Scalable Saturation of Streaming RDF Triples

Mohammad Amin Farvardin

▶ **To cite this version:**

Mohammad Amin Farvardin. Scalable Saturation of Streaming RDF Triples. Document and Text Processing. Université Paris sciences et lettres, 2021. English. NNT : 2021UPSLD019 . tel-03580501

HAL Id: tel-03580501
https://theses.hal.science/tel-03580501

Submitted on 18 Feb 2022

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

PSL
UNIVERSITÉ PARIS

THÈSE DE DOCTORAT
DE L'UNIVERSITÉ PSL

Préparée à Université Paris-Dauphine

Scalable Saturation of Streaming RDF Triples

Soutenue par

Mohammad Amin FARVARDIN

Le **19 Janvier 2021**

École doctorale n° ED 543

Ecole doctorale SDOSE

Spécialité

Informatique

Composition du jury :

Mme. Mirian HALFELD FERRARI ALVES Full Professeur, Université d'Orléans	*Président*
M. Bernd AMANN Full Professeur, Université Sorbonne - LIP6	*Rapporteur*
Mme. Fatiha SAÏS Professeur, Université Paris Saclay	*Rapporteur*
Mme. Mirian HALFELD FERRARI ALVES Full Professeur, Université d'Orléans	*Examinateur*
M. Denis CAROMEL Full Professeur, Université Nice-Sophia-Antipolis	*Examinateur*
M. Khalid BELHAJJAME Maître de Conférences, Université Paris Dauphine - PSL	*Co-Directeur de thèse*
M. Dario COLAZZO Full Professeur, Université Paris Dauphine - PSL	*Directeur de thèse*

Dauphine | PSL

Acknowledgment

First of all, my gratitude goes to my director of the thesis, Prof. Dario COLAZZO, who patiently guided and actively helped me in my research. Thank you for everything you have contributed, scientifically throughout this work. For keeping the doors wide open to answer my questions and help me overcome whatever hardships came to my way. For your friendship and your undeniable support. For all the things that you have taught me during these years. More, in particular, I really admired Dario for his kindness and patience. I will always be grateful to him for being the one who really believed in me and supported me from the first moment. I could not have had a better supervisor.

Likewise, I would also like to thank my co-director of the thesis, Dr. Khalid BELHAJJAME, for all of his contributions, supports, patience, and delighting in the road of the research throughout this thesis. For setting me on the right path when I lost sight of what laid ahead, for his friendship, again patience, motivation, and guidance both on a professional and personal level. For all of our joint meetings and various discussions. I also admire several qualities of my co-supervisor Khalid, but there is one that stands above all, and it is his passion for science. With his enthusiasm and his hard-working attitude, he managed to transmit to me one of the most important motivations that drive the work in academia: believing in what you are doing. Thanks very much for it.

Thanks, both of you, for transforming the thesis duration into enjoyable and unforgettable moments and engraved them in my memory.

Besides, I would like to thank Prof. Bernd AMANN from the Université Sorbonne (LIP6), for accepting to review my thesis and act as a reporter. It is my honor to have you as part of the scientific committee members. Thank you for attending my pre-defense session and proposing guidance lines to improve my work.

It is also my pleasure to have Prof. Fatiha SAÏS from Université Paris Sud, Prof. Miriab Halfelf Ferrari ALVES from Université d'Orléans, and Prof. Denis Caromel from Université Nice Sophia Antipolis (UNSA) in the scientific committee members of my thesis. My sincere thanks go to you for accepting to thoroughly review my thesis, for your insightful comments and suggestions.

I would also like to thank all my colleagues, friends, and in general the LAMSADE family whom I have shared unforgettable moments in recent years. Thank you for the lively and heated discussions, for your friendship, for all the work carried out together, and all the time we spent in laughter and joy.

Finally, my acknowledgment would not be complete if I do not thank my family, who have fundamentally supported me in this way. I would like to, especially thank my dear wife, DR. Fatemeh RAJABI. Dear without your support, this work would not have seen the light and would not have been possible to accomplish. Thanks for your endless support, a constant presence, for believing in me, and for the courage that you always gifted to me. Finally, I want to tell my son, Aiden, the last 6 months of the thesis were combined with your indescribable hope, cheerfulness, and sweetness.

Résumé

À l'ère des Big Data, les données RDF sont produites en grand nombre. Bien qu'il existe des propositions de raisonnement sur de grands graphiques RDF utilisant de grandes plateformes de données, il y a un manque de solutions qui le font dans des environnements où les données RDF sont dynamiques, et où de nouvelles instances et de nouveaux triplets de schéma peuvent arriver à tout moment.

Dans cette thèse, nous présentons la première solution pour raisonner sur de grands flux de données RDF en utilisant de grandes plateformes de données. Ce faisant, nous nous concentrons sur l'opération de saturation, qui cherche à déduire des triples RDF implicites étant donné les contraintes du schéma RDF ou OWL-Horst. En effet, contrairement aux solutions existantes qui saturent les données RDF en masse, notre solution identifie soigneusement le fragment de l'ensemble de données RDF existant (et déjà saturé) qui doit être pris en compte étant donné les nouvelles déclarations RDF délivrées par le flux. Ainsi, elle effectue la saturation de manière incrémentielle. L'analyse expérimentale montre que notre solution est plus performante que les solutions de saturation en masse existantes.

Abstract

In the Big Data era, RDF data are produced in high volumes. While there exist proposals for reasoning over large RDF graphs using big data platforms, there is a dearth of solutions that do so in environments where RDF data are dynamic, and where new instance and schema triples can arrive at any time.

In this thesis, we present the first solution for reasoning over large streams of RDF data using big data platforms. In doing so, we focus on the saturation operation, which seek to infer implicit RDF triples given RDF schema or OWL-Horst constraints. Indeed, unlike existing solutions which saturate RDF data in bulk, our solution carefully identifies the fragment of the existing (and already saturated) RDF dataset that needs to be considered given the fresh RDF statements delivered by the stream. Thereby, it performs the saturation in an incremental manner. An experimental analysis shows that our solution outperforms existing bulk-based saturation solutions.

Contents

List of Figures

List of Tables

1 Introduction

Contents

1.1 Context and Motivation

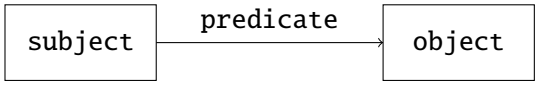
The semantic web [22] is a branch of the present World Wide Web, where computers are able to interpret and explain the semantics of information. The semantic web creates a universal medium for the exchange of data. Simplicity of machine-understandable data on the web are rapidly becoming a high priority for many corporations, individuals, and societies [10].

The semantic web data model relies on Resource Description Framework (RDF) [12], where each RDF statement is made of three different terms: a `subject`, a `predicate` and an `object`. The following example states that Marie Curie was awarded a Nobel prize in physics. In this statement, *Marie_Curie* is the `subject`, *award* is the `predicate` and *Nobel_Prize_in_Physics* is the `object` of a RDF statement.

`<dbpedia:Marie_Curie>` `<dbo:award>` `<dbpedia:Nobel_Prize_in_Physics>` .[1]

[1] In this thesis, we often shorten IRIs (*Internationalized Resource Identifier*) using prefixes for the sake of horizontal space limitation and to enhance expressiveness.

Table 1.1: Representations of an RDF Triple

RDF Type	**An Example**
Graphical	
Triple	`subject predicate object`
Relational	`predicate (subject, object)`

RDF/XML	`<rdf:Description rdf:about="subject">`
	`  <ex:predicate>`
	`  <rdf:Description rdf:about="object" />`
	`  <ex:predicate>`
	`<rdf:description>`

Turtle	`subject ex:predicate object .`
N-Triple	<http://example.org/#subject> <http://example.org/ontology/predicate> <http://example.org/#object> .

The Semantic web contains a massive amount of such statements, which describe information on extremely broad range of domains, from Foods and Agriculture Organization (FOA) [1] and environment [5] to chemical-gene/protein intersections [4] and medical subjects headings (MESH) [11] to government information [6] and social media [7].

The `subject` of a statement is an IRI[2] or a blank node. IRI stands for Internationalized Resource Identifier, and is used in the semantic web to identify resources. The `predicate` must be an IRI, and `object` is an IRI, a blank node or a literal. Blank nodes, denoted as _:b_i, are used to represent unknown resources (i.e., IRIs or literals). It is asserting that the relationship denoted by the predicate *p* holds between the subject *s* and object *o* of the triple. Thus, we assume:

- *U* is the set of IRIs denoting both resources and relationships,
- *L* is the set of literals denoting constants represented by means of strings,
- *B* is the set of blank nodes (corresponding to resources for which neither a

[2]An IRI is just a URI exploiting Unicode in place of US Ascii (American Standard Code for Information Interchange) as the character set.

IRI nor a literal is given/known).

We also assume that these sets are infinite and countable. Blank nodes, here denoted as _:b_i, are essential in RDF to encode incomplete knowledge. They are used to representing unknown resources (IRIs or literals). An RDF triple $s\ p\ o$ is *well-formed* if the subject s belongs to $U \cup B$, the predicate p (also called property) belongs to U and its object o belongs to $U \cup L \cup B$. In this work, we assume triples to be well-formed. An RDF dataset can be viewed as a directed graph, hence the term RDF graph. Each triple $s\ p\ o$ gives rise to an edge labeled by the property p, which connects a node representing the subject s to a node representing the object o.

Table 1.1 illustrates an RDF triple in different ways. They consist of: a Triple, an RDF/XML, Relational form, Turtle form, and N-Triple. For example, the following triple –shows in form of N-Triple– states that Neil Armstrong was on the *Apollo 11* mission:

```
<http://dbpedia.org/resource/Neil_Armstrong> <http://dbpedia.org/ontology/mission> <http://dbpedia.org/resource/Apollo_11> .
```

We consider the N-Triple serialization for the rest of this thesis (Table 1.1 last row). The N-Triple statement is a sequence of a subject, a predicate, and an object terms while separated by whitespace (spaces U+0020 or tabs U+0009) and terminated by a full stop '.' after each triple [13], and every line contains only one N-Triple. N-Triple may contain a 4^{th} term. This term introduces the dataset that the triple belongs to.

1.2 Problem

In the Big Data era, RDF data, just like many other kinds of data, are produced in high volumes. This is partly due to sensor data produced in the context of health monitoring and financial market applications, feeds of user-content provided by social network platforms, as well as long-running scientific experiments that adopt a stream-flow programming model [49]. To take full advantage of semantic data and turn them into actionable knowledge, the semantic web community has devised techniques for processing and reasoning over RDF data (e.g.[23, 70, 80]). This trend generated the need for new solutions for processing and reasoning over RDF datasets since the existing state-of-the-art techniques cannot cope with large volumes of RDF data. Nowadays, aside from the volume of data that, so far produced, processing the frequent generation of massive data in a reasonable time

carries a new challenge. For example, in the context of LOD[3], data are produced continuously, and, since the RDF embraces OWA (Open-World Assumption), therefore there is no restriction to having new schemas with newly data.

A typical and fundamental operation for reasoning about RDF data is *data saturation*. This operation involves a set D of RDF data triples and a set S of semantics properties, expressed in terms of either RDF Schema [24] and/or OWL [54], and aims at inferring the implicit triples that can be derived from D by using properties in S. For example, given the triples below as D and S:

D: <dbpedia:Neil_Armstrong> <dbo:mission> <dbpedia:Apollo_11> .
S: <dbo:mission> <rdfs:range> <dbo:SpaceMission> .

we could infer the following statement as a new knowldge that was not stored in the given dataset.

<dbpedia:Apollo_11> <rdf:type> <dbo:SpaceMission> .

Data saturation is crucial in order to ensure that RDF processing and querying actually work on the *complete* informative content of an RDF database, without ignoring implicit information. To deal with the problem of saturating massive RDF datasets, a few approaches exploiting big data paradigms (namely Map-Reduce [48]) and platforms, notably Hadoop and Spark (see e.g., [75, 35]), have already been proposed. In [75] Urbani *et al.* described the WebPIE system and showed how massive RDF data can be saturated by leveraging on the Map-Reduce paradigm over Hadoop. In [35] Gu *et al.* presented the Cichlid system and showed how to speed up saturation by using Spark and its underlying Resilient Distributed Datasets (RDDs) abstraction. In [57, 59] authors proposed a parallel reasoning method based on P2P self-organizing networks, while in [82] authors propose a parallel approach for RDF reasoning based on MPI. These approaches, however, assume that RDF datasets are fully available before the saturation, and as such, are not instrumented to saturate RDF data produced continuously in streams.

Indeed, when RDF data are produced in streams, such systems must re-process the whole data collection in order to obtain triples entailed by the newly received ones. This is due to the fact that both initial and already obtained triples (by means of past saturation) can entail new triples under the presence of newly received instance/schema triples. The authors of [62] proposed a continuous ASP-based (Answering Set Programming) reasoning over RDF streams on RDF data. In

[3]Open Linked Data https://lod-cloud.net/

the proposed method, the reasoning happens on the windowing-level while not considering static semantics neither massive data. Also, a number of works have addressed the problem of incremental saturation [18, 79, 56, 83], but these approaches, being mostly centralised, do not ensure scalable, distributed, and robust RDF streaming saturation.

To guarantee the fact that the complete knowledge is updated by given new data, the received data has to join with data that already met in the past. Consider that the data volume soon becomes notably high in the presence of massive streams. Accordingly, the main challenges as a big picture of this thesis are:

- *How* `quickly` *and* `when` *received data can be processed?*
- *How* `durable` *will be the proposed method? (if any)*

The objective of this thesis is RDF data saturation under the RDF Schema (RDFS) and OWL-Horst semantics, while data and schema are given partially in different intervals. In this work we are interested in both execution time and memory usage optimization time.

1.3 Solution (Contributions)

Based on the above description of the research context - motivations and problem - in this work we present a distributed technique for saturating *streams* of large RDF data, by relying on `Spark Streaming API` that is an extension of the core Spark API (i.e., as a well-known cluster computing platform for processing massive data), hence ensuring scalability and robustness.

We present our approach in two steps. In the first one, we deal with streaming RDF schema saturation in the presence of RDF Schema statements. Focusing first on RDF Schema is motivated by the fact that, despite its simplicity, RDF Schema is rich enough to make the efficient saturation of streaming large RDF data far from being trivial. As we already said, the main challenge is to quickly process fresh data, that must be joined with past met data, whose volume can soon become particularly high in the presence of massive streams. To this end, unlike existing state-of-the-art solutions [75, 35] for large-scale RDF saturation, upon the arrival of new RDF statements (both schema and instance triples) our solution *finely* identifies the subset of the existing (and already saturated) RDF dataset that needs to be considered. This is obtained by relying on an indexing technique we

devise for our approach. Our indexing algorithm partitions triples into property and object triples, and creates distinct subindexes for each micro-batch; hash maps allow the system to quickly retrieve all triples having a given property or a given object.

In the second part of the thesis, we deal with OWL-Horst rules. In this case we show how our saturation technique, initially developed for RDFS only, can be easily adapted to OWL-Horst: indeed, here we have to deal with weaker constraints on rule application order as well as with the need of computing a fix point.

Finally, we validate our claims of efficiency and scalability through an extensive experimental evaluation, where we analyze the behavior of our algorithm on RDFS-based datasets as well as on OWL-based datasets.

1.4 Thesis Structure

Beside this introductory chapter, this thesis covers five more chapters.

Chapter 2 presents preliminaries information about RDF and RDF reasoning, big data platforms, Apache Spark and Spark Streaming API, and both an RDF reasoning in bulk.

Chapter 3 is dedicated to the related works, where we present several methods, approaches, and techniques falling into the scope of our proposed approach. We present our contribution in chapters 4, 5, and 6.

By considering that data received in a stream fashion, in chapter 4, we first are described the state of RDF triples with/without RDF schema triples that convey for reasoning. Then, to improve saturation performance, we propose a rule pruning. In the second stage, we introduce the core of our contribution as a novel indexing approach. In the following, we design an empirical evaluation to compare our method with the state-of-the-art. The chapter is finished by summary of the chapter.

In chapter 5, we describe our extension of our indexing technique for the OWL-Horst rule set. OWL-Host rules have 23 rules and most of them are much more complicated than RDFS rules [24]. In this chapter, we have considered all the OWL-Horst rules [72]. In the end, by designing an empirical evaluation we compare our improved method for OWL-Horst reasoning with the state-of-the-art. The conclusion of the chapter is take place as the last part of it.

2 Preliminaries

Contents

In this chapter, we will describe the preliminary information and notations of RDF and semantic data reasoning. In the following, we will present the RDF schema rules and explain how the reasoning process infers implicit knowledge from the explicit triples. In the next section, we, briefly, survey on the well-known big data processing platforms, i.e., Apache Hadoop, Apache Spark, and Apache Flink. We, then, explain the reason why we choose the Spark Streaming API as a cluster processing platform in this thesis. The last section of this chapter dedicated to the saturation of large RDF graph in bulk.

2.1 RDF and Semantic Data Reasoning

An RDF dataset is a set of triples of the form $s\ p\ o$. `s` is an IRI[1] or a blank node that represents the subject. IRI stands for Internationalized Resource Identifier, and is used in the semantic web to identify resources. `p` is an IRI that represents

[1] An IRI (*Internationalized Resource Identifier*) is just a URI exploiting Unicode in place of US Ascii as the character set.

the predicate, and o is an IRI, blank node or a literal, and it stands for the object. Blank nodes, denoted as _:b_i, are used to represent unknown resources (IRIs or literals). RDF Schema (or RDFS for short) provides the vocabulary for specifying the following relationships between classes and properties, relying on a simplified notation borrowed from [32]:

- *subClassOf relationship* $\prec_{sc}$: the triple $c_1 \prec_{sc} c_2$ specifies that c_1 is a subclass of c_2;
- *subPropertyOf relationship* $\prec_{sp}$: the triple $p_1 \prec_{sp} p_2$ specifies that p_1 is a sub-property of p_2;
- *property domain* $\hookleftarrow_d$: the triple $p \hookleftarrow_d x$ specifies that the property p has as a domain x; and
- *property range* $\hookrightarrow_r$: the triple $p \hookrightarrow_r z$ specifies that the property p has as a range z.

For the sake of readability, in what follows we use simple strings instead of IRIs to denote predicates, subjects and objects in triples. Also, we abbreviate the `rdf:type` predicate with the τ symbol.

Example 2.1. Figure 2.2 illustrates a set of RDF instance triples that we use as a running example, together with the equivalent graph representation. The graph describes the resource doi_1 that belongs to an unknown class, whose title is "Complexity of Answering Queries Using Materialized Views", whose author is "Serge Abiteboul" and having an unknown contact author. This paper is in the proceedings of an unknown resource whose name is "PODS'98". Lastly, the IRI edbt2013 is a conference and hasName, the property associating names to resources, is created by "John Doe".

Table 2.1 shows how to use triples to describe resources, that is, to represent class (unary relation) and property (binary relation) assertions. The RDF standard in W3C [8] has a set of built-in classes and properties.

Table 2.1: RDF Statements [25]

Assertion	**Triple**	**Relational Notation**
Class	(s, *rdf:type*, o)	o(s)
Property	(s, p, o)	p(s, o)

RDF Schema (or RDFS in short) provides a means to define classes that can be used to type the resources of an RDF graph (Table 2.2)

Table 2.2: RDFS Statements

Constructor	**Triple**	**Notation**
Subclass constraint	$s \prec_{sc} o$	$s \subseteq o$
Subproperty constraint	$s \prec_{sp} o$	$s \subseteq o$
Domain typing constraint	$s \hookleftarrow_{d} o$	Π domain(s) $\subseteq o$
Range typing contraint	$s \hookrightarrow_{r} o$	Π range(s) $\subseteq o$

Figure 2.1 lists schema triples. For example, it specifies that the class *posterCP* is a subclass of *ConfP*, that the property *hasContactA* is a sub-property of *hasAuthor*. It also specifies that the property *hasAuthor* has as domain *paper* and as range a literal.

S = { posterCP $\prec_{sc}$ confP, _:b_0 $\prec_{sc}$ confP,
confP $\prec_{sc}$ paper, hasTitle $\hookleftarrow_{d}$ confP,
hasTitle $\hookrightarrow_{r}$ rdfs:Literal, hasAuthor $\hookleftarrow_{d}$ paper,
hasAuthor $\hookrightarrow_{r}$ rdfs:Literal, hasContractA $\prec_{sp}$ hasAuthor,
inProceesingOf $\hookleftarrow_{d}$ confP, inProceesingOf $\hookrightarrow_{r}$ conference,
hasName $\hookleftarrow_{d}$ conference, hasName $\hookrightarrow_{r}$ rdfs:Literal,
createdBy $\hookrightarrow_{r}$ rdfs:Literal }

Figure 2.1: Instance and schema RDF triples.

As in other works (e.g., [32, 35, 75]) we focus on the core rules of RDFS, the extension to other rules being trivial. In particular, we consider here Rules 2, 3, 5, 7, 9, and 11 among the 13 RDFS rules illustrated in Table 2.3. The main reason for ignoring rules 1,4,6,8,10,12, and 13 is that those rules, in general, do not affect the results of RDFS reasoning and their outcome cannot be used for later derivation. On the other hand, since, all of them have only one antecedent, then by a single pass over the data their outcome can be produced.

The realm of the semantic web embraces the Open World Assumption: facts (triples) that are not *explicitly* stated may hold given a set of RDFS triples expressing constraints. These are usually called *implicit* triples, and, in our work, we

G = { doi_1 τ _:b_0, doi_1 *hasTitle* "CAQU MV",
doi_1 *hasAuthor* "SA", doi_1 *hasContactA* _:b_1,
doi_1 *inProceedingsOf* _:b_2, _:b_2 *hasName* "PODS'98",
hasName createdBy "John Doe", "edbt2013" τ Conference }

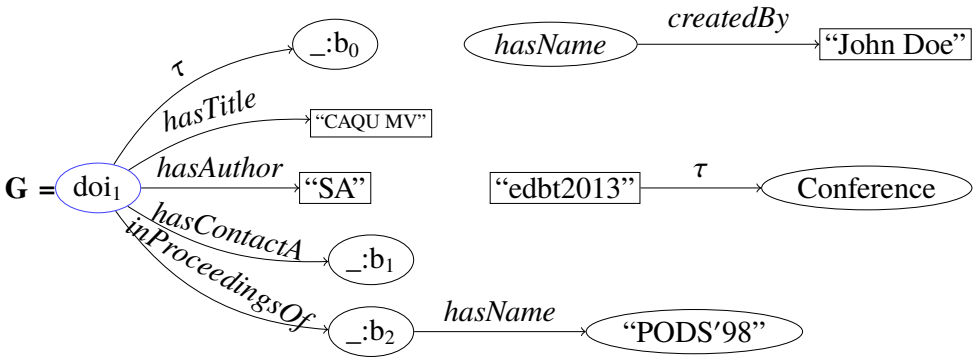


Figure 2.2: RDF graph representation of a conference paper.

Table 2.3: RDFS Ruleset

Rule	**Condition**	**Consequence**	**Level**
rdfs1	*s p o*	*_:b* τ rdfs:Literal	-
rdfs2	$p \hookleftarrow_d x$, *s p o*	*s* τ *x*	instance-level
rdfs3	$p \hookrightarrow_r x$, *s p o*	*o* τ *x*	instance-level
rdfs4	*s p o*	s/o τ rdfs:Resource	-
rdfs5	$p \prec_{sp} q$, $q \prec_{sp} r$	$p \prec_{sp} r$	schema-level
rdfs6	*p* τ *rdf:Property*	$p \prec_{sp} p$	-
rdfs7	s p o, $\text{p} \prec_{\text{sp}} \text{q}$	s q o	instance-level
rdfs8	s τ rdfs:Class	$\text{s} \prec_{\text{sc}}$ rdfs:Resource	-
rdfs9	s τ x, $\text{x} \prec_{\text{sc}} \text{y}$	s τ y	instance-level
rdfs10	s τ rdfs:Class	$\text{s} \prec_{\text{sc}} \text{s}$	-
rdfs11	$\text{x} \prec_{\text{sc}} \text{y}$, $\text{y} \prec_{\text{sc}} \text{z}$	$\text{x} \prec_{\text{sc}} \text{z}$	schema-level
rdfs12	p τ rdfs:ContainerMembershipProperty	$\text{p} \prec_{\text{sp}}$ rdfs:member	-
rdfs13	o τ rdfs:Datatype	$\text{o} \prec_{\text{sc}}$ rdfs:Literal	-

consider the problem of RDF saturation, i.e., given a set of RDFS rules, inferring all possible implicit triples by means of these rules applied on explicit triples, or, recursively, on implicit triples. For example, rule *rdfs2* in Table 2.3 states that, if a property p has a domain x, given a triple $s\ p\ o$, we can infer that s is of type x. Since *rdfs9* specifies that, if s is of type x and x is a subclass of y, then we can infer that s is of type y.

In the remaining part of the thesis, we will use the following notation to indicate derivations/inference of triples. A *derivation tree* is defined as follows.

$$T \ := \ t \ \mid \ \{T \mid T\} - rdfsX \rightarrow t$$

where the rule number X ranges over $\{2, 3, 5, 7, 9, 11\}$. A derivation tree can be empty, hence consisting of a given triple t, or can be of the form $\{T1 \mid T2\} - rdfsX \rightarrow t$, meaning that the tree derives t, by means of rule *rdfs*X whose premises are (matched to) the two triples given by T1 and T2, respectively. So, for instance we can have the following derivation tree T1 for the for G and S previously introduced:

$$\{hasTitle \hookleftarrow_{\mathrm{d}} confP \mid doi_1\ hasTitle\ \text{“CAQU MV”}\} - rdfs2 \rightarrow doi_1\ \tau\ confP$$

Moreover, we can have the following derivation T2 relying on T1:

$$\{T1 \mid confP \prec_{\mathrm{sc}} paper\} - rdfs9 \rightarrow doi_1\ \tau\ paper$$

In the following, given a set of instance RDF triples D and a set of schema triples S, we say that T is over D and S if the derivation tree uses triples in D and S as leaves. Moreover, we define the saturation of D over S as D extended with all the possible instance triples obtained by means of derivation (below, derivation trees are assumed to be over D and S):

$$D_S^* \ = D \cup \ \{t \mid \exists \{T1 \mid T2\} - rdfsX \rightarrow t\ with\ X \in \{2, 3, 7, 9\}\}$$

Notice above that, say, T2 can be a derivation tree totally over S, recursively applying rule 5 (or rule 11) thus deriving a triple in S^*, below defined.

$$S^* \ = S \ \cup \ \{t \mid \exists \{T1 \mid T2\} - rdfsX \rightarrow t\ with\ X \in \{5, 11\}\}$$

Above, in the S^* definition, please note that since $X \in \{5, 11\}$ the whole derivation tree consists of subsequent applications of rule rule 5 (or rule 11).

2.2 Big Data Platforms

In this section, we, briefly, survey the main-stream big data cluster computing platforms, i.e., Apache Hadoop, Spark and Flink.

Apache Hadoop In the last decade, extracting knowledge from the massive data widely was possible for both industrial and academic purpose with the advent of Apache Hadoop [3]. It is an open-source implementation based on MapReduce [30] model with a simple but powerful programming model, i.e., consists of Map and Reduce phases. The Map phase is responsible for distributing and partitioning data, and the Reduce phase utilizes as an execution stage on the partitioned data. The complexity of parallel jobs execution and fault-tolerance is hide from users. The Hadoop API simply forces applications to be implemented in terms of map and reduce functions. Despite the power of this model, it does not suite iterative algorithms that perform numerous cycles of computation on the same data, while some algorithms require iteration to finish their process, e.g., RDF saturation needs iteration until it reaches a fixpoint. To overtake these constraints, the second generation of cluster computing platforms, mainly Spark and Flink, were introduced to process massive data.

Apache Flink Apache Flink [2] is a framework and distributed processing engine for stateful computations over unbounded and bounded data streams. Flink has been designed to run in all common cluster environments, perform computations at in-memory speed and at any scale. It processes data using directed acyclic graphs (DAG) pattern in multi-step data pipelines. At high-level, the control flow of an application is managed by a driver program, which relies on two main parallel programming abstraction: 1) structures to describe the data and 2) parallel operations on these data. Apache Flink is built on top of DataSet, Job Graph and Parallelization Contracts (PACTs) [81]. DataSet is collections of elements of a specific type on operations with an implicit type parameter are defined. Job Graphs performs parallel data flow with arbitrary tasks. PACTs are second-order functions that define properties on the input/output data of their associated UDFs (User Defined Functions) as a first-order function [52]. Flink proposed native closed-loop iteration operation [31] that supports better streaming. Another Flink feature is an automatic cost-based optimizer, that automatically optimizes execution by avoiding costly operations, e.g. shuffle and sort, and cached intermediate data

[52]. The most common dataflow operations in Flink consist of, e.g., *map, reduce, distinct, collect, count, save, groupBy, etc.*

Apache Spark Apache Spark [86] is a widely used in-memory distributed cluster computing framework. It provides the means for specifying DAG-based data flows using operators like *map*, $filter$, *reduceByKey*, *join*, $filter$, etc. over data collections represented by means of Resilient Distributed Datasets (RDDs). In Spark, the data to be processed are mapped into RDDs, where an RDD is an immutable collection of objects (e.g, <key, value> pairs); RDDs are partitioned and distributed over the Spark cluster [52]. RDDs are fragmented as partitions and distributed over the nodes of the spark cluster. In other words, RDDs are a read-only resilient set of objects partitioned across multiple nodes; it effectively supports iterative algorithms as it follows an in-memory data structure with the ability to cache intermediate data across nodes of a cluster. Spark essentially works by applying operations to RDDs. These operations can be divided into *transformations* and *actions* (Table 2.4).

