Scalable Saturation of Streaming RDF Triples

Mohammad Amin Farvardin¹(✉), Dario Colazzo¹, Khalid Belhajjame¹,
and Carlo Sartiani²

¹ Université Paris-Dauphine, PSL, Research University,
Place du Maréchal de Lattre de Tassigny, 75016 Paris, France
{mohammad-amin.farvardin,dario.colazzo,
khalid.belhajjame}@lamsade.dauphine.fr

² Università degli Studi della Basilicata, Potenza, Italy
carlo.sartiani@unibas.it
<https://www.dauphine.psl.eu>,
<https://www.unibas.it>

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 work, 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 or OWL 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.

1 Introduction

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. [4, 23, 27]). However, 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 [16]. 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 RDF data.

Supported by PSL, University Dauphine-Paris.

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 [5] and/or OWL, and aims at inferring the implicit triples that can be derived from D by using properties in S . 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 [15]) and platforms, notably Hadoop and Spark (see e.g., [11, 24]), have already been proposed. In [24] Urbani *et al.* presented WebPIE, a system for RDF data saturation relying on the Map-Reduce paradigm over Hadoop. In [11] 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 [19, 20] authors proposed a parallel reasoning method based on P2P self-organizing networks, while in [28] authors propose a parallel approach for RDF reasoning based on MPI. These approaches, however, assume that RDF datasets are fully available prior to 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. A number of works have addressed the problem of incremental saturation [3, 18, 26, 29], but these approaches, being mostly centralized, do not ensure a scalable, distributed, and robust RDF streaming saturation.

To overcome these limitations, in this work we present the first distributed technique for saturating *streams* of large RDF data, by relying on the Spark Streaming API, hence ensuring scalability and robustness. We present our approach in two steps. In the first one, we deal with streaming RDFS saturation in the presence of RDF Schema statements. The choice of 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. The main challenge here 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 [11, 24] 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 a specific indexing technique we devised 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 presentation, 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.

Paper Outline. The paper is structured as follows. Section 2 presents preliminaries about RDF saturation and Spark Streaming, while Sect. 3 presents an overview of our technique on RDFS by means of examples. In Sect. 4, we describe our extension of our technique for the OWL-Horst rule set, while Sect. 5 is dedicated to the performance evaluation of our approach. Sections 6 and 7, respectively, discuss related works and future perspectives.

The work presented in this paper is an extension of a previous conference paper [8]. The material in Sect. 4, which examines incremental saturation considering OWL-Horst rules is new, as is the evaluation of the effectiveness of saturation in the presence of OWL-Horst rules reported in Sect. 5. We also added in Sect. 3.3 a proof showing the soundness of our solution given the ordering of RDFS rules that we consider.

2 Preliminaries

2.1 RDF and Semantic Data Reasoning

An RDF dataset is a set of triples of the form $s p o$, where s is an IRI¹ or a blank node that represents the subject, p is an IRI that represents 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 [10]:

- *subclass relationship* $\prec_{sc}$: the triple $c_1 \prec_{sc} c_2$ specifies that c_1 is a subclass of c_2 ;
- *subproperty relationship* $\prec_{sp}$: the triple $p_1 \prec_{sp} p_2$ specifies that p_1 is a subproperty 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 .

¹ An IRI (*Internationalized Resource Identifier*) is just a URI exploiting Unicode in place of US Ascii as the character set, and is used in the semantic web community to identify resources.

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 1. Figure 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 `doi1` 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”.

Figure 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.

$$S = \{ \begin{array}{ll} \text{posterCP} \prec_{sc} \text{confP}, & \text{.b}_0 \prec_{sc} \text{confP}, \\ \text{confP} \prec_{sc} \text{paper}, & \text{hasTitle} \leftrightarrow_d \text{confP}, \\ \text{hasTitle} \hookrightarrow_r \text{rdfs:Literal}, & \text{hasAuthor} \leftrightarrow_d \text{paper}, \\ \text{hasAuthor} \hookrightarrow_r \text{rdfs:Literal}, & \text{hasContractA} \prec_{sp} \text{hasAuthor}, \\ \text{inProcessingOf} \leftrightarrow_d \text{confP}, & \text{inProcessingOf} \hookrightarrow_r \text{conference}, \\ \text{hasName} \leftrightarrow_d \text{conference}, & \text{hasName} \hookrightarrow_r \text{rdfs:Literal}, \\ \text{createdBy} \hookrightarrow_r \text{rdfs:Literal} \end{array} \}$$

Fig. 1. Instance and schema RDF triples.

$$G = \{ \begin{array}{l} \text{doi}_1 \tau \text{.b}_0, \quad \text{doi}_1 \text{ hasTitle } \text{“CAQU MV”}, \\ \text{doi}_1 \text{ hasAuthor } \text{“SA”}, \quad \text{doi}_1 \text{ hasContactA } \text{.b}_1, \\ \text{doi}_1 \text{ inProceedingsOf } \text{.b}_2, \quad \text{.b}_2 \text{ hasName } \text{“PODS’98”}, \\ \text{hasName createdBy } \text{“John Doe”}, \quad \text{“edbt2013”} \tau \text{Conference} \end{array} \}$$

Fig. 2. RDF graph representation of a conference paper.

As in other works (e.g., [10,11,24]) 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 1.

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 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 1 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 . Rule *rdfs9*, instead, 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 .

Table 1. RDFS rules.

Rule	Condition	Consequence	Level
<i>rdfs1</i>	$s p o$	$_ :b \ \tau \ \textit{rdfs:Literal}$	–
<i>rdfs2</i>	$p \leftrightarrow_d x, s p o$	$s \ \tau \ x$	Instance-level
<i>rdfs3</i>	$p \hookrightarrow_r x, s p o$	$o \ \tau \ x$	Instance-level
<i>rdfs4</i>	$s p o$	$s/o \ \tau \ \textit{rdfs:Resource}$	–
<i>rdfs5</i>	$p \prec_{sp} q, q \prec_{sp} r$	$p \prec_{sp} r$	Schema-level
<i>rdfs6</i>	$p \ \tau \ \textit{rdf:Property}$	$p \prec_{sp} p$	–
<i>rdfs7</i>	$s p o, p \prec_{sp} q$	$s q o$	Instance-level
<i>rdfs8</i>	$s \ \tau \ \textit{rdfs:Class}$	$s \prec_{sc} \textit{rdfs:Resource}$	–
<i>rdfs9</i>	$s \ \tau \ x, x \prec_{sc} y$	$s \ \tau \ y$	Instance-level
<i>rdfs10</i>	$s \ \tau \ \textit{rdfs:Class}$	$s \prec_{sc} s$	–
<i>rdfs11</i>	$x \prec_{sc} y, y \prec_{sc} z$	$x \prec_{sc} z$	schema-level
<i>rdfs12</i>	$p \ \tau \ \textit{rdfs:ContainerMembershipProperty}$	$p \prec_{sp} \textit{rdfs:member}$	–
<i>rdfs13</i>	$o \ \tau \ \textit{rdfs:Datatype}$	$o \prec_{sc} \textit{rdfs:Literal}$	–

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

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

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 $\{T1 \mid T2\} - \textit{rdfsX} \rightarrow t$, meaning that the tree derives t , by means of rule *rdfsX* 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 G and S previously introduced:

$$\{hasTitle \leftrightarrow_d \textit{confP} \mid doi_1 hasTitle \textit{"CAQUMV"}\} - \textit{rdfs2} \rightarrow doi_1 \ \tau \ \textit{confP}$$

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

$$\{T1 \mid \text{conf}P \prec_{\text{sc}} \text{paper}\} - \text{rdfs}9 \rightarrow \text{doi}_1 \tau \text{paper}$$

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):

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

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.

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

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

2.2 Spark and Spark Streaming

Spark [30] is a widely used in-memory distributed cluster computing framework. It provides the means for specifying DAG-based data flows using operators like `map`, `reduceByKey`, `join`, `filter`, etc. over data collections represented by means of *Resilient Distributed Datasets* (RDDs). For our purposes, we use the streaming capabilities of Spark whereby data come into micro-batches that need to be processed within a time-interval (also referred to as a window).

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.

Spark essentially works by applying operations to RDDs. These operations can be divided into *transformations* and *actions*. Transformations are lazy operations that return a new RDD, and are evaluated only when an action (e.g., `count`, `collect`) is invoked. Typical transformations are `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)`).

