SPOWL: Spark-based OWL 2 Reasoning Materialisation

Yu Liu
Department of Computing
Imperial College London
yu.liu11@imperial.ac.uk

Peter McBrien
Department of Computing
Imperial College London
p.mcbrien@imperial.ac.uk

ABSTRACT

This paper presents SPOWL, which uses Spark to perform OWL reasoning over large ontologies. SPOWL acts as a compiler, which maps axioms in the T-Box of an ontology to Spark programmes, which will be executed iteratively to compute and materialise a closure of reasoning results entailed by the ontology. Such a closure is then available to queries which retrieve information from the ontology. Compared to MapReduce, adopting Spark enables SPOWL to cache data in the distributed memory, to reduce the amount of I/O used, and to also parallelise jobs in a more flexible manner. We further analyse the dependencies among the Spark programmes, and propose an optimised order following the T-Box hierarchy, which makes the materialising process terminate with minimum iterations. Moreover, SPOWL uses a tableaux reasoner to classify the T-Box, and the classified axioms are complied into Spark programmes which are directly related to the ontological data under reasoning. This not only makes the reasoning by SPOWL more complete, but also avoids processing unnecessary rules, as compared to evaluating certain rulesets adopted by most state-of-the-art reasoners. Finally, since SPOWL materialises the reasoning closure for large ontologies, it processes queries retrieving ontology information faster than computing the query answers in real time.

CCS Concepts

•Computing methodologies → Ontology engineering; Distributed computing methodologies;

Keywords

Spark, OWL 2 Reasoning, Scalability, Completeness

1. INTRODUCTION

Approaches to reasoning over large-scale ontologies (e.g. Freebase [9], UniProt [5] and DBpedia [2]) have been researched over the recent years. In order to efficiently process queries over ontologies, reasoners such as Oracle RDF Store [32], SQOWL2 [18] and WebPIE [31] often adopt a materialised approach, which computes and stores a closure of derivations entailed by the ontologies. Such a closure is ready to be read by queries, which often results in faster query processing compared to using the query-rewriting approach adopted by reasoners such as Stardog [25] and Ontop [4]. This is due to query rewriting requires answers be computed at the time of each query execution.

Materialisation-based reasoners usually compute and materialise the reasoning closure by evaluating a set of entailment rules. Well-known rulesets include RDFS entailment rules [23] and OWL ter Horst rules [30], both of which cover some features of OWL 2 RL. The recent OWL 2 standard also releases the OWL 2 RL/RDF rules [22], which cover more complex reasoning than the former two rulesets. To illustrate the evaluating process, we consider a fragment of the **Lehigh University Benchmark** (**LUBM**) [11] composed of the following T-Box axioms[1]:

$$\mathsf{Student} \sqsubseteq \mathsf{Person} \quad (1)$$

$$\mathsf{Student} \sqsubseteq \exists \mathsf{takesCourse}.\mathsf{Course} \quad (2)$$

where (1) expresses that every Student is a Person, and (2) specifies a Student takes at least one Course. We may further assert some A-Box facts that John and Tom are two individuals of the class Student by (3) and (4), and Lewis and Mary are individuals of Person by (5) and (6):

$$\mathsf{Student}(\mathsf{John}) \quad (3) \qquad \mathsf{Person}(\mathsf{Lewis}) \quad (5)$$

$$\mathsf{Student}(\mathsf{Tom}) \quad (4) \qquad \mathsf{Person}(\mathsf{Mary}) \quad (6)$$

We may start the evaluation process over the LUBM fragment (1)–(6) by considering the below entailment rule in the format of **if** ⟨antecedent⟩ **then** ⟨consequent⟩:

$$\textbf{if } C \sqsubseteq D, C(x) \textbf{ then } D(x)$$

which specifies that individuals x of C will be inferred as instances of D if C is a subclass of D. Evaluating this rule checks as to whether there are some ontological statements matching the ⟨antecedent⟩; if so, statements defined in ⟨consequent⟩ are obtained as new derivations. Obviously, the T-Box axiom (1) and A-Box facts (3) and (4) match the ⟨antecedent⟩ of the above entailment rule; therefore, two additional A-Box facts Person(John) and Person(Tom) can be derived by instantiating the above rule, i.e.:

$$\textbf{if } \mathsf{Student} \sqsubseteq \mathsf{Person}, \mathsf{Student}(x) \textbf{ then } \mathsf{Person}(x)$$

Therefore, the class Person will not only explicitly include Lewis and Mary, but also implicitly contain John and Tom.

However, the rule evaluation process suffers some draw-

BeyondMR'17, May 19, 2017, Chicago, IL, USA

ISBN 978-1-4503-5019-8/17/05…$15.00

DOI: http://dx.doi.org/10.1145/3070607.3070609

[1]We adopt the syntax of DL in this paper for a neat representation.

backs. First, the process usually avoids the use of tableaux reasoning [3], even the T-Box of ontologies is small enough for tableaux reasoners (e.g. Pellet [29] and Hermit [28]) to handle. Tableaux reasoners are known to provide complete T-Box reasoning w.r.t. the T-Box, and totally avoiding them might result in less complete reasoning [21]. For example, for two given T-Box axioms, $C \sqsubseteq D \sqcup E$ and $C \sqcap D \sqsubseteq \bot$, a tableaux reasoner is able to infer the subsumption of $C \sqsubseteq E$, which cannot be derived via evaluating the RDFS entailment rules, OWL ter Horst rules, or even the OWL 2 RL/RDF rules.

Second, reasoning via the rule evaluation process totally relies on which set of entailment rules is chosen, and they often sacrifice too much reasoning completeness for the sake of scalable materialisation. For example, no rules in the three aforementioned rulesets handle the T-Box axiom (2), because it brings non-deterministic reasoning, as the expression $\exists$takesCourse.Course is used as a superclass expression. From (2), if an individual x is asserted to be a Student, we only know the property takesCourse relates x to at least one individual y of Course but unable to determine which one. Indeed, the RL profile [22] of OWL 2 does not allow this case to avoid non-determinism. However, results of retrieving individuals which are related by takesCourse (i.e. the subjects of takesCourse) will be incomplete.

Third, the evaluating process often considers large ontologies as whole sets of RDF triples [6], and statements matching the ⟨antecedent⟩ of an entailment rule need to be filtered out every time when evaluating this rule. This filtering process often leads to an issue of inefficient rule matching [14], which slows down the performance of reasoning materialisation, and even query processing (as the materialised results are often subject to queries, which retrieve fragments of the materialisation).

In this paper, we provide our approach named SPOWL, which resolves the above three issues by extending the previous work [17, 20], which handles OWL 2 reasoning in an RDBMS, to now perform OWL 2 reasoning materialisation in a Big Data system. Taking the assumption that the ontology under reasoning has a small T-Box and large A-Boxes, our approach has the following features:

- SPOWL combines a tableaux reasoner for T-Box classification, and generates a set of entailment rules from the classified T-Box. Using a tableaux reasoner not only gives us a complete T-Box reasoning w.r.t. a given T-Box, but also ensures that the entailment rules are only relevant to axioms contained in the T-Box. This completely avoids evaluating entailment rules unrelated to a given large ontology.
- The SPOWL ruleset not only covers OWL 2 RL/RDF rules (except where the **Unique Name Assumption** (**UNA**) conflicts, as SPOWL adopts the UNA), but also contains extra rules for handling some cases which surpass OWL 2 RL. SPOWL is conjectured to be a sound and complete implementation of the OWL 2 RL/RDF rules for OWL 2 RL ontologies [16]. These extra rules result in a more complete materialisation of reasoning; for example, SPOWL handles the case of setting an existential quantification as a superclass expression.
- SPOWL compiles the entailment rules to programmes written in Spark [13]. These programmes are executed iteratively over ontological data to compute and materialise the reasoning results, until no further derivation can be inferred. We also analyse the dependencies among these programmes, and determine an optimised order of executing them, in order to minimise the number of iterations until Spark programme execution can terminate (i.e. when no further reasoning results can be derived).
- In order to avoid repeated filtering, SPOWL inherits the schema used by previous work for representing class and property facts, i.e. we separate instances related to each class or property, and stores them individually. Consequently, when computing reasoning or querying information over a fragment of the ontology, less filtering effort is required.

The remainder of this paper is organised as follows. Section 2 provides an overview of how SPOWL translates a classified T-Box into Spark programmes. Section 3 describes an optimised order for executing these Spark programmes, so that the iteration times of executing them can be minimised, alongside with some tuning strategies to further optimise the reasoning performance. Section 4 evaluates the performance of our approach on the LUBM benchmark. Finally, Section 5 summarises this paper.

2. SPOWL OVERVIEW

The section outlines SPOWL. We illustrate how Spark programmes are generated from a classified T-Box and then applied to the loaded data, so that the results of reasoning can be computed and materialised.