Transformations are lazy operations that return a new RDD, and are evaluated only when an action (e.g., `count`, `collect`, etc.) is invoked. Typical transformation is *map*(), that applies a given function to all the objects in a RDD, and $filter$(), which applies a predicate to the input data (e.g., `rdd.map(x => x + x)`, `rdd.filter(x => x != 3)`.

The authors of [52] found that there is not a single framework for all data types. The same survey shows that Spark is around 1.7x faster than Flink in large graph processing, while Flink up to 1.5x faster than Spark in batch and small graph workloads. For our purposes, we use the streaming capabilities of Spark whereby data comes into micro-batches that need to be processed within a time-interval (also referred to as a window). It is worth mentioning that Spark offers two high-level data collection `DataFrame` and `DataSet` both of which are high-level extensions of `RDD`. Also, Spark offers `Structured Streaming` and `Spark Streaming` engines to process data that convey via stream. In this thesis, we control every single triple from arrival time until writing time on the disk. Since `DataFrame` and `DataSet` as data collections and `Structured Streaming` as a processing engine, despite their powerful functionalities, do not allow us to control every single triple, therefore, we choose `RDD` as data collection and `Spark Streaming` as the processing engine to deal with streaming data.

Table 2.4: Transformations and actions available on RDDs in Spark. Seq[T] denotes a sequence of elements of type T [86]

Transformations	*map*(*f* : T ⇒ U)	:	RDD[T] ⇒ RDD[U]
	filter(*f* : T ⇒ Bool)	:	RDD[T] ⇒ RDD[T]
	flatMap(*f* : T ⇒ Seq[U])	:	RDD[T] ⇒ RDD[U]
	mapValues(*f* : V ⇒ W)	:	RDD[(K, V)] ⇒RDD[(K, W)] (preserve partitioning)
	union()	:	(RDD[T], RDD[T]) ⇒ RDD[T]
	join()	:	(RDD[(K, V), RDD(K, W)]) ⇒ RDD[(K, (V, W))]
	sortBy(*f* : T ⇒ K)	:	RDD[T] ⇒ RDD[T]
	reduceByKey(*f* : (V, V) ⇒ V)	:	RDD[(K, V)] ⇒ RDD[(K, V)]
	partitionBy(*p* : Partitioner[K])	:	RDD[(K, V)] ⇒ RDD[(K, V)]
	mapPartitions (*f* : Iterator[T] ⇒ Iterator[U])	:	RDD[T] ⇒ RDD[T]
	mapPartitionsWithIndex (*f* : (Int, Iterator[T]) ⇒ Iterator[U])	:	RDD[T] ⇒ RDD[T]
Actions	*count*()	:	RDD[T] ⇒ Long
	collect()	:	RDD[T] ⇒ Seq[T]
	reduce(*f* : (T, T) ⇒ T)	:	RDD[T] ⇒ T
	lookup(*k* : K)	:	RDD[[K, V]] ⇒ Seq[V] (On hash/range partitioned RDDs)
	save(*path* : String)	:	Outputs RDD to a storage system. e.g., HDFS

Spark Streaming Spark Streaming is an extension of Spark allowing to process massive data sets made available through data streams. The main idea behind Spark Streaming is to divide the stream in a sequence of subsequent splits called micro-batches each one represented by means of an RDD so that parallel Spark operations can be invoked over them. So a stream is represented as a stream of micro-batches, each one determined in terms of an interval time in which all the received data items take part in the same micro-batch, and transformations applied over micro-batches produce a new stream of micro-batches that can be in turn processed (Figure 2.3). The runtime support of Spark Streaming provides mechanisms for distributing stream processing over the cluster in a resilient fashion, in order to ensure both scalability and robustness.

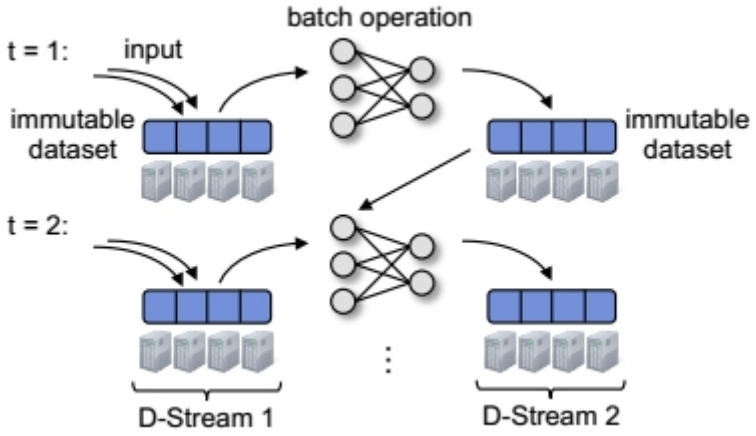


Figure 2.3: D-Stream processing model. In each time interval, the records that arrive are stored reliably across the cluster to form an immutable, partitioned dataset. This is then processed via deterministic parallel operations to compute other distributed datasets that represent program output or state to pass to the next interval. Each series of datasets forms one D-Stream [87]

2.3 Saturation of Large RDF Graphs

The saturation of large RDF graphs can be time and resource consuming, especially when dealing with large RDF graphs. In the state of the art, authors of WebPIE [75] proposed saturation RDF datasets with RDFS and OWL rules on the MapReduce technique on Apache Hadoop. Later, authors of Cichlid [35] represents saturation RDF datasets with RDFS and OWL rules by leveraging the WePIE work on Apache Spark with significant improvement on saturating process.

As in our case, these systems focus on rules 2, 3, 5, 7, 9, and 11, illustrated in Table 2.3. The main reason for ignoring rules 1, 4, 6, 8, 10, 12, and 13 is that those rules, in general, do not affect the inference of RDFS reasoning. Also, their outcome won't be useful for later derivation. On the other hand, all of the mentioned rules have only one antecedent, and by a single pass over the data, their outcome can be produced.

Heuristics and Optimization Techniques

To improve the performance and the scalability of RDF streaming saturation, our indexing scheme alone is not sufficient. Therefore, we also adopt the rule application strategy of Cichlid, and devised new optimization techniques. We will briefly recall the Cichlid strategy, and then focus on our novel techniques.

Rule application order While the outcome of the saturation operation is orthogonal to the order in which the rules are applied, the time and resources consumed by such an operation are not. Because of this, the authors of Cichlid (and WebPIE before them) identified a number of optimisations that influence the rule application order with the view to increasing the efficiency of the saturation. In what follows, we discuss the main ones.

1. *RDF Schema is to be saturated first*. The size of the RDF schema[2] in an RDF graph is usually small, even when saturated. It is usually orders of magnitudes smaller than the size of the remaining *instance* triples. This suggests that the schema of the RDF graph is to be saturated first. By saturating the schema of an RDF graph we mean applying rules that produce triple that describes the vocabulary used in an RDF graph. Furthermore, because the size of the schema is small, schema saturation can be done in centralized fashion. In this respect, the RDFS rules presented in Table 2.3 can be categorised into two disjoint categories: schema-level and instance-level RDFS rules. Schema-level RDFS rules (*rdfs5* and *rdfs11*) designate the rules that produce triples describing the vocabulary (classes, properties, and their relationships). Instance-level triples, on the other hand, specifies resource instances of the classes in the RDF vocabularies and their relationships. Each rule is made up of two premises and one conclusion, each of which is an RDF triple. While premises of schema-level rules are schema triples, premises of instance-level rules are a schema triple and an instance triple. Also, instance-level rules entail an RDF instance triple, while schema-level rules entail an RDF schema triple.

2. *Dependencies between rules*. When determining the rule execution order, the dependencies among rules must be taken into account too. In particular, a rule R_i precedes a rule R_j if the conclusion of R_i is used as a premise for

[2]By Schema, we mean the RDF triples that describe the vocabulary of an RDF graph, i.e., classes, properties and their constraints.

rule R_j. For example *rdfs7* has a conclusion that is used as a premise for rules *rdfs2* and *rdfs3*. Therefore, *rdfs7* should be applied before *rdfs2* and *rdfs3*.

By taking (1) and (2) into consideration, the authors of Cichlid established the orders of applications of rules illustrated in Figure 2.4. To illustrate how rules are implemented in Spark, we will use a concrete example considering *rdfs9*, which can be expressed as follows. If a resource s is of type x, i.e. $s\ \tau\ x$, and x is a sub-class of y, i.e. $x \prec_{sc} y$, then s is also an instance of y, i.e. $s\ \tau\ y$. Note that, as the output of *rdfs2* and *rdfs3* are instance triples with predicate τ, these rules are executed in Cichlid before executing *rdfs9* (see [35] for more details). In our approach we will rely on the same ordering for streaming saturation.

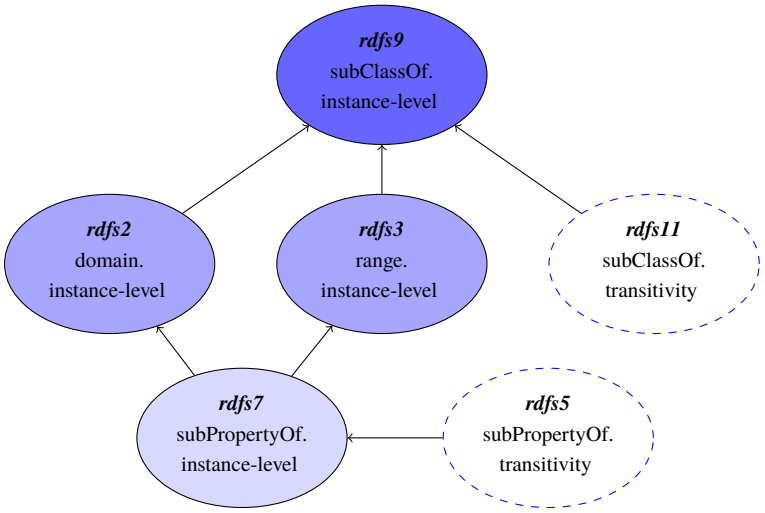


Figure 2.4: The optimized execution RDFS rules order. Dashed ellipses show the schema-level entailment, while the solid ellipses are for the Instance-level entailment. The white colour ellipse has no dependency prerequisite with other RDFS. The lighter ellipses come first.

To implement *rdfs9* in Spark, Cichlid uses the *filter*, *map*, *broadcast*, and *collect* operators in Algorithm 1. The algorithm first retrieves over all the partitions the RDFS schema, the classes and their corresponding sub-classes in the schema, by means of the filter transformation and the collect action (this last one is needed in order to collect on the master/driver machine the total filtered information). This information is then broad-casted[3] (i.e., locally cached in each machine in

[3]Broadcast operation can be used in Spark to cache a copy of data on every node of a cluster.

Algorithm 1: Optimized Parallel reasoning of RDFS *rdfs9*

```
1:  Input: input triple set named triples
2:  Output: reasoning results named results
3:  Begin
4:    schema ← sc.textFile("hdfs://schema-path")
5:    subClassOf ← schema
6:      .filter (t ⇒ t._2.contains("rdfs:subClassOf"))
7:      .map(t ⇒ (t._1, t._3)).collect
8:    bc ← sc.broadcast(subClassOf.toMap)
9:    triples ← sc.textFile("hdfs://instance-path")
10:   results ← triples.filter(t ⇒ bc.value.contains(t._3))
11:     .map(t ⇒ (t._1, τ, bc.value(t._3))
12:   return results
13: End
```

the cluster) as pairs (e.g., $x \rightarrow y$), thereby avoiding the cost of shipping this information every time it is needed. It first retrieves the RDFS schema *(line 4)*, the classes and their corresponding sub-classes *(lines 5-7)*, and the obtained information is then broad-casted *(line 8)*. Therefore, for each broad-casted pair of subclass and superclass, the instances of the subclass are retrieved *(line 9)*, and new triples are derived stating that such instances are also instances of the broad-casted super-class, by means of the map transform *(lines 10-11)*. Spark provides other operators, which are used for implementing other rules, such as *distinct*, *persist*, *union*, *partitionBy*, *reduceByKey*, *mapPartitions*, *mapPartitionsWithIndex*, *etc*. In the following, we breifly describe the most used Spark operations in this thesis:

- *mapPartitionsWithIndex:* In Spark, RDDs are parallelized via partitioning across the nodes, where every partition has a unique *Index*. At the writing time, partitions write into separate files on the disk when each file contains the same file number as the partition index. Therefore, the `mapPartitionsWithIndex` operation lets access to the data (in our case triples) partition-wise when each partition has an index number. In our algorithm, we used this functionality to know each triple is located in memory.

- *partitionBy:* It can be applied on a pair dataset (<`key, value`>) to put all

This helps in avoiding the cost of shipping this information every-time it is used by the nodes.

triples with a same key in one partition. Sometimes more than one type of triples (based on their keys) locating in a partition.

- *reduceByKey:* This operation aggrigates pairs (<`key, value`>) based on their *key*. For instance, by having a sequence of pairs, e.g. Seq((a,1) , (a,3), (b,5)) as an *rdd*, the following operation *rdd*`.reduceByKey(_+_)` keeps one copy of *key* and applies the sum operation on the *values*. In this example, the generated RDD contains the following pairs of data: ((a,4), (b,5)).

- *distinct:* It used to remove duplicates from an RDD. For instance, `rdd.distinct()` removes duplicates from the `rdd`.

- *union:* It concatinates two RDDs, and generates a new RDD containing data from the first and second rdd. The number of partitions of the new RDD is equal to total number of both rdds. `val rdd3 = rdd1.union(rdd2)`.

Notice that as the saturation process may derive triples that are already asserted or have been derived in previous steps of the saturation operation, Cichlid [35] eliminates the duplicated triples from the derived ones, in order to improve efficiency.

2.4 Conclusion

I this chapter, we have explained preliminaries about RDF and semantic data reasoning by given some examples and introduce related notations for that purpose. In the following, we explain the well-known big data platforms and describe their functionalities. We also detailed the saturation of large RDF data from state of the art. In the next chapter we discuss the related work of the thesis domain.

3 Related Works

Contents

The domain of RDF exists for more than 20 years, and a lot of research has been done around it. One of the prominent investigated research problems concern *RDF Reasoning*. It commits to infer implicit triples from the explicit ones. Over the years, with the growing of the volume of generated RDF data, RDF Reasoning became a *Big Data* problem which requires cluster processing to derive the results in an acceptable, efficient time.

RDF is partly generated due to sensor data produced in the context of health monitoring and financial market applications, given user-content provided by social network platforms, long-running scientific experiments that adopt a stream-flow programming model [49]. In some cases, stream data even can be used to improve the streaming experience with experimentation and data science[1] [33]. To prepare and provide a full set of semantic data, the semantic web community has devised techniques for processing and reasoning over RDF data (e.g.[23, 70, 80]). This

[1]https://netflixtechblog.com/a-b-testing-and-beyond-improving-the-netflix-streaming-experience-with-experimentation-and-data-5b0ae9295bdf

trend generated the need for new solutions for processing and reasoning over RDF datasets since existing state-of-the-art techniques cannot cope with large volumes of RDF data. The speed of RDF production has also grown to the point where it became a prominent feature for it. That makes data processing a bold issue at the time of RDF generation.

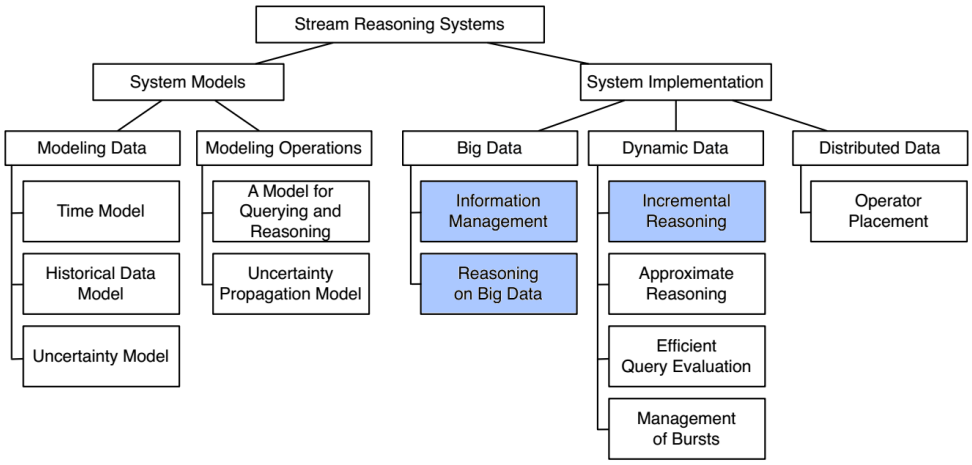


Figure 3.1: A Research Agenda for Stream Reasoning [53]

Figure 3.1 shows a global research agenda for stream reasoning [53], mentioning incremental reasoning as an important problem, which we deal with in this thesis.

In this chapter, we review proposals that are relevant to ours work and, more precisely, related to one of the following fields: *i)* RDF Reasoning contains two subsections called *RDF Partitioning* and *RDF Reasoning Using Big Data Platforms*, *ii)* Incremental Reasoning, and *iii)* Indexing structure for RDF data.

3.1 RDF Reasoning

In this section, we firstly study the RDF partitioning varieties that have been proposed in recent years. Then in the following, we review those literatures that dedicated to RDF reasoning using big data platforms mainly, Hadoop, Apache Spark, and Apache Flink.

3.1.1 RDF Partitioning

Data partitioning is trivial in distributed data management systems. The RDF data is typically split into several partitions and then distributed across the cluster machines. Partitioning aims to reduce the execution time by leveraging parallelism. Data partitioning demands a preprocessing overhead as it needs to be performed over the whole data. In a couple of decades, we have observed the RDF data sources are growing in all dimensions of big data (i.e., variety, velocity, volume, and veracity) remarkably. The volume and velocity of data generation make the storage systems vulnerable in case of keeping them. To get rid of the vulnerability of storage systems, RDF partitioning is also a solution to improve the scalability of the RDF storehouses. Despite the variety of graph partitioning techniques in the state-of-the-art [14, 15, 34, 68, 51, 42], that is hard to find the most proper partitioning system, in the general cases, for a given RDF data.

Recently in [15], Kondylakis *et al.* proposed a learning approach by utilizing two well-known machine learning algorithms, from the schema of a partitioned database to classify new streaming data. In their proposed method, [15], authors use the classical hash partitioning on predicate- and subject-based, e.g., on the triple's subject [66], together with the query workload knowledge, e.g., [39]. Despite other approaches that attempted to reduce the communication between nodes by relying on the predicate of triples [58], they proposed a partitioning method based on the predicate and subject of triples together. The proposed method is slightly closed to our approach on the part that using more than one premise. Moreover, the weak point of this approach is using a union of subject and predicate at the same time. Also, this method did not use any indexing technique to have control over the triple's object that is crucial for reasoning in some rules, e.g., in *domain*, *range*, etc.

Graux *et al.* in SPARQLGX [34] partitioned RDF datasets vertically by utilizing vertical partitioning approach [14], that a triple $s\ p\ o$ is stored in a file named p whose content keeps only subject s and object o entries. The authors of SPARQLGX used vertical partitioning in order to reduce the memory footprint and minimize the query response time. This improvement works for those searches that query RDF dataset based on their properties.

Schätzle *et al.* in [68] uses vertical partitioning (VP) [14] in their method as a based partitioning. The VP is designed based on *predicate* patterns of triples. The main advantage of VP is that it leads to a large reduction of the input size in general. The most drawback of VP is that the size of VP tables is highly skewed

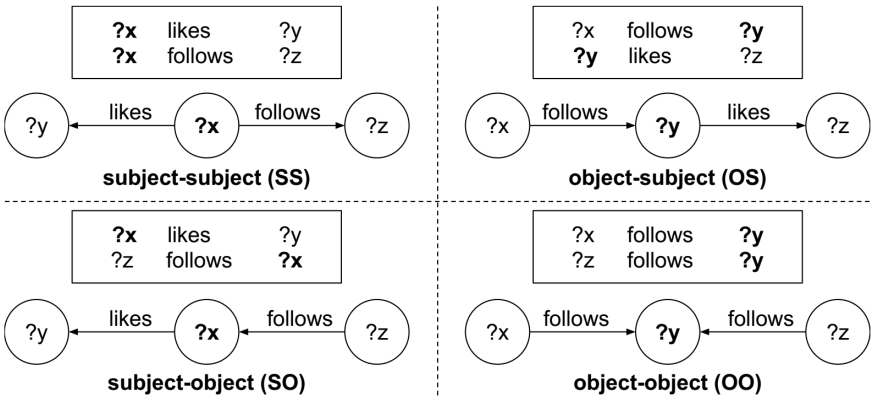


Figure 3.2: Correlation between triple patterns [68]

in a typical RDF dataset with other tables that hold only a few entries, while others include a substantial portion of the whole dataset. To leverage the vertical partitioning, S2RDF [68] presents an extension to the VP schema called `Extended Vertical Partitioning` (`ExtVP`). The proposed technique reduces the input data size of a query by using a preprocessing on semi-join –similar to the concept of Join Indices in the relational databases– to minimize the query input size despite any specific query patterns. ExtVP generates a set of sub-tables corresponding to a vertical partition table. The sub-tables are generated by using the right outer joins between VP tables. Figure 3.2 shows the possible correlations between the triple patterns. The position of a premise that occurs in both triple patterns –called join variable– determines the corresponding column of VP tables that must join. In this case, if the join variable is on a subject position in both triples, then it calls a subject-subject correlation (SS), as both VP tables must join on subjects (s) (Figure 3.2 top-left example). The other possible correlations are subject-object (SO), object-subject (OS), and object-object (OO).

Schätzle *et al.* in S2X [67] combine the graph parallel abstraction of GraphX [84] with the data-parallel computation of Spark to evaluate SPARQL queries in a distributed manner [16]. GraphX uses a vertex-cut partitioning strategy. In this strategy, the edges assign equally to machines and allow vertices to span nodes so that each vertex dedicate to a minimum number of nodes. For that purpose, S2X uses property graph [71] as a RDF data model. The RDF property graph has an ID for every vertex, and every edge has a property and two IDs of the connected

vertices. The edges store the predicate URIs, and the vertices store the subject and object URIs. Vertices also keep data structure for candidate query variables that could match this vertex. Therefore, every vertex in the graph keeps variables of a query where there is a match candidate for that. For that purpose, the first step is to match all triples patterns. Hence, S2X introduced two types of matches: local match and remote match. A local match is a set of the matches for any vertex, and a remote match is a set of adjacent vertices matches. The matched candidates validate with some validation rules by considering the local and remote matched sets. The invalid ones get dropped. In the graph, the locally changed match sets send to their adjacent for the validity of the next step. These processes repeating until no new changes happen. The individual subgraphs from the prior steps create the final output.

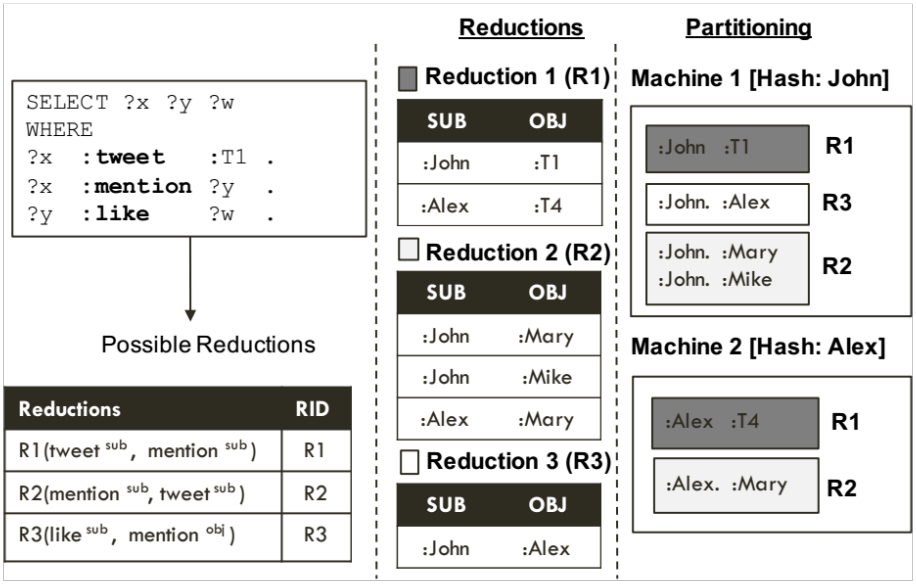


Figure 3.3: Workflow for Workload-Driven Partitioning [51]

The authors of WORQ [51] presented a *W*orkload-driven *R*DF *Q*uery *P*rocessing. The approach strives to minimize the network shuffling overhead based on the query workload. Therefore, WORQ, instead of focusing on a specific part of an RDF premise (e.g., subject only), does partitioning on the RDF data based on the join patterns of the so far received query. To minimize the shuffling overhead, they

Table 3.1: Spark-based RDF systems

System	Query Processing	Partitioning	SPARQL
SPARQLGX [34]	RDD API	Vertical	BGP+
S2RDF [68]	Spark SQL	Extended Vertical	BGP+
S2X [67]	Graph Iteration	Default	BGP+
WORQ [51]	Dataset API	Workload Join Keys	Not given Info

replace the partitions of the reduced sets of intermediate (reductions, in short) results that share the same join attribute on the same node of the cluster. WORQ also partitions the reduction data over the nodes, rather than using vertical partitioning. The reductions that have to partition is determined based on the join attributes, and this partitioning is done once and reused by any other query that has the same join attribute corresponding to the reduction.

Figure 3.3 presents a collection of join patterns from a given query and their similar reduced set of intermediate results. The R1 represents a join pattern on the :tweet property that uses the reduced set of intermediate results on the subject. Also, the R3 denotes that a join pattern on the *like* property that uses the reduced set of intermediate results on the subject as well. WORQ partitions the RDF triples of every reduction based on the subject or object join attribute. As represented in Figure 3.3, since the reduced set of intermediate results is on subject attributes, then R1, R2, and R3 do partitioning on their subjects. Since, the reduced intermediate RDF triples are hash-distributed over nodes, then the partitioning pattern ensures that all data related to the join attributes of a query are located in the same node and let them executed locally.

The general goal of all approaches that are presented above and summarized in Table 3.1 is to improve query performance by exploiting *data parallelization* by utilizing in-memory distributed processing platforms (e.g., Apache Spark). Therefore, data partitioning is a critical key factor in query processing; and it has a significant impact on query answering and RDF reasoning, particularly when they face big data. In the state-of-the-art in general, most of the works in this field use *Vertical Partitioning* [14], *Horizontal Partitioning* [26], a combination, or an extension of them. Notice that, there is not a unique partitioning pattern that suits all situations (e.g., a default partitioning splitting data based on the number of existence executors on every node), even it may cause skew data in one node. It easily could cause long execution time, until the skewed node finished its process,

and maybe the reason for a lot of network communication amongst the nodes of a cluster.

In this thesis, we propose an indexing technique that keeps a very light footprint of received/inferred RDF triples that are given through streamflow of data or derived implicit triples during the reasoning process. The indexing information enables us to pick those potential triples later from the disk, which is related to a new schema that will receive/infer later during streaming RDF data. The indexing technique provides the already received and processed RDF triples for the saturation process, to not lose any implicit triples by given a new schema. A similar partitioning idea has been recognized by WORQ [51]. They do partitioning the RDF data according to the join patterns of the received queries. This proposed technique is based on workload-driven partitioning, which is not suitable for RDF materialization when data and schema receive in-stream that we are exploring in this thesis. Our work does not require the knowledge of the query workloads, nor computes the join reductions for all possible joins. Although, many papers go beyond simple vertical partitioning by the computing all one-step-joints [68], which could be the cause of massive storage overhead and also multiple intermediate results that eventually lead to not fully optimized the execution time.

3.1.2 RDF Reasoning Using Big Data Platforms

To the best of our knowledge, the first proposal to use big data platforms, and MapReduce in particular, to scale the saturation operation is [55], but the authors did not present any experimental result. Other works then addressed the problem of large-scale RDF saturation by exploiting big data systems such as Hadoop and Spark, (see e.g., [76, 75, 35]). For example, Urbani *et al.* [76, 75] proposed a MapReduce-based distributed reasoning system called WebPIE. In doing so, they identified the order in which RDFS rules can be applied to efficiently saturate RDF data (Figure 3.4). Moreover, they specified for each of the RDFS and OWL-Horst rule how it can be implemented using the map and/or reduce functions, run some of the *join* operation in memory to improve the saturation speed, and executed over the Hadoop system. The implemented techniques run on a huge cluster (by default, cluster with 32 nodes) to show the scalability of their approaches.

Building on the work by Urbani *et al.*, the authors of Cichlid [35] implemented RDF saturation over Spark using, in addition to map and reduce, other data transformation operations that are provided by Spark, such as *map*, *filter*, *join*, *union*, etc. Cichlid has shown that the use of Spark can speed up saturation w.r.t the case

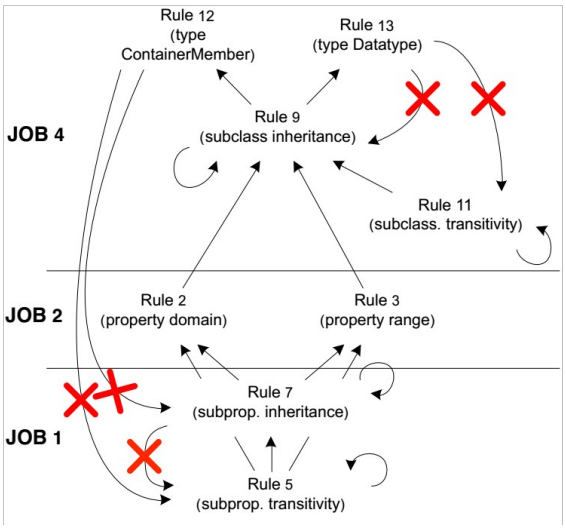


Figure 3.4: Relation between the various RDFS rules [75]

when Hadoop is used. Our solution builds, adapts, and leverages the solutions proposed by WebPIE (such as execution rules order of RDFS, execution rules grouping of OWL-Horst, using the output of some OWL-Horst rules group as an input of other groups, etc.), and Cichlid to cater for the saturation of streams of massive RDF data. It is worth mentioning that the Cichlid authors [35] do not provide any innovation except implementing the WebPIE methods in Apache Spark™.