3 Streaming RDF Saturation

Our goal is to support the saturation of RDF streams by leveraging on Spark stream 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 $[\mathbf{mb}_1, \dots, \mathbf{mb}_n]$, where $i > 0$. A first approach for saturating such a stream using a batch-oriented solution would proceed as follows: when a micro-batch $\mathbf{mb}_i$ arrives, it unions $\mathbf{mb}_i$ 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 aim our saturation approach leverages on a novel indexing scheme for RDF triples, as well as on a few heuristics and optimization techniques.

3.1 Indexing Scheme

Our triple indexing structure allows the system to quickly retrieve a triple given its object or its property. This structure, which is stored on HDFS, comprises two root HDFS directories, called *o* and *p*.

Assume that a new micro-batch $\mathbf{mb}_i$ arrives at time *t*. At $\mathbf{mb}_i$ arrival time, the indexing algorithm creates a new subdirectory *t* inside *o*, as well as a new subdirectory *t* inside *p*. On *o/t* the algorithm stores triples having as predicate *rdf:type*, and, therefore, providing information about the type of a resource; inside *o/t* triples are further partitioned into files according to their actual object, so that triples with the same object are stored in the same file. Notice that triples with the *rdf:type* predicate are used in the premises of *rdfs9*. Given a schema triple of the form $y \prec_{sc} z$, 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*.

On *p/t* the algorithm stores the remaining instance triples of $\mathbf{mb}_i$ i.e., those that do not have *rdf:type* as a predicate. As in the previous case, triples inside *p/t* are partitioned according to their predicate, so that triples with the same predicate are stored in the same file.

Our indexing scheme also exploits two hash maps, stored in RDDs persisted in main memory, that map each object o_i to the corresponding HDFS files, as well as each property p_i to the HDFS files storing the matching triples; these hash maps contain no mapping for objects and properties without triples. 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 maps in order to retrieve only files containing triples with properties needed by these 3 rules.

To illustrate, assume for example that a new micro-batch $\mathbf{mb}_i$ arrives at a given time instant *t*, and that it contains the schema triple $t_{sc}: s_1 \prec_{sc} s_2$. 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 indexing mechanism we designed 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 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 map indexing the files under the $\mathbf{o}$ directories enables the fast recovering of files containing triples with $\mathbf{s}_1$ as an object resource, and that can be combined with the schema triple $\mathbf{t}_{sc}$ to trigger *rdfs9*. The indexing on files in $\mathbf{p}$ directories is exploited 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 more in detail, consider the following example.

Example 2. We assume that we have the initial schema $\mathbf{S}$ of Fig. 1 and that we saturate it by obtaining $\mathbf{S}'$ as indicated below.

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

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

1. The saturated schema $\mathbf{S}'$ is broadcast to each task, that 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. The stream of micro-batches is in Table 2.

Table 2. Instance triples.

<i>mb</i>	Subject	Predicate	Object
1	doi ₁	τ	$_ :b_0$
2	doi ₁	<i>hasTitle</i>	“CAQU MV”
3	doi ₁	<i>hasAuthor</i>	“SA”
4	doi ₁	<i>hasContactA</i>	$_ :b_1$
5	doi ₁	<i>inProceedingsOf</i>	$_ :b_2$
6	$_ :b_2$	<i>hasName</i>	“PODS’98”

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

$$\begin{aligned} \{doi_1 \tau _ :b_0 \mid _ :b_0 \prec_{sc} confP\} - rdfs9 &\rightarrow doi_1 \tau \quad confP \\ \{doi_1 \tau _ :b_0 \mid _ :b_0 \prec_{sc} paper\} - rdfs9 &\rightarrow doi_1 \tau \quad paper \end{aligned}$$

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 the **rdf:type** property, so as to obtain the following file assignment, knowing that t_1 is the time stamp for the current micro-batch:

Table 3. Saturated streaming triples.

mb_i	Received triple(s)	Schema-triple	Entails(E) & Received(R)	Stored path
1	$doi_1 \tau \neg b_0$	$\neg b_0 \prec_{sc} confP$, $\neg b_0 \prec_{sc} paper$	$E. doi_1 \tau confP$, $E. doi_1 \tau paper$, $R. doi_1 \tau \neg b_0$	$o/t_1/file_1$, $o/t_1/file_2$, $o/t_1/file_3$
2	$doi_1 hasTitle \text{“CAQU MV”}$	$hasTitle \leftrightarrow_d confP$	$E. doi_1 \tau confP$, $R. doi_1 hasTitle \text{“CAQU MV”}$	$o/t_2/file_1$, $p/t_2/file_1$ $p/t_2/file_1$
3	$doi_1 hasAuthor \text{“SA”}$	$hasAuthor \leftrightarrow_d paper$	$E. doi_1 \tau paper$, $R. doi_1 hasAuthor \text{“SA”}$	$o/t_3/file_1$, $p/t_3/file_1$
4	$doi_1 hasContactA \neg b_1$	no inference	$R. doi_1 hasContactA \neg b_1$	$p/t_4/file_1$
5	$doi_1 inProceedingOf \neg b_2$	$inProceesingOf \leftrightarrow_d confP$, $inProceesingOf \leftrightarrow_r conference$	$E. doi_1 \tau confP$, $E. \neg b_2 \tau conference$, $R. doi_1 inProceedingOf \neg b_2$	$o/t_5/file_1$, $o/t_5/file_2$, $p/t_5/file_1$
6	$\neg b_2 hasName \text{“PODS’98”}$	$hasName \leftrightarrow_d conference$	$E. \neg b_2 \tau conference$, $R. \neg b_2 hasName \text{“PODS’98”}$	$o/t_6/file_1$, $p/t_6/file_1$

$$\begin{aligned}
NST = \{ & \textit{paper} \prec_{sc} \textit{publication}, \textit{hasContactA} \prec_{sp} \textit{hasAuthor} \\
& \textit{posterCP} \prec_{sc} \textit{publication}, \textit{confP} \prec_{sc} \textit{publication}, \\
& \neg b_0 \prec_{sc} \textit{publication}, \textit{hasContactA} \leftrightarrow_d \textit{paper} \}
\end{aligned}$$

Fig. 3. Newly received and inferred Schema Triples (NST).

$$\begin{aligned}
doi_1 \tau confP &\Rightarrow o/t_1/file_1, \\
doi_1 \tau paper &\Rightarrow o/t_1/file_2, \\
doi_1 \tau \neg b_0 &\Rightarrow o/t_1/file_3
\end{aligned}$$

4. The processing goes on by deriving new instance triples for the micro-batches from 2 to 6, as indicated in Table 3, which also indicates how instance triples are stored/indexed.

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

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

So we have now three steps: **i)** infer the new schema triples by considering the already present schema triples, **ii)** broadcast these schema triples minus the already existing/broadcast schema triples (Fig 3), to enable tasks to locally access them, **iii)** re-process previously met/inferred instance triples by taking into consideration the new schema.

Consider for instance $\{hasContactA \prec_{sp} hasAuthor\}$ as new schema triple. This schema triple triggers *rdfs7*. Therefore, our indexing tells us that *only* file $p/t_4/file_1$ (Table 3, line 4) needs to be loaded to infer new triples, that, of course, will be in turn stored according to our indexing strategy.

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 app-

Algorithm 1. 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 RDDs in memory.
4: Output:  $oIndexingRDD$ ,  $pIndexingRDD$ 
5: Begin
6: // Get a fixed timestamp to save the  $mb'_i$  triples.
7: val  $fts$  = System.currentTimeMillis.toString

8: // The  $mb'_i$  triples partitions by their object where their predicate is rdf:type.
9: val  $oPartition$  =  $mb'_i$ .filter(!_2.contains("rdf-syntax-ns#type")).
10:   map( $t \Rightarrow (t._3, t._1)$ ).partitionBy(number of different object).
11:   mapPartitions(_map( $t \Rightarrow (t._2, t._1)$ ))

12: // The  $mb'_i$  triples partitions by their predicate where their predicate is NOT rdf:type.
13: val  $pPartition$  =  $mb'_i$ .filter(!_2.contains("rdf-syntax-ns#type")).
14:   map( $t \Rightarrow (t._2, t)$ ).partitionBy(number of different predicate).
15:   mapPartitions(_map(_2))

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: //  $oIndexingRDD$  is a HashTable which keeps the object of instance triple as key and their
    physical paths as value.
20:  $oIndexingRDD \cup = oPartition.mapPartitionsWithIndex((index, iterator) \Rightarrow \{$ 
21:    $iterator.map(t \Rightarrow (t._2, fts + "-" + index + "-")) \}$ ).mapPartitions( $\{$ 
22:    $_map(t \Rightarrow (t,1))\}.reduceByKey(_+_).$ mapPartitions(_map(_1))

