improvements. The main limitation is that adopting new hardware would require rewriting the software of many triplestores from scratch to take advantage of the new GPU parallelization techniques. To avoid this, we have shown that a modular architecture for a triplestore can be a good alternative to take advantage of new hardware capabilities. However, our architecture is limited to using GPU optimizations designed for the SQL translation of a SPARQL query. We believe that more research is needed to eliminate the SPARQL/SQL translation and develop GPU optimizations that target SPARQL queries directly.

Appendix A

Other Information

A.1 The OBDA mappings

[**PrefixDeclaration**]
: http :// www.side-sante.fr/sides#
owl: http :// www.w3.org/2002/07/owl#
rdf: http :// www.w3.org/1999/02/22-rdf-syntax-ns#
xml: http :// www.w3.org/XML/1998/namespace
xsd: http :// www.w3.org/2001/XMLSchema#
rdfs: http :// www.w3.org/2000/01/rdf-schema#
spin: http :// spinrdf.org/spin#
sides: http :// www.side-sante.fr/sides#

[**SourceDeclaration**]
sourceUri sides
connectionUrl jdbc:postgresql://XX.XX.XX.XX:9009/sides
username sides
password XXX
driverClass org.postgresql.Driver

[**MappingDeclaration**] **@collection** [[

mappingId urn:test
target sides:test{id} a sides:test .
source select a.id as id, a.title, a.startdate, a.enddate from public.assessment a

mappingId urn:test_has_for_title
target sides:test{id} sides:has_for_title "{ title }"^^xsd:string .

source select a.id as id, a.title as title, a.startdate, a.enddate from public.assessment a

mappingId urn:test_start_date
target sides:test{id} sides:starting_date_of_test "{startdate}"^^xsd:dateTime .
source select a.id as id, a.title, a.startdate as startdate, a.enddate as enddate from public.assessment a where a.startdate is not null

mappingId urn:test_end_date
target sides:test{id} sides:ending_date_of_test "{enddate}"^^xsd:dateTime .
source select a.id as id, a.title, a.startdate as startdate, a.enddate as enddate from public.assessment a where a.enddate is not null

mappingId urn:evaluation_has_for_textual_content
target sides:eval{id} sides:has_for_textual_content "{eval_introduction}"^^xsd:string .
source SELECT di.id as id, string_agg(di.introduction, '') as eval_introduction from ontosides.docimocontent_introduction di
inner join ontosides.evaluation ev on ev.pool_id = di.id
group by di.id

mappingId urn:relation_evaluation_type_question
target sides:eval{pool_id} sides:has_for_question sides:q{question_id} .
source select pq.pool_id as pool_id, pq.question_id as question_id, pq.position from pool_question pq
inner join ontosides.evaluation ev on pq.pool_id = ev.pool_id

mappingId urn:relation_test_evaluation_type_qi
target sides:test{assessment_id} sides:is_made_of sides:eval{pool_id} .
source select ae.assessment_id, ae.pool_id from ontosides.assessment_evaluation ae
inner join docimocontent dc on dc.id = ae.pool_id
where dc.discr in ('qi')

mappingId urn:relation_test_evaluation_type_dp_tcspool_lca
target sides:test{assessment_id} sides:is_made_of sides:eval{docimocontent_id} .
source select assessment_id, docimocontent_id from docimocontentassessment dca
inner join docimocontent dc on dc.id = dca.docimocontent_id
where dc.discr in ('dp', 'tcspool', 'lca')

mappingId urn:relation_test_student
target sides:test{assessment_id} sides:has_for_registrant sides:stu{participant_id} .
source select p.assessment_id as assessment_id, p.participant_id as participant_id from public.participant p

mappingId urn:student
target sides:stu{participant_id} a sides:student ; sides:has_for_id "{participant_id}"^^xsd:integer .
source select participant_id from ontosides.student

mappingId urn:proposal_of_answer
target sides:prop{id} a sides:proposal_of_answer .
source select c.id as id, c.question_id as question_id from public.choice c
inner join ontosides.question q on c.question_id = q.question_id

mappingId urn:relation_question_proposal_of_answer
target sides:q{question_id} sides:has_for_proposal_of_answer sides:prop{id}.
source select c.question_id as question_id, c.id as id from public.choice c
inner join ontosides.question q on c.question_id = q.question_id

mappingId urn:proposal_of_answer_label
target sides:prop{id} sides:has_for_textual_content "{label}"^^xsd:string .
source SELECT cc.id as id, string_agg(cc.label, '') as label from choice c inner join
ontosides.choice_label cc on cc.id = c.id
where coalesce(cc.label, '') != ''
group by cc.id

mappingId urn:proposal_of_answer_valid
target sides:prop{id} sides:has_for_correction "{is_valid}"^^xsd:boolean .
source select c.id as id, ccc.valid, CASE ccc.valid WHEN 't' THEN 'true' WHEN 'f' THEN 'false' END as is_valid
from public.choice c
inner join correction.choice_correction ccc on c.correction_id = ccc.id
inner join ontosides.question q on c.question_id = q.question_id

mappingId urn:speciality
target sides:{uri} a sides:speciality ; rdfs:label "{name}"@fr .
source select s.uri as uri, me.name from meta_speciality me inner join ontosides.speciality s on s.speciality_id = me.id