The authors of [65] explore the ability of modern Big Data platform, mainly Apache Spark and Apache Flink, to examine highly expressive temporal Data-Log/Answer Set Programming over RDF data streams. In this paper, Ren *et al.* (same as Ticker [21] and Laser [19]) borrow LARS [20] theoretical foundation for the concept behind their algorithms. They consider Parallelism Level and Streaming Model as two factors to leverage the scalability and distributed RDF stream reasoning. The Parallelism Level consists of three levels to parallelize the evaluation of a classified Datalog/Answer Set Programming programs, which consists of *Component level, Rules level*, and *Single Rule level*, as defined in [61]. In this paper, Ren *et al.* designed a series of queries to cover all the three parallelism levels and to evaluate the performance impacts of their implementation (based on

LARAS) on the Big Data platform. For Streaming Model, they investigated two classes of streaming models adopted by modern distributed streaming engines, i.e., Bulk Synchronous Processing (handled by Apache Spark) and Record-at-A-Time (handled by Apache Flink). The authors of [65] proposing a method for query answering instead of materialization reasoning on stream. Also, they did not consider the previously received data for the reasoning. On the other hand, they did not use or propose any indexing technique to have fast access into the existed data.

Hu *et al.* in [40] proposed a method to compresses the RDF triples for reducing memory usage over Datalog reasoning. Therefore, the rules are sometimes able to apply to more than one fact at once. They inspire by the columnar database idea from [41], which keeps the common parts of facts only once. Those derived facts can represent by utilizing structure sharing. The proposed method skips some rule applications, and the approach efficiency observed in the case of working with relatively simple rules. [40] implemented in CompMat so that compressing the explicitly given data as part of the materialization process. The given method does not deal with in-stream RDF reasoning/materialization when face with massive RDF data, especially when schemas also convey in-stream fashion along with data.

3.2 Incremental Reasoning

The problem of incremental saturation of RDF data has been investigated by a number of proposals (see e.g., [79, 18, 27, 32, 75]). For example, Volz *et al.* in [79] investigated the problem of maintenance of entailments given changes at the level of the RDF instances as well as at the level of the RDF schema by elaborating their own previous works [77, 78]. In doing so, they adapted a previous state-of-the-art algorithm for incremental view maintenance proposed in the context of deductive database [69].

Barbieri *et al.* [18] builds on the solution proposed by Volz *et al.* [79] by considering the case where the triples are associated with an expiration date in the context of streams (e.g., for data that is location-based). They showed that the deletion, in this case, can be done more efficiently by tagging the inferred RDF triples with an expiration date that is derived based on the expiration dates of the triples used in the derivation. While Volz *et al.* [79] and Barbieri *et al.* [18] seek to reduce the effort required for RDF saturation, they do not leverage any indexing structure to efficiently perform the incremental saturation. As reported by the Volz

et al. [79] in the results of their evaluation study, even if the maintenance was incremental, the inference engine ran out in certain cases of memory. Volz *et al.* in [79] also mentioned that they cannot maintain function symbols other than constants. Regarding, Barbieri *et al.* [18], they considered in their evaluation a single transitive rule (Section 5 in [18]), and did not report on the size of the dataset used, nor the micro-batch size.

Chevalier *et al.* proposed Slider [27], a system for RDF saturation using a distributed architecture. Although the objective of the Slider is similar to our work, it differs in the following aspects. First, in Slider, each rule is implemented in a separate module. We adopt a different approach, where rules are broken into finer operations (*map, reduce, union,* etc.). This creates opportunities for sharing the results of processing at a finer level. For example, the result of a map can be used by multiple rules, thereby reducing the overall processing required. Second, Slider utilizes vertical partitioning [14] for indexing RDF triples. This indexing structure is heavy since it creates a table for each property (i.e., *subject*, *predicate*, and *object*) in the RDF triples. While such an indexing structure proved its efficiency in the context of RDF querying, it is heavy when it comes to RDF saturation. Indeed, we know in the context of RDF saturation the inference rules that can be triggered and therefore can tune the indexing structure needed for this purpose, which we did in our solution. In this approach, we use a lightweight indexing structure that is cheaper to maintain compared with vertical partitioning approach [14].

Guasdoué *et al.* proposed an incremental solution for saturating RDF data [32]. The incrementality comes from the fact that only rules that have a premise triple that is newly asserted or derived are triggered. We adopt a similar approach to Guasdoué *et al.*. However, we utilize an indexing structure to fetch existing triples that have been asserted/derived when processing previous micro-batches. Moreover, Guasdoué *et al.* apply the rules in an arbitrary order, whereas in our approach, we use rule ordering [75] and leverage it for streaming data in a way that minimizes the number of required iterations for saturating the RDF data.

The authors of WebPIE [75] briefly touched on the problem of incrementally saturating RDF data. In doing so, they tamp-stamped the RDF tuples to distinguish new and old tuples. An inference rule R is then activated only if the timestamp associated with one of its premises is new, i.e., greater than the last time the saturation was performed. We proceed similarly in our work. However, unlike our work, WebPIE does not leverage any indexing structures when querying the existing triples to identify those that may be used to activate a given rule R.

Ren *et al.* proposed Strider [63] as an RDF stream processing engine for continuous SPARQL query evaluation, by relying on Spark and Kafka. Ren *et al.* in [64] extended their previous work [63] by combining their SPARQL stream processing engine through encoding the given TBox[2] and ABox[3] triples. Ren *et al.* in [64] works on a trade-off of query rewriting and reasoning. In their approach, they utilize LiteMat [29] technique for query rewriting and reasoning, while in reasoning they focus on *i)* static TBox encoding and *ii)* dynamic Abox stream encoding. Ren *et al.* in both works do not deal with RDF reasoning, especially when schema given in stream.

Lie *et al.* proposed [50] in the continuous work of Heino *et al.* [37], i.e., as the first RDFS reasoning engine that utilizes GPU architecture. Lie *et al.* in [50] proposed the first work in stream reasoning field that using GPU instead of CPU. They formalize stream reasoning into temporal reasoning problems and introducing a compact design of RDF data in the format of the proposed formalization. They introduce a hash-based GPU encoding algorithm for translating string into integer IDs in a fixed-length. They ran their algorithms using CUDA[4] programming language on a PC with an NVIDIA GPU. In the case of dynamic *TBox* (i.e., the triples that contain *subPropertyOf*, *subClassOf*, etc.), to reduce the transitive closure iteration time, the join result applies over inferred triples within two last iterations in their algorithms. Lie *et al.* in [50] claim that their strategy reduces computation and allows a substantial increase in performance. To prove their claim, they did not present any experimental result that deals with a massive enough RDF data in-stream when they are larger than the size of the memory.

The authors of [62] proposed a continuous ASP-based (Answering Set Programming) reasoning over RDF streams on RDF data. The C-ASP processing model merges two developed models: i) Data Stream Management System (DSMS) based on windowing, and ii) Complex Event Processing (CEP) systems where rules applied to timestamped data. The process breaks down in three steps: *a) Windowing*: select subsets of the most recent elements of the input streams; *b) Evaluation*: perform reasoning on the finite and intermediate data portions; and *c) Streaming*: convert the final solutions back into the stream. In all three steps, the reasoning is based on windowing while not considering static semantics in-stream neither using massive data or any big data platform for their process.

To sum up, in comparison with the-state-of-the-art of incremental saturation

[2] An ontology, aka **T**erminological **Box**

[3] A fact base, aka **A**ssertional **Box**

[4] http://www.nvidia.com/object/cuda_home_new.html

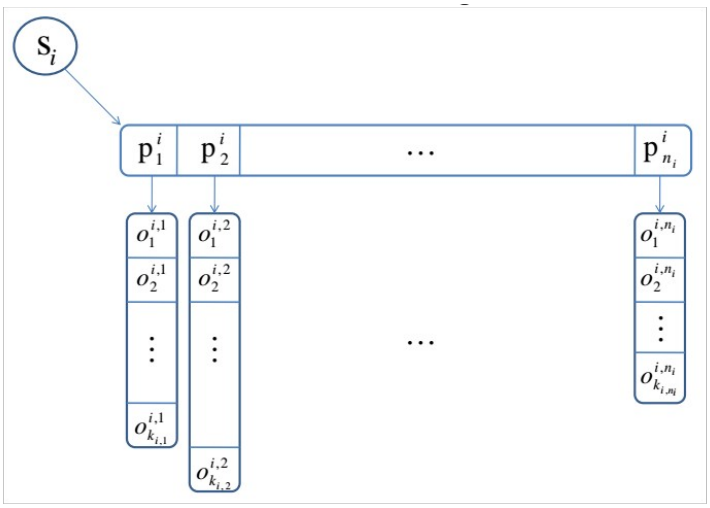


Figure 3.5: spo Indexing in Hexastore [83]

of RDF, we leverage a lightweight indexing structure, a fine-tuned ordering of the execution of the rules, as well as the use of a Big Data platform, namely Spark, to efficiently saturate large in-stream micro-batches of RDF data.

3.3 Indexing Structures for RDF Data

The indexing mechanism we proposed in this thesis is comparable to those proposed by Weiss et al. [83], by Schätzle *et al.* [68] and by Kaoudi *et al.* [43] for efficiently evaluating SPARQL queries. For example, Weiss *et al.* developed Hexastore, a centralized system that maintains six indexes for all triple permutations, namely *spo*, *sop*, *pso*, *pos*, *osp*, and *ops*. For example, using *spo* indexing a subject s_i is associated with a sorted list of properties $\{\mathtt{p}_1^\mathtt{i}, \ldots, \mathtt{p}_\mathtt{n}^\mathtt{i}\}$ Moreover, each property, e.g., $\{\mathtt{p}_1^\mathtt{i}\}$, is associated with an associated sorted list representing the objects $k_{i,j}$, i.e., $\{\mathtt{o}_1^{\mathtt{i},1}, \ldots, \mathtt{o}_{\mathtt{k_i},1}^{\mathtt{i},1}\}$ (Figure 3.5). While this approach allows for efficiently evaluating SPARQL queries, it is expensive in terms of memory usage and index maintenance. According to the authors, Hexastore may require 5 times the size of the storage space required for storing an RDF dataset due to the indexes. The solution developed by Schätzle *et al.* [68], on the other hand, is meant for

distributed evaluation of SPARQL queries using Hadoop. To do so, they use an indexing scheme named ExtVP, which precompute semi-join reductions for between all properties. As shown by the authors, the computation of such indexes is heavy, e.g., it requires 290 seconds to index 100 million triples. To alleviate this, we proposed here an index that is aims to speed up RDF saturation, as opposed to any SPARQL query, and that is amenable to incremental maintenance.

In recent years, considering the nature of the RDF graph, many indices have been proposed for the RDF graph. Udrea *et al.* in GRIN [74] proposed a graph-based lightweight indexing technique by assuming that all inferences on *rdfs:subClassOf* and *rdfs:subPropertyOf* are prepared apriori in the RDF graph where there is a need to traverse edges in the graph determined by an RDF database. Udrea *et al.* identified the radius of a circle, where they draw circles around the selected "center" vertices in the graph where it would include vertices that are at a defined distance of the "center" vertex. To this end, GRIN gathers information within a given distance of vertices by a given radius and utilizes a balanced binary tree to index information. They used Jena API to parse the input RDF files while GRIN was implemented in Java. They reported GRAIN builds the index of 300 MB RDF data in under 50 seconds, while the indexation size is 75 MB, and space to store the data as a hash-table needs 320 MB. So, despite that, they mentioned the indexation is lightweight and just pointed to the data. It is too heavy compared to our indexing technique. Our technique builds a 129 KB index file for a 2.9 GB dataset in 86 seconds an average, i.e., amongst 100 successive micro-batches each 500 MB. Indeed, the proposed indexation by GRIN is heavy for massive streaming RDF data. Consider that, in streaming RDF, indexing needs to update as frequently as a new micro-batch arrives.

Ladwig *et al.* in [46] proposed a keyword search on structured graph data. So, they integrate query translation with query answering to alleviate the overlaps of the query translation process and the necessary process for the query answering. Therefore, they proposed an incremental process for intermediate results computed during query translation aim to reuse them for query answering. The authors of [46] do not show how the proposed algorithm can incrementally update the index structure for RDF data and their schema when they receive them in-stream.

Zou *et al.* in [88] proposed storing and querying RDF data from the graph database. To speed up query processing, they create two indexes, namely VS-tree or VS*-tree index, on each vertex for exact and wildcard SPARQL queries. Zou *et al.* do not investigate their indexation method for streaming RDF. Since our indexing method incrementally update information based on the edges and

vertexes. Our indexing technique keeps information from those edges when they are not `rdf:type`, and also for all vertices as *object* if the connecting edge into it be `rdf:type`.

Kim *et al.* [44] extend the triple filtering method to exploit graph-structured information. To improve the filtering effects, they proposed RG-index for indexing the graph patterns in the RDF graph. To index the graph patterns, Kim *et al.* proposed an adaptation method of the gSpan [85] algorithm for RDF graphs. Also, they introduced a mechanism for caching the intermediate results to efficiently processing graph pattern mining. In this thesis, the proposed index technique is based on the predicate or object of RDF data to empower the system in case of re-materializing only a partial part of RDF data, instead of the whole, so far, given RDF data.

The following studies address decreasing the redundant intermediate results. Kim *et al.* in [45] proposed *R3F*, a filtering method, to exploit the graph-structure information. Therefore, they proposed a path-based index method for filtering RDF triple for the original query processing. Chen *et al.* [47] for RDF data of robotic systems proposed a star-based partitioning and index algorithm. They create an index S-tree based on star structure, while every leaf node represents a star structure. To reduce the storage cost, the encoding of the star structure is compressing data by the Run-Length Encoding method. Then, a compressed-AND operation (designed to avoid the encoding transformation back and forth) is applied directly to the compressed encoding. None of the last two proposed methods did propose any solution in their suggested indexing methods to update their information incrementally.

3.4 Conclusion

In this chapter, we studied literature by considering various compatible aspects of our work. To this end, we investigate the related-works in three categories: i) RDF reasoning, ii)Incremental reasoning, and iii) Indexing structures for RDF triples. We studied RDF reasoning by dividing it into two subsections: RDF partitioning and RDF reasoning by using big data platforms, mainly Apache Spark.

RDF Partitioning It is a fundamental aspect of distributed processing and query answering. Many research [14, 15, 34, 68, 51, 42] benefited from the well-known partitioning technique, e.g., such as Vertical Partitioning [14], directly or as a base for the proposed partitioning system [68]. Since data partitioning is a critical key

factor in query processing, but there is not a unique partitioning pattern that suits all situations. Therefore, in this thesis, for the sake of stream RDF reasoning, we propose an indexing technique that keeps a very light footprint of received/inferred RDF triples that are given through streamflow of data or derived implicit triples during the reasoning process. The indexing information empowers us to pick those potential triples later from the disk, which is related to a new schema that will receive/infer later during streaming RDF data.

RDF Reasoning by Using Big Data Platforms This thesis is built on top WebPie[75] and Cichlid [35]. WebPie proposed a MapReduce-based distributed RDF reasoning, and Cichlid implemented the WebPIE methods in Apache Spark. None of the approaches has proposed a stream RDF reasoning in practice.

Incremental Reasoning The problem of incremental saturation of RDF data has been investigated by several proposals [79, 18, 27, 32, 75]. [79] investigated the problem of maintenance of entailments given changes at the level of the RDF instances and schema by elaborating previous works [77, 78]. [18] builds on [79] by respecting the case where the triples are associated with an expiration date in the context of streams, and they showed that the deletion can be done more efficiently. [79] and [18] do not leverage any indexing structure to efficiently perform incremental saturation. [27] proposed a system for RDF saturation using a distributed architecture. [27] implements each rule in a separate module and utilizes vertical partitioning [14] for indexing RDF triples. The proposed indexing structure by [27] is heavy when it comes to RDF saturation since it creates a table for each property in the RDF triples. [32] proposed an incremental solution for saturating RDF data by relying on the fact that only rules that have a premise triple that is newly asserted or derived are triggered. We adopt a similar approach but utilize an indexing structure to fetch existing triples that have been asserted/derived when processing previous micro-batches. Unlike [32], we use the rule ordering [75] and leverage it for streaming data in a way that minimizes the number of required iterations for saturating the RDF data. [75] quickly discussed the problem of incrementally saturating RDF data by tamp-stamping the RDF tuples to distinguish new and old tuples. We proceed similarly in our work and leverage instead when querying the existing triples to identify those that may be used to activate a given rule.

Indexing Structures for RDF Data The indexing mechanism we proposed in this thesis is comparable to those proposed by [83, 68, 43] for efficiently evaluating SPARQL queries. [83] developed Hexastore, a centralized system that maintains six indexes for all triple permutations. While this approach allows for efficiently

evaluating SPARQL queries, it is expensive in terms of memory usage and index maintenance. On the other hand, [68] is a distributed evaluation of SPARQL queries using Hadoop. To do so, they use an indexing scheme based on [14], which precompute semi-join reductions for between all properties. This approach is heavy for stream RDF reasoning, and to alleviate that, we proposed here an index that aims to speed up RDF saturation that is amenable to incremental maintenance.

Indeed, we know in the context of RDF saturation the inference rules that can be triggered and, therefore, can tune the indexing structure needed for this purpose, as done in our contribution.

4 RDFS Saturation in Streaming

Contents

In this chapter, we describe our contribution in detail on the saturation of RDF streams by leveraging on Spark Streaming processing capabilities. Using Spark, an RDF stream is discretized into a series of timestamped micro-batches that come (and are, therefore, processed) at different time intervals. In our work, we assume that a micro-batch contains a set of instance RDF triples, but may also contain schema (i.e., RDFS) triples.

Consider, for example, an RDF stream composed of the following series of micro-batches $[\mathtt{mb}_1, \ldots, \mathtt{mb}_n]$, where $n \geqslant 2$. A first approach for saturating such a stream using a batch-oriented solution would proceed as follows: when a micro-batch $\mathtt{mb}_i$ arrives, it unions $\mathtt{mb}_i$ with the previous instance dataset (including triples obtained by previous saturation) and then the resulting dataset is totally re-saturated.

On the contrary, our approach allows for RDF saturation in a streaming fashion, by sensibly limiting the amount of data re-processing upon the arrival of a new micro-batch. To this end we have devised the following optimization techniques:

1. *Rule pruning for schema saturation* Given a new micro-batch $\texttt{mb}_\texttt{i}$, we filter all the RDF schema triples contained in it. Note that in the general case it is not likely that these new schema triples trigger all the saturation rules, i.e. it is not the case that the new micro-batch includes all kinds of RDFS triples at once - i.e. *subPropertyOf, domain, range,* and *subClassOf*. So for saturating the schema at the level of the new micro-batch we first filter new schema triples, and then obtain the set of new schema triples named *NST* (Figure 4.2).

NST = `Transitive Closure`*(new received sch* ∪ *present sch) - present sch*

Figure 4.1: *N*ewly *S*chema *T*riples

NST is obtained by considering the effect of existing schema triples on new arrival schema triples. For example, suppose that the schema triple $p \prec_{sp} q$ already exists. By receiving a newly schema triple (e.g., $q \prec_{sp} r$), the *NST* infer, also, an implicit schema triple ($p \prec_{sp} q \;\&\; q \prec_{sp} r \Rightarrow p \prec_{sp} r$), that the passed triples (already received and saturated) never met them before. Since the volume size of schema triples, in general, are small enough to fit in one node, therefore the `Transitive closure` operator (in Figure 4.2) applies locally on the data in purpose of inferring implicit schema triples. Table 4.1 illustrates the rules to be activated given some matching schema triple. For example, if a schema triple specifying the domain of a property exists, then this triggers rule 2. All possible cases are indicated in Table 4.1, and saturation selects one line of this table, depending on the kind of schema predicates met in the new schema triples. This avoids triggering useless rules. Once saturation for $\texttt{mb}_\texttt{i}$ schema triples is done in this optimized fashion, obtained triples (i.e., *NST*) are merged with the existing RDFS schema for a second-pass of global schema saturation, taking into account triples deriving from both $\texttt{mb}_\texttt{i}$ and the pre-existing schema.

2. *Efficiently saturate existing instance triples by leveraging our incremental*

Table 4.1: The 1 and 0 indicate the availability of that particular schema rules in $\texttt{mb}_\texttt{i}$. $X \rightarrow Y$ means: The output of rule X used as an input of rule Y.

	subPropertyOf	**domain**	**range**	**subClassOf**	**Saturation Order**
1	1	1	1	1	*R7 → (R2, R3) → R9*
2	1	1	1	0	*R7 → (R2, R3)*
3	1	1	0	1	*R7 → R2 → R9*
4	1	1	0	0	*R7 → R2*
5	1	0	1	1	*R7 → R3 → R9*
6	1	0	1	0	*R7 → R3*
7	1	0	0	1	*R7, R9*
8	1	0	0	0	*R7*
9	0	1	1	1	*(R2,R3) → R9*
10	0	1	1	0	*R2, R3*
11	0	1	0	1	*R2 → R9*
12	0	1	0	0	*R2*
13	0	0	1	1	*R3 → R9*
14	0	0	1	0	*R3*
15	0	0	0	1	*R9*
16	0	0	0	0	-

indexing scheme Given the new schema triples that are provided by the micro-batch $\texttt{mb}_\texttt{i}$ or inferred in (1), we need to scan existing instances triples to identify those that if combined with the new schema triples will trigger RDFS rules in Table 2.3. This operation can be costly as it involves examining all the instance triples that have been provided and inferred given the micro-batches received before $\texttt{mb}_\texttt{i}$. To alleviate this problem, we have devised an incremental indexing technique that allows for the optimal retrieval of the instance triples that are likely will trigger the RDFS rules given some schema triples. The technique we developed indexes instance triples based on their predicate and object, and, as we will show later, allows to greatly reduce the data processing effort for the saturation under the new schema. Once retrieved, such instances triples are used together with the new schema triples to generate new instance triples. Notice here that we cannot infer new schema triple. This is because the rules for inferring new schema triples require two schema triples as a premise (see Table 2.3 - *rdfs2, rdfs3, rdfs7,*

and rdfs9).

3. *Saturate new instance triples* The instance triples inferred in (2) need to be examined - with existing and *NST* schema triples - as they may be used to infer new instance triples. Specifically, each of those triples is examined to identify the RDFS rule(s) to be triggered. Once identified, such rules are activated to infer instance triples. The instance triples in $\texttt{mb}_\texttt{i}$ as well as those inferred in (2) and (3) are stored and indexed using the method that we will detail next.

4. *Duplicate elimination* In this work, we eliminate the duplicates in two-steps. As a first step that we call partial elimination, we remove the duplicates from every saturated micro-batch before writing them on the disk. So far, same as Cichlid [35] (batch saturation processing), every micro-batch is duplicated free. So, despite that, the micro-batches are duplicate free individually, but still, there may exist duplicates between micro-batches. To avoid those potential duplicates, and to not consider them in the next saturation process, we eliminate duplicates after fetching the relevant triples from *DS′* based on the NST. By this strategy, we only add $\texttt{O(N log N)}$ extra execution time (the time that requires for applying *distinct* operation) on those number of triples, $\texttt{N}$, that they picked relatively. This technique prevents utilizing *distinct* operation among the current and the whole triples *DS′* at the written time by given every micro-batch.

We will now turn our attention to our indexing scheme, mentioned above. For a micro-batch $\texttt{mb}_\texttt{i}$ received at time-stamp `t` we create an HDFS directory named as `t`, in which we store other indexing information related to $\texttt{mb}_\texttt{i}$, as follows. The instance triples that are asserted in $\texttt{mb}_\texttt{i}$, as well as those that are inferred (see (2) and (3) above), are stored into two separate sub-directories, which we name `o` and `p`.

The instance triples in $\texttt{mb}_\texttt{i}$ that provide information about the type of resource, i.e., having as predicate `rdf:type`, are stored in the `o` directory. Such triples are grouped based on their object, and they are stored in files within the `o` directory of the micro-batch $\texttt{mb}_\texttt{i}$. Specifically, instance triples with the same object are stored in the same file. Additionally, our indexing scheme utilizes an associative hash-table stored in the driver memory as a cached RDD, associating each encountered object with the list of HDFS addresses corresponding to files in the `o` directories, which include at least one triple with that object. Notice that triples with the

rdf:type predicate are used in the premises of *rdfs9*. Given a schema triple of the form $\mathtt{y} \prec_{sc} \mathtt{z}$, our indexing approach allows for the fast retrieval of the files in the o directories of the micro-batches that have as an object the resource y, and therefore can be used to trigger *rdfs9*.

The remaining instance triples in $\mathtt{mb}_i$, i.e., those that do not have rdf:type as a predicate, are grouped based on their predicate, and stored within files under the p directory. Additionally, an associative hash-table stored in an RDD persisted in main memory, associating each encountered property with the list of HDFS addresses corresponding to files in the p directories including at least one triple with that property is created and maintained. By means of this kind of indexing, we can optimize application of rules *rdfs2*, *rdfs3* and *rdfs7* to infer new instance triples as we can inspect the previously described hash-table in order to retrieve only files containing triples with properties needed by these 3 rules.

To illustrate, consider for example that a new micro-batch $\mathtt{mb}_i$ arrives at a given time t, and that it contains the schema triple $\mathtt{t}_{sc}$: $s_1 \prec_{sc} s_2$. Such schema triple can contribute to the inference of new schema triples (i.e., by means of *rdfs11*) as well as new instance triples by means of *rdfs9*. Since the indexation mechanism we elaborated is sought for the inference of instance triple, let us focus on *rdfs9*. To identify the instance triples that can be utilized together with the schema triple $\mathtt{t}_{sc}$, we need to examine existing instance triples. Our indexing mechanism allows us to sensibly restrict the set of triples that need to be examined, as the hash-table indexing the files under the o directories enables the fast recovering of files containing triples with $\mathtt{s}_1$ as an object resource, and that can be combined with the schema triple $\mathtt{t}_{sc}$ to trigger *rdfs9*. The indexing of the files in p directories are operated in a similar manner in order to efficiently recover files containing *instance* triples with a given property so as to use included triples to trigger *rdfs2/3/7*, under the arrival of a correspondent schema triple in the stream. To illustrate our approach in more detail let's consider the following example.

Example 4.1. We assume that we have the initial schema **S** of Figure 2.1 and that we saturate it by given two more schema triple to obtaining **S**′ as indicated below.

$$\mathbf{S}' = \texttt{TransitiveClosure}(\ \mathbf{S} \cup \{\mathit{hasContactA} \hookrightarrow_r \mathit{rdfs:Literal},\quad \mathit{\_:b_0} \prec_{sc} \mathit{paper}\}\ )$$

This operation is fast and centralized, as the initial schema is always relatively small in size to fit in memory. Our approach then proceeds according to the

following steps.

1. The saturated schema $\mathbf{S}'$ is broadcasted once to each cluster nodes' executors so that it can access $\mathbf{S}'$ with no further network communication.

2. Then available micro-batches are processed. For the sake of simplicity we make here the, unnatural, assumption that each micro-batch consists of only one triple. An assumption of the stream of micro-batches shown in Table 4.2, while first micro-batch mb_1 convey an instance triples $\text{doi}_1\ \tau\ \_:\text{b}_0$ and so on.

Table 4.2: Instance triples

mb	**Subject**	**Predicate**	**Object**
1	doi_1	τ	$\_:\text{b}_0$
2	doi_1	*hasTitle*	"CAQU MV"
3	doi_1	*hasAuthor*	"SA"
4	doi_1	*hasContactA*	$\_:\text{b}_1$
5	doi_1	*inProceedingsOf*	$\_:\text{b}_2$
6	$\_:\text{b}_2$	*hasName*	"PODS'98"
7	...	...	...

3. The first received micro-batch triggers *rdfs9* so that we have the derivation of two new triples:

$$\{\text{doi}_1\ \tau\ \_:\text{b}_0 \mid \_:\text{b}_0 \prec_{sc} \text{confP}\} \text{ - } rdfs9 \rightarrow \text{doi}_1\ \tau\ \text{confP}$$
$$\{\text{doi}_1\ \tau\ \_:\text{b}_0 \mid \_:\text{b}_0 \prec_{sc} \text{paper}\} \text{ - } rdfs9 \rightarrow \text{doi}_1\ \tau\ \text{paper}$$

The received triple plus the two derived ones are then stored according to our indexing strategy. As already said, triples are grouped by their objects when having `rdf:type` property, so as to obtain the following file assignment, knowing that t_1 is the time stamp for the current micro-batch:

$$doi_1\ \tau\ confP \rightarrow \text{o}/t_1/file_1,$$
$$doi_1\ \tau\ paper \rightarrow \text{o}/t_1/file_2,$$
$$doi_1\ \tau\ \_{:}b_0 \rightarrow \text{o}/t_1/file_3$$

We, also, keep the objects of the above triples along with their physical file location in a hash table for the later fast accessing. In the above case the indexing hash table is:

key	**values**
confP	$\text{o}/t_1/file_1$
paper	$\text{o}/t_1/file_2$
$\_{:}b_0$	$\text{o}/t_1/file_3$

4. The processing goes on by deriving new instance triples for the micro-batches from 2 to 6, as indicated in the Table 4.3, which also indicates how instance triples are stored/indexed.