23: //  $pIndexingRDD$  is a HashTable which keeps the predicate of instance triple as key and
    their physical paths as value.
24:  $pIndexingRDD \cup = pPartition.mapPartitionsWithIndex((index, iterator) \Rightarrow \{$ 
25:    $iterator.map(t \Rightarrow (t._2, fts + "-" + index + "-")) \}$ ).mapPartitions( $\{$ 
26:    $_map(t \Rightarrow (t,1))\}.reduceByKey(_+_).$ mapPartitions(_map(_1))

27: return  $oIndexingRDD$  &  $pIndexingRDD$ 
28: End

```

roach tends to create a non-negligible number of files, but fortunately without compromising efficiency thanks to distribution.

The indexing algorithm (Algorithm 1) is responsible for storing on HDFS at the intended paths, but also for collecting the object/predicate of triples and their paths for indexing variable. We 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 1 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 very first one (*line 7*), are processed in a parallel manner.

3.2 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 schema² 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 new triples describing the vocabulary used in an RDF graph. Furthermore, because the size of the schema is small, schema saturation can be done in a centralized fashion. In this respect, the RDFS rules presented in Table 1 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, specify resource instances of the classes in the RDF vocabularies and their relationships. Each rule is

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

Fig. 4. The optimized order of execution for RDFS. 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. (Color figure online)

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 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 order of application of rules illustrated in Fig. 4.

Rule Pruning for Schema Saturation. Given a new micro-batch $\mathbf{mb}_i$, we filter all the 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*. Therefore, 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

$$NST = \text{Transitive Closure} (\text{new received schema} \cup \text{past schema}) - \text{past schema}$$

The Saturation operation is local and only triggers rules that do need to be applied, in the right order. Table 4 illustrates the rules to be activated given some matching schema triple: the number 1 indicates the availability of a matching schema triple, and 0 indicates it is not. For example, if a schema triple specifying the domain of a property exists, then this triggers rule 2. All possible cases are

Table 4. The 1 and 0 indicate for the availability of that particular schema rules in $\mathbf{mb}_i$. $X \rightarrow Y$ means: the output of rule X used as an input of rule Y .

	subPropertyOf	domain	range	subClassOf	Saturation order
1	1	1	1	1	$R7 \rightarrow (R2, R3) \rightarrow R9$
2	1	1	1	0	$R7 \rightarrow (R2, R3)$
3	1	1	0	1	$R7 \rightarrow R2 \rightarrow R9$
4	1	1	0	0	$R7 \rightarrow R2$
5	1	0	1	1	$R7 \rightarrow R3 \rightarrow R9$
6	1	0	1	0	$R7 \rightarrow R3$
7	1	0	0	1	$R7, R9$
8	1	0	0	0	$R7$
9	0	1	1	1	$(R2, R3) \rightarrow R9$
10	0	1	1	0	$R2, R3$
11	0	1	0	1	$R2 \rightarrow R9$
12	0	1	0	0	$R2$
13	0	0	1	1	$R3 \rightarrow R9$
14	0	0	1	0	$R3$
15	0	0	0	1	$R9$
16	0	0	0	0	—

indicated in Table 4, 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 $\mathbf{mb}_i$ 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 $\mathbf{mb}_i$ and the pre-existing schema.

Efficiently Saturate Existing Instance Triples by Leveraging on our Incremental Indexing Scheme. Given the new schema triples that are provided by the micro-batch $\mathbf{mb}_i$ 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 1. This operation can be costly as it involves examining all the instance triples that have been provided and inferred micro-batches received before $\mathbf{mb}_i$. To alleviate this problem, we exploit the incremental indexing scheme of the previous section; this technique allows for the fast retrieval of the instance triples that will likely trigger the RDFS rules given some schema triples. 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 triples. This is because the rules for inferring new schema triples require two schema triples as a premise (see Table 1).

Incremental Loading. As we previously observed, our indexing technique may lead to the creation of a huge number of files on HDFS, which in turn may increase the risk of a failure when Spark must so many files at once. Therefore, we addressed this reliability issue by loading the index files incrementally (i.e., 150 files per time), until all files have been loading, and then by *unioning* the tuples inside them.

Saturate New Instance Triples. The instance triples inferred in (2) need to be examined 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 $\mathbf{mb}_i$ as well as those inferred in (2) and (3) are stored and indexed using the technique described in Sect. 3.1.

3.3 Streaming Saturation Algorithm

The overall streaming saturation algorithm is shown in Algorithm 2, and commented hereafter.

Given a micro-batch $\mathbf{mb}_i$, we first perform schema saturation if $\mathbf{mb}_i$ contains schema triples (*lines 12, 13*). The related instance triples are retrieved based on $\mathbf{mb}_{\text{NST}}$ (*line 14*). Given newly inferred schema triples, instance triples are retrieved and examined to identify cases where new instance triples may be inferred (*line 15*). The obtained schema triples (i.e., $\mathbf{mb}_{\text{NST}}$) are added and broadcast within the initial schema RDD (*line 17, 18*). The inferred triples, if any, are merged with instance triples of $\mathbf{mb}_i$ (i.e., $\mathbf{mb}_{\text{ins}}$) and saturation is applied to them. In the next step, the received and inferred instance triples are combined and obtained duplicates, if any, are removed (*line 22*). In the last step, the instance triples from the previous step are saved and indexed using our method (*line 24–25*).

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 completeness of our system as well as of WebPIE [24] and Cichlid [11], and reflects rule ordering expressed in Fig. 4. To illustrate the lemma, assume we have $D = \{s \ \tau \ c_1\}$ while the schema includes four triples of the form $c_i \prec_{\text{sc}} c_{i+1}$, for $i = 1 \dots 4$. Over D and S we can have the tree T1 corresponding to:

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

Algorithm 2. Overall Algorithm for Saturating RDF Stream

```

1: Input:  $MB \leftarrow [mb_1, \dots, mb_n]$  // a stream of micro-batches.
2: Output:  $Schemas \leftarrow [Sch_1, \dots, Sch_n]$  //  $Sch_i$  represents the schema triples obtained as a
   result of saturating the micro-batches  $MB = [mb_1, \dots, mb_i]$ .
3: Output:  $Datasets \leftarrow [DS_1, \dots, DS_n]$  //  $DS_i$  represents the instance triples obtained as a
   result of saturating the micro-batches  $MB = [mb_1, \dots, mb_i]$ .
4: Output:  $IndexInformation \leftarrow [oIndex, pIndex]$  //  $oIndex$  and  $pIndex$  keeps object- and
   predicate-based information respectively.
   Begin
5:    $D_{ins} \leftarrow \emptyset$  // Initialize a dataset for instance triples
6:    $D_{sc} \leftarrow \emptyset$  // Initialize a dataset for schema triples
7:    $br \leftarrow$  if  $D_{sc}$  exist then TransitiveClosure and broadcast them
8:   do {
9:     // Separate schema from instance triples in  $mb$ 
      $(mb_{sch}, mb_{ins}) \leftarrow \text{SeparatingTriples}(mb_i)$ 

10:    if  $(mb_{sch} \neq \emptyset)$  then {
11:      // Retrieve the already saturated instance triples and re-saturate them based on com-
        bination of received and existing RDFS triples
12:       $mb'_{sch} \leftarrow (\text{TransitiveClosure}(mb_{sch} \cup D_{sc})) - D_{sc}$ 
13:       $mb_{NST} \leftarrow \text{broadcast}(mb'_{sch})$ 
14:       $D'_{ins} \leftarrow$  Retrieve instance triples to be saturated by using  $mb_{NST}$ 
15:       $mb'_i \leftarrow \text{Saturate}(D'_{ins}, mb_{NST})$ 

16:      // Combine received and existing RDFS triples and re-broadcast them
17:       $D_{sc} \leftarrow mb_{NST} \cup D_{sc}$ 
18:       $br \leftarrow \text{broadcast}(D_{sc})$  // The total schema so far received.
19:    }

20:    // Saturate the received instance triples with total RDFS triples
21:     $mb_{imp} \leftarrow \text{Saturate}(mb_{ins} \cup mb'_i, br)$ 
22:     $mb'_i \leftarrow (mb_{ins} \cup mb'_i \cup mb_{imp}).\text{distinct}$ 