2.1 SPOWL Architecture

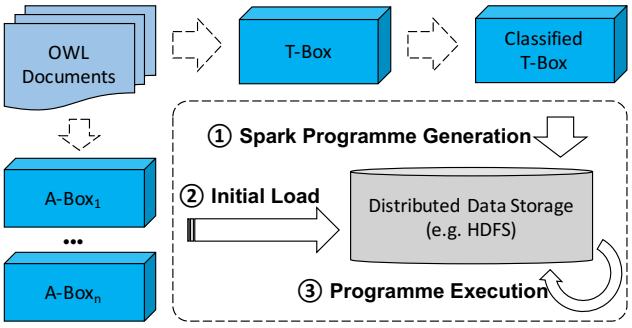

Figure 1: SPOWL Architecture

Figure 1 shows the architecture of SPOWL, which performs reasoning materialisation in three steps:

1. After performing classification using some tableaux reasoners, SPOWL transforms the classified T-Box into a set of entailment rules, which are compiled into Spark programmes.
2. Explicit A-Box facts are loaded into a distributed storage system. The current SPOWL prototype only supports **Hadoop Distributed File System** (**HDFS**) for storing data, but this can be extended to any other distributed storage systems supported by Spark.

3. SPOWL iteratively executes the Spark programmes over the loaded data until no new reasoning can be made, and the results of reasoning are computed and persisted in the HDFS. In particular, the order of executing Spark programmes follows the bottom-up hierarchy of the T-Box, minimising the number of iterations required.

2.2 Outline of the Approach

In order to illustrate the above three steps, we extend the fragmental LUBM ontology (1)–(6) used in Section 1 by the following extra T-Box axioms:

$$\mathsf{GraduateStudent} \sqsubseteq \mathsf{Person} \tag{7}$$

$$\mathsf{GraduateCourse} \sqsubseteq \mathsf{Course} \tag{8}$$

$$\mathsf{GraduateStudent} \sqsubseteq \exists \mathsf{takesCourse}.\mathsf{GraduateCourse} \tag{9}$$

$$\mathsf{Person} \sqcap \exists \mathsf{takesCourse}.\mathsf{Course} \sqsubseteq \mathsf{Student} \tag{10}$$

$$\mathsf{subOrganisationOf} \circ \mathsf{subOrganisationOf} \sqsubseteq \mathsf{subOrganisationOf} \tag{11}$$

Note that Axiom (9) specifies that a GraduateStudent takes at least one GraduateCourse. Axiom (10) expresses that a Person who takes at least one Course is a Student. Axiom (11) states that subOrganisationOf is transitive. We also extend the ontology with the following A-Box data:

$$\mathsf{GraduateStudent}(\mathsf{Jack}) \tag{12}$$

$$\mathsf{Course}(\mathsf{Database}) \tag{13}$$

$$\mathsf{GraduateCourse}(\mathsf{Algorithm}) \tag{14}$$

$$\mathsf{takesCourse}(\mathsf{Tom}, \mathsf{Database}) \tag{15}$$

$$\mathsf{takesCourse}(\mathsf{John}, \mathsf{Algorithm}) \tag{16}$$

$$\mathsf{subOrganisationOf}(\mathsf{Group}, \mathsf{Department}) \tag{17}$$

$$\mathsf{subOrganisationOf}(\mathsf{Department}, \mathsf{College}) \tag{18}$$

$$\mathsf{subOrganisationOf}(\mathsf{College}, \mathsf{University}) \tag{19}$$

2.2.1 *T-Box Classification by Tableaux Reasoner*

SPOWL applies a tableaux reasoner for the T-Box classification, in order to obtain a more complete set of subsumption relations. For example, by classifying axioms (7) – (10), a new subsumption relationship from GraduateStudent to Student can be derived:

$$\mathsf{GraduateStudent} \sqsubseteq \mathsf{Student} \tag{20}$$

It is worth mentioning that because axiom (9) (which uses $\exists$takesCourse.GraduateCourse as a superclass expression) does not meet the OWL 2 RL restrictions, it is not handled by the OWL 2 RL/RDF rules. Consequently, evaluating the OWL 2 RL/RDF rules over the above fragmental LUBM ontology cannot infer (20). Moreover, axioms in the classified T-Box form the schema which is specially for the A-Box of data, and therefore, SPOWL only consider those axioms for reasoning materialisation, rather than evaluating every entailment rule in a ruleset even if some rules are irrelevant.

2.2.2 *Classes & Properties to Spark RDDs*

Spark treats collections of data as **Resilient Distributed Datasets** (**RDDs**). We assume individuals of a class C are stored in an unary RDD $C_{rdd}(id)$; and pairs of individuals connected by a property P are stored in a binary RDD $P_{rdd}(domain, range)$. All RDDs will be initialised to include instances explicitly asserted to the classes or properties which they represent, and they will be eventually built up with implicit instances computed from Spark programmes. For instance, the A-Box facts we have asserted so far for the fragmental LUBM (i.e. (3)–(6) and (12)–(16)) will lead to the initialisation of the following RDDs:

$$\mathsf{Student}_{rdd} = \{\mathsf{John}, \mathsf{Tom}\}$$

$$\mathsf{Person}_{rdd} = \{\mathsf{Lewis}, \mathsf{Mary}\}$$

$$\mathsf{GraduateStudent}_{rdd} = \{\mathsf{Jack}\}$$

$$\mathsf{Course}_{rdd} = \{\mathsf{Database}\}$$

$$\mathsf{GraduateCourse}_{rdd} = \{\mathsf{Algorithm}\}$$

$$\mathsf{takesCourse}_{rdd} = \{(\mathsf{Tom}, \mathsf{Database}), (\mathsf{John}, \mathsf{Algorithm})\}$$

$$\mathsf{subOrganisationOf}_{rdd} = \{(\mathsf{Group.Department}), (\mathsf{Department}, \mathsf{College}), (\mathsf{College}, \mathsf{University})\}$$

As we have mentioned earlier, when storing the A-Box in a **Distributed File System** (**DFS**), many reasoning systems, such as WebPIE, Cichlid [10], SHARD [26] and PigSPARQL [27], simply load ontology files into the DFS and leave the task of partitioning data to applications or users. Our approach (which can be viewed as a variant of the vertical partitioning model used by HadoopRDF [12]) has the advantage that when computing reasoning involving only some classes and properties, only those partitions which store the relevant data need to be accessed. For example, in order to create $\mathsf{Student}_{rdd}$, SPOWL only needs to access the partition which stores instances of the class Student. By contrast, if the data is not partitioned, the whole ontology should be read and a fragment of it should be filtered out.

2.2.3 *Classified T-Box to Spark Programmes*

SPOWL compiles axioms in the classified T-Box to a set of entailment rules (in the format of **if**...**then**...) which will be further implemented as Spark programmes[2]. The Spark programmes will be executed over RDDs of data iteratively to compute new reasoning results, which will build up the RDDs. The iterative execution will terminate when there is no new derivation that can be inferred.

We start from axioms (1), (7), (8) and (20) to illustrate the compiling process from OWL axioms to Spark programmes. Each of these axioms specifies a subsumption relationship from one class to another, which is generally expressed by $C \sqsubseteq D$ in DL. The semantics of $C \sqsubseteq D$ implies that individuals contained in the subsumed class C should be inferred as individuals in the subsumer class D, and SPOWL captures these semantics by mapping in an entailment rule, where $\mathcal{R}_{axiom}$ denotes the entailment rule mapped from an $axiom$:

$$\mathcal{R}_{C \sqsubseteq D} ::= \textbf{if } C_{rdd}(x) \textbf{ then } D_{rdd}(x)$$

The rule states that data in C_{rdd} (representing class C) should be included in D_{rdd} (representing class D). This entailment rule can be implemented in Spark by calling a union function over D_{rdd} to merge D_{rdd} with C_{rdd} (we use $\mathcal{P}_{axiom}$ to denote the Spark programme generated for an $axiom$):

$$\mathcal{P}_{C \sqsubseteq D} ::= D_{rdd} = D_{rdd}.\mathsf{union}(C_{rdd})$$

Note that Spark currently supports three programming languages, namely Scala, Java and Python. In this paper, we adopt a Python-like format to illustrate Spark programmes.

If we follow this process for handling $C \sqsubseteq D$, axioms (1), (7), (8) and (20) are first mapped into respectively

[2]Besides Spark, the set of entailment rules can also be implemented in other programming languages; for example, our previous work [20, 18] supports the implementation as SQL triggers, which compute and materialise reasoning in a relational database.