Table 4.3: Saturated Streaming Triples

mb_i	**Received Triple**	**Schema Triple**	Entails(**E.**) & Received(**R.**)	**Indx. Key**	**Stored Paths & Index Value**
1	$\text{doi}_1\ \tau\ \_{:}\text{b}_0$	$\_{:}\text{b}_0 \prec_{sc} \text{confP}$, $\_{:}\text{b}_0 \prec_{sc} \text{paper}$	**E.** $\text{doi}_1\ \tau\ \text{confP}$, **E.** $\text{doi}_1\ \tau\ \text{paper}$, **R.** $\text{doi}_1\ \tau\ \_{:}\text{b}_0$	confP, paper, $\_{:}\text{b}_0$	$\text{o}/\text{t}_1/\text{file}_1$, $\text{o}/\text{t}_1/\text{file}_2$, $\text{o}/\text{t}_1/\text{file}_3$
2	doi_1 *hasTitle* "CAQUMV"	$\text{hasTitle} \hookleftarrow_d \text{confP}$	**E.** $\text{doi}_1\ \tau$ paper, **R.** doi_1 *hasTitle* "CAQUMV"	paper, *hasTitle*	$\text{o}/\text{t}_2/\text{file}_1$, $\text{p}/\text{t}_2/\text{file}_1$
3	doi_1 *hasAuthor* "SA"	$\text{hasAuthor} \hookleftarrow_d \text{paper}$	**E.** $\text{doi}_1\ \tau$ paper, **R.** doi_1 *hasAuthor* "SA"	paper, *hasAuthor*	$\text{o}/\text{t}_3/\text{file}_1$, $\text{p}/\text{t}_3/\text{file}_1$
4	doi_1 *hasContactA* $\_{:}\text{b}_1$	no inference	**R.** doi_1 *hasContactA* $\_{:}\text{b}_1$	*hasContactA*	$\text{p}/\text{t}_4/\text{file}_1$
5	doi_1 *inProceedingOf* $\_{:}\text{b}_2$	$\text{inProceesingOf} \hookleftarrow_d \text{confP}$, $\text{inProceesingOf} \hookrightarrow_r \text{conference}$	**E.** $\text{doi}_1\ \tau\ \text{confP}$, **E.** $\_{:}\text{b}_2\ \tau$ conference, **R.** doi_1 *inProceedingOf* $\_{:}\text{b}_2$	confP, conference, *inProceedingOf*	$\text{o}/\text{t}_5/\text{file}_1$, $\text{o}/\text{t}_5/\text{file}_2$, $\text{p}/\text{t}_5/\text{file}_1$
6	$\_{:}\text{b}_2$ *hasName* "PODS'98"	$\text{hasName} \hookleftarrow_d \text{conference}$	**E.** $\_{:}\text{b}_2\ \tau$ conference, **R.** $\_{:}\text{b}_2$ *hasName* "PODS'98"	conference, *hasName*	$\text{o}/\text{t}_6/\text{file}_1$, $\text{p}/\text{t}_6/\text{file}_1$

Now assume that in micro-batch 7 we have the following RDF schema triples:

$$paper \prec_{sp} publication,$$
$$hasContractA \prec_{sp} hasAuthor$$

So we have now three steps: **i)** to obtain global newly schema triples, we examine the received plus filtering out already present schema triples in purpose to get NST (figure 4.2), then **ii)** broadcast these schema triples minus the already exist/broadcast schema triples, to enable tasks to locally access them. Next, **iii)** re-processing previously met and inferred instance triples by taking into consideration the new schema (the NST). Consider, for instance, $\{hasContactA \prec_{sp} hasAuthor\}$ as new schema triple. This schema triple triggers *rdfs7*. Therefore, the schema triple affects those instance triples when they have *hasContactA* as the predicate. Thus, by referring to the key of our indexing information, it tells us that *only* file $p/t_4/file_1$ (Table 4.3, line 4) needs to be loaded to, potentially may infer new triple, that, of course, will be in turn stored according to our indexing strategy.

$$\begin{aligned} \texttt{NST} = \{ \quad & \boldsymbol{paper} \prec_{sc} \boldsymbol{publication}, \boldsymbol{hasContactA} \prec_{sp} \boldsymbol{hasAuthor} \\ & posterCP \prec_{sc} publication, confP \prec_{sc} publication, \\ & \_:b_0 \prec_{sc} publication, hasContactA \hookrightarrow_d paper \ \} \end{aligned}$$

Figure 4.2: New received and inferred Schema Triples (NST)

As we will see in our experimental analysis, the pruning of loaded files ensured by our indexing will entail fast incremental saturation. Also, note that our approach tends to create a non-negligible number of files, but fortunately without compromising efficiency thanks to distribution.

4.1 Streaming Saturation Algorithm

The overall streaming saturation algorithm is shown in Algorithm 2, and commented hereafter. Consider that a given micro-batch can convey three different combinations of data. They carry *i)* just instance triples, *ii)* just schema triples, or

iii) instance and schema triples. Here is a global overview of those steps before explaining them in details:

i) Just given instance triples: This case is the most straightforward and easiest situation. We apply the batch saturation process by considering the execution of RDFS rules ordering. At the end of the process, we collect the indexing information from the given and generated implicit RDF triples and store them on the HDFS.

ii) Just RDF schema triples: Phase I) Preparing `NST`[1] from the given and existing schema triples. In the next step, by considering the indexing data, we confirm the presence of the corresponding instance triples, by relying on their footprint from indexing information, without involving the disk. The detected triples fetch and saturate with the `NST`. In the next step, the broadcasted schema triples update by considering the `NST`. *Phase II)*, we collect index information from the inferred implicit instance triples and store them on HDFS.

iii) Combination of instance and schema triples: We firstly extract schema triples from the given micro-batch and apply *Phase I* of the previous step *(ii)*. The inferred implicit triples, if any, are new, so they need to examined with whole schema triples (means, existed schema plus `NST`). Therefore, the inferred implicit triples concatenate into the given instance triples and then following the step *(i)*.

4.1.1 Process, Store, and Index

Given a micro-batch $\texttt{mb}_\texttt{i}$, we first perform schema saturation if $\texttt{mb}_\texttt{i}$ contains schema triples (*lines 12, 13*). The related instance triples are retrived based on $\texttt{mb}_\texttt{NST}$ (*line 14*). Given newly inferred schema triples, instance triples are reterived and examined to identify cases where new instance triples may be inferred (*line 15*). The obtained schema triples (i.e., $\texttt{mb}_\texttt{NST}$) are added and broad-casted within the intial schema RDD (*line 17, 18*). The inferred triples, if any, are merged with instance triples of $\texttt{mb}_\texttt{i}$ (i.e., $\texttt{mb}_\texttt{ins}$) and the saturation is applied to them. In the next step, the received and inferred instance triples are combined and obtained

[1] A small reminder: (*NST* = `Transitive Closure`*(new received sch* ∪ *present sch) - present sch*)

duplicates, if any, are partially removed (*line 22*). In the last step, the instance triples from the previous step are saved and indexed using our method (*line 24-25*).

Algorithm 2: Overall Algorithm for Saturating RDF Stream

1: **Input:** $\texttt{MB} \leftarrow [\texttt{mb}_1, \cdots, \texttt{mb}_n]$ // a stream of micro-batches.
2: **Output:** $\texttt{Schemas} \leftarrow [\texttt{Sch}_1, \cdots, \texttt{Sch}_n]$ // $\texttt{Sch}_i$ represents the schema triples obtained as a result of saturating the micro-batches $\texttt{MB} = [\texttt{mb}_1, \cdots \texttt{mb}_i]$.
3: **Output:** $\texttt{Datasets} \leftarrow [\texttt{DS}_1, \cdots, \texttt{DS}_n]$ // $\texttt{DS}_i$ represents the instance triples obtained as a result of saturating the micro-batches $\texttt{MB} = [\texttt{mb}_1, \cdots \texttt{mb}_i]$.
4: **Output:** $\texttt{IndexInformations} \leftarrow [\texttt{oIndex}, \texttt{pIndex}]$ // oIndex and pIndex keeps object- and predicate-based information respectively.
5: $\texttt{D}_{\texttt{ins}} \leftarrow \emptyset$ // Initialize a dataset for instance triples
6: $\texttt{D}_{\texttt{sc}} \leftarrow \emptyset$ // Initialize a dataset for schema triples
7: *br* ← if $\texttt{D}_{\texttt{sc}}$ exist then TransitiveClosure and broadcast them
8: **do** {
9: $(\texttt{mb}_{\texttt{sch}}, \texttt{mb}_{\texttt{ins}}) \leftarrow \texttt{SeparatingTriples}(\texttt{mb}_i)$ // Separate schema from instance triples of received mb
10: *if*($\texttt{mb}_{\texttt{sch}}$ exist then) {
11: // Build and broadcast a *NST* correspond to the current mb
12: $\texttt{mb}'_{\texttt{sch}} \leftarrow (\texttt{TransitiveClosure}\ (\texttt{mb}_{\texttt{sch}} \cup \texttt{D}_{\texttt{sc}}))$ - $\texttt{D}_{\texttt{sc}}$
13: $\texttt{mb}_{\texttt{NST}} \leftarrow \texttt{broadcast}(\texttt{mb}'_{\texttt{sch}})$
14: // Fetch triples from *DS'* and re-saturate them with the *NST*
15: $\texttt{D}'_{\texttt{ins}} \leftarrow$ Fetch triples using Indexing variable based on $\texttt{mb}_{\texttt{NST}}$
16: $\texttt{mb}'_i \leftarrow \texttt{Saturate}(\texttt{D}'_{\texttt{ins}}, \texttt{mb}_{\texttt{NST}})$
17: // Combine and re-broadcast the received and existing RDFS
18: $\texttt{D}_{\texttt{sc}} \leftarrow \texttt{mb}_{\texttt{NST}} \cup \texttt{D}_{\texttt{sc}}$
19: $\texttt{br} \leftarrow \texttt{broadcast}(\texttt{D}_{\texttt{sc}})$
20: }
21: // Saturate the received instance triples with total RDFS triples
22: $\texttt{mb}_{\texttt{imp}} \leftarrow \texttt{Saturate}(\texttt{mb}_{\texttt{ins}} \cup \texttt{mb}'_i, \texttt{br})$
23: $\texttt{mb}''_i \leftarrow (\texttt{mb}_{\texttt{ins}} \cup \texttt{mb}_i{}' \cup \texttt{mb}_{\texttt{imp}}).\texttt{distinct}$
24: // The following two lines are handled by **Indexing Algorithm 3**
25: Save $\texttt{mb}''_i$ in the HDFS
26: $[\texttt{oIndex}, \texttt{pIndex}] \cup \leftarrow \texttt{indexing}(\texttt{mb}''_i)$
27: } **while**(is there an incoming micro-batch mb?)
28: **End**

The indexing algorithm (Algorithm 3) is responsible for:

i) partitioning the instance triples by their *object*/*predicate*,

ii) store the partitioned triples on HDFS on object- or predicate-based paths,

iii) and finally, keep a hash table in memory that contains the objects'/predicates' of the intended triples as a unique key and their physical stored paths as value.

We rather focus here on the algorithm for indexing, which is central to our contribution. Central to the efficiency of the solution presented in the previous section is the technique that we elaborated for incrementally indexing the new instance triples that are asserted or inferred given a new micro-batch.

Algorithm 3: Incremental RDFS Indexing Algorithm

```
1: // mb'_i is indicated as instance and implicit triples from received mb'_i
2: Input: Saturated mb'_i
3: // The information of mb'_i keeps as two Sets in memory.
4: Output: oIndexingSet, pIndexingSet
5: Begin
6:    // Get a fixed timestamp to save the mb'_i triples.
7:    fts ← TimeMillis

8:    // The mb'_i triples partitions by their object where their predicate is rdf:type.
9:    oPartition ← mb'_i.filter( _._2.contains("rdf-syntax-ns#type")).
10:       map(t ⇒ (t._3, t._1)).partitionBy(number of different object in mb'_i).
11:       mapPartitions(_.map(t ⇒ (t._2, t._1))).persist

12:   // The mb_i' triples partitions by their predicate where their predicate is NOT rdf:type.
13:   pPartition ← mb'_i.filter(!_._2.contains("rdf-syntax-ns#type")).
14:       map(t ⇒ (t._2, (t._1, t._3))).partitionBy(number of different predicate in mb'_i).
15:       mapPartitions(_.map(t ⇒ (t._2._1, t._1, t._2._2))).persist

16:   // The oPartitions and pPartitions store on HDFS at fixed timestamp under o and p sub-directory paths respectively.
17:   oPartition.saveAsTextFile(outputPath + "o/" + fts + "/data/")
18:   pPartition.saveAsTextFile(outputPath + "p/" + fts + "/data/")

19:   // oIndexingSet is a HashTable which keeps the object of instance triple as key and their physical paths as value.
20:   oIndexingSet ∪= oPartition.mapPartitionsWithIndex((index,iterator) ⇒{
21:       iterator.map(t ⇒ (t._2, fts + "-" + index )) }).distinct
22:       .collect.groupBy(_._1).toSeq.map(t ⇒ (t._1, t._2.map(w ⇒ w._2).toSet)))

23:   // pIndexingSet is a HashTable which keeps the predicate of instance triple as key and their physical paths as value.
24:   pIndexingSet ∪= pPartition.mapPartitionsWithIndex((index,iterator) ⇒{
25:       iterator.map(t ⇒ (t._2, fts + "-" + index )) }).distinct
26:       .collect.groupBy(_._1).toSeq.map(t ⇒ (t._1, t._2.map(w ⇒ w._2).toSet)))
27:   // Return the indexing information
28:   return oIndexingSet & pIndexingSet
29: End
```

As mentioned in the previous section, indexed instance triples are classified into two disjoint categories: *object*- or *predicate*-based triples. Specifically, a triple is considered an object-based if its predicate is `rdf:type`. Triples of this kind are used as a premise to *rdfs9* (see Table 2.3). On the other hand, a triple

is considered to be predicate-based if its predicate is different from `rdf:type`. Triples of this kind are used as premise for rules *rdfs2*, *rdfs3* and *rdfs7* (see Table 2.3).

Labeling a new instance triple as object-based or predicate-based is not sufficient. To speed up the retrieval of the triples that are relevant for activating a given RDFS rule, object- and predicate-based triples are grouped in files based on their object and predicate. This allows for triples having a given predicate/object to be located in only one file inside the directory associated with a micro-batch. More specifically, Algorithm 3 details how the indexation operation is performed. It takes as input new instance triples that are asserted or inferred given the last micro-batch `mb′`. It filters the instances triples to create two RDDs. The first RDD is used for storing object-based triples (*line 9-11*). Since the predicate of object-based triples is `rdf:type`, we only store *subject* and *object* of object-based triples. The second RDD is used for predicate-based triples (*line 13-15*). Notice that the triples of the two RDDs are grouped based on their object and predicate, respectively, by utilizing RDD partitioning. The Spark method *partitionBy*() takes as an argument the number of partitions to be created. In the case of the RDD used for storing object-based triples, we use the number of different objects that appear in the triples as an argument. In the case of the RDD used for storing predicate-based triples, we use the number of different predicates that appear in the triples. It is worth mentioning here that we could have used the method *sortBy*() provided by Spark for RDDs instead of *partitionBy*(). However, *sortBy*() is computationally more expensive as it requires a local sort.

Besides grouping the RDDs containing the triples, the algorithm creates two auxiliary lightweight hash structures to keep track of the partitions that store triples with a given object (*line 20-22*) and predicate (*line 24-26*), respectively. Such memory-based hash structures act as indexes. They are lightweight memory-based structures that are utilized during the saturation to quickly identify partitions that contain a given object and predicate, respectively. Note that all the steps of the algorithm, with the exception of the first one (line 7), *oIndexingSet*, and *pIndexingSet* are processed in a parallel manner.

4.2 Soundness and Completeness

We deal now with the proof of soundness and completeness of our approach.

We need the following lemma, which is at the basis of soundness and com-

pleteness of our system as well as of WebPIE [75] and Cichlid [35], and reflects rule ordering expressed in Figure 2.4. To illustrate the lemma, assume we have $D = \{s\ \tau\ c_1\}$ while the schema includes four triples of the form $c_i \prec_{sc} c_{i+1}$, for $i = 1 \ldots 4$. Over D and S we can have the tree T1 corresponding to:

$$\{c_1 \prec_{sc} c_2 \mid c_2 \prec_{sc} c_3\} - rdfs11 \rightarrow c_1 \prec_{sc} c_3$$

A more complex tree is T2 defined in terms of T1:

$$\{s\ \tau\ c_1 \mid T1\} - rdfs9 \rightarrow s\ \tau\ c_3$$

Imagine now we have T3 defined as:

$$\{c_3 \prec_{sc} c_4 \mid c_4 \prec_{sc} c_5\} - rdfs11 \rightarrow c_3 \prec_{sc} c_5$$

We can go on by composing our derivation trees, obtaining T4:

$$\{T2 \mid T3\} - rdfs9 \rightarrow s\ \tau\ c_5$$

Note that the above tree T4 includes two applications of *rdfs*9. At the same time we can have the tree T5:

$$\{T1 \mid T3\} - rdfs11 \rightarrow c_1 \prec_{sc} c_5$$

enabling us to have the tree T4′ which is equivalent to T4, having only one application of rule 9, and consisting of:

$$\{s\ \tau\ c_1 \mid T5\} - rdfs9 \rightarrow s\ \tau\ c_3$$

As shown by this example, and as proved by the following lemma, repeated applications of instance rules $\{2, 3, 7, 9\}$ can be collapsed into only one, provided that this rule is then applied to an instance triple and to a schema triple in S^*, obtained by repeated applications of schema rules 5 and 11. This also proves that it is sound to first saturate the schema S and then applying instance rules $\{2, 3, 7, 9\}$ (each one at most once) over schema rules in S^*.

Lemma 4.2. *Given an RDF data set D of instance triples and a set S of RDFS triples, for any derivation tree T over D and S, deriving $t \in D^*_S$, there exists an equivalent T' deriving t, such that each of the instance rules $\{2, 3, 7, 9\}$ are used at most once, with rule 7 applied before either rule 2 or 3, which in turn is eventually applied before 9 in T'. Moreover, each of these four rules is applied to a S^* triple.*

Proof. Given the above lemma, we can now present the theorem stating the soundness of our approach.

Theorem 4.3. *Given a set of instance triples D and schema triples S, assume the two sets are partitioned in n micro-batches* $\mathtt{mb}_i = D_i \cup S_i$ *with* $i = 1 \ldots n$*, we have that there exists a derivation tree* $\{T1 \mid T2\} - rdfsX \rightarrow t$ *over D and S, with* $t \in D^*_S$*, if and only if there exists* $j \in \{1, \ldots, n\}$ *such t is derived by our system when* $\mathtt{mb}_j$ *is processed, after having processed micro-batches* $\mathtt{mb}_h$ *with* $h = 1 \ldots j - 1$.

Proof. To prove the above lemma, we examine the dependencies between the rules $\{2, 3, 5, 7, 9, 11\}$. A rule $\mathtt{r}$ depends on a rule $\mathtt{r}'$ where possibly r and r' are the same rule, if the activation of $\mathtt{r}'$ produces a triple that can be used as a premise for the activation of $\mathtt{r}$. This examination of rule dependencies reveals that:

- Rule 5 depends on itself only.
- Rule 11 depends on itself only..
- Rule 7 depends on rule 5: rule 7 uses as a premise triples of the form $p \prec_{\text{sp}} q$, which are produced by the activation of rule 5.
- Rules 2 and 3 depend on rule 7: both rules 2 and 3 uses as a premise triples of the form *spo*, which are given in prior and produced by rule 7.
- Rule 9 depends on rules 2, 3 and all given triples in prior with τ as a predicate: both rules produce triples of the form $p\ \tau\ x$, a premise for activating rule 9. It also depends on rule 11.

Figure 4.3 depicts the obtained rule dependency graph. With the exception of rule 5 and 11, the graph is acyclic, meaning that the saturation can be performed in a single pass. Furthermore, the dependency graph shows that in order for the saturation to be made in a single pass schema rules 5 and 11 needs to be first (transitively) applied to saturate the schema, followed by the instance rules. Rule 7 is the first instance rule to be executed, followed by the instance rules 2 and 3 (which can be applied simultaneously or in any order), before applying at the end rule 9. That said, we need to prove now that for an arbitrary T there is exist an equivalent T′ as described in the lemma. This follows from the fact that if (*) T contains more than one rule *rdfsX* with $X \in \{2, 3, 7, 9\}$, then it must be because of subsequent applications of rule 9 (resp. rule 7) each one applied to a schema triple

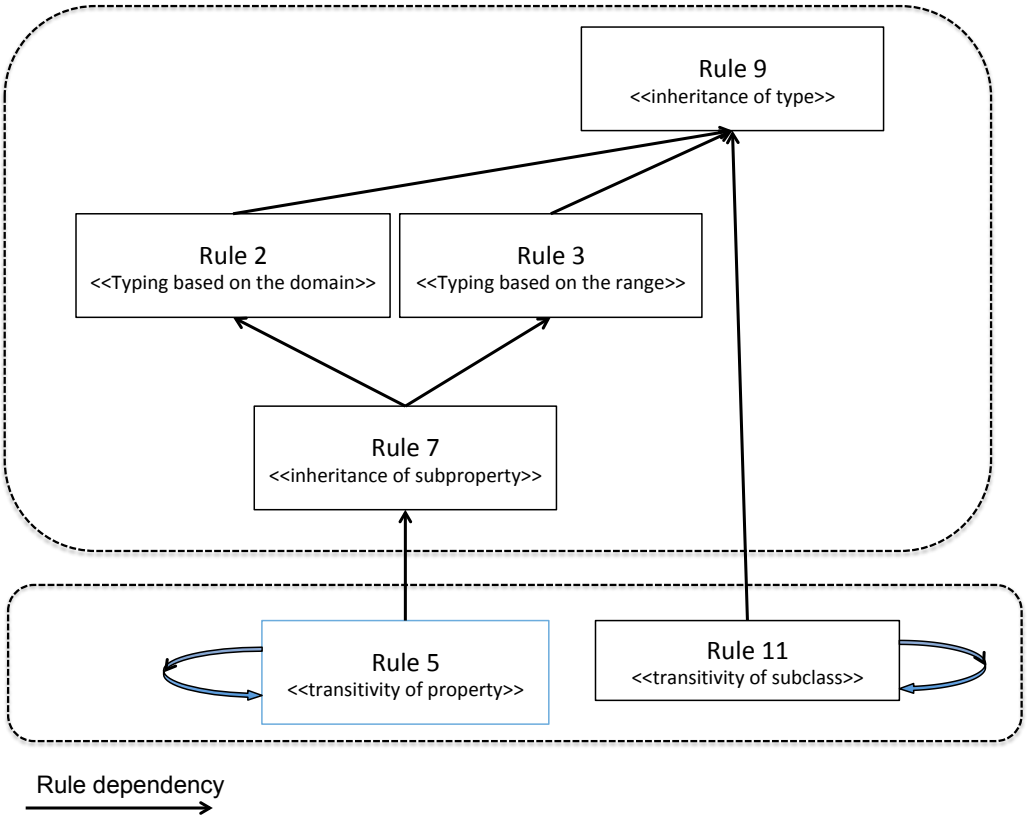


Figure 4.3: RDFS Ruleset Dependencies.

eventually derived by rule 11 (resp. rule 5), exactly as depicted by the example just before the lemma. As shown by the example, this chain of rule 9 (resp. rule 7) applications can be contracted so as to obtain a unique application of rule 9 (resp. rule 7) applied to as schema triple in S^*, obtained by subsequent applications of rule 11 (resp. rule 9). So in case (*) holds, the just described rewriting for chains of rule 9 (resp. rule 7) can be applied to T in order to obtain T′.

□

Given the above lemma, we can now present the theorem stating the soundness of our approach.

Theorem 4.4. *Given a set of instance triples D and schema triples S, assume the two sets are partitioned in n micro-batches $\mathtt{mb}_i = D_i \cup S_i$ with $i = 1 \ldots n$, we have that there exists a derivation tree $\{T1 \mid T2\} - rdfsX \rightarrow t$ over D and S, with $t \in D_S^*$, if and only if there exists $j \in \{1, \ldots, n\}$ such t is derived by our system when $\mathtt{mb}_j$ is*

processed, after having processed micro-batches $\mathtt{mb}_h$ *with* $h = 1 \ldots j-1$.

Proof. The 'if' direction (soundness) is the easiest direction. We prove this case by induction on j. In case one triple t is derived by our system when processing the micro-batch $\mathtt{mb}_1$, then we can see that in Algorithm 2, this triple is obtained by a derivation tree calculated by Saturate(), and including at the leaves instance triple in D_1 and schema triple in ${S_1}^*$. As $D_1 \subseteq D$ and ${S_1}^* \subseteq S^*$, we have that his derivation tree can derive t also from D and for S. Assume now t is derived by our system when processing the micro-batch $\mathtt{mb}_j$ with $j > 1$. Triple t is derived by a derivation tree T possibly using triples t′ derived in $\mathtt{mb}_h$ with $h < j$, as well as triples in D_j and $(\bigcup_1^j S_i)^*$. By induction we have that for each t' derived at step $h < j$ there exists a derivation tree T′ over D and S deriving t'. So to conclude it is sufficient to observe that if in T we replace leaves corresponding to triples t' with the corespondent T' then we obtain the desired derivation tree for t.

Let's now consider the 'only-if' direction (completeness). We proceed by a double induction, first on n, the number of micro-batches, and then on the size of the derivation tree T deriving t. Assume $n = 1$, this means that we only process one micro-batch. By Lemma 4.2 we have that there exists an equivalent T' for t, satisfying the properties stated in the lemma, and hence that can be produced by our algorithm, as we first saturate the schema and then apply instance rules in sequence 7-2-9 or 7-3-9, as in T'.

Assume now $n > 1$. We proceed by induction on the tree derivation $T = \{T1 \mid T2\} - rdfsX \rightarrow t$. The base case is that both T1 and T2 are simple triples in D and S respectively. In this case let j be the minimal index ensuring that both triples have been met in processed micro-batches $\mathtt{mb}_h$, with $h \leq j$. This j exists by hypothesis, and we have that either t1 or t2 is in $\mathtt{mb}_j$. Assume it is t1, a schema triple and that t2 has been met in $\mathtt{mb}_s$ with $s < j$. Then by means of our index we recover t2 (line 14) and saturation for the step j in line 21, builds T to derive the triple t.

Assume now that both T1 and T2 do not consist of a simple triple (the case only one is a triple is similar). By Lemma 4.2, we have that there exists an equivalent $T' = \{T1' \mid T2'\} - rdfsY \rightarrow t$ such that instance rules are use a most once (in the order of Figure 2.4), where each rule uses a schema triple in S^*. This means that, w.l.o.g, T2′ is a schema triple t2′ in S^*. By hypothesis ($S = \bigcup_1^n S_i$) we have that there exists $\mathtt{mb}_h$ such that t2 is obtained by schema saturation (which is globally kept in memory) and that there exists $\mathtt{mb}_s$ in which $t1$ is derived and indexed by our algorithm. Now consider $j = max(s, h)$. At step j our algorithm disposes of both $t1$ (indexed) and $t2$ (in the RAM) and can hence produce $\{t1 \mid t2\} - rdfsY \rightarrow t$.

The remaining cases are similar. □

4.3 Evaluation

The saturation method we have just presented lends itself, at least in principle, to outperform state of the art techniques, notably Cichlid, when dealing with streams of RDF data. This is particularly the case when the information about the RDF schema is also obtained in a stream-based fashion. An empirical evaluation is, however, still needed to be able to answer the following question: *Does our method actually outperform in practice the Cichlid solution* for saturating streams of *RDF*? *And if so, to what extent?* To answer this question, we conducted an experimental analysis that we reported on in this section.

4.3.1 Datasets

Pre-processing data in stream

We make in our experiments the assumption that the RDF data is received along with schema triples. So, dataset arrives in micro-batches in a streaming fashion, while their schema heavenly divided between the micro-batches. Specifically, we used for our experiments three RDF datasets that are widely used in the semantic web community: DBpedia [17], LUBM [36], and dblp[2]. These datasets are not stream-based datasets, and therefore we had to partition them into micro-batches to simulate a setting where the data is received in a streamed manner.

To this end, we created the following stream-based datasets:

1. **DBpedia:** by using DBpedia, we created three stream-based datasets DBpedia-100M, DBpedia-200M, and DBpedia-300M. They contain 100, 200, and 300 million instance triples respectively. The datasets are broken into series of 58, 112, and 162 micro-batches respectively and each micro-batch is 500 MB in size. The existing DBpedia schema has been distributed equally between the micro-batches. Therfore, each micro-batch, also, has a range of [104-128], [58-63], and [40-42] schema triples respectively.

[2]Computer science bibliography (https://dblp.uni-trier.de/faq/What+is+dblp.html)

2. **LUBM:** LUBM[3] [36] is a generator of synthetic RDF datasets. We used it to create three stream-based datasets, LUBM-35M, LUBM-69M, and LUBM-165M, which contained 35, 69, and 165 million triples, respectively. The datasets are partitioned into a series of 11, 23, and 49 micro-batches, respectively, and each micro-batch is 500 MB in size. The existing LUBM schema has been distributed equally between micro-batches. The LUBM in quantity has a tiny schema dataset (i.e., contains 84 schema triples), then each micro-batch also has a range of [7-8], [3-4], and [1-2] schema triples, respectively.

3. **DBLP:** DBLP is a computer science dataset. We created a stream-based DBLP, composed a series of 60 micro-batches containing in total 195 million triples. Each micro-batch is 500 MB in size and contains 6% of schema triple that means only one schema triple per micro-batch.