mappingId urn:relation_eval_medical_specialty
target sides:eval{docimocontent_id} sides:is_linked_to_the_medical_speciality sides:{uri} .
source select ds.docimocontent_id as docimocontent_id, s.uri as uri from docimocontent_speciality ds
inner join meta_speciality me on ds.speciality_id = me.id
inner join ontosides.speciality s on s.speciality_id = me.id
inner join docimocontent d on d.id = ds.docimocontent_id

```
        inner join ontosides.evaluation ev on d.id = ev.pool_id

mappingId urn:relation_question_medical_specialty
target    sides:q{docimocontent_id} sides:is_linked_to_the_medical_speciality sides:{uri} .
source    select ds.docimocontent_id as docimocontent_id, s.uri as uri from
      docimocontent_speciality ds
        inner join meta_speciality me on ds.speciality_id = me.id
        inner join ontosides.speciality s on s.speciality_id = me.id
        inner join docimocontent d on d.id = ds.docimocontent_id
        inner join ontosides.question q on d.id = q.question_id

mappingId urn:meta_cross_knowledge
target    sides:{uri} a sides:referential_entity ; rdfs:label "{name}"@fr .
source    select cke.cke_id as id, cke.uri as uri, mck.name as name from ontosides.
      cross_knowledge_entity cke inner join
            meta_cross_knowledge mck on mck.id = cke.cke_id

mappingId urn:relation_eval_cross_knowledge_entity
target    sides:eval{docimocontent_id} sides:is_linked_to_the_cross_knowledge_entity sides:{
      uri} .
source    select dc.docimocontent_id as docimocontent_id, cke.uri as uri from
      docimocontent_crossknowledge dc
        inner join ontosides.cross_knowledge_entity cke on cke.cke_id = dc.crossknowledge_id
        inner join docimocontent d on d.id = dc.docimocontent_id
        inner join ontosides.evaluation ev on d.id = ev.pool_id

mappingId urn:relation_question_cross_knowledge_entity
target    sides:q{docimocontent_id} sides:is_linked_to_the_cross_knowledge_entity sides:{uri} .
source          select dc.docimocontent_id as docimocontent_id, cke.uri as uri from
      docimocontent_crossknowledge dc
        inner join ontosides.cross_knowledge_entity cke on cke.cke_id = dc.crossknowledge_id
        inner join docimocontent d on d.id = dc.docimocontent_id
        inner join ontosides.question q on d.id = q.question_id

mappingId urn:question_QMA
target    sides:q{id} a sides:QMA ; sides:has_for_title "{title}"^^xsd:string .
source    select d.id as id, d.title as title, d.discr from public.docimocontent d
        inner join ontosides.question q on d.id = q.question_id
        where d.discr = 'qrm'

mappingId urn:question_QUA
target    sides:q{id} a sides:QUA ; sides:has_for_title "{title}"^^xsd:string .
```

source select d.id as id, d.title as title, d.discr from public.docimocontent d
inner join ontosides.question q on d.id = q.question_id
where d.discr = 'qru'

mappingId urn:question_QSOA
target sides:q{id} a sides:QSOA ; sides:has_for_title "{title}"^^xsd:string .
source select d.id as id, d.title as title, d.discr from public.docimocontent d
inner join ontosides.question q on d.id = q.question_id
where d.discr = 'textq'

mappingId urn:question_TCS_question
target sides:q{id} a sides:TCS_question ; sides:has_for_title "{title}"^^xsd:string .
source select d.id as id, d.title as title, d.discr from public.docimocontent d
inner join ontosides.question q on d.id = q.question_id
where d.discr = 'tcsquestion'

mappingId urn:question_has_for_textual_content
target sides:q{question_id} sides:has_for_textual_content "{statement}"^^xsd:string .
source select q.question_id as question_id, string_agg(dc.statement, '') as statement from ontosides.question q
inner join ontosides.docimocontent_statement dc on dc.id = q.question_id
where dc.statement is not null
group by q.question_id

mappingId urn:evaluation_type_set_isolation_question
target sides:eval{id} a sides:set_of_isolated_questions ; sides:has_for_title "{ title }"^^xsd:string .
source select d.id as id, d.title as title, d.discr from public.docimocontent d
inner join ontosides.evaluation ev on d.id = ev.pool_id where d.discr = 'qi'

mappingId urn:evaluation_type_progressive_clinical_case
target sides:eval{id} a sides:progressive_clinical_case ; sides:has_for_title "{ title }"^^xsd:string .
source select d.id as id, d.title as title, d.discr from public.docimocontent d
inner join ontosides.evaluation ev on d.id = ev.pool_id where d.discr = 'dp'

mappingId urn:evaluation_type_TCS
target sides:eval{id} a sides:TCS ; sides:has_for_title "{ title }"^^xsd:string .
source select d.id as id, d.title as title, d.discr from public.docimocontent d
inner join ontosides.evaluation ev on d.id = ev.pool_id where d.discr = 'tcspool'

mappingId urn:evaluation_type_LCA

target sides:eval{id} a sides:LCA ; sides:has_for_title "{ title }"^^xsd:string .
source select d.id as id, d.title as title , d.discr from public.docimocontent d
inner join ontosides.evaluation ev on d.id = ev.pool_id where d.discr = 'lca'

mappingId urn:relation_question_learning_objective
target sides:q{re**source**id} sides:is_linked_to_ECN_referential_entity sides:
learning_objective_{code} .
source select st.re**source**id as re**source**id, lo.code as code from public.skilllink st
inner join ontosides.learning_objective lo on lo.id = st. skillid
inner join ontosides.question q on st.re**source**id = q.question_id

