A Scalable Approach for Distributed Reasoning over Large-scale OWL Datasets

Heba Mohamed\textsuperscript{1,2}, Said Fathalla\textsuperscript{1,2}, Jens Lehmann\textsuperscript{1,3}, and Hajira Jabeen\textsuperscript{4}

\textsuperscript{1}Smart Data Analytics (SDA), University of Bonn, Bonn, Germany
\textsuperscript{2}Faculty of Science, University of Alexandria, Alexandria, Egypt
\textsuperscript{3}NetMedia Department, Fraunhofer IAIS, Dresden Lab, Germany
\textsuperscript{4}Cluster of Excellence on Plant Sciences (CEPLAS), University of Cologne, Cologne, Germany

\{hmohamed, fathalla, jens.lehmann\}@cs.uni-bonn.de, hajira.jabeen@uni-koeln.de

Keywords: Big Data, Distributed Computing, In-Memory Computation, Parallel Reasoning, OWL Horst rules, OWL Axioms.

Abstract: With the tremendous increase in the volume of semantic data on the Web, reasoning over such an amount of data has become a challenging task. On the other hand, the traditional centralized approaches are no longer feasible for large-scale data due to the limitations of software and hardware resources. Therefore, horizontal scalability is desirable. We develop a scalable distributed approach for RDFS and OWL Horst Reasoning over large-scale OWL datasets. The eminent feature of our approach is that it combines an optimized execution strategy, pre-shuffling method, and duplication elimination strategy, thus achieving an efficient distributed reasoning mechanism. We implemented our approach as open-source in Apache Spark using Resilient Distributed Datasets (RDD) as a parallel programming model. As a use case, our approach is used by the SANSA framework for large-scale semantic reasoning over OWL datasets. The evaluation results have shown the strength of the proposed approach for both data and node scalability.

1 INTRODUCTION

The past decades have seen advances in artificial intelligence techniques for enabling various sorts of reasoning. The two primary forms of reasoning performed by inference engines are Forward Chaining and Backward Chaining. Forward chaining (also called data-driven or bottom-up approach) approaches use available data and inference rules to infer implicit information until a goal is reached (Sharma et al., 2012). Contrary, backward chaining (also called goal-driven or up-down approach) approaches use the goal and inference rules to move backward until determining the facts that satisfy the goal (Sharma et al., 2012). Forward chaining techniques can generate a large amount of information from a small amount of data (Al-Ajlan, 2015). As a consequence, it has become the key approach in reasoning systems. RDFS and OWL mainly permit the description of the relationships between the classes and properties used to structure and define entities using rich formal semantics, providing a declarative and extensible space of discourse. RDFS and OWL both define a set of rules which allow new data to be inferred from the original input. The volume of semantic data available on the Web is expanding precipitously, e.g., the English version of DBpedia describes approximately 4.5 million things.\textsuperscript{1} A plethora of new information can be inferred from such large-scale semantic data with the help of RDFS/OWL reasoning rules. However, reasoning over such a vast amount of data presents a series of challenges when developing algorithms that can run in a distributed manner (Gu et al., 2015). Owing to the limitations of software and hardware infrastructures, as well as the complex data flow models, conventional single-node strategies are no longer appropriate for such large-scale semantic data. The limitations of the traditional centralized approaches (i.e., software and hardware resources) for large-scale data have motivated us to develop a distributed approach that is capable of reasoning over...
large amount of data being produced regularly to gain implicit knowledge. Therefore, we introduce an attempt of developing a scalable distributed approach for RDFS and OWL Horst Reasoning over large-scale OWL datasets. In addition, we found that selecting the optimum rule execution order greatly improves the performance. The eminent feature of our approach is that it combines an optimized execution strategy, pre-shuffling method, and duplication elimination strategy, thus achieving an efficient distributed reasoning mechanism.

The Spark framework is an in-memory distributed framework using Resilient Distributed Dataset (RDD) as the primary abstraction for parallel processing provided by Spark. There are many benefits of using RDDs, including in-memory computation, fault-tolerance, partitioning, and persistence across worker nodes. Our approach uses Apache Spark for distributed and scalable in-memory processing. We aim at answering the question: (RQ) How can we infer implicit knowledge from large-scale Linked Data distributed across various nodes? The proposed approach is capable of reasoning over ontologies partitioned across various nodes concurrently. Our experiments showed that the proposed approach is scalable with respect to data, as well as nodes. We affirm that the formal semantics specified by the input ontology are not violated. The sustainability of our approach is demonstrated through SANSA-Stack contributor\[2\] whereas our approach constitutes the inference layer in the SANSA-Stack, which is about reasoning over a very large number of axioms. The SANSA framework\[3\] uses Apache Spark\[4\] as its big data engine for scalable large-scale RDF data processing.