$\mathcal{R}_{\mathsf{Student}\sqsubseteq\mathsf{Person}}$, $\mathcal{R}_{\mathsf{GraduateStudent}\sqsubseteq\mathsf{Person}}$, $\mathcal{R}_{\mathsf{GraduateCourse}\sqsubseteq\mathsf{Course}}$ and $\mathcal{R}_{\mathsf{GraduateStudent}\sqsubseteq\mathsf{Student}}$ as follows:

if $\mathsf{Student}_{rdd}(x)$ **then** $\mathsf{Person}_{rdd}(x)$

if $\mathsf{GraduateStudent}_{rdd}(x)$ **then** $\mathsf{Person}_{rdd}(x)$

if $\mathsf{GraduateCourse}_{rdd}(x)$ **then** $\mathsf{Course}_{rdd}(x)$

if $\mathsf{GraduateStudent}_{rdd}(x)$ **then** $\mathsf{Student}_{rdd}(x)$

which are compiled by $\mathcal{P}_{\mathsf{Student}\sqsubseteq\mathsf{Person}}$, $\mathcal{P}_{\mathsf{GraduateStudent}\sqsubseteq\mathsf{Person}}$, $\mathcal{P}_{\mathsf{GraduateCourse}\sqsubseteq\mathsf{Course}}$ and $\mathcal{P}_{\mathsf{GraduateStudent}\sqsubseteq\mathsf{Student}}$ to:

$\mathsf{Person}_{rdd} = \mathsf{Person}_{rdd}.\mathsf{union}(\mathsf{Student}_{rdd})$

$\mathsf{Person}_{rdd} = \mathsf{Person}_{rdd}.\mathsf{union}(\mathsf{GraduateStudent}_{rdd})$

$\mathsf{Course}_{rdd} = \mathsf{Course}_{rdd}.\mathsf{union}(\mathsf{GraduateCourse}_{rdd})$

$\mathsf{Student}_{rdd} = \mathsf{Student}_{rdd}.\mathsf{union}(\mathsf{GraduateStudent}_{rdd})$

Executing the above will cause Person_{rdd} will be merged with data stored in both $\mathsf{Student}_{rdd}$ and $\mathsf{GraduateStudent}_{rdd}$. Similarly, Course_{rdd} will be merged with $\mathsf{GraduateCourse}_{rdd}$, and $\mathsf{Student}_{rdd}$ will be merged with $\mathsf{GraduateStudent}_{rdd}$. Thus, the RDDs are updated to:

$\mathsf{Student}_{rdd} = \{\mathsf{John}, \mathsf{Tom}, \mathsf{Jack}\}$

$\mathsf{Person}_{rdd} = \{\mathsf{Lewis}, \mathsf{Mary}, \mathsf{John}, \mathsf{Tom}, \mathsf{Jack}\}$

$\mathsf{Course}_{rdd} = \{\mathsf{Database}, \mathsf{Algorithm}\}$

2.2.4 *Beyond OWL 2 RL axioms*

Besides subsumptions, OWL 2 provides a SomeValuesFrom constructor (denoted as $\exists P.D$ in DL) to specify some existential restrictions. $\exists P.D$ specifies a set of individuals x such that each x is related by P to at least one individual y in D. SomeValuesFrom is used by axioms (2) and (9), each of which expresses a subsumption relationship from a class to a SomeValuesFrom expression (i.e. $C \sqsubseteq \exists P.D$), which expresses that every x in C is related by P to at least one y in D. Axioms of $C \sqsubseteq \exists P.D$ might lead to non-deterministic reasoning, because if D contains more than one individual, we only know that P relates every individual of C to at least one of D's individuals, but we cannot determine which one of them.

OWL 2 RL eliminates the case of $C \sqsubseteq \exists P.D$ to avoid the non-determinism. However, this elimination might make the reasoning incomplete; for example, when querying for all individuals that are related by P, individuals of C which should be included as the answer might be missed. However, in SPOWL we specify the following $\mathcal{R}_{C\sqsubseteq\exists P.D}$ to solve this incompleteness.

$$\mathcal{R}_{C\sqsubseteq\exists P.D} ::= \textbf{if } C_{rdd}(x), \neg P_{rdd}(x,y) \textbf{ then } P_{rdd}(x, null) \quad (a)$$
$$\textbf{if } C_{rdd}(x), P_{rdd}(x,y), \neg D_{rdd}(y) \textbf{ then } - \quad (b)$$

As can be seen, $\mathcal{R}_{C\sqsubseteq\exists P.D}$ has two parts: (a) handles the set of x which is recorded in C_{rdd} but no such pairs (x, y) are recorded P_{rdd}, for which we add pairs $(x, null)$ into P_{rdd} (i.e. $null$ denotes some unknown y of D_{rdd} which P_{rdd} relates x to); (b) handles the case that in P_{rdd} there is a pair (x, y) for x in C_{rdd}, but such y is not in D_{rdd}. However, for this case, we do not perform any action (i.e. **then** $-$), because querying for all individuals that are related by P will obtain complete answers in this case. Therefore, $\mathcal{R}_{C\sqsubseteq\exists P.D}$ can be simplified to only contain part (a), which will be implemented as the following Spark programme $\mathcal{P}_{C\sqsubseteq\exists P.D}$:

$$\begin{aligned}\mathcal{P}_{C\sqsubseteq\exists P.D} ::= P_{tmp_1} &= P_{rdd}.\mathsf{map}(\mathsf{lambda}\ (x,y) : x)\\ P_{tmp_2} &= C_{rdd}.\mathsf{subtract}(P_{tmp_1})\\ P_{tmp_3} &= P_{tmp_2}.\mathsf{map}(\mathsf{lambda}\ x : (x, null))\\ P_{rdd} &= P_{rdd}.\mathsf{union}(P_{tmp_3})\end{aligned}$$

Considering axiom (2), SPOWL generates the following $\mathcal{R}_{\mathsf{Student}\sqsubseteq\exists\mathsf{takesCourse}.\mathsf{Course}}$:

if $\mathsf{Student}_{rdd}(x), \neg\mathsf{takesCourse}_{rdd}(x,y)$

then $\mathsf{takesCourse}_{rdd}(x, null)$

which results in the following $\mathcal{P}_{\mathsf{Student}\sqsubseteq\exists\mathsf{takesCourse}.\mathsf{Course}}$:

$$\begin{aligned}\mathsf{takesCourse}_{tmp_1} &= \mathsf{takesCourse}_{rdd}.\mathsf{map}(\mathsf{lambda}\ (x,y) : x)\\ \mathsf{takesCourse}_{tmp_2} &= \mathsf{Student}_{rdd}.\mathsf{subtract}(\mathsf{takesCourse}_{tmp_1})\\ \mathsf{takesCourse}_{tmp_3} &= \mathsf{takesCourse}_{tmp_2}.\mathsf{map}(\mathsf{lambda}\ x : (x, null))\\ \mathsf{takesCourse}_{rdd} &= \mathsf{takesCourse}_{rdd}.\mathsf{union}(\mathsf{takesCourse}_{tmp_3})\end{aligned}$$

The above $\mathcal{P}_{\mathsf{Student}\sqsubseteq\exists\mathsf{takesCourse}.\mathsf{Course}}$ first computes a temporary RDD $\mathsf{takesCourse}_{tmp_1}$ including the individuals recorded as related by takesCourse by using a Spark map function, which projects the field of x from (x, y) pairs contained in $\mathsf{takesCourse}_{rdd}$ (where lambda is a Python construct for creating anonymous functions at runtime). As we have illustrated, $(\mathsf{Tom}, \mathsf{Database})$ and $(\mathsf{John}, \mathsf{Algorithm})$ are included in $\mathsf{takesCourse}_{rdd}$; therefore, Tom from $(\mathsf{Tom}, \mathsf{Database})$, and John in $(\mathsf{John}, \mathsf{Algorithm})$ will be included in $\mathsf{takesCourse}_{tmp_1}$:

$$\mathsf{takesCourse}_{tmp_1} = \{\mathsf{Tom}, \mathsf{John}\}$$

Next, $\mathcal{P}_{\mathsf{Student}\sqsubseteq\exists\mathsf{takesCourse}.\mathsf{Course}}$ computes the data items that are in $\mathsf{Student}_{rdd}$ but not in $\mathsf{takesCourse}_{tmp_1}$ by using a Spark subtract function. Obviously, Jack in $\mathsf{Student}_{rdd}$ is the only one which is not in $\mathsf{takesCourse}_{tmp_1}$, and it will be included in the second temporary RDD $\mathsf{takesCourse}_{tmp_2}$:

$$\mathsf{takesCourse}_{tmp_2} = \{\mathsf{Jack}\}$$

Then, $\mathcal{P}_{\mathsf{Student}\sqsubseteq\exists\mathsf{takesCourse}.\mathsf{Course}}$ calls another map function, which forms $(x, null)$ for each x in $\mathsf{takesCourse}_{tmp_2}$ and includes the $(x, null)$ pairs in $\mathsf{takesCourse}_{tmp_3}$:

$$\mathsf{takesCourse}_{tmp_3} = \{(\mathsf{Jack}, null)\}$$

Finally, after merging with $(\mathsf{Jack}, null)$ in $\mathsf{takesCourse}_{tmp_3}$, $\mathsf{takesCourse}_{rdd}$ becomes:

$$\mathsf{takesCourse}_{rdd} = \{(\mathsf{Tom}, \mathsf{Database}), (\mathsf{John}, \mathsf{Algorithm}), (\mathsf{Jack}, null)\}$$

Thus, if a query asks for individuals related by takesCourse, SPOWL not only returns Tom and John but also Jack as a complete answer to the query.