For each of the above datasets, we ran our saturation algorithm incrementally for each of the succeeding micro-batches. For comparison purposes, for each of the above datasets, we run the Cichlid algorithm on the previous micro-batches if the new schema arrives. To reduce the extra process in successive saturation on Cichlid, we consider `NST` (The `Newest Schema Triples`) for re-saturating the previous micro-batches. We then consider the implicitly inferred triples from the previous processes plus instance triples of the given micro-batch and then saturate them with so far given schema triples. Given that Cichlid is not incremental, for each micro-batch, we had to consider the previous micro-batches.

4.3.2 Experiment Setup

We conducted our experiment on a local cluster[4] with two configurations: 2 nodes and 4 nodes. One node was, also, reserved to act as the driver node and the remaining nodes (2 and 4 respectively) as worker nodes. Each node has a Xeon Octet 2.4 GHz processor, 48 GB memory and 6 TB Hadoop file system. The nodes are connected with 1 Gb/s Ethernet. All the nodes run on Debian 9.3 operating system. The version of the Spark we used is 2.1.0 and Hadoop v2.7.0 with Java v1.8.

[3] `http://swat.cse.lehigh.edu/projects/lubm` - A synthetic benchmark describing the university domain

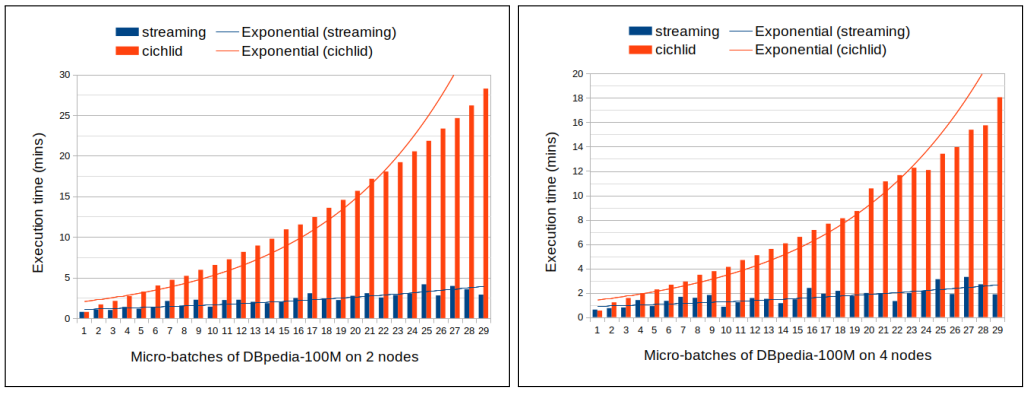
[4] It's located at Université Paris-Dauphine `https://dauphine.psl.eu/`

For each dataset we ran our experiments 5 times, and reported the average running time. We ran our saturation algorithm incrementally for each of the succeeding micro-batches. For comparison purposes, for the similar datasets, we run the Cichlid algorithm on each of the micro-batches as well. Given that Cichlid is not incremental, for each micro-batch, we had to consider the previous micro-batches as well as the micro-batch in question.

4.3.3 Results

The x-axis of histograms represents the micro-batches that composed the dataset. The y-axis reports the execution time required for its saturation in minutes. For each of the succeeding micro-batches, in both scenarios, the y-axis reports the time required for saturating the dataset composed of the current micro-batch, plus previous micro-batches.

DBpedia Datasets



(a) 2 nodes (b) 4 nodes

Figure 4.4: Micro-Batches of DBpedia 100 Million Triples

Figures 4.4 shows the results obtained when saturating the DBpedia-100M dataset in successive micro-batches on 2 and 4 nodes. Figures 4.4a and 4.4b shows that the time required by Cichlid for saturating the streaming data increases substantially by receiving continuous micro-batches.

Figure 4.4a illustrates that saturation on 2 nodes took 349 minutes to process with Cichlid for the entire given dataset, which is 66 minutes by our incremental

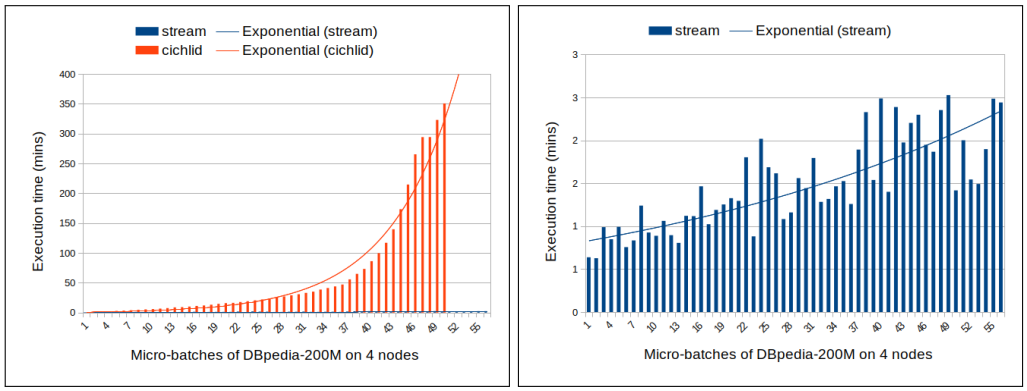
algorithm. That is 5.3x the time required to saturate the same dataset with our method. On average, our method takes 2.3 minutes, whereas Cichlid needs 12 minutes to apply a complete saturation process, by giving a new micro-batch, while that contains a new schema.

Specifically, Figure 4.4b shown the saturation process on 4 nodes that takes more than 219 minutes by Cichlid to saturate the entire process. That is 4.47 times of the time required to saturate the same dataset with our proposed incremental saturation, which is 49 minutes. On average, our method takes 1.7 minutes whereas Cichlid needs 7.5 minutes to apply a complete saturation process by giving new schema per micro-batch. Concerning the exponential trend lines, we observed that growing rate of Cichlid is exponential while that is growing linearly for streaming processes. The experiments show that our method is scalable by increasing the nodes. In 2 nodes (Fig. 4.4a), in some micro-batch, we hit 4.5 minutes while the same micro-batches saturated in 3 minutes with 4 nodes. Cichlid outperforms our method in first micro-batch on Figure 4.4a.

The main reason is that Spark Streaming processes only new data available at the starting point of every interval time. On the other hand, the data that existed before starting the first interval is not considered. Therefore, the data inserted in the middle of the first interval will process in the second interval along with the second micro-batch. Another reason is that the initialization of Spark Streaming takes, a bit longer time than Spark has batch processing.

Cichlid takes almost the same processing time for the early micro-batches. But since they need to apply batch saturation processing on previous micro-batches, as time goes on, the volume of data gets bigger. Therefore, the saturation time increases rapidly. On the other hand, our incremental algorithm fetches related instance triples, regarding the given new schema, among the past data to complete the informative content of the dataset.

Figure 4.5 shows the results obtained when saturating 200 million triples of DBpedia on 4 nodes. The left Figure 4.5a, alike we observed for 100M triples, shows that the execution time required by Cichlid for saturating a successively received data increases substantially as the time goes on. This exponential ratio in saturating causes Cichlid crashed in 51^{th} micro-batch out of 56. The crash is due to the volume of previous data that reloaded to re-saturate with a newly received schema. This figure shows that it takes 3165 minutes to process the first fifty micro-batches from the dataset, whereas it takes 84 minutes with our incremental method for the entire dataset. The Cichlid partial saturation is still 37.6 times the time required to saturate the same dataset compared to our method. On average,



(a) 4 nodes - Both

(b) 4 nodes - Streaming

Figure 4.5: Micro-Batches of DBpedia 200 Million Triples

our method takes 1.5 minutes, whereas, Cichlid requires 63 minutes to apply a complete saturation process by giving a new micro-batch when that contains new schema triples.

Consider that the growing ratio of Cichlid is exponential, whereby our processing time is not visible at the same chart. We then represent it once again separately in Figure 4.5b. We observed that our algorithm processing time starts from 1 minute and in the worst-case reaches 3 minutes to saturate the current, past, and already saturated ones.

Since Cichlid is very time-consuming and also does not tolerate the DBpedia datasets bigger than 200 million triples, we did not run Cichlid for DBpedia-300M.

Figures 4.6 shows the results obtained when saturating 300M of DBpedia in our algorithm in 2 and 4 nodes. The figure shows that the time required for saturating on 2 nodes (blue line) takes a longer time for the same micro-batch with 4 nodes. The yellow and green lines showed required times to fetch related triples based on given schema triples through current micro-batch. The exponential rate of growth on 2 and 4 nodes shows the scalability of our proposed system. In general, the saturation process on 2 nodes takes 167 minutes, while this time is 114 minutes with 4 nodes for the entire dataset. On average, every micro-batch takes 2 minutes to saturate on 2 nodes cluster, while this time for the same micro-batch on 4 nodes on the same cluster is 1.4 minutes.

The good performance of our algorithm is due to its incremental nature, but also to its underlying indexing mechanism. To demonstrate this, Figure 4.7 illustrates for DBpedia, and for each micro-batch, the number of triples that are fetched using

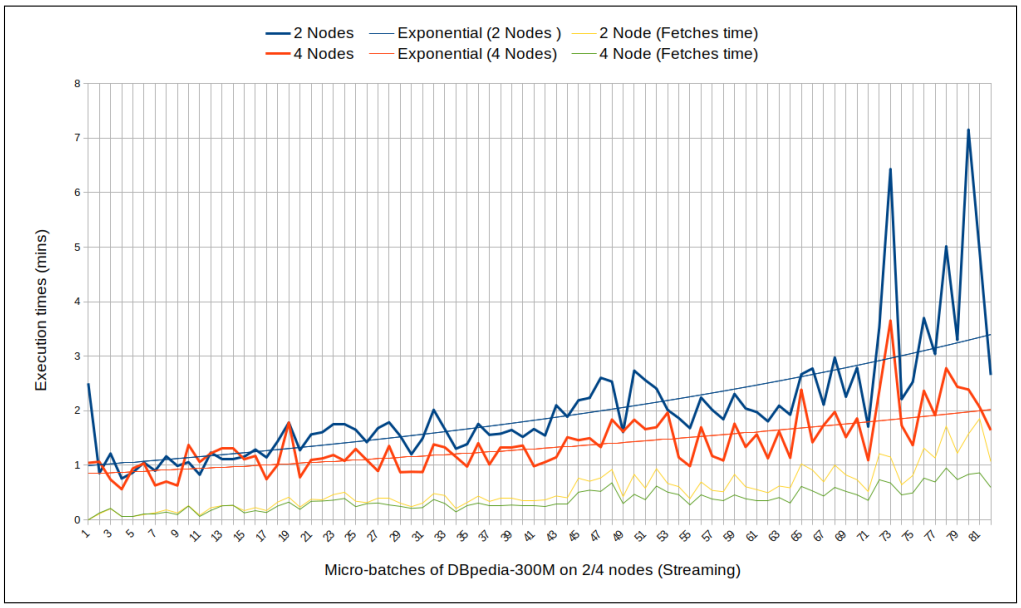


Figure 4.6: DBpedia 300 Million Triples on 2 and 4 nodes - Our approach

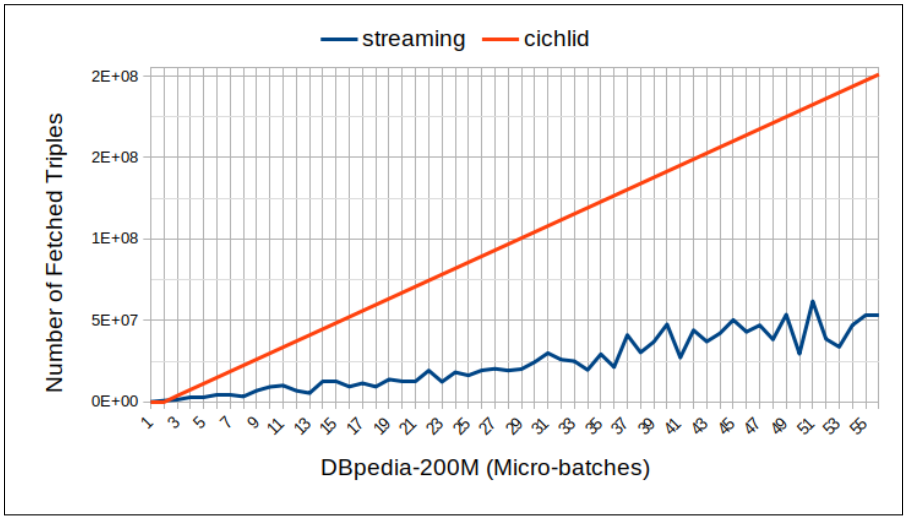


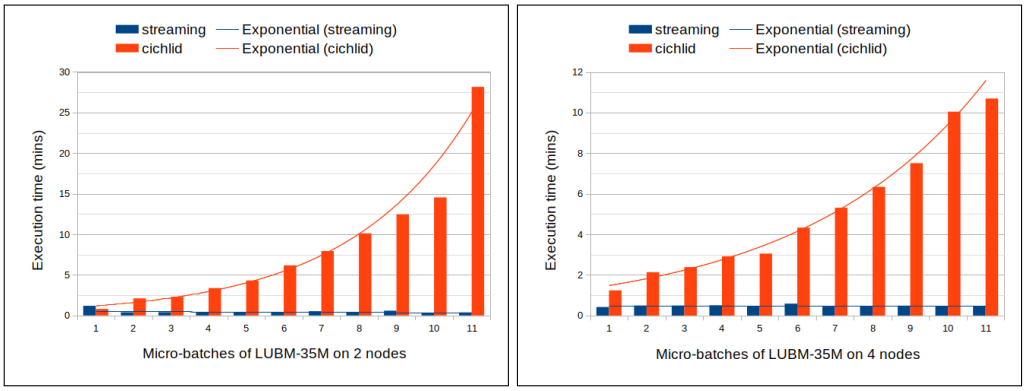
Figure 4.7: Fetching triples per micro-batch. DBpedia 200 Million Triples

the index as well as the total number of triples that the saturation algorithm would have to examine in the absence of the indexing structure (that requires whole amount of triples to load). It shows that the number of triples fetched by the index (blue line) is small compared to the total number of triples that compose the dataset and fetch by Cichlid (red line).

LUBM Dataset

Figure 4.8 shows the results obtained by saturating the LUBM-35M dataset in streaming and successive micro-batches on a 2 and 4 nodes cluster. Both figures showed that the time required by Cichlid increases substantially by receiving continuous micro-batches.

Figure 4.8a shows that saturation on 2 nodes takes 92 minutes for Cichlid to process the entire dataset, while it takes 5.3 minutes by our algorithm. That is 17.7x the time required to saturate compare to us. On average, our method takes less than 30 seconds, whereas Cichlid needs 8.4 minutes to apply a complete saturation process given new schema triples. Expressly, Figure 4.8b showed that



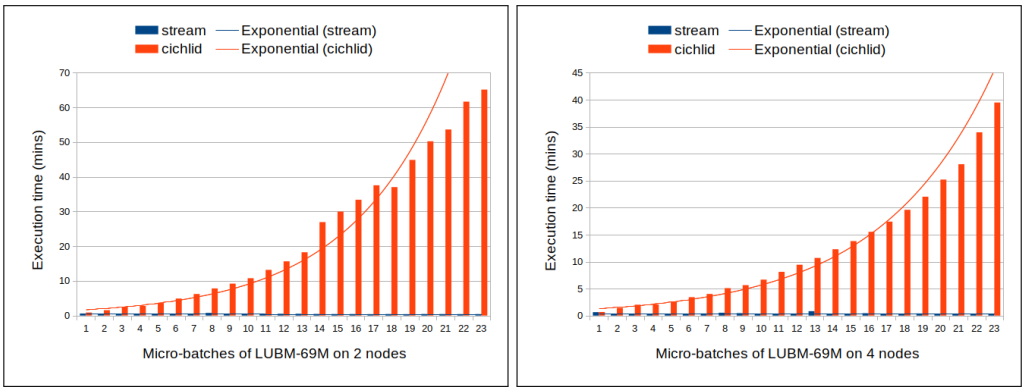
(a) 2 nodes (b) 4 nodes

Figure 4.8: Micro-Batches of LUBM 35 Million Triples

on 4 nodes takes 56 minutes with Cichlid versus 10.2x of the time required to saturate the same dataset with our proposed incremental saturation, which is again 5.2 minutes. Surprisingly, we see that our method does not scale on LUBM-35M. Since the saturation process on the LUBM dataset is not high enough to require more processors, thus adding more nodes does not increase the performance.

Concerning the trend lines, we observed that the growth rate of Cichlid is exponential with a steeper slope compare to the first 11^{th} micro-batches of DBpedia-100M and DBpedia-200M. Due to the LUBM dataset synthetically designed to provide a high computation to testing the system, then the ratio of the slope is understandable.

Figures 4.9 shows the saturation line results of the LUBM dataset of 69 million triples on streaming and successive micro-batches data on 2 and 4 nodes,



(a) 2 nodes (b) 4 nodes

Figure 4.9: Micro-Batches of LUBM 69 Million Triples

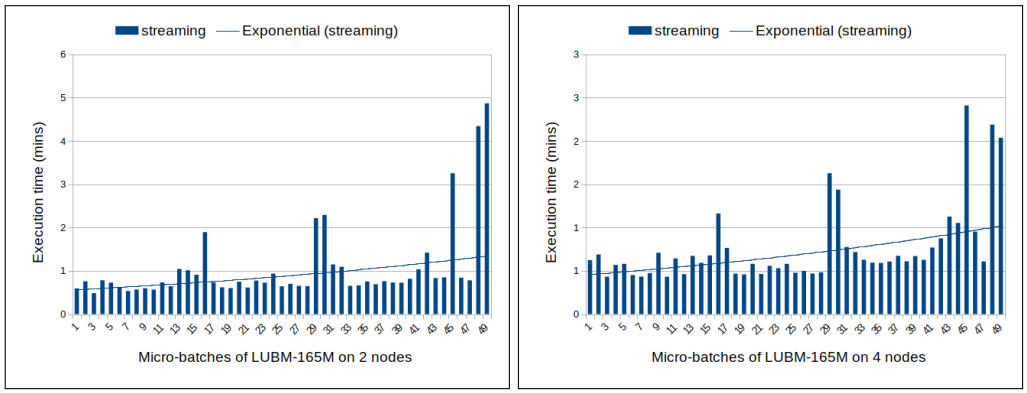
that performed with our method and Cichlid respectively. It shows the same rate as the LUBM-35M of exponential growth. Figure 4.9a details the saturation on 2 nodes that take 538 minutes with Cichlid while it is 49 times the time required to saturate the same dataset in our proposed incremental saturation, which is 11 minutes. On average, our method takes 28 seconds, whereas Cichlid needs 23.4 minutes to apply a complete saturation process on given new schema triples. Figure 4.9b expressed the saturation on 4 nodes. It needed 289 minutes to saturate the entire dataset by Cichlid, whereas 9 minutes with our algorithm. That is 32 times the time required to saturate in comparison with our algorithm. On average, our method takes 24 seconds, whereas Cichlid needs 12.6 minutes to do a complete saturation by a given new schema.

Since processing the LUBM-35M and LUBM-69M were very time-consuming on Cichlid, so we decided to abandon the LUBM-165M.

Figure 4.10 declared the saturation process on 2 and 4 nodes for the LUBM dataset with 165 million triples by applying our incremental method. Figure 4.10a details the saturation on 2 nodes that takes 52 minutes while it is 38 minutes for the same dataset with 4 nodes. It shows we have 27% improvement by doubling the nodes size (Figure 4.10b).

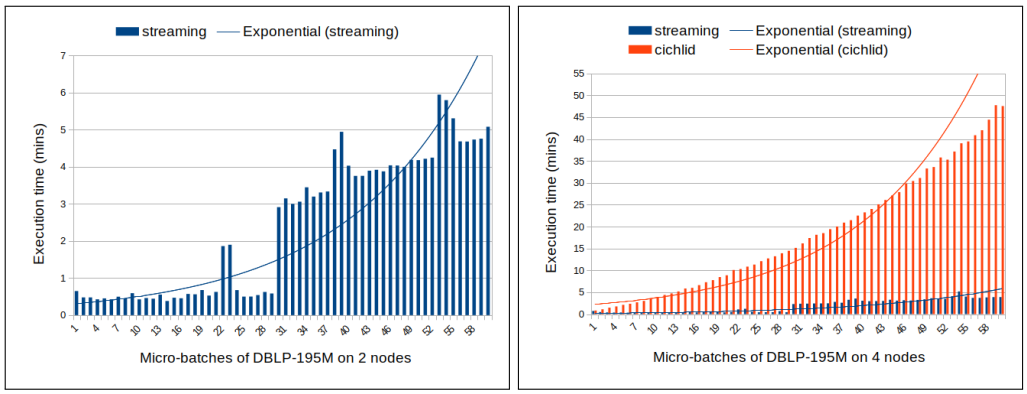
DBLP dataset

Figures 4.11 shows the results of the saturation process on a DBLP dataset containing 195 million triples on 2 and 4 nodes. Figure 4.11b details the saturation results



(a) 2 nodes (b) 4 nodes

Figure 4.10: Micro-Batches of LUBM 165 Million Triples



(a) 2 nodes (b) 4 nodes

Figure 4.11: Micro-Batches of DBLP 195 Million Triples

on 4 nodes, which that takes 1107 minutes with Cichlid, while it is 9.6 times the time required to saturate the same dataset in our proposed incremental saturation, which is 115 minutes. On average, our method takes 1.9 minutes, whereas Cichlid needs 18.4 minutes to apply a complete saturation process on given new schema triples. Execution time on 4 nodes with Cichlid grows exponential. Since saturation certainly needs more time with 2 nodes, therefore we didn't run Cichlid with 2 nodes. Figure 4.11a shows our incremental saturation on 2 nodes on DBLP-195M dataset. It obtained 146 minutes to saturate the entire dataset. On average, our method takes 2.4 minutes to do a complete saturation by a given new schema on 2

nodes. Our method scales 27% with 4 nodes for the same data.

The saturation time increases for $\texttt{mb}_{22}$ and $\texttt{mb}_{23}$ in Figure 4.11a. This happens since the indexing fetched 2.6 and 2.8 million triples for $\texttt{mb}_{22}$ and $\texttt{mb}_{23}$, respectively. This number for mb20 and mb21 is 17k and 18k. Another jump happens for $\texttt{mb}_{38}$, $\texttt{mb}_{39}$, and $\texttt{mb}_{40}$. The indexing fetches 10.4, 10.6, and 10.97 million triples, respectively.

Microbatch size So far, we have considered that the size of the micro-batch is specified apriori. Ultimately, the size of the micro-batch depends, at least partly, on the time interval, the resource we have (cluster configuration), and the amount of processing required that varies from dataset to dataset. To investigate this point, we considered a DBpedia instance of 25.4GB and run 7 different incremental saturations. In saturation i, for $i = 1 \ldots 7$, the size of the micro-batch is $i * 100$MB, resulting in n_i micro-batches, in which the whole set of schema triples have been evenly distributed over the n_i micro-batches. We used for this experiment a cluster with 4 nodes, 11 executors, 4 cores per executor, and 5 GB memory per executor.

Figure 4.12 illustrates the average time required for performing the saturation given a micro-batch in seconds (blue line), the average time required for the index management in seconds (yellow line), and the total execution time to saturate the entire dataset in minutes (red line). Regarding the saturation, the figure shows that micro-batches with different sizes require different times for processing. For example, the time required for processing a $100MB$ micro-batch is smaller compared to the time required for processing micro-batches with larger sizes, but the total execution time in this size is too high compared to others. The increase is not steady. In particular, we observe that micro-batches with $400MB$ and $500MB$ require the same processing time per micro-batch and almost in total execution time. This means the cluster could process a bigger chunk of data within the given time-interval. We can also conclude that the cluster was idle for some time when processing $400MB$ micro-batches.

Regarding the index management (yellow line), it shows that it is comparatively small with respect to the saturation time, and it costs in the worse case less than 6 seconds. Moreover, as with the saturation time, micro-batch size is not the only factor. For example, the micro-batch with a size of $600MB$ required more time for maintaining the index because the number of inferred tuples was higher compared with other micro-batches, including the one with a size of $700MB$. Concerning global execution time (for all micro-batches), experiments showed that when the number of micro-batches decreases (or in other word, size of them increases), this

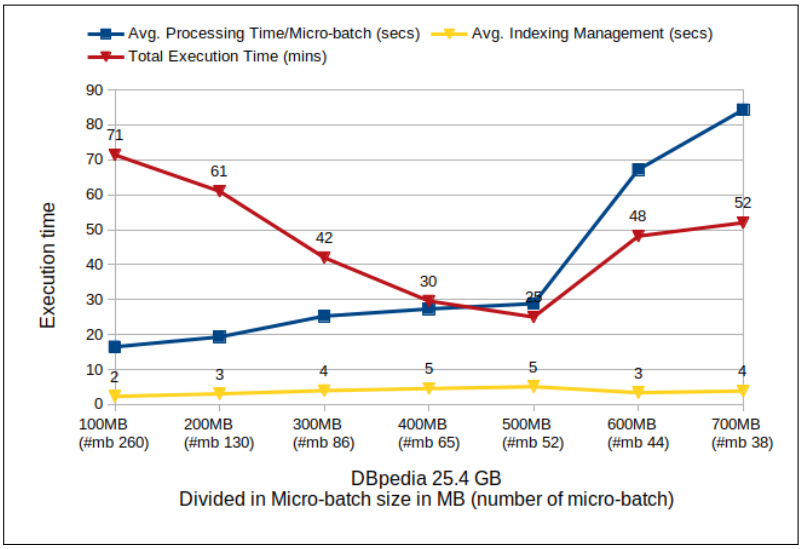


Figure 4.12: Average processing time and indexing management / micro-batch

time can decrease in some cases (this happens in particular for $i \in \{4, 5\}$, Table 4.4).

Table 4.4: **Average time per micro-batch (*mb*)**.

TE: Total Execution time of whole process

PT: Average of Processing Time per micro-batch

Indexing: Average time to fetch the triples by relying on the indexing information

FT: Number of Fetched Triples via indexing information

Size	# of mb	TE(mins)	Indexing(sec)	PT(secs)	# FT (million)
100MB	260	71	2	16	174 m
200MB	130	61	3	19	155 m
300MB	86	42	3	25	174 m
400MB	65	30	5	27	167 m
500MB	52	25	5	29	130 m
600MB	43	48	3	67	22 m
700MB	37	52	4	84	23 m

So far, we proposed, implemented, and observed the performance of our algorithm. But still, an important question remains and that is:

"How sustainable is the proposed incremental algorithm?"

To answer this question, we need to discuss the file system that we used in this thesis; that is HDFS[5]. HDFS has two main components in its architecture: *DataNode* and *NameNode*. DataNode is responsible to keep data in blocks. While NameNode contains the metadata of those files which are present in HDFS and maintains states of all DataNodes. This metadata is present in a serialized form inside a single file, and this file is kept both on-disk and in-memory.

As we already discussed, the outputs of our method are written in multiple files on HDFS by considering the numbers of predicates or objects. This technique creates many files in DataNode while their file names are maintained in NameNode. As saturation goes on the number of files increases too. So, NameNode goes down by reaching its threshold. When NameNode goes down, the file system goes offline, and the saturation platform will disconnect into HDFS.

In default, the size of NameNode is 1 MB and located in java heap. To tackle this limitation, we increased the size of NameNode into 4 MB. From that moment, we did not face any failure in the purpose of the thesis experiments.

4.4 Conclusion

In this chapter, we have presented a stream-based scalable technique for parallel RDFS forward inference. We showed that our proposed approach outperforms state-of-the-art solutions for saturating RDF, namely Cichlid on well-known RDF datasets. In order to innovate and improve the performance, we introduced some key optimizations to handle the given and past received RDF/RDFS triples. These optimizations are: *i)* Rule pruning on given schema triples, *ii)* Reduce the execution time, and *iii)* Reduce the duplicate elimination costs in a stream.

Both in terms of re-saturation and durability, our technique outperforms published approaches (Specifically [35]) by a significant margin, when saturating streams RDF data incrementally using the big data platform, like Apache Spark.

[5]The Hadoop Distributed File System (HDFS) is a distributed file system designed to run on commodity hardware. (http://hadoop.apache.org/docs/r1.2.1/hdfs_design.html)

5 OWL-Horst Saturation in Streaming

Contents

In recent years, `ter OWL-Horst` [72] has gained consistent attention from both research and industrial communities, as it represents a good balance between expressivity and computational tractability. This section is devoted to RDF streaming saturation in the presence of OWL-Horst ontology [73]. The rules are reported in Table 5.1, and, as it can be seen, it is much more expressive than RDFS.

In the previous chapter, 4, we have described how to perform efficient reasoning under the RDFS semantics while data and schema triples receive in successive intervals of time. In this chapter, we will explain an extension of the indexing technique on the ruleset of `Ter OWL-Horst` fragment [73], which is a part of the OWL vocabulary that weakens *D*-entailment [60] and `OWL Full` [9]. However, the techniques we have developed for the RDFS saturation process remain effectual in the context of OWL-Horst rules; but the transposition is not direct. Hence, we had to take particular care regarding the integration of the RDFS and OWL-Horst ruleset in the presence of streaming triples. However, for every

Table 5.1: OWL-Horst rule set. Schemas are indicated by *italic* font.