Furthermore, SANSA has tools for representing, querying, inference, and analysing semantic data. A new release is published almost every six months since 2016. In addition, bugs and improvement suggestions can be submitted through the issue tracker on its GitHub repository.

This paper is organized as follows: Section\[2\] gives a brief overview of the related work on RDF semantic data reasoning. The proposed methodology for axiom-based RDFS/OWL Horst parallel reasoning is presented in Section\[3\]. The experimental setup and discussion of the results are presented in Section\[4\]. Finally, we conclude and the potential extension of the proposed approach in Section\[5\].

\[2\] http://sansa-stack.net/community/#Contributors
\[3\] http://sansa-stack.net/
\[4\] https://spark.apache.org/

2 Related work

For many years, the developments of large-scale reasoning systems were based on the P2P self-organizing networks, which are neither efficient nor scalable (Gu et al., 2015). Distributed in-memory computation systems, such as Apache Spark or Flink, are used to perform inferences over large-scale OWL datasets. These frameworks are robust and can operate on a cluster of multiple nodes, i.e., the workload is distributed over multiple machines.

Several existing approaches can perform reasoning over RDF data, but none of them can reason over large-scale axiom-based OWL data. Jacopo et al. (Urbani et al., 2010) have proposed WebPIE, an inference engine built on top of the Hadoop platform for parallel OWL Horst forward inference. WebPIE has been evaluated using real-world, large-scale RDF datasets (i.e., over 100 billion triples). Results have shown that WebPIE is scalable and significantly outperforms the state-of-the-art systems in terms of language expressivity, data size, and inference performance. In 2012, an optimized version of WebPIE (Urbani et al., 2012) was proposed, which loads the schema triples in memory and, when possible, executes the join on-the-fly instead of in the reduce phase, uses the map function to group the triples to avoid duplicates and execute the RDFS rules in a specific order to minimize the number of MapReduce jobs. Gu et al. (Gu et al., 2015) have proposed Cichlid, a distributed reasoning algorithm for RDFS and OWL Horst rule set using Spark. Several design issues have been considered when developing the algorithm, including a data partitioning model, the execution order of the rules, and eliminating duplicate data. The prominent feature of Cichlid is that the inner Spark execution mechanism is optimized using an off-heap memory storage mechanism for RDD. Liu et al. (Liu et al., 2016) have presented an approach for enhancing the performance of rule-based OWL inference by analyzing the rule interdependence of each class to find the optimal executable strategies. They have implemented a prototype called RORS using Spark. Compared with Cichlid, RORS can infer more implicit triples than Cichlid in less execution time. Yu and Peter (Liu and McBrien, 2017) have presented a Spark-based OWL (SPOWL) reasoning approach that maps axioms in the T-Box to RDDs in order to conclude the reasoning results entailed by the ontology using a tableaux reasoner. SPOWL efficiently caches data in distributed memory to reduce the amount of I/O used, which significantly improves the performance. Furthermore, they have proposed an optimized order of rule execution by analyzing the dependencies between RDDs.
Kim and Park (Kim and Park, 2015) have presented an approach for scalable OWL ontology reasoning in a Hadoop-based distributed computing cluster using SPARK. They have overcome the limitation of I/O delay (i.e., disk-based MapReduce approach) by loading data into the memory of each node. In comparison with WebPIE, this approach outperforms the WebPIE performance when the LUBM dataset is used in the evaluation.

What additionally distinguishes our work from the related work mentioned above is the utilization of the optimal execution strategy proposed by Liu et al. (Liu et al., 2016) and the pre-shuffling method proposed by Gu et al. (Gu et al., 2015) with our duplication elimination strategy (more details in the next section).

3 Parallel Reasoning

In this section, we present the methodology of the proposed approach for axiom-based RDFS/OWL Horst parallel reasoning. Our methodology comprise two tasks; 1) RDFS Parallel Reasoning, in which RDFS rules are applied until no new data is derived, and 2) OWL Parallel Reasoning, in which reasoning over the OWL Horst rule set is performed.

3.1 RDFS Parallel Reasoning

RDFS reasoning is an iterative reasoning process where RDFS rules are applied to the input data until no new data is derived. The RDFS rule set consists of 13 rules, as listed in (Hayes, 2004). The rules with one clause are removed from the reasoning process because they do not influence the reasoning process. Table 1 illustrates the RDFS ruleset (in axiom-based structure) used in this paper. All reasoning steps are implemented using the following RDD operations: map, filter, join, and union operations.

Figure 1 shows the distributed RDFS reasoning approach using Spark RDD operations. First, the approach starts with the input OWL dataset, then converts it to the corresponding RDD of axioms. Next, the approach selects one rule to filter and combine the input RDD based on the semantics of this rule. Next, the derived results (i.e., the inferred axioms after applying the reasoning rule from the previous step) are merged with the original input data (i.e., the initial RDD[axioms]). Finally, the resulting dataset is used as input for the next reasoning rule. All rules are applied iteratively until a fixed-point iteration, in which no new data is produced, and then the reasoning process is terminated.