Note that $\mathcal{P}_{C\sqsubseteq\exists P.D}$ can be alternatively written as:

$$\begin{aligned}P_{rdd} = P_{rdd}.\mathsf{union}(&\\ &C_{rdd}.\mathsf{subtract}(P_{rdd}.\mathsf{map}(\mathsf{lambda}\ (x,y) : x))\\ &\qquad .\mathsf{map}(\mathsf{lambda}\ x : (x, null)))\end{aligned}$$

and we will use this more compact representation for the remaining examples of Spark programs where we do not need to detail the intermediate results during execution.

2.2.5 *DAG for Parallelising Reasoning*

Spark uses its DAG scheduler to provide a more flexible and parallelised job scheduling than MapReduce, which follows a sequential job planing. To illustrate how a more parallelised job scheduling can be used for computing the reasoning materialisation, we consider the axiom (10), which involves another OWL constructor IntersectionOf (symbolised by $\sqcap$). The constructor constructs an IntersectionOf expression $\mathsf{Person}\sqcap\exists\mathsf{takesCourse}.\mathsf{Course}$, which specifies a set of individuals x that are both members of Person and are related by takesCourse to at least one individual of Course. Additionally, by setting the IntersectionOf expression as a subclass of Student, the set of x should be included in Student.

Therefore, SPOWL specifies $\mathcal{R}_{\mathsf{Person}\sqcap\exists\mathsf{takesCourse}.\mathsf{Course}\sqsubseteq\mathsf{Student}}$ for axiom (10) as:

if $\mathsf{Person}_{rdd}(x)$, $\mathsf{takesCourse}_{rdd}(x,y)$, $\mathsf{Course}_{rdd}(y)$
then $\mathsf{Student}_{rdd}(x)$

which means that any x appearing in both $\mathsf{Person}_{rdd}(x)$ and $\mathsf{takesCourse}_{rdd}(x,y)$, where y appears in Course_{rdd}, should be merged into $\mathsf{Student}_{rdd}$. As shown in Figure 2, computing the set of x that should be included in $\mathsf{Student}_{rdd}$ requires three RDDs (i.e. Person_{rdd}, $\mathsf{takesCourse}_{rdd}$ and Course_{rdd}), and by using the DAG scheduler, SPOWL is able to schedule the computations of the three RDDs to three parallelised jobs (i.e. job_a for computing Person_{rdd}, job_b for computing $\mathsf{takesCourse}_{rdd}$ and job_c for computing Course_{rdd}). By contrast, in MapReduce job_a, job_b and job_c have to be scheduled sequentially.

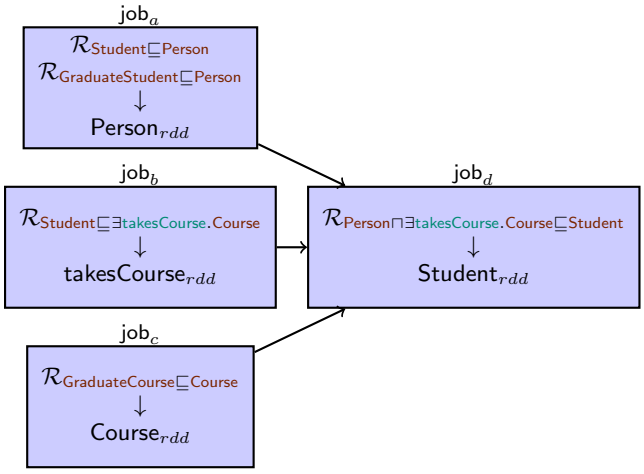

Figure 2: DAG Scheduling for $\mathcal{R}_{\mathsf{Person}\sqcap\exists\mathsf{takesCourse}.\mathsf{Course}\sqsubseteq\mathsf{Student}}$

When Person_{rdd}, $\mathsf{takesCourse}_{rdd}$ and Course_{rdd} have been computed, job_d can start to handle the axiom (10) by implementing the entailment rule $\mathcal{R}_{\mathsf{Person}\sqcap\exists\mathsf{takesCourse}.\mathsf{Course}\sqsubseteq\mathsf{Student}}$ into Spark programmes $\mathcal{P}_{\mathsf{Person}\sqcap\exists\mathsf{takesCourse}.\mathsf{Course}\sqsubseteq\mathsf{Student}}$:

$$\begin{aligned}\mathsf{Student}_{tmp_1} &= \mathsf{takesCourse}_{rdd}.\mathsf{map}(\mathsf{lambda}\ (x_t,y_t):(y_t,x_t))\\ &\quad .\mathsf{join}(\mathsf{Course}_{rdd}.\mathsf{map}(\mathsf{lambda}\ y_c:(y_c,y_c)))\\ &\quad .\mathsf{map}(\mathsf{lambda}\ (y_k,(x_t,y_c)):x_t)\\ \mathsf{Student}_{tmp_2} &= \mathsf{Student}_{tmp_1}.\mathsf{intersection}(\mathsf{Person}_{rdd})\\ \mathsf{Student}_{rdd} &= \mathsf{Student}_{rdd}.\mathsf{union}(\mathsf{Student}_{tmp_2})\end{aligned}$$

The Spark programme first computes $\mathsf{Student}_{tmp_1}$ including the individuals which belong to $\exists\mathsf{takesCourse}.\mathsf{Course}$ by using join and map functions. In Spark, joins should be performed between two RDDs containing key-value pairs, so for (x_t,y_t) pairs in $\mathsf{takesCourse}_{rdd}$, a map function on $\mathsf{takesCourse}_{rdd}$ is applied to create a set of key-value pairs (y_t,x_t), and for data items y_c in Course_{rdd}, another map on Course_{rdd} is specified to generate a set of key-value pairs (y_c,y_c). Continuing with the LUBM fragment, the (y_t,x_t) pairs for $\mathsf{takesCourse}_{rdd}$ will be:

$$\{(\mathsf{Database},\mathsf{Tom}),(\mathsf{Algorithm},\mathsf{John})\}$$

and the (y_c,y_c) pairs for Course_{rdd} will be:

$$\{(\mathsf{Database},\mathsf{Database}),(\mathsf{Algorithm},\mathsf{Algorithm})\}$$

Next, the join function will look for the case of $y_t = y_c$ from the two sets of key-value pairs, and return for each key a set of $(y_k,(x_t,y_c))$ pairs, where x_t (which belongs to $\exists\mathsf{takesCourse}.\mathsf{Course}$) will be projected by a map. Based on the LUBM fragment, $(y_k,(x_t,y_c))$ pairs below will be generated after processing the join function:

$$\{(\mathsf{Database},(\mathsf{Tom},\mathsf{Database})),(\mathsf{Algorithm},(\mathsf{John},\mathsf{Algorithm}))\}$$

Consequently, Tom and John will be selected as members of $\mathsf{takesCourse}_{tmp_1}$, which is shown as follows:

$$\mathsf{takesCourse}_{tmp_1} = \{\mathsf{Tom},\mathsf{John}\}$$

Then, an intersection function is performed to select common data items in both $\mathsf{Student}_{tmp_1}$ and Person_{rdd} (containing Lewis, Mary, John, Tom and Jack). Obviously, {Tom, John} will be computed as common individuals, and they should be merged into $\mathsf{Student}_{rdd}$ by a union function.

2.2.6 *Data Caching in Distributed Memory*

Another important feature of Spark is the capability of caching RDDs in the distributed memory in a cluster of machines. By contrast, MapReduce requires to write/read data to/from the disk, which often leads to a high I/O overhead. We may illustrate the benefits of this by using the axiom (11), which specifies a TransitiveProperty subOrganisationOf, and three A-Box facts (17)–(19) of subOrganisationOf.