23:    // The following two lines are handled by Indexing Algorithm
24:    Save  $mb''_i$  in the HDFS
25:     $[oIndex, pIndex] \cup \leftarrow \text{indexing}(mb''_i)$ 
26:  } while (is there an incoming micro-batch  $mb$ ?)
27: End

```

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 *rdfs9*. 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 1. *Given an RDF dataset 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. To prove the above lemma, we examine the dependencies between the rules $\{2, 3, 5, 7, 9, 11\}$. A rule r depends on a rule r' where possibly r and r' are the same rule, if the activation of r' produces a triple that can be used as a premise for the activation of 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: indeed, rule 7 uses as a premise triples of the form $p \prec_{sp} q$, which are produced by the activation of rule 5;
- Rules 2 and 3 depend on rule 7: both rules 2 and 3 use 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 5.

Fig. 5. RDFS rule dependencies.

Figure 5 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 need 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 derivation tree T there exists an equivalent derivation tree T' as described in the lemma thesis. 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 eventually derived by rule 9 (resp. rule 7) 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 a schema triple in S^* , obtained by subsequent applications of rule 11 (resp. rule 9). So, in the case $(*)$ holds, the just described rewriting for chains of rule 9 (resp. rule 7) can be applied to T in order to obtain T' . $\square$

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

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

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 $\mathbf{mb}_1$, 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 triples in D_1 and schema triples in S_1^* . As $D_1 \subseteq D$ and $S_1^* \subseteq S^*$, we have that this derivation tree can derive t also from D and for S . Assume now t is derived by our system when processing the micro-batch $\mathbf{mb}_j$ with $j > 1$. Triple t is derived by a derivation tree T possibly using triples t' derived in $\mathbf{mb}_h$ 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 corresponding 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 1 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 derivation tree $T = \{T1 \mid T2\} - \text{rdfs}X \rightarrow t$. The base case is that both $T1$ and $T2$ are simple triples $t1$ and $t2$ 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 $\mathbf{mb}_h$, with $h \leq j$. This j exists by hypothesis, and we have that either $t1$ or $t2$ is in $\mathbf{mb}_j$. Assume it is $t1$, a schema triple and that $t2$ has been met in $\mathbf{mb}_s$ 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 when only one of $T1$ and $T2$ is a triple is similar). By Lemma 1, we have that there exists an equivalent $T' = \{T1' \mid T2'\} - \text{rdfs}Y \rightarrow t$ such that instance rules are used a most once (in the order of Fig. 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 $\mathbf{mb}_h$ such that $t2$ is obtained by schema saturation (which is globally kept in memory) and that there exists $\mathbf{mb}_s$ in which $t1$ is derived and indexed by our algorithm. Now consider $j = \max(s, h)$. At step j both $t1$ (indexed) and $t2$ (in the main memory) are available to our algorithm, which can hence produce $\{t1 \mid t2\} - \text{rdfs}Y \rightarrow t$.

The remaining cases are similar. □

4 Extension to Streaming OWL-Horst Saturation

This section is devoted to RDF streaming saturation in the presence of OWL-Horst ontology [13]. For space reason, we do not present OWL-Horst in detail, and assume the reader is already familiar with it.

In recent years, OWL-Horst has gained consistent attention by both research and industrial communities, as it represents a good balance between expressivity and computational tractability. The rules are reported in Table 5, and, as it can be seen, OWL-Horst is much more expressive than RDFS. However, the techniques we have developed for RDFS saturation still remain effective in the context of OWL-Horst rules, but the transposition is not direct, and we had to take particular care of the integration of RDFS and OWL-Horst saturation in the presence of streaming instance and schema data. However, for each single OWL-Horst rule the extension of our RDFS approach is almost direct, validating the effectiveness of our previously introduced indexing technique.

As already observed in [11], 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 (Fig. 5). 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: $\mathcal{G}_1^{\text{sh}}$, $\mathcal{G}_2^{\text{sh}}$ and $\mathcal{G}_3^{\text{sh}}$, each of which is composed of the following rules:

- $\mathcal{G}_1^{\text{sh}} = \{\text{OWL-Horst}(9, 10, 12_{(a,b)}, 13_{(a,b)}) + \text{RDFS}(5, 11)\}$
- $\mathcal{G}_2^{\text{sh}} = \{\text{RDFS}(7, 2, 3, 9)\}$
- $\mathcal{G}_3^{\text{sh}} = \{\text{Rules}(14_{(a,b)}, 3, 8_{(a,b)}, 15, 16, 4, 1, 2)\}$

The first group contains OWL-Horst rules together with the two RDFS rules that produce schema triples, viz RDFS5 and RDFS11. The OWL-Horst in the group can be applied in any order once, however RDFS rules 5 and 7 need to be applied multiple times until a fix-point is reached. The second group $\mathcal{G}_2^{\text{sh}}$ contains RDFS rules that use as premise triples that are produced by the rules in $\mathcal{G}_1^{\text{sh}}$. The third group $\mathcal{G}_3^{\text{sh}}$, on the other hand, is independent of the other two groups. The above analysis suggests the following order of application of rules given newly acquired schema triples: the rules in group $\mathcal{G}_1^{\text{sh}}$ need to be applied before applying those in group $\mathcal{G}_2^{\text{sh}}$, whereas the rules in the third group $\mathcal{G}_3^{\text{sh}}$ can be applied in parallel to those in $\mathcal{G}_1^{\text{sh}}$ and $\mathcal{G}_2^{\text{sh}}$.

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 Fig. 6:

- $\mathcal{G}_1^{\text{ins}} = \{\text{RDFS}(7, 2, 3, 9)\}$
- $\mathcal{G}_2^{\text{ins}} = \{\text{OWL-Horst}(14_{(a,b)}, 3, 8_{(a,b)})\}$

Table 5. OWL-Horst rule set. Schemas are indicated by *italic* font.

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

- $G_3^{\text{ins}} = \{\text{OWL-Horst}(15, 16, 4)\}$
- $G_4^{\text{ins}} = \{\text{OWL-Horst}(15, 16, 4)\}$
- $G_5^{\text{ins}} = \{\text{OWL-Horst}(1, 2, 7, 11)\}$

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 for step B5). 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 $\mathcal{G}_2^{\text{ins}}$ groups, in the sense that the triples produced by these can be used as a premise by the rules in $\mathcal{G}_3^{\text{ins}}$. Also, the rules in $\mathcal{G}_1^{\text{ins}}$ depends on those in $\mathcal{G}_3^{\text{ins}}$. Notice that this introduces a loop between the three first groups. The rules in the forth group $\mathcal{G}_4^{\text{ins}}$ depend on those triples that are produced by the last two schema groups ($\mathcal{G}_2^{\text{sch}}$ and $\mathcal{G}_3^{\text{sch}}$) and the first three instance groups ($\mathcal{G}_1^{\text{ins}}$, $\mathcal{G}_2^{\text{ins}}$, and $\mathcal{G}_3^{\text{ins}}$) plus those triples that are received by the current micro-batch. Since the rules in the fourth group $\mathcal{G}_4^{\text{ins}}$ needs two instance triples to trigger the rules, thus, the objective of the fourth group $\mathcal{G}_4^{\text{ins}}$ is to find the complement part of the received instance triples from DS' by assuming that both schema triples plus one of the instance triples exist. The analysis of the rules in the fifth group, $\mathcal{G}_5^{\text{ins}}$, 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 $\mathcal{G}_5^{\text{ins}}$, as already assumed by Cichlid [11]. It is worth recalling that *sameAs* saturation, performed by $\mathcal{G}_5^{\text{ins}}$, 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 [24] and then re-used by Cichlid [11], 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 [11, 24], 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 14_a 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 design an algorithm, which is depicted in Fig. 6, for 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 [11], with the notable difference that we strive to perform the saturation incrementally.

As shown in Fig. 6, 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).

Fig. 6. The global overview of saturation process on OWL-horst rules.

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 are applied. In A1.1, once the new schema triples are derived, these ones plus the new schema triples in the current micro-batch (the old schema rules are not used here) are used for RDFS instance triple saturation, as happened for our algorithm for RDFS saturation (Sect. 3). The novelty in step A is that now we have step A2 applying OWL-Horst rules for instance triple derivation, by using new schema triples plus the old ones, and by using our indexing approach to retrieve needed instance triples derived in the past (stored on HDFS). For instance, for rule 2, once the indicated schema triple identifies a property p , then we use this to retrieve, by means of our indexing scheme, only the triples having p as property and then we perform the join required by the second and third premise (note that such kind of joins does not occur for RDFS saturation). In this way the number of triples involved in the join operation is sensibly reduced. 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 are 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 triples still computed and broadcast in step A. The main part of step B consists of a loop for iterating saturation until a fix point 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) involving rules that could be triggered by RDFS derivation in step B1; in case these two steps produce new triples, then step B3 uses those triples for applying OWL-Horst rules 15, 16 and 4. Once the fix point of the loop is reached, step B4 needs to fetch instance triples from HDFS by using our indexing technique, once in the driver the existence of schema triples to trigger rules (e.g., rule 15) is detected. 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.
4. Rules that take two schema triples and two instance triples and produce an instance triple, namely OWL-Horst rules 15 and 16.

Algorithm 3. OWL-Horst *Rule 15* (Step A2)