No	Condition	Consequence
1	*p* τ *owl:FunctionalProperty* u p v , u p w	v owl:sameAs w
2	*p* τ *owl:InverseFunctionalProperty* v p u , w p u	v owl:sameAs w
3	*p* τ *owl:SymmetricProperty* v p u	u p v
4	*p* τ *owl:TransitiveProperty* u p w , w p v	u p v
5a	u p v	u owl:sameAs u
5b	u p v	v owl:sameAs v
6	v owl:sameAs w	w owl:sameAs v
7	v owl:sameAs w, w owl:sameAs u	v owl:sameAs u
8a	*p owl:inverseOf q* , v p w	w q v
8b	*p owl:inverseOf q* , v q w	w p v
9	*v* τ *owl:Class* , v owl:sameAs w	$v \prec_{sc} w$
10	*p* τ *owl:Property* , p owl:sameAs q	$p \prec_{sp} q$
11	u p v, u owl:sameAs x , v owl:sameAs y	x p y
12a	*v owl:equivalentClass w*	$v \prec_{sc} w$
12b	*v owl:equivalentClass w*	$w \prec_{sc} v$
12c	$v \prec_{sc} w$, $w \prec_{sc} v$	*v owl:equivalentClass w*
13a	*v owl:equivalentProperty w*	$v \prec_{sp} w$
13b	*v owl:equivalentProperty w*	$w \prec_{sp} v$
13c	$v \prec_{sp} w$, $w \prec_{sp} v$	*v owl:equivalentProperty w*
14a	*v owl:hasValue w*, *v owl:onProperty p*, u p v	u τ v
14b	*v owl:hasValue w*, *v owl:onProperty p* u τ v	u p v
15	*v owl:someValuesFrom w*, *v owl:onProperty p* u p x , x τ w	u τ v
16	*v owl:allValuesFrom u*, *v owl:onProperty p* w τ v, w p x	x τ u

single OWL-Horst rule, the extension of the indexing technique for the RDFS approach is almost direct, validating the effectiveness of our previously introduced indexing technique. The reasons that we were choosing the OWL-Horst ruleset are *i*) it is an existing standard for scalable OWL reasoning when its implemented by industrial-strength triples stores such as GraphDB[1]; *ii*) also, compared to the RDFS rule set, it has a more complex set of rules to express; *iii*) it keeps, computationally, a balance between the OWL full and the limited articulation of RDFS; *iv*), and finally, in the streaming data, it brings a new challenge to maintain the complete set of saturated data up-to-date.

Despite the normalization of OWL-Horst rules rather than OWL full, they still have very complicated rules to provide an extensive reasoning process in a stream. For instance, dissimilar from RDFS rules, rules 12c, 13c, 14a, 14b, 15, and 16 from OWL-Horst require more than one schema triple, while some of them require more than one instance triple to infer an implicit triple (i.e., 1, 2, 4, 15, 16, etc.). Those characteristics of OWL-Horst rules need several *join* operation between the given and saturated data.

Same as RDFS, to dismiss some straightforward rules from our process, we omit rules with one antecedent, i.e., rules $5_{a,b}$, while those rules can be parallelized efficiently [75] at a query time.

5.1 Challenges on OWL-Horst Reasoning in Stream

In the OWL-Horst rule stream reasoning, we face two main challenges:

i) Unlike RDFS, most of the OWL-Horst rules require more than one schema and instance triple to star the saturation process. So, they need data communication between nodes to the corresponding instance matches.

ii) The previous challenge (i) will be more serious when we assume that a part of the instance triple has already arrived and stored on the disk.

On the other hand, some rules (i.e., Rule 15, 16, 4) require iteration until reaching a fixpoint. Fortunately, we still have some rule dependencies. It implies that the correctness of our algorithm follows naturally, as the fixpoint semantics is natural for RDF saturation. Figure 5.1 illustrates the overall steps of our complete RDF streaming saturation to deal with OWL-Horst schema rules in a streaming fashion. It is worth observing that the particular rule ordering we adopt is that of

[1] https://www.w3.org/2001/sw/wiki/Owlim

Cichlid [35]. Notice that there is no *join* operation required between schema and instance triples in the RDFS rule set, whereas in some OWL-Horst ruleset needs multiple join operations in schema- and instance-level (i.e., Rule 15 and 16) to infer an implicit triple.

As already observed in [35], an important difference wrt RDFS saturation is that for OWL-Horst it is not possible to identify an ordering in rule application which is as fine-grained as that for RDFS (Figure 4.3). That said, a careful analysis that distinguishes the setting where new schema triples are considered from the setting where new instance triples are considered, allows us to establish some partial ordering among the rules.

Specifically, given some new schema triples, which come with a new micro-batch, the examination of rules dependencies allows us to identify three groups of rules that may be triggered: G_1^{sch}, G_2^{sch} and G_3^{sch}, each of which is composed of the following rules:

- $G_1^{sch} = \{\texttt{OWL} - \texttt{Horst}(9, 10, 12_{(a,b)}, 13_{(a,b)}) + \texttt{RDFS}(5, 11)\}$
- $G_2^{sch} = \{\texttt{RDFS}(7, 2, 3, 9)\}$
- $G_3^{sch} = \{\texttt{OWL} - \texttt{Horst}(14_{(a,b)}, 3, 8_{(a,b)}, 15, 16, 4, 1, 2)\}$

The first group contains OWL-Horst rules together with the two RDFS rules that produce schema triples, viz RDFS 5 and 11. The OWL-Horst in the group can be applied in any order once however, RDFS rules 5 and 11 need to be applied multiple times until reaching a fix-point (note that RDFS 5 and 11 operate centralized). The second group G_2^{sch} contains RDFS rules that use as premise triples that are produced by the rules in G_1^{sch}, but also, the related instance triples for this group fetch from DS′ by utilizing the indexing information regarding the G_1^{sch}. The third group G_3^{sch}, on the other hand, is independent of the other two group. The above analysis suggests the following order of application of rules given newly acquired schema triples: the rules in group G_1^{sch} need to be applied before applying those in group G_2^{sch}, whereas the rules in the third group G_3^{sch} can applied in parallel to those in G_1^{sch} and G_2^{sch}.

The analysis of rule dependencies considering newly acquired or inferred instance triples is less conclusive since we cannot escape the iterative application of rules. That said, we identified the following groups of rules, which are exploited in the saturation algorithm shown in Figure 5.1:

- $G_1^{ins} = \{\texttt{RDFS}(7, 2, 3, 9)\}$

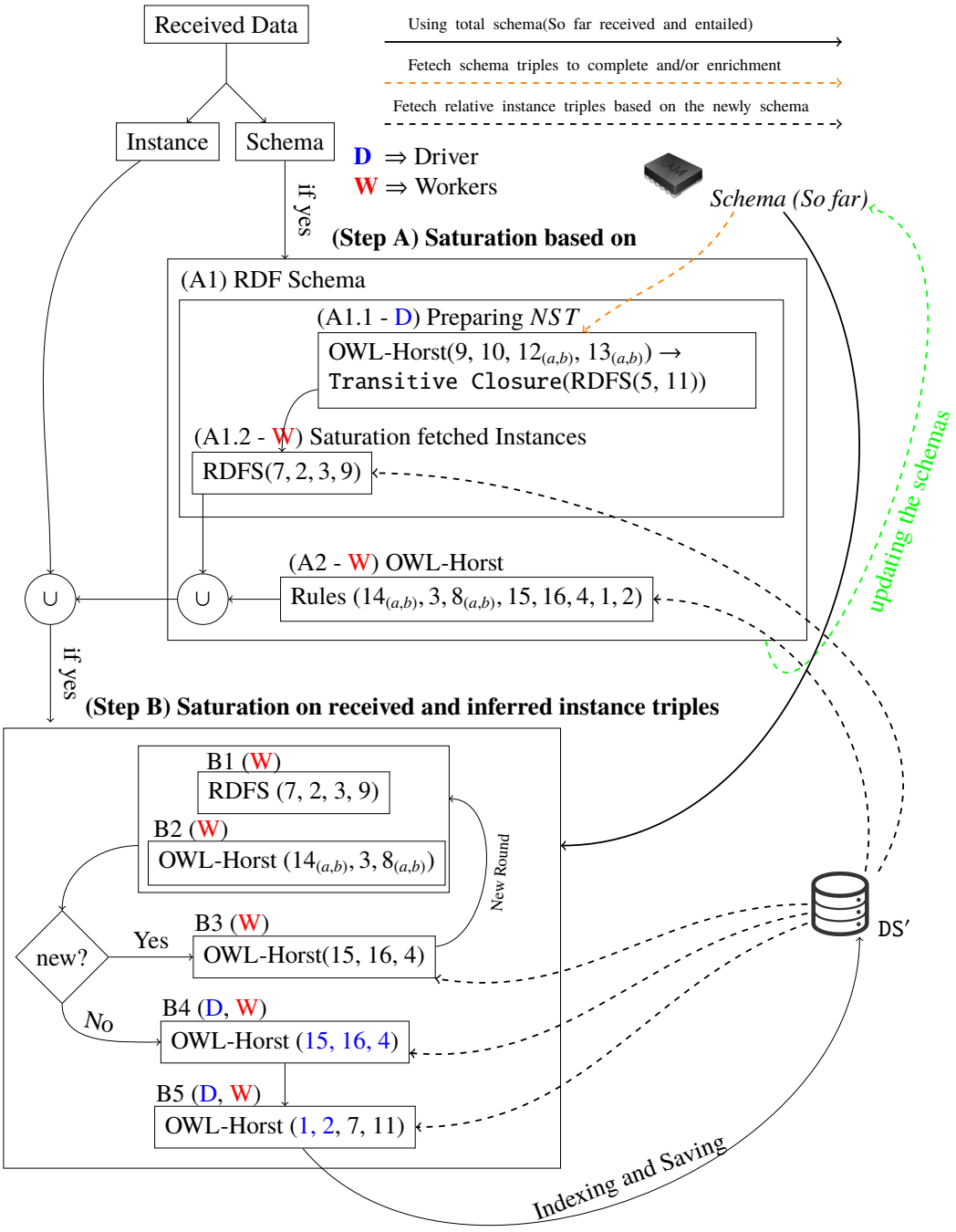


Figure 5.1: The Global Overview of Saturation Process on OWL-Horst Rules.

- $G_2^{ins} = \{OWL - Horst(14_{(a,b)}, 3, 8_{(a,b)})\}$
- $G_3^{ins} = \{OWL - Horst(15, 16, 4)\}$
- $G_4^{ins} = \{OWL - Horst(15, 16, 4)\}$
- $G_5^{ins} = \{OWL - Horst(1, 2, 7, 11)\}$

The first group G_1^{ins} contains RDFS rules that need to be applied in the order specified. The remaining groups contain OWL-Horst rules that can be applied once in any order (except Step B5 in Figure 5.1). The rules in the second group G_2^{ins} are independent of those in the first group G_1^{ins} and they can produce in parallel with the first group G_1^{ins}. The rules in G_3^{ins} depends on those in the first G_1^{ins} and the second G_2^{ins} groups in the sense that the triples produced by these can be used as a premise by the rules in G_3^{ins}. Also, the rules in G_1^{ins} depends on those in G_3^{ins}. Notice that this introduces a loop between the three first groups. The rules in the fourth group G_4^{ins} depend on those triples produced by the last two schema groups (G_2^{sch} and G_3^{sch}) and the first three instance groups (G_1^{ins}, G_2^{ins}, and G_3^{ins}), plus those triples that were received by the current micro-batch. The fourth group needs two instance triples to trigger the rules. Thus, the objective of the fourth group G_4^{ins} is to find the complementary part of the current given instance triple from DS′ by assuming that the schema triples exist.

The analysis of the rules in the fifth group, G_5^{ins}, reveals that these depend on the rules in the first four groups, whereas none of the rules in the first four groups depend on the rules in G_5^{ins}, as already assumed by Cichlid [35]. It is worth recalling that *sameAs* saturation, performed by G_5^{ins}, needs to be dealt with in a careful way in order to avoid a blow up in triple creation. We use the approach introduced by WebPIE [75] and the re-used by Cichlid [35], which we do not describe here again and which, in a nutshell, creates and manages in an efficient way a *sameAsTable* in which, for instance, if a, b, c, and d are the same according to the sameAs relation, then those resources will be stored in a unique line of the table, essentially containing one equivalence class induced by sameAs. Also observe that we could imagine that OWL-Horst rule 11 could trigger again other rules, say rule 14_a. Actually, as shown in [35, 75], these triggered rules would produce triples already inferred by our step B5. For instance if rule 11 produces $x\ \tau\ y$ where (*) x and y are, respectively, sameAs u and v (already used by rule 14a in the premise $u\ \tau\ v$), then if we assume rule 14_a produces again (once re-triggered) a triple by using $x\ \tau\ y$, that triple would be $x\ p\ y$. Since step B5 takes as input $u\ p\ v$

(produced in step B2 by rule 14_a), we have that $x\ p\ y$ is produced by step B5 due to (*), so there is no need to trigger rule 14_a again.

The above analysis allowed us to elaborate on an algorithm, which is depicted in Figure 5.1 for an efficiently saturating RDF streams considering both RDFS and OWL-Horst. It is worth observing that the rule ordering in our solution is similar to that proposed by Cichlid [35], with the notable difference that we strive to perform the saturation incrementally.

As shown in Figure 5.1, when a new micro-batch arrives, first a simple filtering separates new instance triples from new schema triples. Our algorithm first performs step A, in which saturation for new schema rules is performed, by also taking into account the previously inferred *schema* triples. Note that this step is needed in order to avoid inferring many times the same triples starting from newly arrived schema triples. The idea is to infer the new schema and perform a first wave of instance triple derivation in terms of the new schema, only once (as we will see in Step B, newly derived instance triples will be considered for fix point computation).

In step A, first the driver saturates the new schema rules in sub-step A1.1 dedicated to the derivation of new RDFS triples, and in which, first, OWL-Horst rules that can produce premises for RDFS rules 5 and 11 are applied, and then these last ones (RDFS 5 and 11) are applied. In A1.1, once the new schema triples are derived by considering the combinations of the newly received and old schema (those that could infer a new schema, that not met yet), plus the new schema triples in the current micro-batch are used for RDFS instance triple saturation (A1.2), as happened for our algorithm for RDFS saturation (Section 4.1). The novelty in step A is used in step A2, by applying OWL-Horst rules to using the newly obtained schema triples (from the new schema triples plus the old ones) on instance triples. The instance triples will fetch from DS′ (instance triples that derived in the past and stored on HDFS) by utilizing our indexing approach. For instance, for rule 2, once the indicated schema triple identifies a property p. We use our indexing information to retrieve only the triples having p as property and then, we perform the join required by the second and third premise (note such kind of join does not occur for RDFS saturation). In this way, the number of triples involved in the join operation reduces sensibly. Finally, the existed schema triples are updated with new schemas received and obtained due to this micro-batch. To summarize, step A is totally along the lines of our algorithm for RDFS saturation: we obtain new schema triples and use them to infer new instance triples, that is then used in step B, which we comment below.

Step B follows step A and takes as input: the newly received instance triples in the current micro-batch, plus the instance triples derived in step A, plus the new global RDFS-OWL schema still computed and broadcast in step A. The main part of step B consists of a loop for iterating saturation until a fixpoint is reached. The body of the iteration consists of three subsequent steps: a first one concerning RDFS rules for instance triple derivation (Step B1), followed by OWL-Horst derivation (Step B2), while unlike Step A1 there are any relation between the Steps B1, B2; in case these two steps produce new triples, then (Step B3) uses those triples for applying OWL-Horst rules 15, 16 and 4. In Step B3, the rules 15 and 16 trigger, in the case that new object-based triples (`rdf:type` as the predicate) are derived in this round of loop except for the already generated ones. Rule 4, in this step, eligibles to trigger by the existence of new triples excluding the existed triples from the previous rounds.

Once the fixpoint is reached, we jump to Step B4. Explanations are needed to determine the difference between Step B4 and Step B3. As already presented in the RDFS section, few schema rules, i.e., RDFS 5 and 11, required two premises as input and infer a schema triple as output. In the streaming context, any of the premises may be given later. Therefore, by receiving any premises, the second part should search among the existed schemas. The same scenario applies for rules 15, 16, and 4, the difference is that they require two instance triples. Therefore, in Step B4, we assume that one of the instance triples is received by the current micro-batch and the second one needs to fetch from HDFS. By using our indexing technique, we prevent retrieving the whole dataset DS', once in the existence of schema triples to trigger the rules, i.e., 15, 16, 4 is detected.

It is noteworthy that Rules 5 and 11 of RDFS are not complex and resource-consuming to find the missing part since the size of schema triples is small enough to fit in memory. On the contrary, the same procedure is very resource-consuming to find the matching instances in the rules since the rules require two instance triples while one of them may have arrived earlier and located on disk.

For Rule 4 in Step 4, firstly, we fetch all predicate-based triples when their predicate corresponds to the subject of a given schema triple e.g., the subject p from $p\ \tau$ *owl:TransitiveProperty*, when one out of two instance triple is given by the current micro-batch (either received and or derived one). Rules 15 and 16 follow the same strategy as Rule 4, except that, first, they need to find matched schema triples. This part proceeds in a centralized computing fashion, and by considering the size of the schemas in constant time. We then filter the triples for every candidate schema. Then, we do another filter among the filtered triples

and the schemas to find the object/predicate of the second triple to be able to fetch the most potential and relevant triples from the disk by utilizing the indexing information.

In case B4 produces new instance triples, then (step B5) uses those triples as well for applying OWL-Horst rules 1, 2, 7 and 11. Their application requires the system to fetch instance triples through our indexes, plus the indexing of newly inferred triples.

Regarding the implementation of individual rules, we distinguish the following kinds of rules:

1. Rules that take one or two schema triples and produce a schema triple, e.g., OWL Horst rules $12_{(a,b,c)}$.

2. Rules that take one schema triple and one or two instance triples and produce an instance triple, e.g., OWL Horst rules 4 and $8_{(a,b)}$.

3. Rules that take one instance triple and produce one instance triple that we omit them, e.g., rule $5_{(a,b)}$.

4. Rules that take two schema triples and two instance triples and produce an instance triple, namely OWL Horst rules 15 and 16.

Rules in (1) and (2) can be implemented similarly to the RDFS rules presented earlier. Rules in (3) can be implemented straightforwardly at the query time since they involve a single instance triple. Rules in (4), however, need to be processed differently. For this reason, we focus on detailing the processing of Rule 15. Other rules in (4) can be implemented similarly.

For the sake of clarity, we recall rule 15 definition.

Schemas: *v owl:someValuesFrom w*, *v owl:onProperty p*
Instances: *u p x*, *x rdf:type w*
Result: ⇒ *u rdf:type v*

This rule is processed differently depending on whether it is triggered given a newly acquired schema triple (see Box A2 in Figure 5.1), given an instance triple (see Box B3 in Figure 5.1), or based on received a new instance triple (see Box B4 in Figure 5.1). Algorithm 4 details the processing of Rule 15 (Figure 5.1 Step A2), when given corresponding new schema triples. It starts by retrieving the two kinds of schema triples that are necessary for triggering the rule, namely *onProperty* triples and *someValuesFrom* triples (*lines 6-7*). If such triples exist, then the algorithm attempts to find their match. For example, if a newly acquired

Algorithm 4: OWL-Horst *Rule 15* (Step A2)

Input: $\texttt{mb}_{\texttt{sch}}$ – Received schema triples
Input: op' and sv' – Two datasets that represent schema triples with *owl:onProperty* and *owl:someValuesFrom* as predicate respectively except those they received by the current `mb`
Output: $A2_{r_{15}}$ – reasoning results
Begin
  // Extract owl:onProperty *and* owl:someValuesFrom *schema from* $\texttt{mb}_{\texttt{sch}}$
  $new\_op \leftarrow \texttt{mb}_{\texttt{sch}}$.
    filter(t ⇒ t._2.equals(*"owl:onProperty"*)).
    map(t ⇒ (t._1, t._3)).collect.toSet
  $new\_sv \leftarrow \texttt{mb}_{\texttt{sch}}$.
    filter(t ⇒ t._2.equals(*"owl:someValuesFrom"*)).
    map(t ⇒ (t._1, t._3)).collect.toSet

  // Rule 15 won't trigger if there is no new schema triple arrives
  if(new_op.isEmpty && new_sv.isEmpty) **then**
    return empty
  // Keep those schema triples if the second match of schema is arrived too.
  $op_1 \leftarrow$ `findMatches`(new_op, new_sv)
  $sv_1 \leftarrow$ `findMatches`(new_sv, new_op)
  // Also find other matches among the previous schema based on the current schemas
  $op_2 \leftarrow$ `findMatches`(op', new_sv)
  $sv_2 \leftarrow$ `findMatches`(sv', new_op)
  op ← (op_1.toSeq ++ op_2.toSeq)
  sv ← (sv_1.toSeq ++ sv_2.toSeq)

  // Every new op must have, at least, a new and/or old sv/sv′ and so on for every new sv too.
  if((op_1.toSeq ++ op_2.toSeq).isEmpty || (sv_1.toSeq ++ sv_2.toSeq).isEmpty) **then**
    return empty
  // Retrieve related instance triples from DS′ by relying on indexing information
  $pTriples \leftarrow$ Fetch triples from `predicate_based` paths of DS' based on the op
  $oTriples \leftarrow$ Fetch triples from `object_based` paths of DS' based on the sv
  // Saturation process among the fetched triples
  $in_{15} \leftarrow pTriples$.**map**(t ⇒ ((op.value(t._2), t._3), t._1))
  $t_{15} \leftarrow oTriples$.**map**(t ⇒ ((sv.value(t._2), t._1), Nil))
  $A2_{r_{15}} \leftarrow in_{15}$.**join**($t_{15}$).**map**(t ⇒ (t._2._1, t._1._1))
  return $A2_{r_{15}}$
End

triple is an onProperty triple, e.g., (v_1 owl:onProperty p_1), then the algorithm tries to find a matching triples, e.g., (v_1 owl:someValuesFrom w), from received and

already existed schema, and vice versa (*lines 10-13*). For every matching pair of someValuesFrom and onProperty triples (*lines 14-15*), the algorithm retrieves instance triples that can be used for triggering the rule using our index (*lines 18-19*), and inferring implicit triples (*lines 20-22*) accordingly as specified by the rule.

Algorithm 5: OWL-Horst *Rule 15* (Step B3)

1: **Input:** $\texttt{mb}_{\texttt{inst}}$ – Instance Triples of micro-batch
2: **Input:** $op'Swap$ and $sv'Swap$: Datasets that represent schema triples with *owl:onProperty* and *owl:someValuesFrom* as predicate respectively. i.e., Swap means (Object, Subject)
3: **Output:** $B3_{r_{15}}$ – Reasoning results
4: **Begin**
// Extract the given triples and types based on the entire (so far) received schema.
5: $potential\_Triples \leftarrow \texttt{mb}_{\texttt{inst}}$.
filter(t ⇒ $op'Swap$.value.contains(t._2))
6: $potential\_Types \leftarrow \texttt{mb}_{\texttt{inst}}$.
filter(t ⇒ t._2.equals("*rdf:type*") && $sv'Swap$.value.contains(t._3))
7: $tr_{15} \leftarrow potential\_Triples$.
map(t ⇒ (($op'Swap$.value(t._2), t._3), t._1))
8: $ty_{15} \leftarrow potential\_Types$.
map(t ⇒ (($sv'Swap$.value(t._2), t._1), Nil))
9: $B3_{r_{15}} \leftarrow tr_{15}$.**join**($ty_{15}$).**map**(t ⇒ (t._2._1, t._1._1))
10: **return** $B3_{r_{15}}$
11: **End**

Algorithm 5 details the processing of rule 15 given the received and inferred instance triples.

This algorithm 5 relies on received instance triples and total schema. The algorithm starts by retrieving the required instance triples from the received and inferred instance triples (*lines 5-6*) and trigger the rule by considering the schema triples that were present before and along the given micro-batch (*lines 7-9*). The inferred results (*line 10*) will use into another round of saturation process in case that the previous round of saturation process infers new instance triple (i.e., with predicate *"rdf:type"*). Otherwise, they will go to the next saturation step (Step B4).

Algorithm 6 aims to find one of the bipartite instance triples of rule 15 from

Algorithm 6: OWL-Horst *Rule 15* (Step B4)

Input: $\texttt{mb}_{\texttt{inst/inf}}$ – Instance and inferred triples based on the current $\texttt{mb}_\texttt{i}$.
Input: *Tsch* – The total schema that received so far
Output: $B4_{r_{15}}$ – Reasoning results
Begin
// Extract the respective schema triples from the total schema
$sv_{set} \leftarrow$ *Tsch*.
filter(t ⇒ `t._2`.equals(*"owl:someValuesFrom"*)).**map**(t ⇒ (`t._1`, `t._3`))
$op_{set} \leftarrow$ *Tsch*.
filter(t ⇒ `t._2`.equals(*"owl:onProperty"*)).**map**(t ⇒ (`t._1`, `t._3`))
// Choose that someValuesFrom triples if exists any related onProperty for that and vice versa.
$svs \leftarrow$ `findMatches`(sv_{set}, op_{set})
$ops \leftarrow$ `findMatches`(op_{set}, sv_{set})
// Assure that both schemas exist
if(svs.isEmpty || ops.isEmpty) **then return** empty
// Broadcast the svs and ops and the swap version of them, i.e.,(Object, Subject)
$sv_{br} \leftarrow$ **broadcast**(svs)
$op_{br} \leftarrow$ **broadcast**(ops)
$svSwap_{br} \leftarrow$ **broadcast**(svs.**map**(t ⇒ (`t._2, t._1`)))
$opSwap_{br} \leftarrow$ **broadcast**(ops.**map**(t ⇒ (`t._2, t._1`)))
// Filter nominated types, find triples with a matching signature, fetch triples from disk, saturation process
nominatedTypes ← $\texttt{mb}_{\texttt{inst/inf}}$.
filter(t ⇒ $svSwap_{br}$.value.contains(`t._3`) && `t._2`.equals(*"rdf:type"*))
$vs_1 \leftarrow$ *nominatedTypes*.**map**(t ⇒ $svSwap_{br}$.value(`t._2`)).**distinct**
$ps_1 \leftarrow vs_1$.**map**(t ⇒ op_{br}.value(t))
relatedPs ← Fetch triples from predicate-based paths based on ps_1
$tr_{15}\_1 \leftarrow$ *relatedPs*.**map**(t ⇒ (($opSwap_{br}$.value(t._2), t._3), t._1))
$ty_{15}\_1 \leftarrow$ *nominatedTypes*.**map**(t ⇒ (($svSwap_{br}$.value(t._2), t._1), Nil))
$r_{15}\_1 \leftarrow tr_{15}\_1$.**join**($ty_{15}\_1$).**map**(t ⇒ (t._2._1, t._1._1))
// Filter nominated types, find triples with a matching signature, fetch types from disk, saturation process
nominatedTriples ← $\texttt{mb}_{\texttt{inst/inf}}$.**filter**(t ⇒ $opSwap_{br}$.value.contains(t._2))
$ps_2 \leftarrow$ *nominatedTriples*.**map**(t ⇒ $opSwap_{br}$.value(t._2).**distinct**
$vs_2 \leftarrow ps_2$.**map**(t ⇒ sv_{br}.value(t)).distinct.collect.toList)
relatedTs ← Fetch types from object-based paths based on vs_2
$tr_{15}\_2 \leftarrow$ *newP*.**map**(t ⇒ (($opSwap_{br}$.value(t._2), t._3), t._1))
$ty_{15}\_2 \leftarrow$ *relatedTs*.**map**(t ⇒ (($svSwap_{br}$.value(t._2), t._1), Nil))
$r_{15}\_2 \leftarrow tr_{15}\_2$.**join**($ty_{15}\_2$).**map**(t ⇒ (t._2._1, t._1._1))

$B4_{r_{15}} \leftarrow r_{15}\_1$.**union**($r_{15}\_2$)
return $B4_{r_{15}}$
End

the already existing triples (*DS′*) received through previous micro-batches. In this step, the necessary condition is the existence of both schema triples and at least one of the instance triple. In this regard, we suppose that both necessary schema triples exist, and one of the related instances triple given and/or inferred via current micro-batch. Therefore as a first step, we extract the schema triples with predicate *owl:onProperty* and *owl:someValuesFrom*, that required for rule 15, among the so far schema triples that we received up to this moment of the process (*lines 5-6*). As we said, just a complete set of the schema triples is eligible to trigger. For this purpose, the algorithm makes use of the `findMatches`() subroutine to find those matches for every intended schema triple which given *owl:onProperty* triples returns corresponding *owl:someValuesFrom* triples and vice-versa (*lines 7-8*). In the next step, by considering that both schema exist (*line 9*), we examine the provided instance triples with the selected schema triples to pick those instances that both schema triples exist for them. For this purpose, we broadcast the collected schema triples via *broadcast* operation (*lines 10-13*). Then, we pick those triples from the $\texttt{mb}_{\texttt{inst/inf}}$ when they have *rdf:type* as a predicate with the same object as the collected *someValuesFrom* objects (*line 14*). We extract all corresponding *owl:onProperty* schema triples based on the *owl:someValuesFrom* schema and the candidate triples –those with rdf:types as predicate– (lines 15-16). In the following, by utilizing the indexing information, we fetch the related triples among the predicate-based triples from the disk *DS′* (line 17). It is worth mentioning that the number of distinct objects and predicates in datasets is small enough to fit in memory. For example, the examined dataset in this section contains only 116 different distinct numbers of objects and 83 different numbers of predicates for object- and predicate-based triples, respectively. Finally, we apply the saturation process between the chosen schema triples, the candidate *rdf:type* triples, and their corresponding predicate-based triples fetched from the disk *DS′ lines 18-20*. So far, we have done a complete informative saturation for every triple with *rdf:type* as a predicate that we got and inferred via the current $\texttt{mb}_{\texttt{inst/inf}}$. The algorithm 6, *lines 21-24*, dedicated to the same process as *lines 14-20* except to find the right *rdf:type* triples with correspondence objects. For this purpose, we fetch those triples from the object-based triples located on the disk *DS′*. Finally, we apply the saturate process between the selected and fetch triples, and both matched schema triples by considering both matched schema triples. Finally, the results of the saturation processes (i.e., r_{15}_1 and r_{15}_2) are concatenating and return (*line 28*).