When a property P is defined as a TransitiveProperty, the semantics of transitivity specifies that if (x,y) and (y,z) are both instances of P, then (x,z) is an instance of P. SPOWL translates this into an entailment rule $\mathcal{R}_{P\circ P\sqsubseteq P}$:

if $P_{rdd}(x,y), P_{rdd}(y,z)$ **then** $P_{rdd}(x,z)$

Therefore, the transitivity of subOrganisationOf is handled by $\mathcal{R}_{\mathsf{subOrganisationOf}\circ\mathsf{subOrganisationOf}\sqsubseteq\mathsf{subOrganisationOf}}$:

if $\mathsf{subOrganisationOf}_{rdd}(x,y), \mathsf{subOrganisationOf}_{rdd}(y,z)$
then $\mathsf{subOrganisationOf}_{rdd}(x,z)$

Materialising the reasoning results for a TransitiveProperty P is also known as the problem of computing its transitive closure, which has been researched by many studies, such as [7] and [24]. In SPOWL, we adopt a simple recursive-doubling method described in [15] for compiling $\mathcal{R}_{P\circ P\sqsubseteq P}$ to the Spark programmes $\mathcal{P}_{P\circ P\sqsubseteq P}$:

while True **do**
  $P_{tmp} = P_{rdd}.\mathsf{map}(\mathsf{lambda}\ (x_p,y_p):(y_p,x_p)).\mathsf{join}(P_{rdd})$
    $.\mathsf{map}(\mathsf{lambda}\ (y_k,(x_p,z_p)):(x_p,z_p))$
  if $P_{tmp}.\mathsf{isEmpty}()$ **then** break
  $P_{rdd} = P_{rdd}.\mathsf{union}(P_{tmp})$
end

As can be seen, $\mathcal{P}_{P\circ P\sqsubseteq P}$ contains a while loop, which computes the transitive closure for P iteratively. In each iteration, a self join on P_{rdd} (which initially contains explicit instances of P) is performed to see whether new transitive pairs (x_p,z_p) can be computed (from pairs (x_p,y_p) and (y_p,z_p)). If so, the new pairs are stored in P_{tmp} (i.e. P_{tmp} is not empty), which is merged into P_{rdd} at the end of this iteration, and the updated P_{rdd} will be used for the next iteration. Otherwise, if no transitive pairs can be calculated (i.e. P_{tmp} is empty), the computation of transitive closure terminates.

Since Spark is able to cache RDDs in the distributed memory, at the end of each computation iteration of $\mathcal{P}_{P\circ P\sqsubseteq P}$ we can call a Spark function cache (or persist) to cache P_{rdd} in memory without needing to write the intermediate results to disk. Thus, during the next iteration, P_{rdd} can be read directly from the memory without data exchange with the disk.

To determine the number of iterations required for computing a transitive closure, we interpret P as a graph, where each vertex x represents an individual x, an arc from x to y, denoted as $Arc\langle x,y\rangle$, represents x is explicitly related to

y by P, and a path from x to y, $Path\langle x, y\rangle$, denotes that x is explicitly or implicitly related to y by P (i.e. through one or more arcs y is reachable from x in the graph). Thus, computing the transitive closure for P can be interpreted as the problem of computing all $Path\langle x, y\rangle$ in the graph of this property.

The number of iterations required to terminate the computation depends on the longest path in a graph of P. If the length of an arc $Arc\langle x, y\rangle$ is set as 1, the length of $Path\langle x, y\rangle$ is the number of arcs from x to y. For example, in the graph of P, if there are arcs $Arc\langle x, a\rangle$, $Arc\langle a, b\rangle$ and $Arc\langle b, y\rangle$, then y is reachable from x via a and b, and such a path $Path\langle x, y\rangle$ is of the length 3. Note that for the case that y is reachable from x by more than one path, we consider the shortest one as its length. Continuing with the example, if the graph further contains arcs $Arc\langle x, c\rangle$ and $Arc\langle c, y\rangle$, then y is also reachable from x via c, and the length $Path\langle x, y\rangle$ should be 2, which is shorter than 3. If the longest path in a graph is of length d, a simple recursive-doubling method requires $\log_2 d$ iterations at most to finish computation of the transitive closure. However, unless d of a graph is pre-known, an extra iteration (i.e. totally $\log_2 d+1$ iterations) is necessarily required to check as to whether P_{tmp} is empty.

Using cache not only helps iterative computation (such as handling transitive properties), but also benefits the situations in which certain RDDs are used repeatedly.

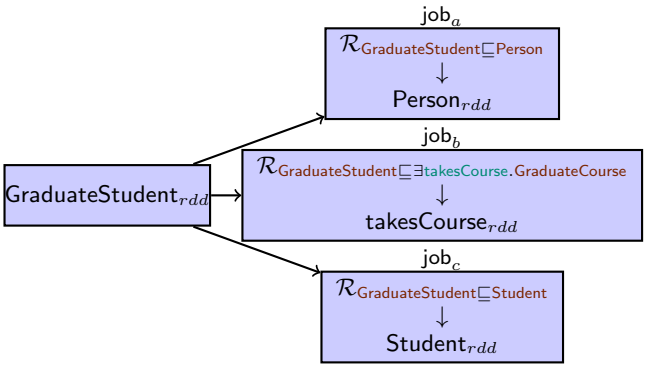

Figure 3: Caching GraduateStudent$_{rdd}$ for Repeated Usage

As illustrated in Figure 3, because axioms (7), (9) and (20) all specify GraduateStudent as their subclass expressions, GraduateStudent$_{rdd}$ will be used by job$_a$, job$_b$ and job$_c$, which respectively handle axioms (7), (9) and (20). By caching GraduateStudent$_{rdd}$ in memory, Spark can read it directly from the memory for repeated use without needing to write and then read intermediate results to and from disk, which is often required by MapReduce. Note that as Spark adopts a DAG scheduler, job$_a$, job$_b$ and job$_c$ can also be parallelised.

2.2.7 *Optimising Programme Execution Order*

We have provided an overview of how SPOWL specifies for an OWL *axiom* an entailment rule $\mathcal{R}_{axiom}$, which will be further implemented as a Spark programme $\mathcal{P}_{axiom}$. These Spark programmes will be executed iteratively until no new data is produced, which implies the termination of the reasoning materialisation. In distributed computing, because scheduling, starting and terminating distributed computation jobs often has a large overhead, even one more iteration will significantly affect the total performance. Therefore, in SPOWL we wish that the reasoning materialisation terminates with as fewer iterations as possible.

Take Figure 2 for handling the axiom (10) as an example again: since job$_d$ takes Person$_{rdd}$ from job$_a$, takesCourse$_{rdd}$ from job$_b$ and Course$_{rdd}$ from job$_c$ as inputs, executing job$_a$, job$_b$ and job$_c$ before job$_d$ is the best order. Otherwise, if job$_d$ is executed before any of job$_a$, job$_b$ and job$_c$, then job$_d$ still should be executed again to ensure that the new derivations from job$_a$, job$_b$ and job$_c$ are considered. Thus, we now consider a general method for optimising the execution order of our Spark programmes.

3. OPTIMISATION IN SPOWL

As we have illustrated in Section 2, SPOWL translates axioms in a classified T-Box into Spark programmes, which are specific to the ontology being reasoned over. Spark programmes are then launched to execute iteratively to calculate and materialise the reasoning closure. In order to optimise SPOWL for terminating the reasoning materialisation with minimum iterations, we further analyse the dependencies of Spark programmes, and execute them in an order following the bottom-up hierarchy of the T-Box. In addition, we apply some tuning techniques provided by Spark to further improve the performance of SPOWL.

3.1 Ordering Spark Programmes

We define that an entailment rule $\mathcal{R}_{axiom_1}$ is **higher** than another one $\mathcal{R}_{axiom_2}$ (or $\mathcal{R}_{axiom_2}$ is **lower** than $\mathcal{R}_{axiom_1}$), if $\mathcal{R}_{axiom_1}$ takes data inferred from $\mathcal{R}_{axiom_2}$ as its input. For example, if we have a T-Box hierarchy composed of two axioms $C_1 \sqsubseteq C_2$ and $C_2 \sqsubseteq C_3$, the entailment rule $\mathcal{R}_{C_1\sqsubseteq C_2}$ is lower than $\mathcal{R}_{C_2\sqsubseteq C_3}$, as the new data inferred because of $C_1 \sqsubseteq C_2$ contributes to the reasoning of $C_2 \sqsubseteq C_3$. To minimise the materialising iterations, we should execute the Spark programmes generated from the lowest entailment rules to the highest one. Thus, executing $\mathcal{P}_{C_1\sqsubseteq C_2}$ (compiled from $\mathcal{R}_{C_1\sqsubseteq C_2}$) before $\mathcal{P}_{C_2\sqsubseteq C_3}$ (compiled from $\mathcal{R}_{C_2\sqsubseteq C_3}$) should terminate the materialisation with only one iteration; however, executing $\mathcal{P}_{C_2\sqsubseteq C_3}$ before $\mathcal{P}_{C_1\sqsubseteq C_2}$ might require two iterations.

Depending on the types of OWL 2 axioms, the entailment rules specified by SPOWL can be divided into three groups and the dependencies among the rules are illustrated in Figure 4.