```

1: Input:  $mb_{sch}$  – Received schema triples
2: 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  $mb$ 
3: Output: results – reasoning results
4: Begin
5:   // Extract owl:onProperty and owl:someValuesFrom schema from  $mb_{sch}$ 
6:   val  $new\_op =$ 
      $mb_{sch}.filter(t \Rightarrow t..2.equals("owl:onProperty")).map(t \Rightarrow (t..1, t..3)).collect.toSet$ 
7:   val  $new\_sv =$ 
      $mb_{sch}.filter(t \Rightarrow t..2.equals("owl:someValuesFrom")).map(t \Rightarrow (t..1, t..3)).collect.toSet$ 

   // Rule 15 won't trigger if there is no new schema triple arrives
8:   if( $new\_op.isEmpty$  &&  $new\_sv.isEmpty$ ) then
9:     return empty

   // Keep those schema triples if the second match of schema is arrived too.
10:  val  $op_1 = findMatches(new\_op, new\_sv)$ 
11:  val  $sv_1 = findMatches(new\_sv, new\_op)$ 

   // Also find other matches among the previous schema based on the current schemas
12:  val  $op_2 = findMatches(op', new\_sv)$ 
13:  val  $sv_2 = findMatches(sv', new\_op)$ 

14:  val  $op = (op_1.toSeq ++ op_2.toSeq)$ 
15:  val  $sv = (sv_1.toSeq ++ sv_2.toSeq)$ 

   // There is a guarantee that for every new op, at least, a new and/or old sv/sv' exists and
   so on for every new sv too
16:  if( $(op_1.toSeq ++ op_2.toSeq).isEmpty \vee (sv_1.toSeq ++ sv_2.toSeq).isEmpty$ ) then
17:    return empty

   // Retrieve related instance triples from  $DS'$  by relying on indexing information
18:  val  $pTriples =$  Fetch triples from predicate-based paths of  $DS'$  based on the  $op$ 
19:  val  $oTriples =$  Fetch triples from object-based paths of  $DS'$  based on the  $sv$ 

   // Saturation process among the fetched triples
20:  val  $in_{15} = pTriples.map(t \Rightarrow ((op.value(t..2), t..3), t..1))$ 
21:  val  $t_{15} = oTriples.map(t \Rightarrow ((sv.value(t..2), t..1), Nil))$ 
22:   $results = in_{15}.join(t_{15}).map(t \Rightarrow (t..2..1, t..1..1))$ 

23:  return results
24: End

```

Rules in (1) and (2) can be implemented similarly to the RDFS rules presented earlier. Rules in (3) can be implemented straightforwardly 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 rule 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: $\Rightarrow$ 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 Fig. 6), given an instance triple (see Box B3 in Fig. 6), or based on received a new instance triple (see Box B4 in Fig. 6). Algorithm 3 details the processing of rule 15 (Fig. 6 Step A2), when

Algorithm 4. OWL-Horst *Rule 15* (Step B3)

```

1: Input:  $mb_{inst}$  – Instance Triples of micro-batch
2: Input:  $op'Swap$  and  $sv'Swap$  – Two datasets that represent schema triple with owl:onProperty
   and owl:someValuesFrom as predicate respectively. i.e., Swap means (Object, Subject)
3: Output:  $results$  – Reasoning results
4: Begin
   // Extract the given triples and types based on the entire (so far) received schema.
5: val  $potential\_Triples = mb_{inst}.filter(t \Rightarrow op'Swap.value.contains(t._2))$ 
6: val  $potential\_Types =$ 
    $mb_{inst}.filter(t \Rightarrow t._2.equals("rdf:type") \&\& sv'Swap.value.contains(t._3))$ 

7: val  $tr_{15} = potential\_Triples.map(t \Rightarrow ((op'Swap.value(t._2), t._3), t._1))$ 
8: var  $ty_{15} = potential\_Types.map(t \Rightarrow ((sv'Swap.value(t._2), t._1), Nil)).persist$ 
9: val  $r_{15} = tr_{15}.join(ty_{15}).map(t \Rightarrow (t._2._1, t._1._1))$ 

10: return  $r_{15}$ 
11: End

```

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 tries to find their match. For example, if a newly acquired triple is an *onProperty* triple, e.g., $(v_1 \text{ owl:onProperty } p_1)$, then the algorithm attempts to find a matching triples, e.g., $(v_1 \text{ owl:someValuesFrom } w)$, from received and already existing 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 4 details the processing of rule 15 given the received and inferred instance triples. This algorithm 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 triggers 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 be used into another round of saturation process if the previous round of the saturation process infers new triples. Otherwise, the saturation process jumps out of the loop and goes to the next saturation step (Step B4).

Algorithm 5. OWL-Horst *Rule 15* (Step B4)

```

1: Input:  $\mathbf{mb}_{\text{inst}/\text{inf}}$  – Instance and inferred triples based on the current  $\mathbf{mb}_i$ .
2: Input:  $T_{\text{sch}}$  – The total schema that received so far
3: Output:  $\text{results}$  – Reasoning results
4: Begin
5:   // Extract the respective schema triples from the total schema
6:   val  $sv_{\text{set}} = T_{\text{sch}}.\text{filter}(t \Rightarrow t._2.\text{equals}(\text{"owl:someValuesFrom"})).\text{map}(t \Rightarrow (t._1, t._3))$ 
7:   val  $op_{\text{set}} = T_{\text{sch}}.\text{filter}(t \Rightarrow t._2.\text{equals}(\text{"owl:onProperty"})).\text{map}(t \Rightarrow (t._1, t._3))$ 

8:   // Choose all someValuesFrom if exists any related onProperty for that and vice versa.
9:   val  $svs = \text{findMatches}(sv_{\text{set}}, op_{\text{set}})$ 
10:  val  $ops = \text{findMatches}(op_{\text{set}}, sv_{\text{set}})$ 

11:  // Assuring that both schemas exist
12:  if( $svs.\text{isEmpty} \parallel ops.\text{isEmpty}$ ) return empty

13:  // Broadcast the  $svs$  and  $ops$  and the swap version of them, i.e., (Object, Subject)
14:  val  $sv_{br} = \text{broadcast}(svs.\text{toMap})$ 
15:  val  $op_{br} = \text{broadcast}(ops.\text{toMap})$ 

16:  val  $svSwap_{br} = \text{broadcast}(svs.\text{map}(t \Rightarrow (t._2, t._1)).\text{toMap})$ 
17:  val  $opSwap_{br} = \text{broadcast}(ops.\text{map}(t \Rightarrow (t._2, t._1)).\text{toMap})$ 

18:  // Filter types, find triples with a matching signature, fetch triples from disk, saturation
19:  process
20:    val  $\text{CandidatedTypes} =$ 
21:       $\mathbf{mb}_{\text{inst}/\text{inf}}.\text{filter}(t \Rightarrow svSwap_{br}.\text{value}.\text{contains}(t._3) \ \&\& \ t._2.\text{equals}(\text{"rdf:type"}))$ 
22:    val  $vs_1 = \text{CandidatedTypes}.\text{map}(t \Rightarrow svSwap_{br}.\text{value}(t._2)).\text{distinct}$ 
23:    val  $ps_1 = vs_1.\text{map}(t \Rightarrow op_{br}.\text{value}(t)).\text{collect.toList}$ 
24:    val  $\text{relatedPs} = \text{Fetch triples from predicate-based paths based on } ps_1$ 

25:    val  $tr_{15\_1} = \text{relatedPs}.\text{map}(t \Rightarrow ((opSwap_{br}.\text{value}(t._2), t._3), t._1))$ 
26:    val  $ty_{15\_1} = \text{CandidatedTypes}.\text{map}(t \Rightarrow ((svSwap_{br}.\text{value}(t._2), t._1), \text{Nil})).\text{persist}$ 
27:    val  $r_{15\_1} = tr_{15\_1}.\text{join}(ty_{15\_1}).\text{map}(t \Rightarrow (t._2._1, t._1._1))$ 