mappingId urn:relation_question_learning_sub_objective
target sides:q{re**source**id} sides:is_linked_to_ECN_referential_entity sides:
learning_sub_objective_{code_parent}_{code_child} .
source select st.re**source**id as re**source**id, lso.code_parent as code_parent, lso.code_child
as code_child from public.skilllink st
inner join ontosides.learning_sub_objective lso on lso.id_child = st. skillid
inner join ontosides.question q on st.re**source**id = q.question_id

mappingId urn:relation_eval_learning_sub_objective
target sides:eval{re**source**id} sides:is_linked_to_ECN_referential_entity sides:
learning_sub_objective_{code_parent}_{code_child} .
source select st.re**source**id as re**source**id, lso.code_parent as code_parent, lso.code_child
as code_child from public.skilllink st
inner join ontosides.learning_sub_objective lso on lso.id_child = st. skillid
inner join ontosides.evaluation ev on st.re**source**id = ev.pool_id

mappingId urn:relation_eval_learning_objective
target sides:eval{re**source**id} sides:is_linked_to_ECN_referential_entity sides:
learning_objective_{code} .
source select st.re**source**id as re**source**id, lo.code as code from public.skilllink st
inner join ontosides.learning_objective lo on lo.id = st. skillid
inner join ontosides.evaluation ev on st.re**source**id = ev.pool_id

mappingId urn:sequence_of_questions_by_evaluation
target sides:lq{pool_id} rdf:_{pos} sides:q{question_id} .
source select pq.pool_id as pool_id, pq.question_id as question_id, (pq.position +1) as pos
from public.pool_question pq
inner join ontosides.evaluation ev on ev.pool_id = pq.pool_id

mappingId urn:relation_evaluation_list_of_questions_dp

target sides:lq{pool_id} a rdf:Seq . sides:eval{pool_id} sides:has_for_list_of_questions sides:lq{pool_id} .
source select ev.pool_id as pool_id from ontosides.evaluation ev inner join public.docimocontent d on d.id = ev.pool_id where d.discr = 'dp'

mappingId urn:relation_evaluation_list_of_questions_lca
target sides:lq{pool_id} a rdf:Seq . sides:eval{pool_id} sides:has_for_list_of_questions sides:lq{pool_id} .
source select ev.pool_id as pool_id from ontosides.evaluation ev inner join public.docimocontent d on d.id = ev.pool_id where d.discr = 'lca'

mappingId urn:relation_answer_question
target sides:answer{response_id} sides:correspond_to_question sides:q{question_id} .
source select concat(cast(response_id as text), cast(choice_id as text)) as response_id_new ,
rt.id as response_id, rc.choice_id as choice_id, c.question_id as question_id
from response.responses_choices rc
inner join response.response_table rt on rt.id = rc.response_id
inner join choice c on c.id = rc.choice_id
inner join correction.choice_correction ccc on ccc.id = c.correction_id
where response_id not in (select response_id from ontosides.excluded_responses_ids)

mappingId urn:action_to_answer
target sides:adr{response_id_new} a sides:action_to_answer .
source select response_id_new, begindate from ontosides.response where response_id not in (select response_id from ontosides.excluded_responses_ids)

mappingId urn:relation_action_to_answer_test
target sides:adr{response_id_new} sides:done_during sides:test{assessment_id} .
source select response_id_new, assessment_id from ontosides.response where response_id not in (select response_id from ontosides.excluded_responses_ids)

mappingId urn:answer
target sides:answer{response_id} a sides:answer .
source select response_id from ontosides.response where response_id not in (select response_id from ontosides.excluded_responses_ids)

mappingId urn:relation_answer_has_for_timestamp
target sides:answer{response_id} sides:has_for_timestamp "{begindate}"^^xsd:dateTime .
source select distinct response_id, begindate from ontosides.response where begindate is not null and response_id not in (select response_id from ontosides.

excluded_responses_ids)

mappingId urn:relation_answer_action_to_answer
target sides:adr{response_id_new} sides:is_part_of sides:answer{response_id} .
source select response_id_new, response_id from ontosides.response where response_id not in (select response_id from ontosides.excluded_responses_ids)

mappingId urn:relation_answer_student
target sides:answer{response_id} sides:done_by sides:stu{participant_id} .
source select response_id, participant_id from ontosides.response where response_id not in (select response_id from ontosides.excluded_responses_ids)

mappingId urn:relation_has_rightly_ticked
target sides:adr{response_id_new} sides:has_rightly_ticked sides:prop{choice_id} .
source select response_id_new, choice_id ,valid from ontosides.response r where r.valid = true and response_id not in (select response_id from ontosides.excluded_responses_ids)

mappingId urn:relation_has_wrongly_ticked
target sides:adr{response_id_new} sides:has_wrongly_ticked sides:prop{choice_id} .
source select response_id_new, choice_id, valid from ontosides.response r where r.invalid = true and response_id not in (select response_id from ontosides.excluded_responses_ids)

mappingId urn:relation_answer_question_fixed
target sides:answer{response_id_fixed} sides:correspond_to_question sides:q{question_id} .
source select response_id_fixed, question_id from ontosides.responses_fixed

mappingId urn:action_to_answer_fixed
target sides:adr{response_id_new} a sides:action_to_answer .
source select response_id_new, begindate from ontosides.responses_fixed