In many parallel RDFS reasoning approaches, each rule is applied through join operations, and the related data is read from various nodes. Consequently, due to the massive volume of data traffic between nodes, we encounter high network overhead. We can broadcast the schema axioms to every node before conducting the reasoning process, as the number of schema axioms is minimal and conserves constants in the real world (Heino and Pan, 2012; Gu et al., 2015). The use of broadcast variables avoids network communication overhead, whereas the corresponding data used by the join operation is now available in each node. Algorithm 1 illustrates the usage of the broadcast variables to optimize the execution of a rule $R_6$. Line 2 extracts all the OWLClassAssertion axioms from the ax RDD. Next, the algorithm extracts all the OWLSubClassOf axioms and converts it
Table 2: RDFS rule set in terms of OWL Axioms. The following notation conventions are used: "C" is an OWLClass, "R" is an OWLObjectProperty, "P" is OWLDataProperty, "A" is an OWLAnnotationProperty, "i" is an OWLIndividual, "v" is OWLLiteral, "s" is an OWLAnnotationSubject, and "u", "w", "x", and "y" are instances. All names may have subscripts ($i_1$ ... $i_n$).

<table>
<thead>
<tr>
<th>Rule</th>
<th>Condition</th>
<th>Axiom to Add</th>
</tr>
</thead>
<tbody>
<tr>
<td>R1</td>
<td>SubClassOf $(C_1, C_2)$. SubClassOf $(C_1, C_2)$</td>
<td>SubClassOf $(C, C_2)$</td>
</tr>
<tr>
<td>R2(a)</td>
<td>SubDataPropertyOf $(P_1, P_2)$. SubDataPropertyOf $(P_1, P_2)$</td>
<td>SubDataPropertyOf $(P, P_2)$</td>
</tr>
<tr>
<td>R2(b)</td>
<td>SubObjectPropertyOf $(R_1, R_2)$. SubObjectPropertyOf $(R_1, R_2)$</td>
<td>SubObjectPropertyOf $(R, R_2)$</td>
</tr>
<tr>
<td>R2(c)</td>
<td>SubAnnotationPropertyOf $(A, A_1)$. SubAnnotationPropertyOf $(A, A_1)$</td>
<td>SubAnnotationPropertyOf $(A, A_2)$</td>
</tr>
<tr>
<td>R3(a)</td>
<td>DataPropertyAssertion $(P_i v)$. DataPropertyAssertion $(P_i v)$</td>
<td>DataPropertyAssertion $(P_1, i v)$</td>
</tr>
<tr>
<td>R3(b)</td>
<td>ObjectPropertyAssertion $(R_i, i_v)$. SubObjectPropertyOf $(R, R_i)$</td>
<td>ObjectPropertyAssertion $(R_1, i_1 v)$</td>
</tr>
<tr>
<td>R3(c)</td>
<td>AnnotationAssertion $(A, v)$. SubAnnotationPropertyOf $(A, A_1)$</td>
<td>AnnotationAssertion $(A, 1_i v)$</td>
</tr>
<tr>
<td>R4(a)</td>
<td>DataPropertyDomain $(P, C)$. DataPropertyAssertion $(P, i v)$</td>
<td>ClassAssertion $(C, i)$</td>
</tr>
<tr>
<td>R4(b)</td>
<td>ObjectPropertyDomain $(R, C)$. ObjectPropertyAssertion $(R, i_1 v)$</td>
<td>ClassAssertion $(C, i_1)$</td>
</tr>
<tr>
<td>R5(a)</td>
<td>DataPropertyRange $(P, C)$. DataPropertyAssertion $(P, i v)$</td>
<td>ClassAssertion $(C, i)$</td>
</tr>
<tr>
<td>R5(b)</td>
<td>ObjectPropertyRange $(R, C)$. ObjectPropertyAssertion $(R, i_1 v)$</td>
<td>ClassAssertion $(C, i_1)$</td>
</tr>
<tr>
<td>R6</td>
<td>SubClassOf $(C, C_1)$ . ClassAssertion $(C, i)$</td>
<td>ClassAssertion $(C_1, i)$</td>
</tr>
</tbody>
</table>

Algorithm 1: Parallel reasoning algorithm of rule $R_6$ using broadcast variables

```
Input : ax: RDD of OWL Axioms
Output: Inferred axioms from rule $R_6$
1 begin
   2   val ta = extract(ax, AxiomType.ClassAssertion)
   3   val sc = extract(ax, AxiomType.SubClassOf)
   4   .map(a => (a/SubClass, a/SuperClass))
   5   .collect() toMap
   6   val scBC = sc broadcast(sc)
   7   val R6 = ta.filter(a =>
                     .scBC.value.contains(a/ClassExpression))
   8   .flatMap(a => scBC.value.a/ClassExpression)
   9   .map(s =>
             ClassAssertionAxiom(s, a/GetIndividual))
10 end
```

to the corresponding Map (lines 3-5). Line 6 broadcasts the OWLSubClassOf map to all the nodes of the cluster. Finally, the OWLSubClassOf axioms are filtered out using the OWLSubClassOf broadcast variable $scBC$ and mapped to get the inferred axioms based on the semantics of $R_6$ (lines 7-11).