28:    // Filter types, find triples with a matching signature, fetch types from disk, saturation process
29:    val  $\text{CandidatedTriples} = \mathbf{mb}_{\text{inst}/\text{inf}}.\text{filter}(t \Rightarrow opSwap_{br}.\text{value}.\text{contains}(t._2))$ 
30:    val  $ps_2 = \text{CandidatedTriples}.\text{map}(t \Rightarrow opSwap_{br}.\text{value}(t._2)).\text{distinct}$ 
31:    val  $vs_2 = ps_2.\text{map}(t \Rightarrow sv_{br}.\text{value}(t)).\text{distinct.collect.toList}$ 
32:    val  $\text{relatedTs} = \text{Fetch types from object-based paths based on } vs_2$ 

33:    val  $tr_{15\_2} = \text{newP}.\text{map}(t \Rightarrow ((opSwap_{br}.\text{value}(t._2), t._3), t._1))$ 
34:    val  $ty_{15\_2} = \text{relatedTs}.\text{map}(t \Rightarrow ((svSwap_{br}.\text{value}(t._2), t._1), \text{Nil}))$ 
35:    val  $r_{15\_2} = tr_{15\_2}.\text{join}(ty_{15\_2}).\text{map}(t \Rightarrow (t._2._1, t._1._1))$ 