mappingId urn:relation_action_to_answer_test_fixed
target sides:adr{response_id_new} sides:done_during sides:test{assessment_id} .
source select response_id_new, assessment_id from ontosides.responses_fixed

mappingId urn:answer_fixed
target sides:answer{response_id_fixed} a sides:answer .
source select response_id_fixed from ontosides.responses_fixed

mappingId urn:relation_answer_has_for_timestamp_fixed
target sides:answer{response_id_fixed} sides:has_for_timestamp "{begindate}"^^xsd:dateTime .
source select distinct response_id_fixed, begindate from ontosides.responses_fixed where

begindate is not null

mappingId urn:relation_answer_action_to_answer_fixed
target sides:adr{response_id_new} sides:is_part_of sides:answer{response_id_fixed} .
source select response_id_new, response_id_fixed from ontosides.responses_fixed

mappingId urn:relation_answer_student_fixed
target sides:answer{response_id_fixed} sides:done_by sides:stu{participant_id} .
source select response_id_fixed, participant_id from ontosides.responses_fixed

mappingId urn:relation_has_rightly_ticked_fixed
target sides:adr{response_id_new} sides:has_rightly_ticked sides:prop{choice_id} .
source select response_id_new, choice_id ,valid from ontosides.responses_fixed r where r.
valid = true

mappingId urn:relation_has_wrongly_ticked_fixed
target sides:adr{response_id_new} sides:has_wrongly_ticked sides:prop{choice_id} .
source select response_id_new, choice_id, valid from ontosides.responses_fixed r where r.
invalid = true

mappingId urn:picture
target sides:picture{file_id} a sides:picture .
source select id, file_id from (
select id, file_id from ontosides.docimocontent_comment_images
union
select id, file_id from ontosides.docimocontent_introduction_images
union
select id, file_id from ontosides.docimocontent_statement_images
) as t

mappingId urn:video
target sides:video{file_id} a sides:video .
source select id, file_id from (
select id, file_id from ontosides.docimocontent_comment_videos
union
select id, file_id from ontosides.docimocontent_introduction_videos
union
select id, file_id from ontosides.docimocontent_statement_videos
) as t

mappingId urn:relation_question_picture_multimedia_content
target sides:q{question_id} sides:has_for_multimedia_content sides:picture{file_id} .

```
source    select question_id, file_id from (
              select q.question_id, dci.file_id from ontosides.question q
              inner join ontosides.docimocontent_comment_images dci on dci.id = q.
    question_id
              union
              select q.question_id, dii.file_id from ontosides.question q
              inner join ontosides.docimocontent_introduction_images dii on dii.id = q.
    question_id
              union
              select q.question_id, dsi.file_id from ontosides.question q
              inner join ontosides.docimocontent_statement_images dsi on dsi.id = q.
    question_id
              ) as t

mappingId urn:relation_question_video_multimedia_content
target    sides:q{question_id} sides:has_for_multimedia_content sides:video{file_id} .
source    select question_id, file_id from (
              select q.question_id, dcv.file_id from ontosides.question q
              inner join ontosides.docimocontent_comment_videos dcv on dcv.id = q.
    question_id
              union
              select q.question_id, div.file_id from ontosides.question q
              inner join ontosides.docimocontent_introduction_videos div on div.id = q.
    question_id
              union
              select q.question_id, dsv.file_id from ontosides.question q
              inner join ontosides.docimocontent_statement_videos dsv on dsv.id = q.
    question_id
              ) as t


mappingId urn:relation_evaluation_picture_multimedia_content
target    sides:eval{pool_id} sides:has_for_multimedia_content sides:picture{file_id} .
source    select pool_id, file_id from (
              select ev.pool_id, dci.file_id from ontosides.evaluation ev
              inner join ontosides.docimocontent_comment_images dci on dci.id = ev.pool_id
              union
              select ev.pool_id, dii.file_id from ontosides.evaluation ev
              inner join ontosides.docimocontent_introduction_images dii on dii.id = ev.pool_id
              union
              select ev.pool_id, dsi.file_id from ontosides.evaluation ev
              inner join ontosides.docimocontent_statement_images dsi on dsi.id = ev.pool_id
```

```
        ) as t

mappingId urn:relation_evaluation_video_multimedia_content
target    sides:eval{pool_id} sides:has_for_multimedia_content sides:video{file_id} .
source    select pool_id, file_id from (
            select ev.pool_id, dcv.file_id from ontosides.evaluation ev
            inner join ontosides.docimocontent_comment_videos dcv on dcv.id = ev.pool_id
            union
            select ev.pool_id, div.file_id from ontosides.evaluation ev
            inner join ontosides.docimocontent_introduction_videos div on div.id = ev.pool_id
            union
            select ev.pool_id, dsv.file_id from ontosides.evaluation ev
            inner join ontosides.docimocontent_statement_videos dsv on dsv.id = ev.pool_id
            ) as t


mappingId urn:question_comment_has_for_textual_content
target    sides:q{id} sides:has_for_comment sides:comment{comment_block_id} . sides:
    comment{comment_block_id} a sides:comment ; sides:has_for_textual_content "{comment
    }"^^xsd:string .
source    select c.id as id, comment_block_id, string_agg(c.comment, '') as comment from
    ontosides.docimocontent_comment c
            inner join ontosides.question q on q.question_id = c.id
            where coalesce(c.comment, '') != ''
            group by c.id, comment_block_id


mappingId urn:question_comment_has_for_multimedia_content_images
target    sides:q{id} sides:has_for_comment sides:comment{comment_block_id} . sides:
    comment{comment_block_id} a sides:comment ; sides:has_for_multimedia_content sides:
    picture{file_id} .
source    select c.id as id, c.comment_block_id, c.file_id from ontosides.
    docimocontent_comment_images c
            inner join ontosides.question q on q.question_id = c.id


mappingId urn:question_comment_has_for_multimedia_content_videos
target    sides:q{id} sides:has_for_comment sides:comment{comment_block_id} . sides:
    comment{comment_block_id} a sides:comment ; sides:has_for_multimedia_content sides:
    video{file_id} .
source    select v.id as id, v.comment_block_id, v.file_id from ontosides.
    docimocontent_comment_videos v
```