3.2 OWL Horst Parallel Reasoning

In addition to RDFS reasoning, there is another more powerful and complex rule set called OWL Horst reasoning rule set. Reasoning over the OWL Horst rule set is known as OWL reasoning (Ter Horst, 2005). Table 2 lists the set of OWL Horst rules.

3.2.1 Rule Analysis

The rules of OWL Horst are more complicated than those of RDFS, and we need to conduct more jobs to evaluate the complete closure (Urbani et al., 2009). The OWL datasets contain several OWL Axioms structures. In this approach, we divide the OWL Axioms into three main categories: 1) SameAs axioms, which are OWLSameIndividual axioms, 2) Type axioms, which are OWLClassAssertion axioms, and 3) SPO, which are the remaining axioms. Due to the complexity of the OWL Horst rules, we have analysed the set of rules and divide them into four main rule categories:

- **SameAs rules**: Rules whose antecedent or consequence include OWLSameIndividual axiom. Those rules include $O_1, O_2, O_5, O_9, O_9$, and $O_{10}$. According to (Kim and Park, 2015) [Gu et al., 2015], rules $O_9$ and $O_{10}$ are used for ontology merging and can be excluded from the reasoning process.

- **Type rules**: Rules whose antecedent or consequence include OWLClassAssertion axiom. Those rules include $R_1, R_2, R_6, O_{13}, O_{14}, O_{15}$, and $O_{16}$.

- **SPO rules**: Rules whose antecedent or consequence include axioms from SPO category. Those rules include $R_1, O_3, O_4$, and $O_7$.

- **Schema rules**: The remaining rules which are $R_1, R_2, O_{11}$, and $O_{12}$. 