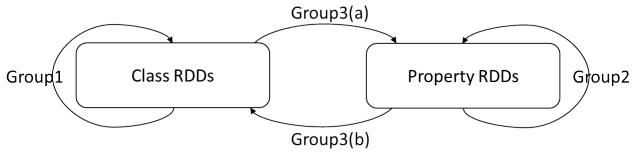

Figure 4: Dependence among entailment rules

1. The first group contains entailment rules which infer new data items to class RDDs from class RDDs, taking no property RDDs as input. Entailment rules fall into this group are $\mathcal{R}_{C\sqsubseteq D}$, $\mathcal{R}_{C\equiv D}$, $\mathcal{R}_{C_1\sqcup\ldots\sqcup C_n\sqsubseteq D}$ and $\mathcal{R}_{C_1\sqcap\ldots\sqcap C_n\sqsubseteq D}$.

2. The second group of entailment rules infer new data to property RDDs from property RDDs, taking no

class RDDs as input. Such entailment rules are $\mathcal{R}_{P \sqsubseteq Q}$, $\mathcal{R}_{P \equiv Q}$, $\mathcal{R}_{P \equiv P^-}$, $\mathcal{R}_{P \equiv Q^-}$, $\mathcal{R}_{P \circ P \sqsubseteq P}$ and $\mathcal{R}_{P_1 \circ \ldots \circ P_n \sqsubseteq P}$.

3. Entailment rules in the third group compute new data items to class RDDs from property RDDs, or infer new data items to property RDDs from class RDDs:

   (a) Entailment rules which take some class RDDs as input (or part of the input) and generate new data to property RDDs are $\mathcal{R}_{C \sqsubseteq \exists P.\{a\}}$, $\mathcal{R}_{C \sqsubseteq \exists P.D}$ and $\mathcal{R}_{C \sqsubseteq \exists P.\mathsf{Self}}$.

   (b) Entailment rules which compute new data to class RDDs from some property RDDs are $\mathcal{R}_{C \sqsubseteq \forall P.D}$, $\mathcal{R}_{\exists P.D \sqsubseteq C}$, $\mathcal{R}_{\exists P.\{a\} \sqsubseteq C}$, $\mathcal{R}_{\exists P.\mathsf{Self} \sqsubseteq C}$, $\mathcal{R}_{\geqslant n\, P \sqsubseteq C}$ (and $\mathcal{R}_{\geqslant n\, P.D \sqsubseteq C}$), $\mathcal{R}_{\top \sqsubseteq \forall P^-.C}$ and $\mathcal{R}_{\top \sqsubseteq \forall P.D}$.

Note that some of the above entailment rules considered by SPOWL are not included in the OWL 2 RL/RDF rules, such as $\mathcal{R}_{\exists P.\mathsf{Self} \sqsubseteq C}$ and $\mathcal{R}_{\geqslant n\, P.D \sqsubseteq C}$. By constrast, most materialisation-based systems such as WebPIE, Cichlid and RORS [19] only analyse dependencies of RDFS entailment rules and OWL ter Horst rules. Entailment rules in the first group (inferring data to class RDDs from class RDDs) are independent of those in the second group (inferring data to property RDDs from property RDDs). Therefore, in each of the first two groups, we follow the bottom-up class hierarchy or property hierarchy as the optimised order of executing Spark programmes. We illustrate this by taking property axioms $P_1 \sqsubseteq P_2$, $P_2 \circ P_2 \sqsubseteq P_2$ and $P_2 \sqsubseteq P_3$ as an example, the property hierarchy is displayed in Figure 5.

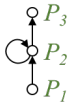

Figure 5: Acyclic property hierarchy

As can be seen, the property hierarchy is acyclic, we can easily obtain the dependencies among the entailment rules $\mathcal{R}_{P_1 \sqsubseteq P_2}$, $\mathcal{R}_{P_2 \circ P_2 \sqsubseteq P_2}$ and $\mathcal{R}_{P_2 \sqsubseteq P_3}$: $\mathcal{R}_{P_2 \circ P_2 \sqsubseteq P_2}$ is higher than $\mathcal{R}_{P_1 \sqsubseteq P_2}$ and is lower than $\mathcal{R}_{P_2 \sqsubseteq P_3}$. In other words, Spark programmes should be executed as the order of $\mathcal{P}_{P_1 \sqsubseteq P_2}$ followed by $\mathcal{P}_{P_2 \circ P_2 \sqsubseteq P_2}$ followed by $\mathcal{P}_{P_2 \sqsubseteq P_3}$. Indeed, by $\mathcal{P}_{P_1 \sqsubseteq P_2}$, new data items are inferred to $P2_{rdd}$ (representing P_2) from $P1_{rdd}$ (representing P_1). Next, the transitive closure of $P2_{rdd}$ is computed by executing $\mathcal{P}_{P_2 \circ P_2 \sqsubseteq P_2}$, and will be merged into $P3_{rdd}$ (representing P_3) by processing $\mathcal{P}_{P_2 \sqsubseteq P_3}$.

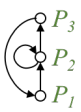

Figure 6: Cyclic property hierarchy

However, if we consider an extra property axiom $P_3 \equiv {P_1}^-$, the property hierarchy in Figure 5 becomes cyclic as shown in Figure 6. For a cyclic hierarchy, we cannot determine which entailment rule is the lowest nor which one is the highest; therefore, we could randomly select one of them as the first one to process. Suppose we choose $\mathcal{R}_{P_1 \sqsubseteq P_2}$ as the first entailment rule again, we should thereafter sequentially consider $\mathcal{R}_{P_2 \circ P_2 \sqsubseteq P_2}$ and $\mathcal{R}_{P_2 \sqsubseteq P_3}$. Here, due to the extra axiom $P_3 \equiv {P_1}^-$, which might infer new data to $P1_{rdd}$, $\mathcal{R}_{P_1 \sqsubseteq P_2}$ might need to be considered again, and so does $\mathcal{R}_{P_2 \circ P_2 \sqsubseteq P_2}$ and $\mathcal{R}_{P_2 \sqsubseteq P_3}$. In essence, the execution should terminate whenever there is no new reasoning to the input RDDs taken by all entailment rules.

Entailment rules in the third group bring the reasoning from class RDDs to property RDDs and vice versa, which makes it difficult for SPOWL to determine an optimised order. However, since PropertyDomain axioms (i.e. $\top \sqsubseteq \forall P^-.C$) and PropertyRange axioms (i.e. $\top \sqsubseteq \forall P.D$), which lead to data reasoning from property RDDs to class RDDs, are frequently used in most ontologies, we tend to consider entailment rules in the first group are higher than those in the second group. Therefore, in general, our approach adopts an optimised order of considering entailment rules as:

1. Spark programmes compiled from the second group of entailment rules are executed to infer new data from property RDDs to property RDDs following the property hierarchy.

2. Spark programmes compiled from group 3(b) deriving reasoning from property RDDs to class RDDs are executed. Note that since entailment rules involved in this step might depend on each other, we also process them from lower entailment rules to higher rules.

3. Spark programmes generated from the first group of entailment rules are processed to derive new data from class RDDs to class RDDs following the class hierarchy.

4. Whenever the ontology has axioms which could derive new reasoning from class RDDs to property RDDs (i.e. entailment rules in group 3(a)), we check whether new data items are inferred because of them; if so we reconduct the previous three steps until the input taken by higher-level entailment rules contains no newly inferred data.

3.2 Tuning Spark Programmes

Spark provides numerous tuning techniques[3]. In this section, we discuss those techniques which have used to improve SPOWL's performance of reasoning materialisation.

- **Caching Data in Distributed Memory**

  In the physical Spark programmes, we cache an RDD which is repeatedly used to avoid re-computation of this RDD. For instance, for an IntersectionOf axiom $C \sqsubseteq C_1 \sqcap \ldots \sqcap C_n$, because a tableaux reasoner classifies it to n subsumption relationships (i.e. $C \sqsubseteq C_1, \ldots, C \sqsubseteq C_n$), the data in C_{rdd} (representing C) will be used n times; therefore, caching C_{rdd} by the Spark function cache or persist in the distributed memory will improve the performance of reasoning this axiom. Moreover, if an entailment rule requires iterative computation (e.g. handling a TransitiveProperty), we tend to cache the intermediate result after each iteration, so that they can be quickly accessed by the next iteration.

  Note that, because of the in-memory nature of Spark, the amount of the distributed memory in a cluster

[3]https://spark.apache.org/docs/latest/tuning.html

could become a bottleneck. When the memory is not enough for caching all items of data, Spark will write some old data to disk (which in a way, may be regarded as reverting to MapReduce), and consequently SPOWL will benefit less from the memory caching. In order to use the distributed memory more efficiently, we also adopt the Kryo library[4] to serialise RDDs, which will use less memory than without serialisation.