36:    // Return the results of both phases
37:    return  $r_{15\_1}.\text{union}(r_{15\_2})$ 
38: End

```

Algorithm 5 aims at finding one of the bipartite instance triples of rule 15 from 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 instance triples received and/or inferred via current micro-batch. Therefore as a first step, we extract the related schema triple (*owl:onProperty* and *owl:someValuesFrom*), that are required for rule 15, among the schema triples 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 for *owl:onProperty*

triples returns corresponding *owl:someValuesFrom* triples and vice-versa (lines 7–8). In the next step, by considering that both schema triples 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 $\text{mb}_{\text{inst}/\text{inf}}$ 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), those triples that have the same predicate as the *owl:onProperty*'s object. 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 objects and 83 different 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 $\text{mb}_{\text{inst}/\text{inf}}$. The Algorithm 5, lines 21–24, is dedicated to the same process (lines 14–20) except to find the right *rdf:type* triples with corresponding objects. For this purpose, we fetch those triples from the object-based triples located on the disk, in DS' . Finally, we apply the saturation process between over the selected and fetched triples by considering both matched schema triples. Finally, the results of the saturation processes (i.e., r_{15-1} and r_{15-2}) are concatenated and returned (line 28).

5 Evaluation

The saturation method we have just presented³ 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 or OWL-Horst ruleset is also obtained in a stream-based fashion.

To validate this claim and to understand to which extent our method outperforms its competitors, we performed an empirical evaluation on real-life RDF datasets. The results of this evaluation are shown in the next sections.

5.1 Datasets

We used for our experiments three RDF datasets for RDFS saturation that are widely used in the semantic web community: DBpedia [2], LUBM [12], and dblp⁴

³ <https://git.lamsade.fr/afarvardin/RDFInStream>.

⁴ Computer science bibliography (<https://dblp.uni-trier.de/faq/What+is+dblp.html>).

Since these three datasets do not have any OWL-Horst schema triples, we choose a portion of UniProt[7] for OWL-Horst saturation

These datasets are not stream-based datasets, and therefore we had to partition them into micro-batches to simulate a setting where the data are received in a streamed manner. We make in our experiments the assumption that a substantial part of the data is received initially and that micro-batches arrive then in a streaming fashion. We consider this to be a realistic assumption, in those scenarios where a substantial part of the data is known initially, and new triples arrive as time goes by. In what follows, and for space sake, we report on the experiments we ran against DBpedia for RDFS and UniProt for OWL-Horst saturation.

Using DBpedia, we created three stream-based datasets DBpedia-100, DBpedia-200, and DBpedia-300. They are composed of initial chunks that contain 100, 200, and 300 million instance triples respectively, and a series of 15 micro-batches, each composed 160K triples plus between 64 and 2500 schema triples. For the initial chunk we reserve 25% of schema triples, while the remaining ones are spread over the micro-batches as indicated above. Regarding saturation using OWL-Horst ruleset, we used the *UniProt* dataset, which contains 320 million triples and occupies 49.6 GB. For evaluation purposes, we partition the Uniprot datasets into micro-batches. In doing so, we set the size of micro-batches to 512 MB. The schema triples of Uniprot (549 triples) are divided equally between the micro-batches. Thus, each micro-batch has a range of [5–6] schema triples.

5.2 Experiment Setup

In the case of RDFS saturation, for each of the above datasets, we ran our saturation algorithm initially for the first chunk, and then incrementally for each remaining micro-batch. For comparison purposes, for each of the above datasets, we ran the Cichlid algorithm on the initial chunk, and then on each of the micro-batches. Given that Cichlid is not incremental, for each micro-batch, we had to consider the previous micro-batches and the initial chunk as well as the current micro-batch.

Alike RDFS datasets in the OWL-Horst dataset, every micro-batch contains schema triples. Therefore, Cichlid needs to reload every past micro-batches by receiving a new micro-batch.

We performed our experiments on a cluster with 4 nodes (and 8 nodes (check the extended version [9])), connected with 1 Gbps Ethernet. One node was reserved to act as the master node and the remaining 3 nodes as worker nodes. Each node has a Xeon O2.4 GHz processor, 48 GB memory, and 33 TB Hadoop file system, and runs Linux Debian 9.3, Spark 2.1.0, Hadoop 2.7.0, and Java 1.8.

For each dataset we ran our experiment 5 times, and reported the average running time.

5.3 Results

Saturation Considering RDF Schema Rules

Figure 7 shows the results obtained when saturating 300 million triples from the DBpedia dataset. The x-axis represents the initial chunk and the micro-batches that composed the dataset. For the initial chunk, the y-axis reports the time required for its saturation. For each of the following micro-batches, the y-axis reports the time required for saturating the dataset composed of the current micro-batch, the previous micro-batches, and the initial chunk put together.

Fig. 7. DBpedia 300 million triples - run on 4 nodes.

The figure shows that the time required by Cichlid for saturating the stream increases substantially as the number of micro-batches increases, and is significantly higher than the one required by our algorithm. Specifically, the saturation takes more than 1000min given the last micro-batch, that is 22 times the amount of time required to saturate the first micro-batch, namely 45 min. On the other hand, our incremental algorithm takes almost the same time for all micro-batches. Specifically, it takes 41 min given the first micro-batch, and 78 min given the last micro-batch.

We obtained similar trends using other datasets: the dblp dataset (see Fig. 8), and the LUBM dataset (see Fig. 9); these datasets have smaller sizes wrt to the DBpedia one (190M and 69M, respectively).

Fig. 8. DBLP 190M - run on 4 nodes. **Fig. 9.** DBLP 190M - run on 4 nodes.

The good performance of our algorithm is due to its incremental nature, but also to its underlying indexing mechanism. To demonstrate this, Fig. 10 illustrates for DBpedia, and for each micro-batch, the number of triples that are fetched using 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 the whole amount of triples to be loaded). As it can be observed, the number of triples fetched by the index is a small fraction of the total number of triples that compose the dataset.

Fig. 10. DBpedia 300M - retrieved triples when receiving new schema triples.

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

Figure 11 illustrates the average time required for performing the saturation given a micro-batch (blue line), and the average time required for the index management (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 100 MB micro-batch is smaller compared to the time required for processing micro-batches with larger sizes. The increase is not steady. In particular, we observe that micro-batches with 400 MB and 500 MB require the same processing 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 400 MB micro-batches.

Regarding the index management time (red line), our experiment shows that it is significantly small with respect to the saturation time, and it costs in the worse case less than half a minute. Besides the time-interval, the configuration of the cluster impacts stream saturation. As shown in Fig. 11, 500 MB micro-batches require the same time as 400 MB micro-batches for maintaining the index.

Concerning global execution time (for all micro-batches), experiments showed that, when the number of micro-batches decreases, this time can decrease in some cases (this happens in particular for $i \in \{1, 2, 3\}$, see [9] for details, Table 5).

To summarize, the results we presented here show that it is possible to saturate streams of RDF data in an incremental manner by using big data platforms, and that our approach outperforms the state of the art.

Fig. 11. Average processing and indexing management time/micro-batch. (Color figure online)

Saturation Considering RDF Schema and OWL-Horst Rules

Figure 12 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. The figure also shows for both approaches the exponential trendline. 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 6) per micro-batch. The y-axis reports the execution time required for saturation of each micro-batch in seconds.

Figure 12 shows that the time required by Cichlid to saturate the dataset (depicted using a red line) increases substantially as the number of micro-batches does. Furthermore, Cichlid fails to saturate the dataset starting from the 18th micro-batch. This is in contrast to our incremental solution (depicted using a blue line), which manages to efficiently saturate the dataset given the received micro-batches. Specifically, we observe that the time required for saturation varies slightly between micro-batches, and is far smaller than the time required by Cichlid, especially in later iterations.

Fig. 12. UniProt 320 million triples - comparison with Cichlid (4 nodes).

Table 6. Types and numbers of schema triple per micro-batch in Fig. 12.

μ_b	Schemas
μ_b0	<code>owl(onProperty → 1), rdfs(domain → 2, range → 1, subClassOf → 2)</code>
μ_b1	<code>owl(allValuesFrom → 1), rdfs(domain → 1, range → 2, subClassOf → 2)</code>
μ_b2	<code>owl(onProperty → 1), rdfs(range → 2, subPropertyOf → 1, subClassOf → 2)</code>
μ_b3	<code>owl(allValuesFrom → 1), rdfs(domain → 3, subClassOf → 2)</code>
μ_b4	<code>rdfs(domain → 1, range → 2, subClassOf → 3)</code>
μ_b5	<code>owl(onProperty → 1), rdfs(range → 1, domain → 1, subPropertyOf → 1, subClassOf → 2)</code>
μ_b6	<code>owl(inverseOf → 1, allValuesFrom → 1), rdfs(range → 2, subClassOf → 2)</code>
μ_b7	<code>owl(onProperty → 1, equivalentClass → 1), rdfs(range → 1, domain → 2, subClassOf → 1)</code>
μ_b8	<code>owl(onProperty → 1, equivalentClass → 1), rdfs(range → 1, domain → 1, subClassOf → 2)</code>
μ_b9	<code>owl(onProperty → 1), rdfs(domain → 1, subClassOf → 3)</code>
μ_b10	<code>owl(onProperty → 1), rdfs(range → 2, subClassOf → 3)</code>
μ_b11	<code>rdfs(range → 2, domain → 1, subClassOf → 3)</code>
μ_b12	<code>owl(equivalentClass → 1), rdfs(domain → 3, subClassOf → 2)</code>
μ_b13	<code>owl(onProperty → 2), rdfs(range → 1, domain → 2, subClassOf → 1)</code>
...	...
μ_b82	<code>owl(hasValue → 1), rdfs(range → 1, domain → 1, subPropertyOf → 1, subClassOf → 1)</code>
μ_b83	<code>rdfs(range → 1, domain → 2, subPropertyOf → 1, subClassOf → 2)</code>
...	...
μ_b100	<code>owl(allValuesFrom → 1), rdfs(domain → 2, range → 1, subClassOf → 2)</code>

Figures 12 illustrates that using Cichlid, the saturation of the first 17 micro-batches takes 1086 min, while our solution takes 15mn to process those batches. Our solution takes 257 min (which is still far smaller than the time required by Cichlid to process the initial 17 micro-batches) to process the entire dataset, which consists of 100 micro-batches.

Fig. 13. UniProt 320 million triples - incremental streaming (4 nodes).

Figure 13 illustrates the time required for each step of our incremental solution using 4 nodes. The blue line illustrates the total time required to process each micro-batch. The green line illustrates the time required to process the schema (Step A in Fig. 6). The yellow line illustrates the time required to process the instance triples within the micro-batch (Step B in Fig. 6). The purple line illustrates the time required for index management, which consists of partitioning, compressing (using a default Spark compressor, i.e., *gzip*), storage time, in addition to collecting data information for our indexing technique from the saturated micro-batch. Finally, the gray line illustrates the time required to detect and retrieve data from the existing dataset DS' (this time is embedded in both steps A and B). The figure shows that the total fetching time (the gray line) for the entire process is 61 min. That is 23.7% of the total processing time, which is 257 min.