<table>
<thead>
<tr>
<th>Rule</th>
<th>Condition</th>
<th>Axiom to Add</th>
</tr>
</thead>
<tbody>
<tr>
<td>O1</td>
<td>FunctionalObjectProperty (R). ObjectPropertyAssertion (R i u). ObjectPropertyAssertion (R i w)</td>
<td>SameIndividual (u w)</td>
</tr>
<tr>
<td>O2</td>
<td>InverseFunctionalObjectProperty (R). ObjectPropertyAssertion (R i u). ObjectPropertyAssertion (R w i)</td>
<td>SameIndividual (u w)</td>
</tr>
<tr>
<td>O3</td>
<td>SymmetricObjectProperty (R). ObjectPropertyAssertion (R u w). ObjectPropertyAssertion (R u w)</td>
<td>ObjectPropertyAssertion (R w u)</td>
</tr>
<tr>
<td>O4</td>
<td>TransitiveObjectProperty (R). ObjectPropertyAssertion (R i u). ObjectPropertyAssertion (R i w)</td>
<td>ObjectPropertyAssertion (R w u)</td>
</tr>
<tr>
<td>O5</td>
<td>SameIndividual (u w).</td>
<td>SameIndividual (u w)</td>
</tr>
<tr>
<td>O6</td>
<td>SameIndividual (i u), SameIndividual (u w)</td>
<td>SameIndividual (i w)</td>
</tr>
<tr>
<td>O7(a)</td>
<td>InverseObjectProperties (R1 R2). ObjectPropertyAssertion (R1 u w). ObjectPropertyAssertion (R2 u w)</td>
<td>ObjectPropertyAssertion (R2 w u)</td>
</tr>
<tr>
<td>O7(b)</td>
<td>InverseObjectProperties (R1 R2). ObjectPropertyAssertion (R1 u w). ObjectPropertyAssertion (R2 u w)</td>
<td>ObjectPropertyAssertion (R2 w u)</td>
</tr>
<tr>
<td>O8</td>
<td>Declaration (Class (C1)). SameIndividual (C1, C2). SubClassOf (C1, C2)</td>
<td>SubObjectPropertyOf (R1, R2)</td>
</tr>
<tr>
<td>O9(a)</td>
<td>Declaration (DataProperty (P1)). SameIndividual (P1, P2).</td>
<td>SubDataPropertyOf (P1, P2)</td>
</tr>
<tr>
<td>O9(b)</td>
<td>Declaration (ObjectProperty (R1)). SameIndividual (R1, R2).</td>
<td>SubObjectPropertyOf (R1, R2)</td>
</tr>
<tr>
<td>O10(a)</td>
<td>DataPropertyAssertion (P x y). SameIndividual (u x), SameIndividual (w y)</td>
<td>DataPropertyAssertion (P x y)</td>
</tr>
<tr>
<td>O10(b)</td>
<td>ObjectPropertyAssertion (R u w). SameIndividual (u x), SameIndividual (w y)</td>
<td>ObjectPropertyAssertion (R w y)</td>
</tr>
<tr>
<td>O11(a)</td>
<td>EquivalentClasses (C1, C2).</td>
<td>SubClassOf (C1, C2)</td>
</tr>
<tr>
<td>O11(b)</td>
<td>EquivalentClasses (C1, C2).</td>
<td>SubClassOf (C1, C2)</td>
</tr>
<tr>
<td>O11(c)</td>
<td>SubClassOf (C1, C2). SubClassOf (C2, C1).</td>
<td>EquivalentClasses (C1, C2)</td>
</tr>
<tr>
<td>O12(a1)</td>
<td>EquivalentDataProperty (P1 P2).</td>
<td>SubDataPropertyOf (P1, P2)</td>
</tr>
<tr>
<td>O12(b1)</td>
<td>EquivalentDataProperty (P1 P2).</td>
<td>SubDataPropertyOf (P1, P2)</td>
</tr>
<tr>
<td>O12(c1)</td>
<td>SubDataPropertyOf (P1 P2).</td>
<td>EquivalentDataProperty (P1 P2)</td>
</tr>
<tr>
<td>O12(a2)</td>
<td>EquivalentObjectProperty (R1 R2).</td>
<td>SubObjectPropertyOf (R1, R2)</td>
</tr>
<tr>
<td>O12(b2)</td>
<td>EquivalentObjectProperty (R1 R2).</td>
<td>SubObjectPropertyOf (R2, R1)</td>
</tr>
<tr>
<td>O12(c2)</td>
<td>SubObjectPropertyOf (R1 R2). SubObjectPropertyOf (R2, R1).</td>
<td>EquivalentObjectProperty (R1 R2)</td>
</tr>
<tr>
<td>O13(a1)</td>
<td>EquivalentClasses (C, DataHasValue (P v)). DataPropertyAssertion (P i v).</td>
<td>ClassAssertion (C i)</td>
</tr>
<tr>
<td>O13(b1)</td>
<td>SubClassOf (C, DataHasValue (P v)). DataPropertyAssertion (P i v).</td>
<td>ClassAssertion (C i)</td>
</tr>
<tr>
<td>O13(a2)</td>
<td>EquivalentClasses (C, ObjectHasValue (R v)). ObjectPropertyAssertion (R i v).</td>
<td>ClassAssertion (C i)</td>
</tr>
<tr>
<td>O13(b2)</td>
<td>SubClassOf (C, ObjectHasValue (R v)). ObjectPropertyAssertion (R i v).</td>
<td>ClassAssertion (C i)</td>
</tr>
<tr>
<td>O14(a1)</td>
<td>ClassAssertion (DataHasValue (P v), i).</td>
<td>DataPropertyAssertion (P i v)</td>
</tr>
<tr>
<td>O14(b1)</td>
<td>SubClassOf (C, DataHasValue (P v)). ClassAssertion (C i).</td>
<td>DataPropertyAssertion (P i v)</td>
</tr>
<tr>
<td>O14(c1)</td>
<td>EquivalentClasses (C, DataHasValue (P v)).</td>
<td>DataPropertyAssertion (P i v)</td>
</tr>
<tr>
<td>O14(a2)</td>
<td>ClassAssertion (ObjectHasValue (R v), i).</td>
<td>ObjectPropertyAssertion (R i v)</td>
</tr>
<tr>
<td>O14(b2)</td>
<td>SubClassOf (C, ObjectHasValue (R v)). ClassAssertion (C i).</td>
<td>ObjectPropertyAssertion (R i v)</td>
</tr>
<tr>
<td>O14(c2)</td>
<td>EquivalentClasses (C, ObjectHasValue (R v)). ClassAssertion (C i).</td>
<td>ObjectPropertyAssertion (R i v)</td>
</tr>
<tr>
<td>O15</td>
<td>ObjectPropertyAssertion (R u w). ClassAssertion (C u).</td>
<td>ClassAssertion (C u)</td>
</tr>
<tr>
<td>O16</td>
<td>ObjectPropertyAssertion (R u w). ClassAssertion (C u).</td>
<td>ClassAssertion (C u)</td>
</tr>
</tbody>
</table>
Figure 2 illustrates the architecture of the proposed approach. In the beginning, SANSA framework (SANSA-OWL layer\(^5\)) is used to convert the input functional syntax ontology to the corresponding RDD[OWLAXiom]. Afterwards, the filter() transformation is applied on the input RDD based on the semantic of each rule (i.e., extract the intended axioms for each specific rule). Then, the execution rule order is performed, as illustrated in the reasoning step. Relying on the interdependence of rules in each category, we selected the optimum rule execution strategy based on the order proposed in (Liu et al., 2016). The execution order for schema category is \(O_{11a} \rightarrow O_{11b} \rightarrow R_1 \rightarrow O_{11c} \rightarrow O_{12a}, O_{12b} \rightarrow R_2 \rightarrow O_{12c}\). After applying the schema rules, we choose the rule execution order for SPO category as \(O_3 \rightarrow R_3 \rightarrow O_7 \rightarrow O_1\). Afterwards, comes the rule execution order for Type rule category which is \(R_4 \rightarrow R_5 \rightarrow R_6 \rightarrow O_{14} \rightarrow O_{13} \rightarrow O_{15} \rightarrow O_{16}\). Finally, we ended with sameAs rule execution order which is \(O_1 \rightarrow O_{10} \rightarrow O_2 \rightarrow O_6 \rightarrow O_3\). Ultimately, the inferred OWL axioms are merged with the input ontology.