inner join ontosides.question q on q.question_id = v.id

mappingId urn:proposal_of_answer_comment_has_for_textual_content
target sides:prop{id} sides:has_for_comment sides:comment{comment_block_id} . sides:comment{comment_block_id} a sides:comment ; sides:has_for_textual_content "{comment}"^^xsd:string .
source select c.id as id, comment_block_id, string_agg(c.comment, '') as comment from ontosides.choice_comment c
where coalesce(c.comment, '') != ''
group by c.id, comment_block_id

mappingId urn:proposal_of_answer_comment_has_for_multimedia_content_images
target sides:q{id} sides:has_for_comment sides:comment{comment_block_id} . sides:comment{comment_block_id} a sides:comment ; sides:has_for_multimedia_content sides:picture{file_id} .
source select id, comment_block_id, file_id from ontosides.
choice_comment_content_images

mappingId urn:proposal_of_answer_comment_has_for_multimedia_content_videos
target sides:q{id} sides:has_for_comment sides:comment{comment_block_id} . sides:comment{comment_block_id} a sides:comment ; sides:has_for_multimedia_content sides:video{file_id} .
source select id, comment_block_id, file_id from ontosides.choice_comment_content_videos

mappingId urn:multimedia_image_metadata
target sides:picture{id} sides:has_filename "{filename}"^^xsd:string; sides:has_basepath "{basepath}"^^xsd:string .
source select f.filename, f.basepath, f.id from file f
inner join filecontext fc on fc.file_id = f.id
inner join theia_block tb on tb.filecontext_id = fc.id
where tb.dtype = 'image';

mappingId urn:multimedia_video_metadata
target sides:video{id} sides:has_filename "{filename}"^^xsd:string; sides:has_basepath "{basepath}"^^xsd:string .
source select f.filename, f.basepath, f.id from file f
inner join filecontext fc on fc.file_id = f.id
inner join theia_block tb on tb.filecontext_id = fc.id
where tb.dtype = 'image';

mappingId urn:test_has_for_context_national_training
target sides:test{id} sides:has_for_context "{context}"^^xsd:string .
source select id, 'national training' as context from assessment
where discr = 'docimocontenttraining' and isnational is true

mappingId urn:test_has_for_context_local_training
target sides:test{id} sides:has_for_context "{context}"^^xsd:string .
source select id, 'local training' as context from assessment
where discr = 'docimocontenttraining' and isnational is not true

mappingId urn:test_has_for_context_exam
target sides:test{id} sides:has_for_context "{context}"^^xsd:string .
source select id, 'exam' as context from assessment
where discr = 'exam'

mappingId urn:relation_has_for_weight_of_correction
target sides:prop{id} sides:has_for_weight_of_correction "{weight_of_correction}"^^xsd:string .
source select c.id, cc.fatal, cc.valid,
case
when cc.fatal = false then 'Normal'
when cc.fatal = true and cc.valid =false then 'Unacceptable'
when cc.fatal = true and cc.valid =true then 'Indispensable'
end
as weight_of_correction
from choice c
inner join correction.choice_correction cc on cc.id = c.correction_id

mappingId urn:question_has_for_weight
target sides:q{id} sides:has_for_weight "{weight}".
source select id, weight from docimocontent dc where dc.discr in ('qrm', 'qru', 'textq', 'tcsquestion')

mappingId urn:rating_questions
target sides:q{question_id} sides:has_rating sides:rating{rating_id}. sides:rating{rating_id} a sides:rating ; sides:given_by sides:stu{user_id} ; sides:has_for_timestamp "{updatedat

}"^^xsd:dateTime ; sides:has_for_value "{value}"^^xsd:integer .
source select ass.docimocontent_id as question_id, tur.id as rating_id, tur.user_id, tur.createdat, tur.updatedat, tur.value
from theiacar_user_rating tur
left join theiacar_comment_rating_thread tc on tur.commentratingthread_id=tc.id
left join docimocontentassessment ass on tc.identity = ass.assessment_id
left join assessment as ass1 on ass1.id = ass.assessment_id
left join docimocontent dc on dc.id=ass.docimocontent_id
where ass1.isnational=true and user_id is not null
and (dc.discr = 'qrm' or dc.discr = 'qru')

mappingId urn:rating_evaluations
target sides:eval{eval_id} sides:has_rating sides:rating{rating_id}. sides:rating{rating_id} a sides:rating ; sides:given_by sides:stu{user_id} ; sides:has_for_timestamp "{updatedat}"^^xsd:dateTime ; sides:has_for_value "{value}"^^xsd:integer .
source select ass.docimocontent_id as eval_id, tur.id as rating_id, tur.user_id, tur.createdat, tur.updatedat, tur.value
from theiacar_user_rating tur
left join theiacar_comment_rating_thread tc on tur.commentratingthread_id=tc.id
left join docimocontentassessment ass on tc.identity = ass.assessment_id
left join assessment as ass1 on ass1.id = ass.assessment_id
left join docimocontent dc on dc.id=ass.docimocontent_id
where ass1.isnational=true and user_id is not null
and dc.discr = 'dp'

mappingId urn:has_for_expected_answer_text
target sides:q{question_id} sides:has_for_expected_answer_text "{textvalue}" .
source select question_id, textvalue from correction.text_correction where textvalue is not null and textvalue <> '' and question_id is not null