On average, our indexing technique takes 37s to detect and fetch the necessary triples from DS' (stored on disk) when new schema triples are received. Notice that in Fig. 13, we have a spike on the micro-batch 84 (mb_{84}). To find out the reason for this leap, we compared what happens during the saturation of micro-batch 82 (mb_{82}), which takes 224 s, with what happens during the saturation of micro-batch mb_{84} , which takes instead 593 s.

- *Fetching time:* The fetching time for mb_{84} is 87 s, while it is 51 s for the mb_{82} . That time corresponds to the retrieval of almost 111 and 31 million potential RDF triples (object- and predicate-based triples) from DS' for mb_{84} and mb_{82} , respectively. Those triples are retrieved from 1075 files for mb_{84} , and 964 files for 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 mb_{84} takes 60 s (including the required fetching times), while it takes 25 s for mb_{82} .
 - *Step B*: The time required in Step B, including fetching time for this step, is 314 s for mb_{84} , while it is 142 s for mb_{82} .
 - *Indexing Management*: The indexing management time for mb_{84} is 219 s. This time is divided into 187 and 32 s for partitioning, indexing, and saving the data into object- and the predicate-based triples, respectively. The indexing management time for mb_{82} is 58 s that consists of 27 and 31 s 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 mb_{84} , we 110 K predicate-based triples 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 mb_{82} 111 K predicate-based triples 36 K object-based triples in every iteration. To a considerable extent, this explains the difference in the processing time required for saturating data in mb_{84} compared to time needed for mb_{82} .

Fig. 14. Fetches triples per mb. UniProt 320 million triples.

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 14 depicts the results. It shows that our method retrieves far smaller numbers of triples compared with the Cichlid.

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 `mb84`, we fetch 111 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.

6 Related Work

RDF Saturation 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 [17], 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., [11, 24, 25]). For example, Urbani *et al.* [24, 25] 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. Moreover, they specified for each of the RDFS rule how it can be implemented using map and/or reduce functions, and executed over the Hadoop system. Building on the work by Urbani *et al.*, the authors of Cichlid [11] implemented RDF saturation over Spark using, in addition to map and reduce, other transformations that are provided by Spark, such as filter, union, etc. Cichlid has shown that the use of Spark can speed up saturation wrt the case when Hadoop is used. Our solution builds on and adapts the techniques proposed by WebPie and Cichlid to cater for the saturation of streams of massive RDF data.

Incremental Saturation. The problem of incremental saturation of RDF data has been investigated by a number of proposals (see e.g., [3, 6, 10, 24, 26]). For example, Volz *et al.* 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 [26]. In doing so, they adapted a previous state of the art algorithm for incremental view maintenance proposed in the context of deductive database [22]. Barbieri *et al.* [3] builds on the solution proposed by Volz *et al.* by considering the case where the triples are associated with an expiration date in the context of streams (e.g., for data that are 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.* and Barbieri *et al.* [3] 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 Volz *et al.* in the results of their evaluation study, even if the maintenance was incremental, the inference engine ran out of memory in certain cases. Regarding, Barbieri *et al.* [3], they considered in their evaluation a single transitive

rule (Sect. 5 in [3]), and did not report on the size of the dataset used, nor the micro-batch size.

Chevalier *et al.* proposed Slider, a system for RDF saturation using a distributed architecture [6]. Although the objective of 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 [1] for indexing RDF triples. This indexing structure is heavy since it creates a table for each property in the RDF. 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.

Guasdoué *et al.* proposed an incremental solution for saturating RDF data [10]. 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 work, we order the rules in a way to minimize the number of iterations required for saturating the RDF data.

The authors of WebPie [24] briefly touched on the problem of incrementally saturating RDF data. In doing so, they time-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 .

To sum up, compared with the existing state of the art in incremental saturation 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 micro-batches of RDF data.

Indexing Structures for RDF Data. The indexing mechanism we proposed here is comparable to those proposed by Weiss *et al.* [29], by Schätzle *et al.* [21] and by Kaoudi [14] *et al.* 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 $\{p_1^i, \dots, p_n^i\}$. Moreover, each property is associated with a sorted list representing the objects. 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.* [21], 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 between all properties. As shown by the authors, the computation of such indexes is heavy, e.g., it requires 290 s to index 100 million triples. To alleviate this, we proposed here an index that is aimed at speeding up RDF saturation, as opposed to any SPARQL query, and that is amenable to incremental maintenance.

7 Conclusion and Future Work

In this paper we presented a solution for incrementally saturating streams of massive RDF datasets considering RDFS rules and OWL-Horst rules. In our solution, we strive to cater for the incremental processing of the saturation operation. To do so, we make use of an indexing scheme that allows us to retrieve only the instance triples that are necessary for saturation given a newly arrived micro-batch. We have shown that our approach and techniques are effective for both RDFS and OWL-Horst rules and outperform the state of the art solution, viz Cichlid. As future work, we would like to extend our algorithm for query answering problem, for which we believe that our technique could still entail possible optimizations.

References

1. Abadi, D.J., Marcus, A., Madden, S., Hollenbach, K.J.: Scalable semantic web data management using vertical partitioning. In: Proceedings of the 33rd International Conference on Very Large Data Bases, University of Vienna, Austria, 23–27 September 2007, pp. 411–422. ACM (2007). <http://www.vldb.org/conf/2007/papers/research/p411-abadi.pdf>
2. Auer, S., Bizer, C., Kobilarov, G., Lehmann, J., Cyganiak, R., Ives, Z.: DBpedia: a nucleus for a web of open data. In: Aberer, K., et al. (eds.) ASWC/ISWC -2007. LNCS, vol. 4825, pp. 722–735. Springer, Heidelberg (2007). https://doi.org/10.1007/978-3-540-76298-0_52
3. Barbieri, D.F., Braga, D., Ceri, S., Valle, E.D., Grossniklaus, M.: 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, 30 May–3 June 2010, Proceedings, Part I, pp. 1–15 (2010). https://doi.org/10.1007/978-3-642-13486-9_1
4. Bizer, C., Heath, T., Berners-Lee, T.: Linked data-the story so far. *Int. J. Semant. Web Inf. Syst.* **5**(3), 1–22 (2009)
5. Brickley, D., Guha, R.: RDF Schema 1.1. Technical report W3C (2014). W3C Recommendation
6. Chevalier, J., Subercaze, J., Gravier, C., Laforest, F.: Slider: an efficient incremental reasoner. In: Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data, Melbourne, Victoria, Australia, 31 May–4 June 2015, pp. 1081–1086 (2015). <https://doi.org/10.1145/2723372.2735363>

7. Consortium, T.U.: UniProt: a worldwide hub of protein knowledge. *Nucleic Acids Res.* **47**(D1), D506–D515 (2018). <https://doi.org/10.1093/nar/gky1049>
8. Farvardin, M.A., Colazzo, D., Belhajjame, K., Sartiani, C.: Streaming saturation for large RDF graphs with dynamic schema information. In: *Proceedings of the 17th ACM SIGPLAN International Symposium on Database Programming Languages, DBPL 2019, Phoenix, AZ, USA, 23 June 2019*, pp. 42–52. ACM (2019). <https://doi.org/10.1145/3315507.3330201>
9. Farvardin, M.A., Colazzo, D., Belhajjame, K., Sartiani, C.: Streaming saturation for large RDF graphs with dynamic schema information, February 2019. <https://www.lamsade.dauphine.fr/~afarvardin/submissions/2019/technical-paper.pdf>
10. Goasdoué, F., Manolescu, I., Roatis, A.: Efficient query answering against dynamic RDF databases. In: *Proceedings of the 16th International Conference on Extending Database Technology*, pp. 299–310. ACM (2013)
11. Gu, R., Wang, S., Wang, F., Yuan, C., Huang, Y.: Cichlid: efficient large scale RDFS/OWL reasoning with spark. In: *2015 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pp. 700–709. IEEE (2015)
12. Guo, Y., Pan, Z., Hefflin, J.: LUBM: a benchmark for OWL knowledge base systems. *J. Web Sem.* **3**(2–3), 158–182 (2005)
13. ter Horst, H.J.: Completeness, decidability and complexity of entailment for RDF schema and a semantic extension involving the owl vocabulary. *Web Semant. Sci. Serv. Agents WWW* **3**(2–3), 79–115 (2005)
14. Kaoudi, Z., Miliaraki, I., Koubarakis, M.: RDFS reasoning and query answering on top of DHTs. In: Sheth, A., et al. (eds.) *ISWC 2008. LNCS*, vol. 5318, pp. 499–516. Springer, Heidelberg (2008). https://doi.org/10.1007/978-3-540-88564-1_32
15. Leskovec, J., Rajaraman, A., Ullman, J.D.: *Mining of Massive Datasets*. Cambridge University Press, Cambridge (2014)
16. Liew, C.S., Atkinson, M.P., Galea, M., Ang, T.F., Martin, P., Hemert, J.I.V.: Scientific workflows: moving across paradigms. *ACM Comput. Surv. (CSUR)* **49**(4), 66 (2017)
17. Mika, P., Tummarello, G.: Web semantics in the clouds. *IEEE Intell. Syst.* **23**(5), 82–87 (2008)
18. Motik, B., Nenov, Y., Piro, R.E.F., Horrocks, I.: Incremental update of datalog materialisation: the backward/forward algorithm. In: *Proceedings of the Twenty-Ninth AAAI Conference on Artificial Intelligence*, 25–30 January 2015, Austin, Texas, USA, pp. 1560–1568 (2015). <http://www.aaai.org/ocs/index.php/AAAI/AAAI15/paper/view/9660>
19. Mühleisen, H., Dentler, K.: Large-scale storage and reasoning for semantic data using swarms. *IEEE Comp. Int. Mag.* **7**(2), 32–44 (2012)
20. Oren, E., Kotoulas, S., Anadiotis, G., Siebes, R., ten Teije, A., van Harmelen, F.: Marvin: Distributed reasoning over large-scale semantic web data. *J. Web Semant.* **7**(4), 305–316 (2009)
21. Schätzle, A., Przyjaciół-Zablocki, M., Skilevic, S., Lausen, G.: S2RDF: RDF querying with SPARQL on spark. *PVLDB* **9**(10), 804–815 (2016). <https://doi.org/10.14778/2977797.2977806>. <http://www.vldb.org/pvldb/vol9/p804-schaetzle.pdf>
22. Staudt, M., Jarke, M.: Incremental maintenance of externally materialized views. In: *VLDB*, pp. 75–86. Morgan Kaufmann (1996)
23. Stocker, M., Sirin, E.: Pelletspatial: a hybrid RCC-8 and RDF/OWL reasoning and query engine. In: *OWLED*, vol. 529 (2009)
24. Urbani, J., Kotoulas, S., Maassen, J., Van Harmelen, F., Bal, H.: Webpie: a web-scale parallel inference engine using mapreduce. *Web Semant.: Sci. Serv. Agents WWW* **10**, 59–75 (2012)

25. Urbani, J., Kotoulas, S., Oren, E., van Harmelen, F.: Scalable distributed reasoning using MapReduce. In: Bernstein, A., et al. (eds.) ISWC 2009. LNCS, vol. 5823, pp. 634–649. Springer, Heidelberg (2009). https://doi.org/10.1007/978-3-642-04930-9_40
26. Volz, R., Staab, S., Motik, B.: Incrementally maintaining materializations of ontologies stored in logic databases. *J. Data Semant.* **2**, 1–34 (2005). https://doi.org/10.1007/978-3-540-30567-5_1
27. Wang, X.H., Zhang, D.Q., Gu, T., Pung, H.K.: Ontology based context modeling and reasoning using owl. In: Proceedings of the Second IEEE Annual Conference on Pervasive Computing and Communications Workshops, 2004, pp. 18–22. IEEE (2004)
28. Weaver, J., Hendler, J.A.: Parallel materialization of the finite RDFS closure for hundreds of millions of triples. In: Bernstein, A., et al. (eds.) ISWC 2009. LNCS, vol. 5823, pp. 682–697. Springer, Heidelberg (2009). https://doi.org/10.1007/978-3-642-04930-9_43
29. Weiss, C., Karras, P., Bernstein, A.: Hexastore: sextuple indexing for semantic web data management. *PVLDB* **1**(1), 1008–1019 (2008). <https://doi.org/10.14778/1453856.1453965>. <http://www.vldb.org/pvldb/1/1453965.pdf>
30. Zaharia, M., et al.: Resilient distributed datasets: a fault-tolerant abstraction for in-memory cluster computing. In: NSDI, pp. 15–28. USENIX Association (2012)