5.2 Evaluation

Like RDFS section, our extended method, that we presented in this chapter, outperform the state-of-the-art approaches in reasoning of the OWL-Horst ruleset, mainly Cichlid, when dealing with streams of RDF data. This is particularly the case when the OWL-Horst schema is also given in a stream-based fashion along with RDF data. An empirical evaluation alike RDFS section needs to answer the following questions:

– Once again, *does our method actually outperform in practice the Cichlid method in OWL-Horst ruleset* for saturating streams of *RDF*?

– *And if yes, for how long and what circumstances?*

To answer this questions, we design and implement[2] an experimental analysis that we reported on in this section.

5.2.1 Datasets

Pre-processing Data in Stream

We make in our experiments the assumption that the RDF data received along with OWL-Horst schema triples. So, the dataset arrives in micro-batches in a streaming fashion, while their schema is heavenly divided between the micro-batches. We consider this situation to simulate RDF data in reality production. Specifically, we used for our experiments an RDF dataset that widely uses in the semantic web community: UniProt [28][3]. The UniProt dataset is not stream-based in default. Therefore, we partitioned them into micro-batches to be able to push them out into the streaming saturation process to simulate a data flow where the data will receive in a streamed manner.

For this purpose, we picked a sub-dataset of the UniProt dataset that contains 320 million. In doing so, the dataset partitions into a series of 100 micro-batches each 512 MB in size. The schema triples of UniProt contains 549 triples, divides, almost, equally between the micro-batches. Thus, each micro-batch has a range of [5-6] schema triples.

[2] https://git.lamsade.fr/afarvardin/RDFInStream

[3] The universal protein knowledge base (https://www.uniprot.org/)

5.2.2 Experiment Setup

We conducted our experiment on a local cluster[4] with two configurations: 2 nodes and 4 nodes. One node was, also, reserved to act as the driver node and the remaining nodes (2 and 4 respectively) as worker nodes. Each node has a Xeon Octet 2.4 GHz processor, 48 GB memory, and the cluster provides 33 TB Hadoop file system. The nodes are connected with 1 Gb/s Ethernet. All the nodes run on Debian 9.3 operating system. The version of the Spark we used is 2.1.0 and Hadoop v2.7.0 with Java v1.8 is installed on the cluster.

For each dataset, we ran our experiment 5 times and reported the average running time. We ran our saturation algorithm incrementally for each of the succeeding micro-batches. For comparison purposes, for the similar datasets, we run the Cichlid algorithm on the micro-batches from the same dataset as well. Given that Cichlid is not incremental, for each micro-batch, we had to consider the previous micro-batches as well as the micro-batch in question.

5.2.3 Results

In general, the x-axis of histograms represents the micro-batches that composed the dataset. The y-axis reports the execution time required for its saturation in seconds. For each of the succeeding micro-batches, in both scenarios, the y-axis reports the time required for saturating the dataset composed of the current micro-batch, the previous micro-batches put together.

Figures 5.2 shows the execution time required by our incremental streaming method and the execution time required by the state of the art, i.e., Cichlid, to saturate the UniProt dataset on 2 nodes. The figure also shows for both approaches the exponential trend line. The x-axis represents the received micro-batches, each one composed of instance triples, 512 MB, and a few schema triples, i.e., 5 to 6 schema triple (Table 5.2) per micro-batch. The y-axis reports the execution time required for saturation of each micro-batch in seconds.

Since the processing time of our incremental method is indistinguishable in Figure 5.2, therefore, Figure 5.3 shown only our method results. Figures 5.2 and 5.3 illustrated that the saturation on 2 nodes took 1023 minutes to process with Cichlid method (depicted using a red line) for the first 17 micro-batches from the given dataset, which is 18 minutes for the same number of micro-batches with our incremental algorithm. That is 56.8 times the time required to saturate the

[4]It's located at Université Paris-Dauphine https://dauphine.psl.eu/

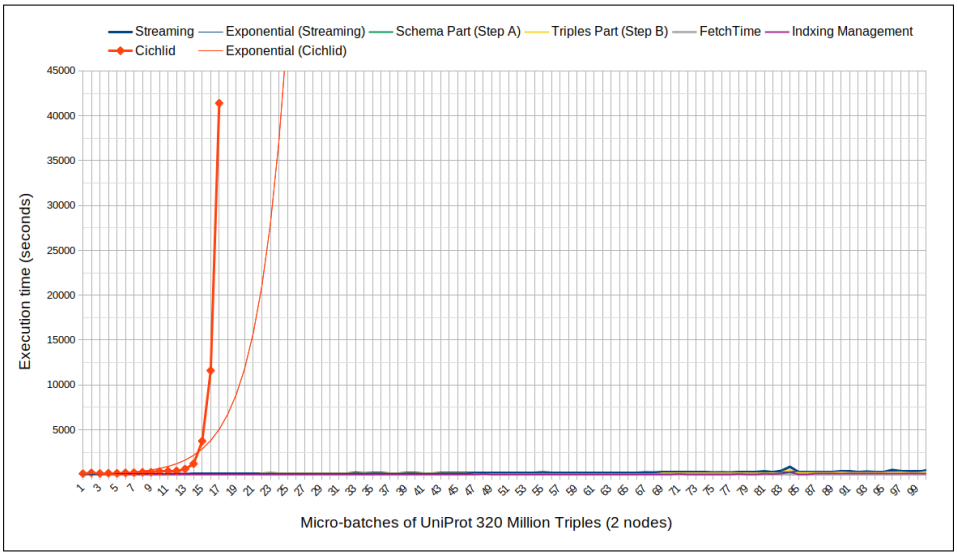


Figure 5.2: UniProt 320 Million Triples - Comparison with Cichlid (2 nodes)

same number of micro-batches (the first 17 ones) compare to our method. Our incremental algorithm (depicted using a blue line) takes 346 minutes to saturate the entire dataset that consists of 100 micro-batches. On average, our method takes 3.5 minutes, whereas (only by considering the first 17 micro-batches) Cichlid needs 64 minutes to apply a complete saturation process, by giving a new micro-batch, while that contains a new schema.

The following reasons are the main cause of failure by Cichlid:

- The volume of RDF is increasing over time;
- The higher rate of computation required for OWL-Horst compare to RDFS ruleset;
- Itreative saturating process required by some rules until a fixpoint;
- Limited resources for the given data.

In Cichlid algorithm, by considering that the size of RDF is growing over time in one hand, and the higher saturation processing rate of the OWL-Horst ruleset in compare to RDFS in the other hand. Also the nature of OWL-Horst rules that needs an uncertain number of loops until a fixpoint, and finally the number of resources that we choose (just 2 nodes) at the designing time of this experiment are the main reasons of failure for the Cichlid in this part.

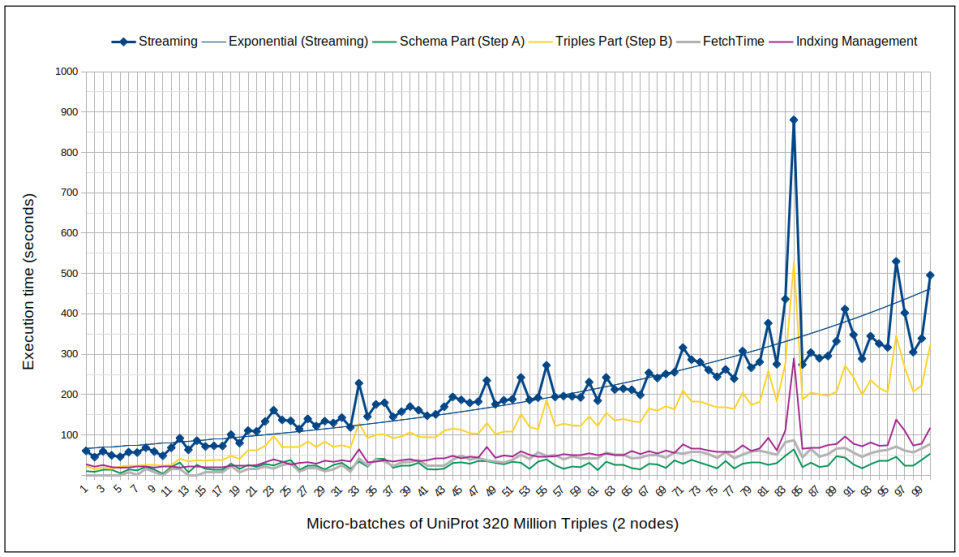


Figure 5.3: UniProt 320 Million Triples - Incremental Streaming (2 nodes)

Specifically, Figure 5.3 shown our incremental streaming saturation on 2 nodes. The streaming depicted using a blue line illustrates the time required for each step of our incremental algorithm solution to do a complete saturation by given a new micro-batch with or without carrying schema triples. This time creates from four different operations: *i)* processing time for Schema Part (Step A in Figure 5.1) (in green line), *ii)* processing time for the triple part (Step B in Figure 5.1), and *iii)* the *indexing management* time that required for partitioning, compressing, storing, and collect the data information for our indexing technique from the saturated micro-batch. In our case, all micro-batches contain new schema triples, and the incremental saturation processing for new schema presents by *Step A* (depicted using a green line). The fetches time (gray line) represents the time that our algorithm needs to detect and retrieve data among the already existed triples (located in *DS′*). This time embedded in Step A and Step B as well. The total fetches time for the entire processes is 66 minutes, which is 19% of the total processing time that is 346 minutes. On average, our indexing technique takes 40 seconds in average to detect and fetches the necessary triples from *DS′* based on new schema triples.

In Figure 5.3, we have a big jump on the micro-batch 84 ($\texttt{mb}_{84}$). To interpret that, we get help from micro-batch 83 ($\texttt{mb}_{83}$) for the comparison point, since the total execution time for $\texttt{mb}_{84}$ is 880 seconds while it is 436 for $\texttt{mb}_{83}$. In this purpose, fetching times, processing time of Step A and B, indexing management and, the

number of iteration that requires to get into fixed point.

Table 5.2: Types and numbers of schema triple per micro-batch in Figure 5.4.

μ_b	**Schemas**
$\mu_b 0$	owl(*onProperty* → 1), rdfs(*domain* → 2, *range* → 1, *subClassOf* → 2)
$\mu_b 1$	owl(*allValuesFrom* → 1), rdfs(*domain* → 1, *range* → 2, *subClassOf* → 2)
$\mu_b 2$	owl(*onProperty* → 1), rdfs(*range* → 2, *subPropertyOf* → 1, *subClassOf* → 2)
$\mu_b 3$	owl(*allValuesFrom* → 1), rdfs(*domain* → 3, *subClassOf* → 2)
$\mu_b 4$	rdfs(*domain* → 1, *range* → 2, *subClassOf* → 3)
$\mu_b 5$	owl(*onProperty* → 1), rdfs(*range* → 1, *domain* → 1, *subPropertyOf* → 1, *subClassOf* → 2)
$\mu_b 6$	owl(*inverseOf* → 1, *allValuesFrom* → 1), rdfs(*range* → 2, *subClassOf* → 2)
$\mu_b 7$	owl(*onProperty* → 1, *equivalentClass* → 1), rdfs(*range* → 1, *domain* → 2, *subClassOf* → 1)
$\mu_b 8$	owl(*onProperty* → 1, *equivalentClass* → 1), rdfs(*range* → 1, *domain* → 1, *subClassOf* → 2)
$\mu_b 9$	owl(*onProperty* → 1), rdfs(*domain* → 1, *subClassOf* → 3)
$\mu_b 10$	owl(*onProperty* → 1), rdfs(*range* → 2, *subClassOf* → 3)
$\mu_b 11$	rdfs(*range* → 2, *domain* → 1, *subClassOf* → 3)
$\mu_b 12$	owl(*equivalentClass* → 1), rdfs(*domain* → 3, *subClassOf* → 2)
$\mu_b 13$	owl(*onProperty* → 2), rdfs(*range* → *1*, *domain* → 2, *subClassOf* → 1)
...	...
$\mu_b 82$	owl(*hasValue* → 1), rdfs(*range* → 1, *domain* → 1, *subPropertyOf* → 1, *subClassOf* → 1)
$\mu_b 83$	rdfs(*range* → 1, *domain* → 2, *subPropertyOf* → 1, *subClassOf* → 2)
$\mu_b 84$	rdfs(*range* → 3, *subClassOf* → 2)
...	...
$\mu_b 100$	owl(*allValuesFrom* → 1), rdfs(*domain* → 2, *range* → 1, *subClassOf* → 2)

– *Fetching time:* The fetching time for $\texttt{mb}_{84}$ is 95 seconds, while it is 79 seconds for the $\texttt{mb}_{83}$. That time corresponds to the retrieval of almost 111 and 69 million potential RDF triples (object- and predicate-based triples) from *DS′* for $\texttt{mb}_{84}$ and $\texttt{mb}_{83}$, respectively. Those triples are retrieved from 1075 files for $\texttt{mb}_{84}$, and 1142 files for $\texttt{mb}_{83}$, respectively. We, therefore, conclude that the fetching time is not the main reason for the difference in processing the two micro-batches. Furthermore, given the growing volume of data to be fetched, we can observe that our fetching algorithm retrieves them in a reasonable time.

– *Step A:* This step for $\texttt{mb}_{84}$ takes 64 seconds (including the required fetching times), while it takes 48 seconds for $\texttt{mb}_{83}$.

– *Step B:* The time required in Step B (including the fetchint time) is 527 seconds for $\texttt{mb}_{84}$, while it is 277 seconds for $\texttt{mb}_{83}$.

– *Indexing Management:* The indexing management time for $\texttt{mb}_{84}$ is 289 seconds. This time is divided into 251 and 38 seconds for partitioning,

indexing, and saving the data into object- and predicate-based triples, respectively. The indexing management time for $\texttt{mb}_{83}$ is 111 seconds that consist of 78 and 33 seconds for the object- and predicate-based triples respectively. Concerning this step, We observed that the dominant execution time belongs to object-based triples.

– *Fixpoint:* On the other hand, both mbes used four iterations to reach a fixpoint. In the $\texttt{mb}_{84}$, we observed 240K predicate-based triples and 1.45 million object-based triples per partition (i.e., there are 20 partitions in 2 nodes) that are required to saturate in every iteration. These numbers for $\texttt{mb}_{83}$ are 240K predicate-based triples and 460K object-based triples in every iteration. TO a considerable extent, this explains the difference in the processing time required for saturating data in $\texttt{mb}_{84}$ compared to time needed for $\texttt{mb}_{83}$.

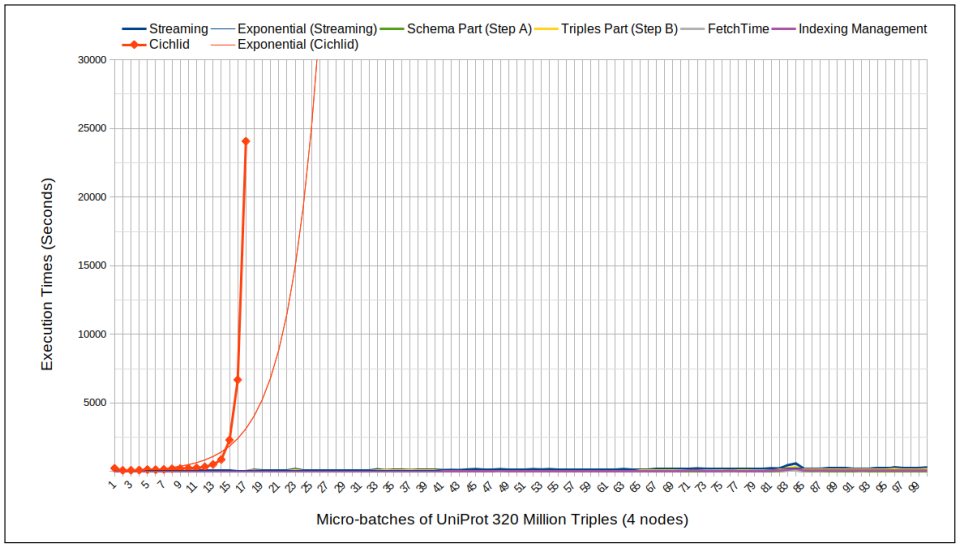


Figure 5.4: UniProt 320 Million Triples - Comparison with Cichlid (4 nodes)

Figures 5.4 shows the obtained results by the saturation process of the UniProt-320M dataset in successive micro-batches on four nodes. It shows that the time required by Cichlid for saturating the streaming data is still increasing significantly by receiving successive micro-batches so that the processing time of our incremental method becomes indistinguishable. Therefore, we draw our method results in Figure 5.5.

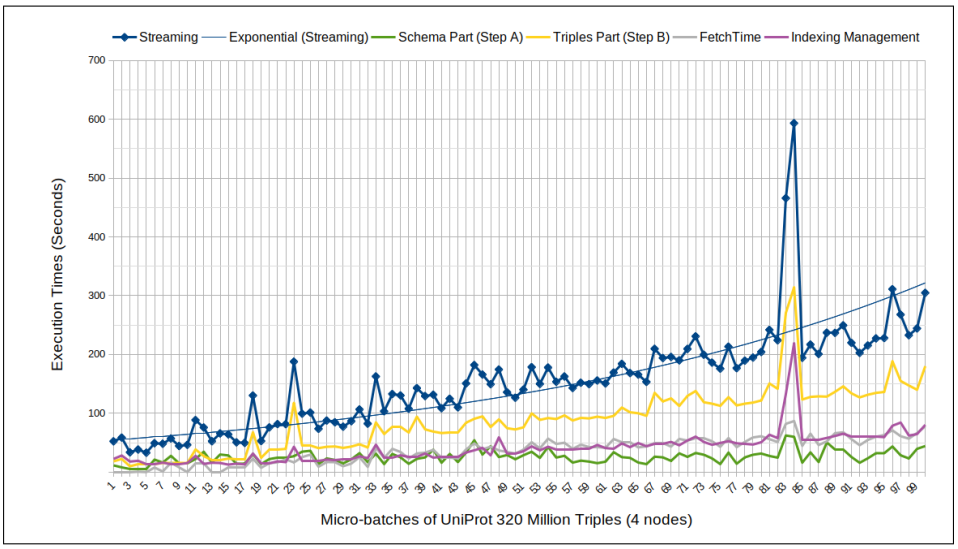


Figure 5.5: UniProt 320 Million Triples - Incremental Streaming (4 nodes)

Figures 5.4 and 5.5 illustrated the saturation process on four nodes. It took 1086 minutes with Cichlid for the first 17 micro-batches, while it takes 15 minutes for the same number of micro-batches with our incremental algorithm. That is 72.4 times the time required to saturate the same number of micro-batches (the first 17 ones). Our incremental algorithm takes 257 minutes to process the entire dataset consists of 100 micro-batches. On average, our method takes 2.7 minutes, whereas (by considering the first 17 micro-batches), the Cichlid needs 36 minutes to apply a complete saturation process by giving a new micro-batch contains a new schema.

Figure 5.6 shown a comparison of the same process between 2 and 4 nodes for the Cichlid algorithm. Despite the size of the nodes that became double, the sum of saturation processing time in 2 nodes for the last three micro-batches takes two times of 4 nodes execution time, which are 946 and 551 minutes on 2 and 4 nodes, respectively. However, the first 14 micro-batches take 78 and 58 minutes for 2 and 4 nodes, respectively. The algorithm on four nodes provides a better computing rate for the same volume of data moreover scaled almost 2x in every micro-batches. The processing trend line for $\mathtt{mb}_{15}$ on two nodes grows faster. This growth only on the last micro-batch ($\mathtt{mb}_{17}$) takes more processing time than the recent three micro-batches ($\mathtt{mb}_{15,16,17}$) with four nodes.

Figure 5.5 shown our incremental streaming saturation on 4 nodes. The streaming line (blue) shows a complete saturation processing time by given a new micro-

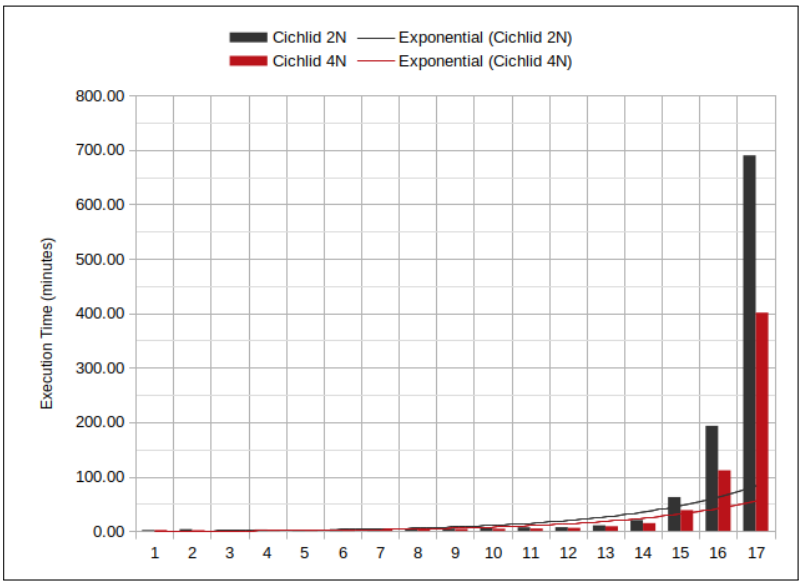


Figure 5.6: The First 17 mbes of UniProt 320 Million Triples - Cichlid

batch with or without carrying schema triples. In our case, all micro-batches contain new schema triples, and the *Step A* illustrates processing time based on the new schema. The Streaming line (in blue) shown the total execution time required for a micro-batch. This time creates from four different operations: *i)* processing time for Schema Part (Step A in Figure 5.1) (in green), *ii)* processing time for the triple part (Step B in Figure 5.1), and *iii)* the *indexing management* time that required for partitioning, compressing, storing, and collect the data information for our indexing technique from the saturated micro-batch. The fetches time (in gray) represents the time that our algorithm needs to detect and retrieve data among the existed triples (i.e., from *DS'*). This time is embedded in Step A and Step B.

The total fetches time for the entire process is 61 minutes, which is 23.7% of the total processing time, which is 257 minutes. In the 4 node cluster, the fetch time seems to take more time related to the same saturation process on the 2 node cluster. The fetching time is related to the number of files that are going to fetch from the same HDFS. Therefore it takes the same time for both cases to retrieve the triples. We obtained that the fetch time is 1.7% faster on the 4 node cluster.

On average, our indexing technique takes 37 seconds to detect and fetches the necessary triples from *DS'* by given new schema triples. In Figure 5.5 (alike figure 5.3), we have a big jump on the micro-batch 84 (mb_{84}). We use micro-batch 82 (mb_{82}) for the comparison point, because the total execution time for mb_{84} is 593 seconds while it is 224 for mb_{82}. In this purpose, fetching times, processing time

of Step A and B, indexing management and, the number of iteration that requires to get into fixed point.

- *Fetching time:* We know that the total fetching time for the entire process of $\mathtt{mb}_{84}$ is 87 seconds, while it is 51 seconds for the $\mathtt{mb}_{82}$. That time correspond to the retrieval of almost 111 and 69 million potential RDF triples (object- and predicate-based triples) from DS' for $\mathtt{mb}_{84}$ and $\mathtt{mb}_{82}$ Those triples are retrieved from 1075 and 964 files for $\mathtt{mb}_{84}$ and $\mathtt{mb}_{82}$, respectively. We, therefore, conclude that the fetching time is not the main reason for the difference in processing the two micro-batches. Furthermore, given the growing volume of data to be fetched, we can observe that our fetching algorithm retrieves them in a reasonable time.

- *Step A:* This step for $\mathtt{mb}_{84}$ takes 60 seconds including the required fetching times, while it takes 25 seconds for $\mathtt{mb}_{82}$.

- *Step B:* The time required in Step B, including the fetch time for this step, is 314 seconds for $\mathtt{mb}_{84}$ while it is 142 seconds for $\mathtt{mb}_{82}$.

- *Indexing Management:* Moreover, the indexing management time for $\mathtt{mb}_{84}$ is 219 seconds. This time divided into 187 and 32 seconds for partitioning, indexing, and saving the data into object- and predicate-based triples, respectively. The indexing management time for $\mathtt{mb}_{82}$ is 58 seconds that consist of 27 and 31 seconds for the object- and predicate-based triples respectively. Concerning this step, We observed that the dominant execution time belongs to object-based triples.

- *Fixpoint:* On the other hand, both mbes used four iterations to reach a fixpoint. In the $\mathtt{mb}_{84}$, we observed 110K predicate-based triples and 660K object-based triples per partition (i.e., there are 44 partitions in 4 nodes) that are required to saturate in every iteration. These numbers for $\mathtt{mb}_{82}$ are 111K predicate-based triples and 36K object-based triples in every iteration. To a considerable extent, this explains the difference in the processing time required for saturating data in $\mathtt{mb}_{84}$ compared to time needed for $\mathtt{mb}_{82}$

Cichlid takes 40 to 60 extra seconds, which is negligible compare to the later ones, for the early micro-batches processing time. But since they need to apply batch saturation processing on the previous micro-batches, therefore by time goes on as the volume of data gets larger, the saturation processing time increasing

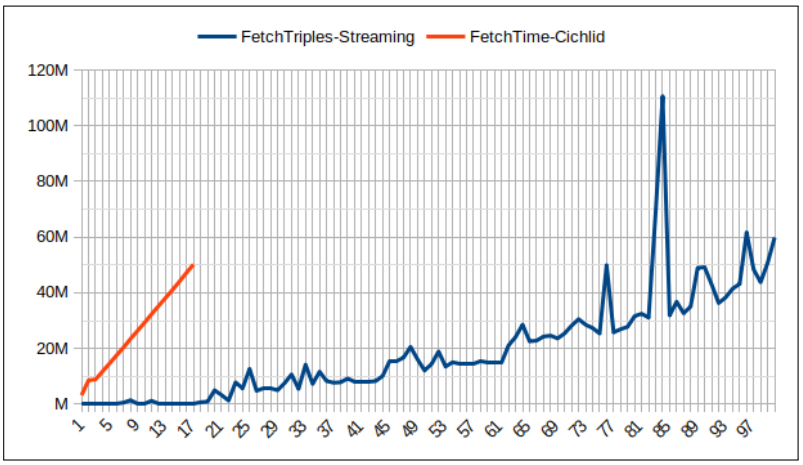


Figure 5.7: Fetches triples per mb. UniProt 320 Million Triples

rapidly. On the other hand, our incremental algorithm fetches just related instance triples, based on the given new schema, among the past data to do a complete informative content of the dataset. As we saw earlier in the RDFS chapter, the good performance of our algorithm is due to its incremental nature, but also its underlying indexing mechanism.

We also recorded the number of triples that are retrieved by a given new micro-batch using our incremental method and compared it with the number of triples that are retrieved by Cichlid. Figure 5.7 depicts the results. It shows that our method (depicted with the blue line) retrieves far smaller numbers of triples compared with the Cichlid (depicted with the red). This can be explained by the fact that our method utilizes indexing structures designed to retrieve only the triples that are likely to yield the activation of a saturation rule. It is worth noting that for $\mathtt{mb}_{84}$, we fetch 94 million triples (around one-third of the whole dataset) from the already existing dataset DS'. Thanks to the incremental loading of our indexing data structures, our approach can fetch a massive number of triples successfully in a reasonable time by utilizing a relatively small cluster.

5.3 Conclusion

In this chapter, we described an extension of our stream-based reasoning technique for the OWL-Horst forward inference and demonstrated it on well-known RDF datasets. We obtained that the devised indexing technique is adaptable to the new scenario of the OWL-Horst. To this end, we checked with the OWL-Horst rules

using the indexing technique to fetch the relative triples and apply the saturation process on the eligible OWL-Horst rules. In other words, we do not trigger a rule if any of their premises are missing. Also, we compare our method with state of the art, namely Cichlid, and show that our strategy successfully adopts into the streaming RDF data when they convey OWL-Horst rules along with them. This is obtained by reducing the number of loading triples by fetching them wisely from *DS′* based on the received new OWL-Horst rules and RDF triples. Finally, we trigger only the eligible rules when all of their premises exist at the time of saturation. We observed that this strategy reduces IO throughput and execution time significantly.

To innovate and improve the performance, we have introduced several optimizations to handle the given and already received triples. These optimizations consist of: *i)* Rule pruning on the given schema triples, *iii)* Reduce the execution time, and *iii)* Reduce the duplicate elimination costs in a stream.

Both in terms of re-saturation and durability, our technique outperforms the published approaches ([35]) by a significant margin. To summarize , we show that our approach outperforms the state-of-the-art in both durability and processing time when saturating RDF streams data incrementally using the Apache Spark platform.

6

Conclusions, Discussions, and Future Work

Contents

Nowadays, RDF data, just like many other kinds of data, are produced in high volumes. A wide range of systems is generating data in RDF format including data produced in the context of health monitoring and financial market applications, feed of user-content provided by social network platforms, as well as long-running scientific experiments that adopt a stream-flow programming model, LOD, etc. To take full advantage of semantic data and turn them into actionable knowledge, the semantic web community has devised techniques for processing and reasoning over RDF data. This trend generated the need for new solutions for processing and reasoning over RDF datasets since existing state-of-the-art techniques cannot cope with large volumes of data. Nowadays, aside from the volume of data that is so far produced and processed, the frequent generation of massive data and process them in a reasonable time carries a new challenge.

Data saturation is crucial to ensure that RDF processing and querying work on the *complete* informative content of an RDF database, without ignoring implicit information. Taking this into consideration, we work in this thesis on the reasoning over streams of RDF triples (including schema triples). Indeed, when RDF data is produced in streams, the state-of-the-art approaches must re-process the whole data collection to obtain triples entailed by the newly received ones. It is since both received (via current micro-batch) and inferred triples (those already saturated

and stored triples on the disk) can entail new triples under the presence of newly received instance/schema triples.