Algorithm 2 describes the parallel OWL-Horst reasoning approach. The algorithm starts by applying the schema rules over the input axioms (line 2). Then, the reasoning process repeatedly applies rules over the input data (lines 3-12). After each step, the derived data is merged with the original data, i.e., the output of one rule is input to the next one. Next, sameAs rules are applied over the output (lines 13-14). Finally, before returning the inferred axioms, we apply distinct() transformations to eliminate the duplicated axioms (line 15), in which the degree of parallelism is passed. The degree of parallelism is specified based on the number of cores in the cluster.

### 3.2.2 Optimization for join reasoning.

In OWL Horst rules, there is more than one rule that requires multiple join operations. Such rules contain more than one OWLAssertionAxiom. Since the schema axioms are usually small, the optimization algorithm (using broadcast variables) is also used to optimize the OWL Horst reasoning.

---

\(^5\) https://github.com/SANSA-Stack/SANSA-OWL
3.2.3 Eliminating Duplicate Axioms

The pre-shuffle optimization method (proposed in [Gu et al., 2015]) is used to eliminate the duplicated axioms using distinct operation over the resultant Spark RDD. In the pre-shuffle strategy, an RDD is partitioned and cached in memory before invoking the distinct operation on it, to optimize the wide dependencies among RDDs against the narrow dependencies. Wide dependencies can generally be avoided when constructing distributed algorithms to prevent overhead communication. After studying the state of the art, we observed that almost all of them eliminate duplicate data after the reasoning process, which greatly increases the running time. Therefore, in the proposed approach, we eliminate the duplicated data from the derived results of each set of rules in each category, i.e., SameAs, Type, SPO, and Schema categories. Therefore, pre-shuffle method is applied before performing the duplicate elimination strategy. We found that the elimination of duplication before performing reasoning of the rules from the next category has reduced the number of axioms used in the next process, which decreases the reasoning time dramatically. Experiment 3 (in subsection 4.2) illustrates the performance improvement with and without using the proposed duplicate elimination strategy.

3.2.4 Transitive Rules

To conduct the transitive closure, a self-join operation is needed on the set of transitive pairs. Performing the self-join over the transitive pairs is not efficient because it requires a large amount of redundant work. Moreover, it performs too many iterative processes which significantly increases the running time. For example, consider the transitive chain \( a \rightarrow b \rightarrow c \rightarrow d \rightarrow e \). The implicit transitive pair \((a, e)\) can be produced multiple times, such as computed from \((a, c)\) and \((c, e)\), or from \((a, b)\) and \((b, e)\), etc. To enhance the reasoning performance, we adopt the transitive closure algorithm proposed in [Liu et al., 2016].

Algorithm 3 describes in detail the essential steps carried out by the parallel transitive closure algorithm. The transitive pairs \( tc \) are passed as input to the algorithm. For example, to compute the transitive rule \( R_1 \), we pass all pairs of OWLSubClassOf axioms as input. Consider that we have the two axioms OWLSubClassOf\((a, b)\) and OWLSubClassOf\((b, c)\), then the \( tc \) RDD will contains the two pairs \([a, b], [b, c]\). In the beginning, the transitive pairs are cached in memory for further computations (line 3). Then, the pairs are swapped \( y = ([b, a], [c, b]) \) and partitioned across the cluster using \( p \), which is a defined hash partitioner (lines 5-6). Line 9 performs the join operation on the original and swapped pairs. For example, we perform join over the pairs \((a, b), (b, c)\) to get RDD of \([b, c, (a, c)]\) then map the output to get the new \((a, c)\) paths. After that, line 11 gets only the new inferred pair which is \((a, c)\) pair. If we get at least one new pair, then the process is repeated to get the new pair in the transitive chain (lines 14-18). Finally, line 19 adds the inferred pairs to the original RDD of transitive pairs (i.e., \( tc \)) and hence \( tc = [(a, b), (b, c), (a, c)] \). The process is continued until no new pairs are inferred. In Algorithm 3 the cache() operation is invoked over the joined RDDs. Caching the data before the join operation substantially enhances the performance of the algorithm. The transitive closure algorithm can also be used to compute \( O_b \).