- **Partitioning before Join**

  In Spark, a normal join between two sets of key-value pairs will shuffle the pairs whose keys are the same to the same executor, so that join pairs can be computed. However, in the case of one set of key-value pairs being very large while the other set is quite small, this normal join might result in a slow shuffle process because of shuffling the large set. Instead, we partition the large set of key-value pairs by their keys, and copy the small set to the node where each partition of the large set is stored. Since this reduces the amount of data transferred through the network of a cluster, the join can be performed much faster. Partitioning an RDD can be achieved by a Spark function partitionBy.

4. EVALUATION OF SPOWL

In this section, we evaluate SPOWL on its performance of materialising the reasoning closure. All experiments were performed on a cluster of 9 machines running on a private cloud environment[5]. The cluster contains a master node and 8 slave nodes (each with CPU @ 2.5GHz, 4 Cores, and 16 GB of Memory). It ran Hadoop version 2.6.0-cdh5.5.0 (with 2.08 TB configured capacity), and Apache Spark 1.6.0. SPOWL used OWL API v3.4.3 for T-Box loading, and supports the use of Pellet v2.3.1 or Hermit v1.3.8 for T-Box classification.

In this evaluation, we use the well-known LUBM benchmark, which consists of a T-Box, a data generator, and 14 queries. The T-Box contains 43 classes, 32 properties, and approximately 200 axioms (as we have illustrated previously, some axioms are beyond OWL 2 RL). The data generator is used to produce LUBM A-Boxes with different sizes. We use LUBM-n to denote a dataset that contains n universities of A-Box facts. Each university of data has about 100,000 class and property facts; for instance, LUBM-2000 has approximately 270 million A-Box facts and is about 44GB in size. Each experiment was repeated 10 times, of which the average value is reported.

The time used by SPOWL for initial data loading and materialising the reasoning closure is reported in Section 4.1. Although the performance results are rather preliminary, we compare them to another materialisation-based system, WebPIE v1.1.1, and the comparison is discussed in Section 4.2. Note that the reasoning materialisation is stored in the HDFS, which can be read by any HDFS-supported query languages such as Pig[6] and Hive[7]. SPOWL current supports LUBM queries in Spark[8], and its performance of query processing is not included in this evaluation because of space restrictions, but can be found in [16].

[4]https://github.com/EsotericSoftware/kryo
[5]https://www.doc.ic.ac.uk/csg/services/cloud
[6]https://pig.apache.org/
[7]https://hive.apache.org/
[8]https://github.com/yl12510/thesis/blob/master/lubm

4.1 Performance of Reasoning Materialisation

4.1.1 Initial Load

We used SPOWL to load the original A-Boxes for LUBM-400, LUBM-800, LUBM-1200, LUBM-1600 and LUBM-2000. During the stage of initial loading, instances of every class or property were filtered out and materialised in separate folders in the HDFS. We recorded the time which SPOWL used for loading each dataset in Table 1.

Table 1: Reasoning Materialisation by SPOWL (Total Caching)

SPOWL	LUBM-400	LUBM-800	LUBM-1200	LUBM-1600	LUBM-2000
Initial Load	9m08s	20m30s	27m50s	41m20s	54m10s
Reasoning	10m19s	16m28s	33m20s	38m58s	58m08s
Total Time	19m27s	36m58s	1h01m10s	1h20m18s	1h52m18s

As can be seen, the time used by SPOWL increased almost linearly when loading datasets from LUBM-400 to LUBM-2000. In particular, SPOWL was able to initially load LUBM-2000 in 55 minutes (i.e. the loading speed was at about 81,818 facts/s). We may highlight this linear increase by translating the results in Table 1 into a line chart in Figure 7. Larger LUBM sizes resulted in out of memory errors.

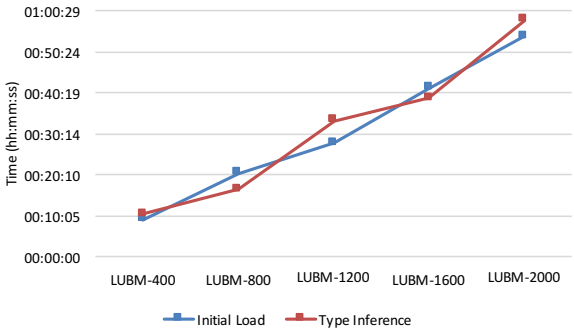

Figure 7: Reasoning Materialisation by SPOWL (Total Caching)

4.1.2 Reasoning Materialisation

Over the loaded data, we used SPOWL to perform reasoning by launching a Spark application which executed Spark programmes generated from T-Box axioms. The results of reasoning were computed and materialised in the HDFS. In this stage, we tuned SPOWL to cache as many datasets as possible in the distributed memory. The time used by SPOWL for the five datasets is also shown in Table 1, and is available in the line chart shown in Figure 7. Again, the time used for reasoning grew almost linearly from LUBM-400 to LUBM-2000. Furthermore, for the largest dataset LUBM-2000, SPOWL materialised about 246 million implicit facts in 59 minutes (i.e. at the speed of 70,690 facts/s).

4.1.3 Data Caching Strategies

One key advantage of Spark compared to MapReduce is the ability to cache data in the distributed memory to improve computation performance. However, the size of the distributed memory in a cluster often becomes a bottleneck, when memory is smaller than the size of data to be

cached, and some data will be written to disk. In such circumstances, Spark becomes similar to MapReduce. Indeed, when we used SPOWL to process LUBM-2000, an out of memory error erratically occurred. This also explains the reason why SPOWL needed about 22m more for reasoning from LUBM-1600 to LUBM-2000, but required only 5m more from LUBM-1200 to LUBM-1600.

We verified this by adjusting SPOWL to use a partial caching strategy, as compared to the total caching strategy reported in Table 1. In partial caching, we only cache the data used for Spark programmes generated for one axiom, rather than always caching all data for the whole reasoning materialisation. Thus, when SPOWL starts to process a new set of Spark programmes for an axiom, it needs to read related data from the disk, and after the programmes have been finished, SPOWL needs to write newly derived results back to disk. Performance results of SPOWL after this change are provided in Table 2, and they show SPOWL now was able to handle up to LUBM-4000, which is about 90GB of size (with the total caching strategy, SPOWL was only able to scale up to LUBM-2000 over the evaluation cluster).

Table 2: Reasoning Materialisation by SPOWL (Partial Caching)

SPOWL	LUBM-2000	LUBM-3000	LUBM-4000
Initial Load	53m05s	1h16m54s	1h54m41s
Reasoning	1h57m56s	3h06m59s	4h41m07s
Total	2h51m00s	4h23m52s	6h35m48s

However, SPOWL required much longer time for reasoning (i.e. 1h57m56s) over LUBM-2000 now as compared to the time (i.e. 58m08s) it needed in Table 1. This is as expected because now SPOWL spent more time on exchanging data between memory and disk. Inspired from this finding, we plan to investigate how a hybrid data caching strategy might maximise the scalability of SPOWL without slowing down the reasoning too much.

4.2 Comparing SPOWL to WebPIE

The reason we choose WebPIE as a comparison system is mainly because WebPIE uses MapReduce as a computational mechanism to materialise reasoning by evaluating a set of entailment rules, while SPOWL uses Spark, which has certain advantages over MapReduce. Unlike SPOWL, WebPIE does not use a tableaux reasoner, and its reasoning completeness is limited to the evaluated entailment rules. In particular, it covers the OWL ter Horst rules, but does not fully handle OWL 2 RL/RDF rules. Moreover, WebPIE treats ontology data as a single set of RDF triples without any partitioning, but it compresses ontology data before executing MapReduce programmes, which could accelerate the materialising. Consequently, decompressing the generated reasoning materialisation is required. In this evaluation, we recorded the time used by WebPIE for processing LUBM-1000, LUBM-2000, LUBM-3000 and LUBM-4000, which is provided in Table 3.

Since WebPIE compresses ontological data under reasoning, which could accelerate the reasoning process, it is more fair to compare the total time used by WebPIE and SPOWL, than only comparing the reasoning time. When SPOWL uses the total caching strategy, even without compressing the data, it required 1h52m18s for materialising reasoning results of LUBM-2000 as shown in Table 1, and the performance is faster than 2h14m14s performed by WebPIE. The reason for the outperformance is not only because SPOWL uses Spark's capability of data caching, but also due to the fact that SPOWL compiles a classified T-Box to Spark programmes directly related to the ontology rather than the simple rule evaluation adopted by WebPIE. Indeed, if we consider the situation in which SPOWL uses the partial caching strategy (in Table 2), it needed 2h51m in total for LUBM-2000, which is just slightly slower than WebPIE. Remember that SPOWL does not compress the ontology data, and we plan to add this to SPOWL in the future. In addition, SPOWL handles some axioms beyond OWL 2 RL, such as axioms (2) and (9) in LUBM, and thus is more complete than WebPIE.