]]

References

[1] T. Chawla, G. Singh, E. S. Pilli, M. Govil, Storage, partitioning, indexing and retrieval in big rdf frameworks: A survey, Computer Science Review 38 (2020) 100309. URL: https://www.sciencedirect.com/science/article/pii/S1574013720304093. doi:https://doi.org/10.1016/j.cosrev.2020.100309.

[2] Hadoop architecture, https://hadoop.apache.org/docs/r3.3.0/hadoop-project-dist/hadoop-hdfs/HdfsDesign.html, 2022. Accessed: 2022-10-26.

[3] Acelerating apache spark 3, https://www.nvidia.com/en-us/deep-learning-ai/solutions/data-science/apache-spark-3/ebook-sign-up/, 2022. Accessed: 2022-11-30.

[4] Spark ai summit 2020 highlights: Innovations to improve spark 3.0 performance, https://www.infoq.com/news/2020/07/spark-ai-summit-performance-gpu/, 2022. Accessed: 2022-11-30.

[5] Z. Kaoudi, I. Manolescu, RDF in the clouds: a survey, VLDB J. 24 (2015) 67–91. URL: https://doi.org/10.1007/s00778-014-0364-z. doi:10.1007/s00778-014-0364-z.

[6] W. Ali, M. Saleem, B. Yao, A. Hogan, A. N. Ngomo, Storage, indexing, query processing, and benchmarking in centralized and distributed RDF engines: A survey, CoRR abs/2009.10331 (2020). URL: https://arxiv.org/abs/2009.10331. arXiv:2009.10331.

[7] G. Agathangelos, G. Troullinou, H. Kondylakis, K. Stefanidis, D. Plexousakis, RDF query answering using apache spark: Review and assessment, in: 34th IEEE International Conference on Data Engineering Workshops, ICDE Workshops 2018, Paris, France, April 16-20, 2018, IEEE Computer Society, 2018, pp. 54–59. URL: https://doi.org/10.1109/ICDEW.2018.00016. doi:10.1109/ICDEW.2018.00016.

[8] A. Schätzle, M. Przyjaciel-Zablocki, S. Skilevic, G. Lausen, S2rdf: Rdf querying with sparql on spark, Proc. VLDB Endow. 9 (2016) 804–815. URL: https://doi.org/10.14778/2977797.2977806. doi:10.14778/2977797.2977806.

[9] H. Naacke, O. Curé, B. Amann, SPARQL query processing with apache spark, CoRR abs/1604.08903 (2016). URL: http://arxiv.org/abs/1604.08903. arXiv:1604.08903.

[10] O. Palombi, F. Jouanot, N. Nziengam, B. Omidvar-Tehrani, M.-C. Rousset, A. Sanchez, Ontosides: Ontology-based student progress monitoring on the national evaluation system of french medical schools, Artificial intelligence in medicine 96 (2019) 59–67.

[11] A. Sanchez-Ayte, F. Jouanot, M.-C. Rousset, Construct queries performance on a spark-based big rdf triplestore, in: P. Groth, M.-E. Vidal, F. Suchanek, P. Szekley, P. Kapanipathi,

C. Pesquita, H. Skaf-Molli, M. Tamper (Eds.), The Semantic Web, Springer International Publishing, Cham, 2022, pp. 444–460.

[12] Rdf 1.1 concepts and abstract syntax, https://www.w3.org/TR/rdf11-concepts/, 2023. Accessed: 2023-02-19.

[13] Rdf 1.1 datasets, https://www.w3.org/TR/2014/REC-rdf11-concepts-20140225/#section-dataset, 2023. Accessed: 2023-02-19.

[14] Sparql 1.1 query language, https://www.w3.org/TR/sparql11-query, 2023. Accessed: 2023-02-19.

[15] Owl 2 web ontology language document overview, https://www.w3.org/TR/owl2-overview/, 2023. Accessed: 2023-02-19.

[16] Owl 2 web ontology language profiles (second edition), https://www.w3.org/TR/owl2-profiles/, 2023. Accessed: 2023-02-19.

[17] D. Calvanese, G. De Giacomo, D. Lemho, M. Lenzerini, R. Rosati, Dl-lite: Tractable description logics for ontologies, in: Proceedings of the 20th National Conference on Artificial Intelligence - Volume 2, AAAI'05, AAAI Press, 2005, p. 602–607.

[18] D. Maier, K. T. Tekle, M. Kifer, D. S. Warren, Datalog: Concepts, History, and Outlook, Association for Computing Machinery and Morgan amp; Claypool, 2018, p. 3–100. URL: https://doi.org/10.1145/3191315.3191317.