4 Evaluation

In this section, we describe the evaluation of the proposed approach. We conducted three types of experiments; 1) Experiment 1: we evaluate the reasoning time of our distributed approach with different data sizes to analyze the data scalability and the performance of the reasoning process, 2) Experiment 2: we measure the speedup performance by increasing
the number of working nodes (i.e., machines) in the cluster environment to evaluate the horizontal node scalability, and 3) Experiment 3: we measure the reasoning time of our approach with and without the proposed duplicate elimination strategy to analyze the performance of the reasoning process in both cases. We start with the experimental setup, then we present and discuss the results. The objective is to answer the following evaluation questions (EQs):

- **EQ1** How does the runtime of the proposed approach be influenced with and without caching?
- **EQ2** How does the distribution of the workload over multiple machines affect the runtime?
- **EQ3** How does the speedup ratio vary regarding the number of worker nodes?
- **EQ4** How efficient is the proposed approach when the reasoning is distributed over multiple machines?

**Metrics.** Two metrics are used to measure the performance of parallel algorithms; Speedup ratio and Efficiency. The speedup ratio \( S \) is a significant metric for measuring the performance of parallel algorithms against serial ones. Mathematically, the speedup ratio is defined as \( S = \frac{T_L}{T_N} \), where \( T_L \) is the execution time of the algorithm in local mode, and \( T_N \) is the time of \( N \) workers. Efficiency \( E = \frac{S}{N} \) represents how well parallel algorithms utilize the computational resources by measuring the speedup per worker. It is the time taken to run the algorithm on \( N \) workers in comparison with the time to run it on a local machine.

### 4.1 Experimental Setup

**System configuration.** All distributed experiments have been run on a cluster with five nodes. Among these nodes, one is reserved to act as the master, and the other four nodes are the computing workers. Each node has AMD Opteron 2.3 GHz processors (64 Cores) with 250 GB of memory, and the configured capacity is 1.7 TB. The nodes are connected with 1 Gb/s Ethernet. Furthermore, Spark v2.4.4 and Hadoop v2.8.1 with Java 1.8.0 are installed on this cluster. All local-mode experiments are carried out on a single cluster instance. All distributed experiments have been run three times, and we reported the average execution time.

**Benchmark.** Lehigh University (LUBM) (Guo et al., 2005) synthetic benchmark has been used for the experiment. For the evaluation of Semantic Web repositories, LUBM is used to assess systems with various reasoning and storage process capacities over large datasets. The benchmark is expressed in OWL Lite language. We used the LUBM data generator in our experiment to generate five datasets with different sizes: LUBM-5, LUBM-10, LUBM-20, LUBM-50, and LUBM-100, where the number attached to the benchmark name, e.g., 20 in LUBM-20, is the number of generated universities. In addition, to evaluate the efficiency of the proposed approach with more complex OWL languages, we used the University Ontology Benchmark (UOBM) UOBM extends LUBM (Guo et al., 2005) and generates more realistic datasets. UOBM covers a complete set of OWL 2 constructs by including both OWL Lite and OWL DL ontologies. UOBM generates three different data sets: one, five, and ten universities. Properties of the generated datasets, loading time to the HDFS, the number of axioms of each dataset, and the number of inferred axioms are listed in Table 3.

### 4.2 Results and Discussion

**Experiment 1 (Data Scalability):** In this experiment, we increase the size of the input OWL datasets to measure the effectiveness of the proposed reasoning approach. In the cluster environment, we preserve a constant number of nodes (workers) at five and raise the size of datasets to determine whether larger datasets can be processed by the proposed approach. We run the experiments on five different datasets from the LUBM benchmark to measure the data scalability. We start by creating a dataset of five universities (i.e.,

---

6. https://www.cs.ox.ac.uk/isg/tools/UOBMGenerator/
Table 3: LUBM benchmark datasets (functional syntax)

<table>
<thead>
<tr>
<th>Dataset</th>
<th>Size</th>
<th>Load time (sec)</th>
<th>#Axioms</th>
<th>#Inferred</th>
<th>Speedup</th>
<th>Efficiency</th>
</tr>
</thead>
<tbody>
<tr>
<td>LUBM-5</td>
<td>118 MB</td>
<td>3</td>
<td>636,446</td>
<td>274,563</td>
<td>1.4x</td>
<td>0.3</td>
</tr>
<tr>
<td>LUBM-10</td>
<td>284 MB</td>
<td>9</td>
<td>1,316,517</td>
<td>559,873</td>
<td>1.7x</td>
<td>0.3</td>
</tr>
<tr>
<td>LUBM-20</td>
<td>514 MB</td>
<td>10</td>
<td>2,781,477</td>
<td>1,182,585</td>
<td>1.8x</td>
<td>0.4</td>
</tr>
<tr>
<td>LUBM-50</td>
<td>1.3 GB</td>
<td>17</td>
<td>6,654,755</td>
<td>2,924,925</td>
<td>2.1x</td>
<td>0.4</td>
</tr>
<tr>
<td>LUBM-100</td>
<td>2.7 GB</td>
<td>48</td>
<td>13,310,784</td>
<td>6,204,835</td>
<td>2.5x</td>
<td>0.5</td>
</tr>
</tbody>
</table>

LUBM-5), then we increase the number of universi-

ties (i.e., scaling up the size).