Table 3: Reasoning Materialisation by WebPIE

WebPIE	LUBM-1000	LUBM-2000	LUBM-3000	LUBM-4000
compress	29m04s	59m37s	1h31m52s	2h01m59s
reasoning	30m36s	46m02s	58m27s	70m13s
decompress	14m03s	28m35s	49m16s	1h03m7s
Total	1h13m43s	2h14m14s	3h19m35s	4h15m19s

WebPIE stores the reasoning materialisation as a whole set of RDF triples without considering to partition the ontological data. This means when there are queries retrieving information from the materialisation, WebPIE would require longer time to process the queries than SPOWL. Moreover, since it performs reasoning by the way of evaluating a fixed set of entailment rules, its reasoning completeness is limited by the choice of the ruleset. However, SPOWL uses the classified T-Box for generating entailment rules, which are specially for the ontology under reasoning, and handle some extra axioms beyond OWL 2 RL, which leads to a more complete query answering.

5. SUMMARY AND CONCLUSIONS

To summarise, this paper has described how our approach to compiling OWL T-Boxes into Spark programmes as a system named SPOWL, which supports reasoning over large ontologies in a Big Data system. Unlike most large scale reasoners, which simply evaluate a set of entailment rules for materialising the reasoning closure, SPOWL uses a classified T-Box for a more complete T-Box reasoning. Moreover, compared to reasoners using MapReduce, SPOWL benefits from Spark which uses distributed memory as much as possible, and schedules jobs in a more flexible and parallelised manner by the DAG scheduler. In particular, we have divided Spark programmes into three groups, and have introduced an optimised order of executing the three groups of Spark programmes, which might reduce execution iterations until the computation of reasoning closure can terminate.

However, our approach restricts itself to consider simple and small T-Boxes, while ontologies with complex and large structure (e.g. SNOMED-CT [8] and Gene ontology [1]) are beyond the scope of SPOWL. Also, if dependencies among the Spark programmes contain many cycles, an optimised order of execution is difficult to obtain, but we consider this is highly unlikely in real-world ontologies.

6. REFERENCES

[1] M. Ashburner, C. A. Ball, J. A. Blake, D. Botstein, H. Butler, J. M. Cherry, A. P. Davis, K. Dolinski,

S. S. Dwight, J. T. Eppig, et al. Gene Ontology: Tool for the Unification of Biology. *Nature genetics*, 25(1):25–29, 2000.

[2] S. Auer, C. Bizer, G. Kobilarov, J. Lehmann, R. Cyganiak, and Z. G. Ives. DBpedia: A Nucleus for a Web of Open Data. In *ISWC/ASWC*, volume 4825, pages 722–735. Springer, 2007.

[3] F. Baader, D. Calvanese, D. L. McGuinness, D. Nardi, and P. F. Patel-Schneider, editors. *The Description Logic Handbook: Theory, Implementation, and Applications (Second Edition)*. Cambridge University Press, Cambridge, UK, 2010.

[4] T. Bagosi, D. Calvanese, J. Hardi, S. Komla-Ebri, D. Lanti, M. Rezk, M. Rodríguez-Muro, M. Slusnys, and G. Xiao. The Ontop Framework for Ontology Based Data Access. In *Proceedings of CSWS 2014*, pages 67–77, Wuhan, China, 8–12 Aug. 2014. Springer.

[5] T. U. Consortium. UniProt: a Hub for Protein Information. *Nucleic Acids Research*, 43(D1):D204–D212, 2015.

[6] R. Cyganiak, D. Wood, and M. Lanthaler, editors. *RDF 1.1 concepts and abstract syntax*. W3C Recommendation, 2014. Latest version available at https://www.w3.org/TR/rdf11-concepts/.

[7] G. Dong, L. Libkin, J. Su, and L. Wong. Maintaining Transitive Closure of Graphs in SQL. *International Journal of Information Technology*, 51(1):46, 1999.

[8] K. Donnelly. SNOMED-CT: The advanced terminology and coding system for eHealth. *Studies in health technology and informatics*, 121:279, 2006.

[9] Google. Freebase Data Dumps. https://developers.google.com/freebase, 2016.

[10] R. Gu, S. Wang, F. Wang, C. Yuan, and Y. Huang. Cichlid: Efficient Large Scale RDFS/OWL Reasoning with Spark. In *IPDPS*, pages 700–709. IEEE, 2015.

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

[12] M. F. Husain, L. Khan, M. Kantarcioglu, and B. Thuraisingham. Data intensive query processing for large RDF graphs using cloud computing tools. In *IEEE CLOUD*, pages 1–10. IEEE, 2010.

[13] H. Karau, A. Konwinski, P. Wendell, and M. Zaharia. *Learning Spark: Lightning-Fast Big Data Analysis*. " O'Reilly Media, Inc.", 2015.

[14] M. Krötzsch. OWL 2 Profiles: An Introduction to Lightweight Ontology Languages. In *Reasoning Web International Summer School*, Vienna, Austria, 2012. Springer.

[15] J. Leskovec, A. Rajaraman, and J. D. Ullman. *Mining of Massive Datasets*. Cambridge University Press, 2014.

[16] Y. Liu. *Inference as a Data Management Problem*. PhD thesis, Imperial College London, October 2016. Available at http://hdl.handle.net/10044/1/44960.

[17] Y. Liu and P. McBrien. Transactional and Incremental Type Inference from Data Updates. In *Proceedings of BICOD 2015*, pages 206–219, Edinburgh, UK, 6–8 July 2015. Springer.

[18] Y. Liu and P. McBrien. Transactional and Incremental Type Inference from Data Updates. *The Computer Journal*, 60(3), 2016.

[19] Z. Liu, Z. Feng, X. Zhang, X. Wang, and G. Rao. RORS: Enhanced Rule-based OWL Reasoning on Spark. In *Asia-Pacific Web Conference*, pages 444–448. Springer, 2016.

[20] P. McBrien, N. Rizopoulos, and A. C. Smith. SQOWL: Type Inference in an RDBMS. In *Proceedings of ER 2010*, pages 362–376, Vancouver, BC, Canada, 1–4 Nov. 2010. Springer.

[21] G. Meditskos and N. Bassiliades. Combining a DL Reasoner and a Rule Engine for Improving Entailment-Based OWL Reasoning. In *Proceedings of ISWC 2008*, pages 277–292, Karlsruhe, Germany, 2008. Springer.

[22] B. Motik, B. C. Grau, I. Horrocks, Z. Wu, A. Fokoue, and C. Lutz, editors. *OWL 2 Web Ontology Language: Profiles (Second Edition)*. W3C Recommendation, 2012. Latest version available at http://www.w3.org/TR/owl2-profiles/.

[23] S. Muñoz-Venegas, J. Pérez, and C. Gutierrez. Simple and Efficient Minimal RDFS. *J. Web Sem.*, 7(3):220–234, 2009.

[24] C. Pang, G. Dong, and K. Ramamohanarao. Incremental Maintenance of Shortest Distance and Transitive Closure in First-Order Logic and SQL. *ACM Transactions on Database Systems (TODS)*, 30(3):698–721, 2005.

[25] H. Pérez-Urbina, E. Rodríguez-Díaz, M. Grove, G. Konstantinidis, and E. Sirin. Evaluation of Query Rewriting Approaches for OWL 2. In *Proceedings of SSWS+HPCSW*, volume 943, pages 32–44, Boston, USA, 2012. CEUR-WS.org.

[26] K. Rohloff and R. E. Schantz. High-performance, massively scalable distributed systems using the mapreduce software framework: the SHARD triple-store. In *PSI EtA*, page 4. ACM, 2010.

[27] A. Schätzle, M. Przyjaciel-Zablocki, and G. Lausen. PigSPARQL: Mapping SPARQL to pig latin. In *Proceedings of the International Workshop on Semantic Web Information Management*, page 4. ACM, 2011.

[28] R. Shearer, B. Motik, and I. Horrocks. HermiT: A Highly-Efficient OWL Reasoner. In *OWLED*, volume 432, page 91, 2008.

[29] E. Sirin, B. Parsia, B. C. Grau, A. Kalyanpur, and Y. Katz. Pellet: A practical OWL-DL reasoner. *J. Web Sem.*, 5(2):51–53, 2007.

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

[31] J. Urbani, S. Kotoulas, J. Maassen, F. Van Harmelen, and H. Bal. WebPIE: A Web-scale Parallel Inference Engine using MapReduce. *J. Web Sem.*, 10:59–75, 2012.

[32] Z. Wu, G. Eadon, S. Das, E. I. Chong, V. Kolovski, M. Annamalai, and J. Srinivasan. Implementing an Inference Engine for RDFS/OWL Constructs and User-Defined Rules in Oracle. In *Proceedings of ICDE 2008*, pages 1239–1248, Cancún, México, 7–12 Apr. 2008. IEEE, New York.