[19] A. Calì, G. Gottlob, T. Lukasiewicz, Datalog±: A unified approach to ontologies and integrity constraints, in: Proceedings of the 12th International Conference on Database Theory, ICDT '09, Association for Computing Machinery, New York, NY, USA, 2009, p. 14–30. URL: https://doi.org/10.1145/1514894.1514897. doi:10.1145/1514894.1514897.

[20] Reasoning and validation with spin, https://rdf4j.org/documentation/programming/spin/, 2021. Accessed: 2021-02-11.

[21] D. Calvanese, G. D. Giacomo, D. Lembo, M. Lenzerini, R. Rosati, Ontology-Based Data Access and Integration, Springer New York, New York, NY, 2018, pp. 2590–2596. URL: https://doi.org/10.1007/978-1-4614-8265-9_80667. doi:10.1007/978-1-4614-8265-9_80667.

[22] R2rml: Rdb to rdf mapping language, https://www.w3.org/TR/r2rml/, 2023. Accessed: 2023-02-19.

[23] Ontop, a virtual knowledge graph, https://ontop-vkg.org/guide://ontop-vkg.org/guide/l, 2023. Accessed: 2023-02-19.

[24] Rdfs semantics, https://www.w3.org/TR/rdf-mt/#RDFSRules, 2023. Accessed: 2023-02-19.

[25] J. Urbani, S. Kotoulas, E. Oren, F. van Harmelen, Scalable distributed reasoning using mapreduce, in: A. Bernstein, D. R. Karger, T. Heath, L. Feigenbaum, D. Maynard, E. Motta, K. Thirunarayan (Eds.), The Semantic Web - ISWC 2009, Springer Berlin Heidelberg, Berlin, Heidelberg, 2009, pp. 634–649.

[26] M. Buron, Efficient reasoning on large and heterogeneous graphs, Theses, École Polytechnique, 2020. URL: https://hal.inria.fr/tel-03107689.

[27] O. Palombi, M.-C. Rousset, F. Jouanot, Ontosides ontology, https://perscido.univ-grenoble-alpes.fr/datasets/DS388, 2023. Accessed: 2023-04-07.

[28] Top braid composer from topquadrant software, http://www.topquadrant.com/, 2019. Accessed: 2019-09-01.

[29] O. Palombi, F. Ulliana, V. Favier, J.-C. Leon, M.-C. Rousset, My corporis fabrica: an ontology-based tool for reasoning and querying on complex anatomical models, Journal of Biomedical Semantics (JOBS 2014) 5 (2014).

[30] P.-Y. Rabattu, B. Masse, F. Ulliana, M.-C. Rousset, D. Rohmer, J.-C. Leon, O. Palombi, My corporis fabrica embryo: An ontology-based 3d spatio-temporal modeling of human embryo development, Journal of Biomedical Semantics (JOBS 2015) 6 (2015).

[31] French Ministry for Higher Education and Research, Etudes Médicales, http://www.enseignementsup-recherche.gouv.fr/pid20536/bulletin-officiel.html?cid_bo=71544&cbo=1, 2013. Bulletin Officiel n^o 20, May 16th.

[32] S. Lohmann, S. Negru, F. Haag, T. Ertl, Visualizing ontologies with VOWL, Semantic Web 7 (2016) 399–419. URL: http://dx.doi.org/10.3233/SW-150200. doi:10.3233/SW-150200.

[33] B. Grau, I. Horrocks, Y. Kazakov, U. Sattler, Just the right amount: Extracting modules from ontologies, in: 16th International World Wide Web Conference, WWW2007|Int. World Wide Web Conf., 2007, pp. 717–726. URL: http://dblp.uni-trier.de/rec/bibtex/conf/www/KolovskiHP07. doi:10.1145/1242572.1242669, 16th International World Wide Web Conference, WWW2007 ; Conference date: 01-07-2007.

[34] P. Doran, I. Palmisano, V. A. M. Tamma, SOMET: algorithm and tool for SPARQL based ontology module extraction, in: U. Sattler, A. Tamilin (Eds.), Proceedings of the Workshop on Ontologies: Reasoning and Modularity, WoMO 2008, Tenerife, Spain, June 2, 2008, volume 348 of *CEUR Workshop Proceedings*, CEUR-WS.org, 2008. URL: http://ceur-ws.org/Vol-348/worm08_contribution_8.pdf.

[35] P. Doran, V. Tamma, L. Iannone, Ontology module extraction for ontology reuse: An ontology engineering perspective, in: Proceedings of the Sixteenth ACM Conference on Conference on Information and Knowledge Management, CIKM '07, Association for Computing Machinery, New York, NY, USA, 2007, p. 61–70. URL: https://doi.org/10.1145/1321440.1321451. doi:10.1145/1321440.1321451.

[36] M. d'Aquin, M. Sabou, E. Motta, Modularization: a key for the dynamic selection of relevant knowledge components, in: International Workshop on Modular Ontologies, 2006.

[37] J. Seidenberg, A. Rector, Web ontology segmentation: Analysis, classification and use, in: Proceedings of the 15th International Conference on World Wide Web, WWW '06, Association for Computing Machinery, New York, NY, USA, 2006, p. 13–22. URL: https://doi.org/10.1145/1135777.1135785. doi:10.1145/1135777.1135785.

[38] N. Noy, M. A. Musen, Prompt: Algorithm and tool for automated ontology merging and alignment, in: AAAI/IAAI, 2000.