Figure 3 indicates the run time of the proposed ap-

proach with and without caching mechanism for each
dataset. Caching is a technique for speeding up pro-

cesses with multiple access to the same RDD, which
holds the data in memory and speeds up the computa-
tions. The x-axis represents the produced LUBM
datasets with an increase in the number of universi-
ties, while the y-axis represents the time of execu-
tion in hours. As illustrated in Figure 3, the reduc-
tion in execution time between the proposed approach (blue columns) and without caching (red ones) is clear (response to EQ1). For instance, OWL Horst rea-

soning on LUBM-100 costs around 23 hours with-

out caching, while the time is decreased by 52% (i.e., becomes 11.3 hours) after the caching was trig-

gered. Therefore, we conclude that the proposed ap-

proach is scalable in terms of data scalability. Fig-

ure 4 displays the performance of the proposed ap-

proach obtained in a cluster environment compris-
ing five worker machines. For example, consider the

LUBM-100 dataset; the execution time in a single (local) machine environment is decreased from 28.3

hours to 11.3 hours compared to the cluster environ-

ment. The reason is the distribution of the computa-
tions across multiple machines. The usage of clus-
ter mode has decreased the running time by 60% for

LUBM-100 (response to EQ2).

Experiment 2 (Node Scalability): In this experi-

ment, we vary the number of nodes in the cluster (with

one node each time from one to five) with an eye to-

wards affirming the node scalability. Figure 5 illus-

trates the scalability performance of our approach by

increasing the number of worker nodes from one to

two for the LUBM-10 dataset. The execution time

was dramatically decreased from 146 to 40 minutes, i.e., approximately one-third. It is apparent that

the time of execution drops linearly as the number of

workers increases (response to EQ3). The speedup

and efficiency ratios for the five datasets are shown in

Table 3. Concerning the number of workers, the

speed of the selected data is improved sequentially

(response to EQ4). In conclusion, the results illustrate

that the proposed OWL Horst reasoning approach has

achieved near-linear output scalability in the context

of speedup.

Experiment 3 (Duplicate Elimination): In this

experiment, we study the effect of eliminating du-

plicates on the performance using our strategy. This

strategy aims to reduce the amount of data that poten-
tial jobs can handle by removing duplicates as early

as possible. To do so, we recorded the execution time

with and without the proposed duplicate elimination

strategy, i.e., eliminating duplicates after the com-

pletion of the reasoning process. As shown in Figure 6,

the execution time is dramatically decreased when our

elimination strategy is applied.
5 Conclusion and Future Work

We proposed a novel approach for performing distributed reasoning using RDFS and OWL Horst rules. Surprisingly, after reviewing the literature, we found no tool that can reason over large-scale OWL datasets. The proposed approach is implemented as an open-source distributed system for reasoning large-scale OWL datasets using Spark. Compared to Hadoop MapReduce, Spark enables efficient distributed processing by supporting running multiple jobs at the same node simultaneously as well as the ability to cache data required for computations in the memory. The use of data storage in memory greatly decreases the average time spent on network communication (i.e., communication overhead) and data read/write using disk-based approaches. We exploit these advantages for supporting Semantic Web reasoning. Furthermore, the proposed approach combines the contributions introduced by state-of-the-art (i.e., the optimized execution strategy and the pre-shuffling method). Besides, we proposed a novel duplicate elimination strategy that drastically reduces the reasoning time. These tasks are considered the most time-consuming tasks in the reasoning process. The experiments proved that the proposed approach is scalable in terms of both data and node scalability. Our approach has successfully inferred around six million axioms in 11 hours using only five nodes. In conclusion, our approach achieved near-linear scalability of output in the sense of speedup. We have successfully integrated the proposed approach into the SANSA framework, which ensures its sustainability and usability.

To further our research, we plan to perform several improvements, including code optimization, such as using different persisting strategies, and build an optimal execution strategy based on the statistics of the input OWL dataset using OWLStats [Mohamed et al., 2020] approach from the SANSA framework. Moreover, we aim to design more reasoning profiles, such as OWL EL and OWL RL.

Acknowledgements

This work has been supported by the following EU Horizon2020 projects: LAMBDA project (GA no. 809965), CLEOPATRA project (GA no. 812997), and PLATOON project (GA no. 872592).

REFERENCES