Throughout the chapters of this thesis, we described the main approaches that constitute the techniques presented in this work. We utilize the devised incremental indexing for each entailment rule, which is in charge of a specific part of the overall target. Each of the described steps outputs some results that can be susceptible to the next steps to querying and analysis. The outcomes collected from all the steps allow achieving the final goal, i.e., scalable saturation of streaming RDF triples.

Based on our literature study applied to the existing approaches and their drawbacks, we have built an incremental indexing technique that composes of different steps and responsibilities. We can summarize the contributions of these approaches as follows: i) partitioning the saturated instance triples by their object/predicate, ii) storing the partitioned triples on HDFS on an object- or predicate-based paths, iii) and finally, maintaining a hash table in memory that contains the objects'/predicates' of the intended triples as a unique key and their physical stored paths as value to access stored triples later.

In this thesis, we have shown how RDF data streams can be saturated in an optimized way. We also showed that our approach outperforms state-of-the-art solutions for incrementally saturating RDF, namely Cichlid [35].

To this end, we have presented a stream-based scalable indexing technique for parallel RDFS and OWL-Horst forward inference and applied it on well-known RDF datasets. To innovate and improve the performance, we introduced some key optimizations:

- *Rule pruning on given schema:* By analyzing a given newly schema triple, we ignore some unnecessary rules from the execution rules' order to reduce the fetching time from DS';

- *Reduce the execution time:* Thanks to indexing technique for reducing the number of fetching triples among DS' (already received and saturated) and subsequently, reduce the saturation process by leveraging our incremental indexing scheme;

- *Duplicate elimination*; In this work, we, slightly, compromise with duplicates. Since the storage cost is far cheaper than the processing cost, therefore, we decided to minimize the number of duplicates on the disk to the goal of reducing the processing costs. For this purpose, duplicate elimination

applies to every micro-batch before saving on the disk, and never consider the duplicates among micro-batches at the re-saturation process time.

6.1 Discussion of Scope

Despite all efforts to optimize our implementation, there are a number of assumptions behind our algorithms: (a) Although the size of the cluster's memory is usually large enough, the schema must be small enough to fit in the memory; (b) Same as the state of the art [75], we assume that there is no ontology hijacking [38]; (c) The used datasets are available locally. We divide them into small files and push them into the saturation process to simulate a stream-based data convey; (d) Our algorithm, somehow, depends on the HDFS configurations. The NameNode in HDFS responsible for storing the available file names on the HDFS. Thus, our method is highly dependent on this feature since each micro-batch is stored in several new files. In general, the java heap size plays a vital role in our devised technique.

Despite the scalable method that we have devised the cluster needs to be carefully tuned to utilize its total power. In other words, configuring multi-node and dedicating a large amount of memory per executor does not guarantee to perform saturation efficiently.

6.2 Discussion on Streaming Platform

In this work, we have devised a stream reasoning method using Apache Spark Streaming. The indexing technique that we have developed is not dedicated just for RDFS reasoning and can be applied for other problems, i.e., query answering. This technique focuses on the related instance triples for re-saturating based on the given newly schema triples. The system can work continuously; until HDFS has space, and the java heap could tolerate storing new file names.

We also described an extension of our stream-based technique for parallel OWL-Horst forward inference and applied it on well-known RDF datasets (i.e., UniProt). We observed that the devised indexing technique is adaptable to a new scenario. In this case, we reason over the OWL-Horst rules using the indexing technique to fetch the relevant triples and apply the saturation process on the eligible OWL-Horst rules. In other words, we do not trigger a rule if any of their premises are missing. We compare our method with the state-of-the-art,

namely Cichlid, and it shows that our strategy successfully operate over RDF data when they convey OWL-Horst rules along with them. This important obtained by reducing the numbers of loaded triples, thanks to the indexing technique, based on the received new OWL-Horst rules and RDF triples. Finally, we trigger the rules when all of their premises exist at the time of saturation. We observed that this strategy reduces IO throughput and execution time significantly.

6.3 Future challenges

In fact, our achieved work opens up doors for multiple further research. There exist several potential perspectives based on the obtained results.

In the empirical evaluations, we were limited to not use some potential Apache Spark feature that is "Dynamic Resource Allocation". We intend to examine this capability in our future work. Then, since Apache Flink is a well-known distributed platform for process streaming, one of the perspectives is to re-implement the proposed approach with Apache Flink and conduct a comparison with the current system. This comparison gives us a broader perspective on the capabilities of the RDF saturation in the context of data stream.

Another important potential work is to apply our technique for query answering problems. For this purpose, we require to analyze the required triple model based on the premises of the given query. By obtaining this information, we can fetch the relevant triples from the existed dataset. We believe that our technique can be seamlessly adapted for this purpose.

Bibliography

[1] AGROVOC is a controlled vocabulary covering all areas of interest of the Food and Agriculture Organization (FAO) of the United Nations, including food, nutrition, agriculture, forestry, fisheries, scientific and common names of animals and plants, environment, biological notions, techniques of plant cultivation and more. https://lod-cloud.net/dataset/agrovoc.

[2] Apache Flink. https://flink.apache.org/.

[3] Apache Hadoop. https://hadoop.apache.org/.

[4] CTD includes manually curated data describing cross-species chemical-gene/protein interactions and chemical- and gene-disease relationships to illuminate molecular mechanisms underlying variable susceptibility and environmentally influenced diseases. https://lod-cloud.net/dataset/bio2rdf-ctd.

[5] Farming statistics (farm sizes, land use, livestock) on local authority level. https://lod-cloud.net/dataset/environment-data-gov-uk.

[6] Reference data for linked UK government data: * People * Departments * Namespace for various time intervals. https://lod-cloud.net/dataset/reference-data-gov-uk.

[7] SocialLink, a publicly available Linked Open Data dataset that matches social media accounts on Twitter to the corresponding entities in multiple language chapters of DBpedia. https://lod-cloud.net/dataset/SocialLink.

[8] W3C Recommandation: Resource description framework(RDF). http://www.w3.org/RDF/.

[9] W3C: OWL Web Ontology Language. https://www.w3.org/TR/owl-ref/#OWLFull, 10 February 2004.

[10] Semantic Web Activity Statement. https://www.w3.org/2001/sw/Activity/, 2001.

[11] Medical Subjects Headings Thesaurus 2012, OWL version. https://lod-cloud.net/dataset/bioportal-mesh-owl, 2012.

[12] W3C Recommandation: RDF Primer. http://www.w3.org/TR/rdf-primer/, 2012.

[13] RDF 1.1 N-Triples. https://www.w3.org/TR/n-triples/, 2014.

[14] D. J. Abadi, A. Marcus, S. Madden, and K. J. Hollenbach. Scalable semantic web data management using vertical partitioning. In *Proceedings of the 33rd International Conference on Very Large Data Bases, University of Vienna, Austria, September 23-27, 2007*, pages 411–422. ACM, 2007.

[15] G. Agathangelos, G. Troullinou, H. Kondylakis, K. Stefanidis, and D. Plexousakis. Incremental data partitioning of RDF data in SPARK. In A. Gangemi, A. L. Gentile, A. G. Nuzzolese, S. Rudolph, M. Maleshkova, H. Paulheim, J. Z. Pan, and M. Alam, editors, *The Semantic Web: ESWC 2018 Satellite Events - ESWC 2018 Satellite Events, Heraklion, Crete, Greece, June 3-7, 2018, Revised Selected Papers*, volume 11155 of *Lecture Notes in Computer Science*, pages 50–54. Springer, 2018.

[16] G. Agathangelos, G. Troullinou, H. Kondylakis, K. Stefanidis, and 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*, pages 54–59. IEEE Computer Society, 2018.

[17] S. Auer, C. Bizer, G. Kobilarov, J. Lehmann, R. Cyganiak, and Z. G. Ives. Dbpedia: A nucleus for a web of open data. In *ISWC/ASWC*, volume 4825 of *Lecture Notes in Computer Science*, pages 722–735. Springer, 2007.

[18] D. F. Barbieri, D. Braga, S. Ceri, E. D. Valle, and M. Grossniklaus. Incremental reasoning on streams and rich background knowledge. In *The Semantic Web: Research and Applications, 7th Extended Semantic Web Conference, ESWC 2010, Heraklion, Crete, Greece, May 30 - June 3, 2010, Proceedings, Part I*, pages 1–15, 2010.

[19] H. R. Bazoobandi, H. Beck, and J. Urbani. Expressive stream reasoning with laser. In C. d'Amato, M. Fernández, V. A. M. Tamma, F. Lécué, P. Cudré-Mauroux, J. F. Sequeda, C. Lange, and J. Heflin, editors, *The Semantic Web - ISWC 2017 - 16th International Semantic Web Conference, Vienna, Austria, October 21-25, 2017, Proceedings, Part I*, volume 10587 of *Lecture Notes in Computer Science*, pages 87–103. Springer, 2017.

[20] H. Beck, M. Dao-Tran, T. Eiter, and M. Fink. LARS: A logic-based framework for analyzing reasoning over streams. In B. Bonet and S. Koenig, editors, *Proceedings of the Twenty-Ninth AAAI Conference on Artificial Intelligence, January 25-30, 2015, Austin, Texas, USA*, pages 1431–1438. AAAI Press, 2015.

[21] H. Beck, T. Eiter, and C. Folie. Ticker: A system for incremental asp-based stream reasoning. *Theory Pract. Log. Program.*, 17(5-6):744–763, 2017.

[22] T. Berners-Lee, J. Hendler, and O. Lassila. The semantic web. *Scientific American*, 284(5):34–43, May 2001.

[23] C. Bizer, T. Heath, and T. Berners-Lee. Linked data-the story so far. *International journal on semantic web and information systems*, 5(3):1–22, 2009.

[24] D. Brickley, R. V. Guha, and A. Layman. Resource description framework (rdf) schema specification. 1999.

[25] S. Cebiric, F. Goasdoué, H. Kondylakis, D. Kotzinos, I. Manolescu, G. Troullinou, and M. Zneika. Summarizing semantic graphs: a survey. *VLDB J.*, 28(3):295–327, 2019.

[26] S. Ceri, M. Negri, and G. Pelagatti. Horizontal data partitioning in database design. In M. Schkolnick, editor, *Proceedings of the 1982 ACM SIGMOD International Conference on Management of Data, Orlando, Florida, USA, June 2-4, 1982*, pages 128–136. ACM Press, 1982.

[27] J. Chevalier, J. Subercaze, C. Gravier, and F. Laforest. Slider: An efficient incremental reasoner. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data, Melbourne, Victoria, Australia, May 31 - June 4, 2015*, pages 1081–1086, 2015.

[28] T. U. Consortium. Uniprot: the universal protein knowledgebase. *Nucleic Acids Research*, 45(Database-Issue):D158–D169, 2017.

[29] O. Curé, H. Naacke, T. Randriamalala, and B. Amann. Litemat: A scalable, cost-efficient inference encoding scheme for large RDF graphs. In *2015 IEEE International Conference on Big Data, Big Data 2015, Santa Clara, CA, USA, October 29 - November 1, 2015*, pages 1823–1830. IEEE Computer Society, 2015.

[30] J. Dean and S. Ghemawat. Mapreduce: simplified data processing on large clusters. *Commun. ACM*, 51(1):107–113, 2008.

[31] S. Ewen, K. Tzoumas, M. Kaufmann, and V. Markl. Spinning fast iterative data flows. *PVLDB*, 5(11):1268–1279, 2012.

[32] F. Goasdoué, I. Manolescu, and A. Roatiş. Efficient query answering against dynamic rdf databases. In *Proceedings of the 16th International Conference on Extending Database Technology*, pages 299–310. ACM, 2013.

[33] N. Govind. A/B Testing and Beyond: Improving the Netflix Streaming Experience with Experimentation and Data Science. https://bit.ly/3npxbx1/.

[34] D. Graux, L. Jachiet, P. Genevès, and N. Layaïda. SPARQLGX: efficient distributed evaluation of SPARQL with apache spark. In P. T. Groth, E. Simperl, A. J. G. Gray, M. Sabou, M. Krötzsch, F. Lécué, F. Flöck, and Y. Gil, editors, *The Semantic Web - ISWC 2016 - 15th International Semantic Web Conference, Kobe, Japan, October 17-21, 2016, Proceedings, Part II*, volume 9982 of *Lecture Notes in Computer Science*, pages 80–87, 2016.

[35] R. Gu, S. Wang, F. Wang, C. Yuan, and Y. Huang. Cichlid: efficient large scale rdfs/owl reasoning with spark. In *Parallel and Distributed Processing Symposium (IPDPS), 2015 IEEE International*, pages 700–709. IEEE, 2015.

[36] Y. Guo, Z. Pan, and J. Heflin. LUBM: A benchmark for OWL knowledge base systems. *J. Web Sem.*, 3(2-3):158–182, 2005.

[37] N. Heino and J. Z. Pan. RDFS reasoning on massively parallel hardware. In P. Cudré-Mauroux, J. Heflin, E. Sirin, T. Tudorache, J. Euzenat, M. Hauswirth, J. X. Parreira, J. Hendler, G. Schreiber, A. Bernstein, and E. Blomqvist, editors, *The Semantic Web - ISWC 2012 - 11th International Semantic Web Conference, Boston, MA, USA, November 11-15, 2012, Proceedings, Part I*, volume 7649 of *Lecture Notes in Computer Science*, pages 133–148. Springer, 2012.

[38] A. Hogan, A. Harth, and A. Polleres. Scalable authoritative OWL reasoning for the web. *Int. J. Semantic Web Inf. Syst.*, 5(2):49–90, 2009.

[39] K. Hose and R. Schenkel. WARP: workload-aware replication and partitioning for RDF. In C. Y. Chan, J. Lu, K. Nørvåg, and E. Tanin, editors, *Workshops Proceedings of the 29th IEEE International Conference on Data Engineering, ICDE 2013, Brisbane, Australia, April 8-12, 2013*, pages 1–6. IEEE Computer Society, 2013.

[40] P. Hu, J. Urbani, B. Motik, and I. Horrocks. Datalog reasoning over compressed RDF knowledge bases. In W. Zhu, D. Tao, X. Cheng, P. Cui, E. A. Rundensteiner, D. Carmel, Q. He, and J. X. Yu, editors, *Proceedings of the 28th ACM International Conference on Information and Knowledge Management, CIKM 2019, Beijing, China, November 3-7, 2019*, pages 2065–2068. ACM, 2019.

[41] S. Idreos, F. Groffen, N. Nes, S. Manegold, K. S. Mullender, and M. L. Kersten. Monetdb: Two decades of research in column-oriented database architectures. *IEEE Data Eng. Bull.*, 35(1):40–45, 2012.

[42] Z. Kaoudi and I. Manolescu. RDF in the clouds: a survey. *VLDB J.*, 24(1):67–91, 2015.

[43] Z. Kaoudi, I. Miliaraki, and M. Koubarakis. RDFS reasoning and query answering on top of DHTs. In *International Semantic Web Conference*, volume 5318 of *Lecture Notes in Computer Science*, pages 499–516. Springer, 2008.

[44] K. Kim, B. Moon, and H. Kim. RG-index: an RDF graph index for efficient SPARQL query processing. *Expert Syst. Appl.*, 41(10):4596–4607, 2014.

[45] K. Kim, B. Moon, and H. Kim. R3F: RDF triple filtering method for efficient SPARQL query processing. *World Wide Web*, 18(2):317–357, 2015.

[46] G. Ladwig and T. Tran. Combining query translation with query answering for efficient keyword search. In L. Aroyo, G. Antoniou, E. Hyvönen, A. ten Teije, H. Stuckenschmidt, L. Cabral, and T. Tudorache, editors, *The Semantic Web: Research and Applications, 7th Extended Semantic Web Conference, ESWC 2010, Heraklion, Crete, Greece, May 30 - June 3, 2010, Proceedings, Part II*, volume 6089 of *Lecture Notes in Computer Science*, pages 288–303. Springer, 2010.

[47] Y. Leng, Z. Chen, H. Wang, and F. Zhong. A partitioning and index algorithm for RDF data of cloud-based robotic systems. *IEEE Access*, 6:29836–29845, 2018.

[48] J. Leskovec, A. Rajaraman, and J. D. Ullman. *Mining of massive datasets*. Cambridge university press, 2014.

[49] C. S. Liew, M. P. Atkinson, M. Galea, T. F. Ang, P. Martin, and J. I. V. Hemert. Scientific workflows: moving across paradigms. *ACM Computing Surveys (CSUR)*, 49(4):66, 2017.

[50] C. Liu, J. Urbani, and G. Qi. Efficient RDF stream reasoning with graphics processingunits (gpus). In C. Chung, A. Z. Broder, K. Shim, and T. Suel, editors, *23rd International World Wide Web Conference, WWW '14, Seoul, Republic of Korea, April 7-11, 2014, Companion Volume*, pages 343–344. ACM, 2014.

[51] A. Madkour, A. M. Aly, and W. G. Aref. WORQ: workload-driven RDF query processing. In D. Vrandecic, K. Bontcheva, M. C. Suárez-Figueroa, V. Presutti, I. Celino, M. Sabou, L. Kaffee, and E. Simperl, editors, *The Semantic Web - ISWC 2018 - 17th International Semantic Web Conference, Monterey, CA, USA, October 8-12, 2018, Proceedings, Part I*, volume 11136 of *Lecture Notes in Computer Science*, pages 583–599. Springer, 2018.

[52] O. Marcu, A. Costan, G. Antoniu, and M. S. Pérez-Hernández. Spark versus Flink: Understanding performance in big data analytics frameworks. In *2016 IEEE International Conference on Cluster Computing, CLUSTER 2016, Taipei, Taiwan, September 12-16, 2016*, pages 433–442. IEEE Computer Society, 2016.

[53] A. Margara, J. Urbani, F. van Harmelen, and H. E. Bal. Streaming the web: Reasoning over dynamic data. *J. Web Semant.*, 25:24–44, 2014.

[54] D. L. McGuinness, F. Van Harmelen, et al. Owl web ontology language overview. *W3C recommendation*, 10(10):2004, 2004.

[55] P. Mika and G. Tummarello. Web semantics in the clouds. *IEEE Intelligent Systems*, 23(5):82–87, 2008.

[56] B. Motik, Y. Nenov, R. E. F. Piro, and I. Horrocks. Incremental update of datalog materialisation: the backward/forward algorithm. In *Proceedings of the Twenty-Ninth AAAI Conference on Artificial Intelligence, January 25-30, 2015, Austin, Texas, USA.*, pages 1560–1568, 2015.

[57] H. Mühleisen and K. Dentler. Large-scale storage and reasoning for semantic data using swarms. *IEEE Comp. Int. Mag.*, 7(2):32–44, 2012.

[58] T. Neumann and G. Weikum. The RDF-3X engine for scalable management of RDF data. *VLDB J.*, 19(1):91–113, 2010.

[59] E. Oren, S. Kotoulas, G. Anadiotis, R. Siebes, A. ten Teije, and F. van Harmelen. Marvin: Distributed reasoning over large-scale semantic web data. *J. Web Sem.*, 7(4):305–316, 2009.

[60] P. Hayes (Ed.). RDF Semantics, W3C Recommendation. http://www.w3.org/TR/2004/REC-rdf-mt-20040210/, 10 February 2004.

[61] S. Perri, F. Ricca, and M. Sirianni. Parallel instantiation of ASP programs: techniques and experiments. *Theory Pract. Log. Program.*, 13(2):253–278, 2013.

[62] T. Pham, M. I. Ali, and A. Mileo. C-ASP: Continuous ASP-Based Reasoning over RDF Streams. In M. Balduccini, Y. Lierler, and S. Woltran, editors, *Logic Programming and Nonmonotonic Reasoning - 15th International Conference, LPNMR 2019, Philadelphia, PA, USA, June 3-7, 2019, Proceedings*, volume 11481 of *Lecture Notes in Computer Science*, pages 45–50. Springer, 2019.

[63] X. Ren and O. Curé. Strider: A hybrid adaptive distributed RDF stream processing engine. In *The Semantic Web - ISWC 2017 - 16th International Semantic Web Conference, Vienna, Austria, October 21-25, 2017, Proceedings, Part I*, pages 559–576, 2017.

[64] X. Ren, O. Curé, H. Naacke, and L. Ke. Strider-lsa: Massive RDF stream reasoning in the cloud. *CoRR*, abs/1708.06521, 2017.

[65] X. Ren, O. Curé, H. Naacke, and G. Xiao. RDF stream reasoning via answer set programming on modern big data platform. In M. van Erp, M. Atre, V. López, K. Srinivas, and C. Fortuna, editors, *Proceedings of the ISWC 2018 Posters & Demonstrations, Industry and Blue Sky Ideas Tracks co-located with 17th International Semantic Web Conference (ISWC 2018), Monterey, USA, October 8th - to - 12th, 2018*, volume 2180 of *CEUR Workshop Proceedings*. CEUR-WS.org, 2018.

[66] K. Rohloff and R. E. Schantz. High-performance, massively scalable distributed systems using the mapreduce software framework: the SHARD triple-store. In E. Tilevich and P. Eugster, editors, *SPLASH Workshop on Programming Support Innovations for Emerging Distributed Applications SPLASH/OOPSLA 2010), October 17, 2010, Reno/Tahoe, Nevada, USA*, page 4. ACM, 2010.

[67] A. Schätzle, M. Przyjaciel-Zablocki, T. Berberich, and G. Lausen. S2X: graph-parallel querying of RDF with graphx. In F. Wang, G. Luo, C. Weng, A. Khan, P. Mitra, and C. Yu, editors, *Biomedical Data Management and Graph Online Querying - VLDB 2015 Workshops, Big-O(Q) and DMAH, Waikoloa, HI, USA, August 31 - September 4, 2015, Revised Selected Papers*, volume 9579 of *Lecture Notes in Computer Science*, pages 155–168. Springer, 2015.

[68] A. Schätzle, M. Przyjaciel-Zablocki, S. Skilevic, and G. Lausen. S2RDF: RDF querying with SPARQL on spark. *PVLDB*, 9(10):804–815, 2016.

[69] M. Staudt and M. Jarke. Incremental maintenance of externally materialized views. In T. M. Vijayaraman, A. P. Buchmann, C. Mohan, and N. L. Sarda, editors, *VLDB'96, Proceedings of 22th International Conference on Very Large Data Bases, September 3-6, 1996, Mumbai (Bombay), India*, pages 75–86. Morgan Kaufmann, 1996.

[70] M. Stocker and E. Sirin. Pelletspatial: A hybrid rcc-8 and rdf/owl reasoning and query engine. In *OWLED*, volume 529, 2009.

[71] W. Sun, A. Fokoue, K. Srinivas, A. Kementsietsidis, G. Hu, and G. T. Xie. Sqlgraph: An efficient relational-based property graph store. In T. K. Sellis, S. B. Davidson, and Z. G. Ives, editors, *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data, Melbourne, Victoria, Australia, May 31 - June 4, 2015*, pages 1887–1901. ACM, 2015.

[72] H. J. ter Horst. Completeness, decidability and complexity of entailment for RDF schema and a semantic extension involving the OWL vocabulary. *J. Web Semant.*, 3(2-3):79–115, 2005.

[73] H. J. ter Horst. Completeness, decidability and complexity of entailment for rdf schema and a semantic extension involving the owl vocabulary. *Web Semantics: Science, Services and Agents on the World Wide Web*, 3(2-3):79–115, 2005.

[74] O. Udrea, A. Pugliese, and V. S. Subrahmanian. GRIN: A graph based RDF index. In *Proceedings of the Twenty-Second AAAI Conference on Artificial Intelligence, July 22-26, 2007, Vancouver, British Columbia, Canada*, pages 1465–1470. AAAI Press, 2007.

[75] J. Urbani, S. Kotoulas, J. Maassen, F. Van Harmelen, and H. Bal. WebPIE: A web-scale parallel inference engine using mapreduce. *Web Semantics: Science, Services and Agents on the World Wide Web*, 10:59–75, 2012.

[76] J. Urbani, S. Kotoulas, E. Oren, and F. van Harmelen. Scalable distributed reasoning using mapreduce. In *International Semantic Web Conference*, volume 5823 of *Lecture Notes in Computer Science*, pages 634–649. Springer, 2009.

[77] R. Volz, S. Staab, and B. Motik. Incremental maintenance of dynamic datalog programs. In R. Volz, S. Decker, and I. F. Cruz, editors, *PSSS1 - Practical and Scalable Semantic Systems, Proceedings of the First International Workshop on Practical and Scalable Semantic Systems, Sanibel Island, Florida, USA, October 20, 2003*, volume 89 of *CEUR Workshop Proceedings*. CEUR-WS.org, 2003.

[78] R. Volz, S. Staab, and B. Motik. Incremental maintenance of materialized ontologies. In R. Meersman, Z. Tari, and D. C. Schmidt, editors, *On The Move to Meaningful Internet Systems 2003: CoopIS, DOA, and ODBASE - OTM Confederated International Conferences, CoopIS, DOA, and ODBASE 2003, Catania, Sicily, Italy, November 3-7, 2003*, volume 2888 of *Lecture Notes in Computer Science*, pages 707–724. Springer, 2003.

[79] R. Volz, S. Staab, and B. Motik. Incrementally maintaining materializations of ontologies stored in logic databases. *J. Data Semantics*, 2:1–34, 2005.

[80] X. H. Wang, D. Q. Zhang, T. Gu, and H. K. Pung. Ontology based context modeling and reasoning using owl. In *Pervasive Computing and Communications Workshops, 2004. Proceedings of the Second IEEE Annual Conference on*, pages 18–22. Ieee, 2004.

[81] D. Warneke and O. Kao. Nephele: efficient parallel data processing in the cloud. In I. Raicu, I. T. Foster, and Y. Zhao, editors, *Proceedings of the 2nd Workshop on Many-Task Computing on Grids and Supercomputers, MTAGS 2009, November 16, 2009, Portland, Oregon, USA*. ACM, 2009.

[82] J. Weaver and J. A. Hendler. Parallel materialization of the finite RDFS closure for hundreds of millions of triples. In *International Semantic Web Conference*, volume 5823 of *Lecture Notes in Computer Science*, pages 682–697. Springer, 2009.

[83] C. Weiss, P. Karras, and A. Bernstein. Hexastore: sextuple indexing for semantic web data management. *PVLDB*, 1(1):1008–1019, 2008.

[84] R. S. Xin, J. E. Gonzalez, M. J. Franklin, and I. Stoica. Graphx: a resilient distributed graph system on spark. In P. A. Boncz and T. Neumann, editors, *First International Workshop on Graph Data Management Experiences and Systems, GRADES 2013, co-located with SIGMOD/PODS 2013, New York, NY, USA, June 24, 2013*, page 2. CWI/ACM, 2013.

[85] X. Yan and J. Han. gspan: Graph-based substructure pattern mining. In *Proceedings of the 2002 IEEE International Conference on Data Mining (ICDM 2002), 9-12 December 2002, Maebashi City, Japan*, pages 721–724. IEEE Computer Society, 2002.

[86] M. Zaharia, M. Chowdhury, T. Das, A. Dave, J. Ma, M. McCauly, M. J. Franklin, S. Shenker, and I. Stoica. Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing. In *NSDI*, pages 15–28. USENIX Association, 2012.

[87] M. Zaharia, T. Das, H. Li, T. Hunter, S. Shenker, and I. Stoica. Discretized streams: fault-tolerant streaming computation at scale. In *SOSP*, pages 423–438. ACM, 2013.

[88] L. Zou, J. Mo, L. Chen, M. T. Özsu, and D. Zhao. gStore: Answering SPARQL queries via subgraph matching. *Proc. VLDB Endow.*, 4(8):482–493, 2011.

RÉSUMÉ

À l'ère des Big Data, les données RDF sont produites en grand nombre. Bien qu'il existe des propositions de raisonnement sur de grands graphiques RDF utilisant de grandes plateformes de données, il y a un manque de solutions qui le font dans des environnements où les données RDF sont dynamiques, et où de nouvelles instances et de nouveaux triplets de schéma peuvent arriver à tout moment.

Dans cette thèse, nous présentons la première solution pour raisonner sur de grands flux de données RDF en utilisant de grandes plateformes de données. Ce faisant, nous nous concentrons sur l'opération de saturation, qui cherche à déduire des triples RDF implicites étant donné les contraintes du schéma RDF. En effet, contrairement aux solutions existantes qui saturent les données RDF en masse, notre solution identifie soigneusement le fragment de l'ensemble de données RDF existant (et déjà saturé) qui doit être pris en compte étant donné les nouvelles déclarations RDF délivrées par le flux. Ainsi, elle effectue la saturation de manière incrémentielle. L'analyse expérimentale montre que notre solution est plus performante que les solutions de saturation en masse existantes.

MOTS CLÉS

RDF Saturation, RDF Streams, Big Data, Apache Spark

ABSTRACT

In the Big Data era, RDF data are producing in high volumes. While there exist proposals for reasoning over large RDF graphs using big data platforms, there is a deficiency of solutions that do so in environments where RDF data are dynamic, and where new instance and schema triples can arrive at any time.

In this thesis, we present the first solution for reasoning over large streams of RDF data using big data platforms. In doing so, we focus on the saturation operation, which seeks to infer implicit RDF triples given RDF schema constraints. Indeed, unlike existing solutions which saturate RDF data in bulk, our solution carefully identifies the fragment of the existing (and already saturated) RDF dataset that needs to be considered given the fresh RDF statements delivered by the stream. Thereby, it performs the saturation in an incremental manner. Experimental analysis shows that our solution outperforms existing bulk-based saturation solutions.

KEYWORDS

RDF Saturation, RDF Streams, Big Data, Apache Spark