[39] S. Ben Abbès, A. Scheuermann, T. Meilender, M. d'Aquin, Characterizing Modular Ontologies, in: 7th International Conference on Formal Ontologies in Information Systems - FOIS 2012, Graz, Austria, 2012, pp. 13–25. URL: https://hal.science/hal-00710035.

[40] M.-C. Rousset, F. Ulliana, Extracting bounded-level modules from deductive rdf triplestores, in: AAAI Conference on Artificial Intelligence, 2015.

[41] What does "big data" mean and who will win, https://youtu.be/KRcecxdGxvQ?t=228, 2022. Accessed: 2022-12-05.
[42] The Apache Software Foundation, Apache spark, https://spark.apache.org/, 2021. Accessed: 2021-12-05.
[43] Sansa Stack, Apache spark, https://sansa-stack.net/, 2021. Accessed: 2022-11-30.
[44] G. GSK, Bellman, https://github.com/asanchez75/bellman, 2022. Accessed: 2022-11-30.
[45] O. Curé, G. Blin, RDF Database Systems: Triples Storage and SPARQL Query Processing, 1st ed., Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2014.
[46] G. Agathangelos, G. Troullinou, H. Kondylakis, K. Stefanidis, D. Plexousakis, Rdf query answering using apache spark: Review and assessment, in: 2018 IEEE 34th International Conference on Data Engineering Workshops (ICDEW), 2018, pp. 54–59. doi:10.1109/ICDEW.2018.00016.
[47] C. Stadler, G. Sejdiu, D. Graux, J. L. 0001, Querying large-scale rdf datasets using the sansa framework, in: M. C. Suárez-Figueroa, G. Cheng, A. L. Gentile, C. Guéret, C. M. Keet, A. Bernstein (Eds.), Proceedings of the ISWC 2019 Satellite Tracks (Posters Demonstrations, Industry, and Outrageous Ideas) co-located with 18th International Semantic Web Conference (ISWC 2019), Auckland, New Zealand, October 26-30, 2019, volume 2456 of *CEUR Workshop Proceedings*, CEUR-WS.org, 2019, pp. 285–288. URL: http://ceur-ws.org/Vol-2456/paper74.pdf.
[48] A. Aranda-Andújar, F. Bugiotti, J. Camacho-Rodríguez, D. Colazzo, F. Goasdoué, Z. Kaoudi, I. Manolescu, AMADA: Web Data Repositories in the Amazon Cloud, in: ACM CIKM - International Conference on Information and Knowledge Management, Maui, United States, 2012. URL: https://hal.inria.fr/hal-00730687.
[49] J. Dean, S. Ghemawat, Mapreduce: Simplified data processing on large clusters, Commun. ACM 51 (2008) 107–113. URL: https://doi.org/10.1145/1327452.1327492. doi:10.1145/1327452.1327492.
[50] M. Zaharia, An Architecture for Fast and General Data Processing on Large Clusters, volume 11, Association for Computing Machinery and Morgan amp; Claypool, 2016.
[51] J. Paul, S. Lu, B. He, Database Systems on GPUs, Now Publishers, 2021.
[52] Cedar, activity report 2021, https://raweb.inria.fr/rapportsactivite/RA2021/cedar/CEDAR-RA-2021.pdf, 2022. Accessed: 2022-11-12.
[53] The Linux Foundation, Delta lake documentation, https://delta.io/, 2021. Accessed: 2021-12-05.
[54] J. Laskowski, The internals of delta lake, https://books.japila.pl/delta-lake-internals/, 2021. Accessed: 2021-12-05.
[55] M. Ragab, S. Sakr, R. Tommasini, Benchmarking spark-sql under alliterative rdf relational storage backends., 2019.
[56] R. Gu, S. Wang, F. Wang, C. Yuan, Y. Huang, Cichlid: Efficient large scale rdfs/owl reasoning with spark, in: 2015 IEEE International Parallel and Distributed Processing Symposium, 2015, pp. 700–709. doi:10.1109/IPDPS.2015.14.
[57] M. A. Farvardin, Scalable Saturation of Streaming RDF Triples, Theses, Université Paris sciences et lettres, 2021. URL: https://theses.hal.science/tel-03580501.

[58] Hdfs erasure coding, https://hadoop.apache.org/docs/stable/hadoop-project-dist/hadoop-hdfs/HDFSErasureCoding.html, 2022. Accessed: 2022-12-05.

[59] Saving capacity with hdfs raid, https://engineering.fb.com/2014/06/05/core-data/saving-capacity-with-hdfs-raid/, 2022. Accessed: 2022-12-07.

[60] Parse one quad, https://lists.apache.org/thread/l9otgzvb5szhvjy6y0wk5l4xlzg3bbm8, 2022. Accessed: 2022-12-07.

[61] xxhash - extremely fast hash algorithm, https://github.com/Cyan4973/xxHas://github.com/Cyan4973/xxHash, 2022. Accessed: 2022-12-07.

[62] OpenLink Software, Virtuoso universal server, https://virtuoso.openlinksw.com/, 2021. Accessed: 2021-12-05.

[63] M. Jovanovik, M. Spasić, Benchmarking virtuoso 8 at the mighty storage challenge 2018: Challenge results, in: D. Buscaldi, A. Gangemi, D. Reforgiato Recupero (Eds.), Semantic Web Challenges, Springer International Publishing, Cham, 2018, pp. 24–35.