SWRL Parallel Reasoning Implementation with Spark SQL

Wan Li, Huaai Kang, Dongbo Ma and Weiwei Wei

Software and Information School, Beijing Information Technology College, Beijing, China.
Email: lee_wan@sina.com

Abstract. With the rapid development of semantic Web and big data technology, ontology data has the characteristics of large-scale, high-speed growth and diversity which big data has. On one hand, the conventional ontology reasoners do not scale well for large amounts of ontologies because they are designed for run on a single machine. On the other hand, the existing scalable reasoners are not perfect enough, for example, to completely support the widely used Semantic Web Rule Language (SWRL) rules. This paper presents an implementation for SWRL scalable parallel reasoning using the Spark SQL programming model, and optimizes and processes some of the problems in the implementation.

1. Introduction

In the field of Semantic Web[1][2], based on RDF [3], RDFS [4] and OWL [5], W3C has introduced SWRL [6] with more logical expression ability combining description logic and rule. With the rapid development of the Semantic Web and big data technologies, ontology data has presented big data characteristics such as large-scale, high-speed growth, and diversity. To this end, researchers have introduced distributed computing technology into the field of Semantic Web research to explore high-efficiency ontology reasoning methods in distributed environments [7][8][9][10].

In our previous work [11], based on the analysis of existing semantic reasoning algorithms, combined with Spark SQL[12] which is a newer and higher level parallel computing platform with structured data processing capabilities, we proposed a SWRL parallel reasoning method based on Spark SQL. This paper is about the specific implementation of the method. We implemented SWRL parallel reasoning based on Spark SQL and optimized the parallel reasoning algorithm to improve performance.

2. Preliminary

2.1. SWRL Semantic Reasoning

A RDF triple, which is a triple of resource <s, p, o>, asserts that the relationship denoted by the predicate is held between the subject and object of the triple. RDF defines a simple graph model to denote relationships between resources using the format of RDF triple.

Based on RDF, both RDFS and OWL define a set of rules respectively with the ability to represent implicit information. SWRL not only includes the RDFS/OWL Horst semantics, but also can be used to express application-specific semantics. Reasoning is the process of deducing implicit information from existing RDF data by using the RDFS or OWL or SWRL reasoning rules. Given an RDF graph G, some new triples denoted as T can be derived from the RDFS or OWL or SWRL rules. Add T to G then we can obtain a bigger RDF graph G’. The process from G to G’ is called reasoning [13]. Reasoning is an iterative process that does not end until new results are not derived.

A SWRL rule r has the form (1), where $B = B_0 \wedge \cdots \wedge B_m$ is the body of the rule, and $H = H_0 \wedge \cdots \wedge H_n$ is the head of the rule. Both B_i and H_j are RDF atoms, which are triples of resources or variables. B_i is a body atom, and H_j is a head atom. To make a rule safe, SWRL constrain that a variable in the head must occur at less one time in the body.

$$B_0 \wedge \cdots \wedge B_m \Rightarrow H_0 \wedge \cdots \wedge H_n \qquad (1)$$

A variable substitution [10][14] $\varphi_t^{?x}$ is an operation that apply to an atom to replace the occurrences of the variable $?x$ with a resource t. A substitution application (SA) $A[\varphi]$ is a triple obtained by applying a substitution φ to an atom A. Composition of substitutions φ_1 and φ_2 is also a substitution and defined as usual in [15]. Given a fact base I and a rule r, the inference result $r(I)$ is the smallest set containing $H_j(r)[\varphi]$ (for each $0 \leq j \leq n$) for each substitution φ such that $B_i(r)[\varphi] \in I$ (for each $0 \leq i \leq m$).

2.2. Our Previous Work: SWRL Parallel Reasoning Method with Spark SQL

We divide the SWRL rule execution procedure into three stages [11]: (1) finding SA set for each body atom (FindSA), (2) joining SA sets of all the body atoms of the rule to find out the variable substitutions of the rule head (JoinSA) (3) applying the variable substitutions of the rule head to each atom of the rule head (ApplySH). SQL queries are executed by Spark SQL to implement parallelized rule execution.

Take the following rule r1 given in [10] as an example. The reasoning process is shown in figure 1.

$$r_1\text{: } B_0\text{: } (?x, \mathit{worksFor}, ?y_1) \wedge B_1\text{: } (?y_1, \mathit{subOrganizationOf}, ?y_2) \Rightarrow H_0\text{: } (?x, \mathit{worksFor}, ?y_2)$$

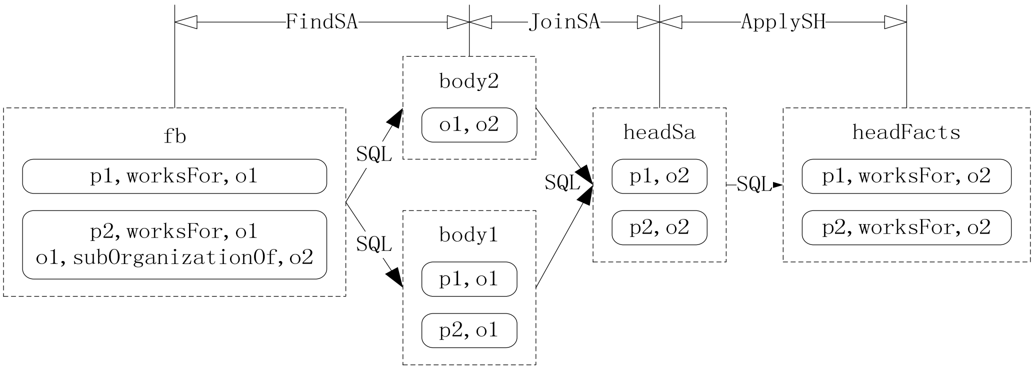

Figure 1. The execution workflow of rule r1

The SQL statements of the three stages are as follows:
FindSA:

SELECT t0 AS ?x, t2 AS ?y1 FROM fb WHERE t1='worksFor'

SELECT t0 AS ?y1, t2 AS ?y2 FROM fb WHERE t1='subOrganizationOf'

JoinSA:

SELECT body0.?x, body1.?y2 FROM body0, body1 WHERE body0.?y1=body1.?y1

ApplySH:

SELECT DISTINCT ?x AS t0, 'worksFor' AS t1, ?y2 AS t2 FROM headSa

3. SWRL Parallel Reasoning Implementation with Spark SQL

3.1. Reasoning Implementation for a Rule

As mentioned above, the reasoning of a rule is to use Spark SQL to execute the SQL statements of the three stages of the rule's reasoning plan (FindSA, JoinSA and ApplySH). The SQL statements for the three stages are already generated in the generateing rule planning step.

Algorithm 1 shows the process of reasoning a rule by sequentially executing these SQL statements of the three stages. Firstly call the Spark DataFrame's *createOrReplaceTempView* function to register the input triples library *tb* as a SQL temporary view. Then call the *findSa* function to find the substitution application (SA) set of each rule body atom. This is done by executing the previously generated SQL statement *findBodyAtomSaSql* to query in the view of *tb*. The SA data sets of each rule body atom are each registered as a SQL temporary view. Then call the *joinSa* function to join the SA sets of all the rule body atoms to find the variable substitution of the rule header. This is done by executing the previously generated SQL statement *joinSaSql* to query in the view of the SA data set of each rule body atom obtained in the FindSA phase. The variable substitution data set of the rule header is registered as a SQL temporary view. Then call the *applySh* function to apply variable substitution to each rule header atom. This is done by executing the previously generated SQL statement *applyHeadAtomSaSql* to query in the view of the variable substitution data set of the rule header obtained in the JoinSA phase. Union the triple sets obtained by each rule header atom to get the rule's reasoning result triple set.

Algorithm 1. Reasoning a rule with Spark SQL

```
Input: rulePlan (reasoning plan of a rule), tb (current triples)
Output: (derived triples by the rule)
def reasoningRule(rulePlan: RulePlan, tb: DataFrame): DataFrame = {
  tb.createOrReplaceTempView(TB_VIEW_NAME)
  findSa(rulePlan.findSaPlan)
  joinSa(rulePlan.joinSaPlan)
  return applySh(rulePlan.applyShPlan)
}
def findSa(findSaPlan: List[FindBodyAtomSaPlan]) = {
  for (findBodyAtomSaPlan <- findSaPlan)
    findBodyAtomSa(findBodyAtomSaPlan)
}
def findBodyAtomSa(findBodyAtomSaPlan: FindBodyAtomSaPlan) = {
  val bodyAtomSa = spark.sql(findBodyAtomSaPlan.findBodyAtomSaSql)
  bodyAtomSa.createOrReplaceTempView(findBodyAtomSaPlan.bodyAtomName)
}
def joinSa(joinSaPlan: JoinSaPlan) = {
  val headSa = spark.sql(joinSaPlan.joinSaSql)
  headSa.createOrReplaceTempView(HEADSA_VIEW_NAME)
}
def applySh(applyShPlan: List[ApplyHeadAtomSaPlan]): DataFrame = {
  var headTriples = createEmptyTripleDataFrame()
  for (applyHeadAtomSaPlan <- applyShPlan) {
    val headAtomTriples = applyHeadAtomSa(applyHeadAtomSaPlan)
    headTriples = headTriples.union(headAtomTriples)
  }
  return headTriples
}
def applyHeadAtomSa(applyHeadAtomSaPlan: ApplyHeadAtomSaPlan): DataFrame = {
  return spark.sql(applyHeadAtomSaPlan.applyHeadAtomSaSql)
}
```

3.2. Reasoning Implementation for Rule Base

The reasoning process of a rule base is divided into two steps: the first step is to parse each rule to generate the reasoning plan in the form of SQL statements with Spark SQL and save it; the second step is to execute the reasoning plan SQL statements with Spark SQL for reasoning. The first step and the second step are separable. The performance of the reasonor is mainly determined by the second step. We did some work to improve the performance of the second step. Here, we explain the implementation of the second step.

To put it simply, the second step of the reasoning process is to execute the rule's reasoning plan one by one, add the derived new triple facts into the fact base, and loop the reasoning until no new facts are derived, then the reasoning ends. However, in practice, the implementation of the reasoning process is more complicated due to the DAG mechanism of the Spark computing framework.

The reasoning of the rule base requires iterative calculations. Therefore, Spark's calculation process DAG will be particularly long. The entire DAG calculation needs to be completed before the result is obtained. During this period, datasets with a large amount of lineage will be generated, which will result out of memory. Therefore, it is necessary to truncate the lineage. It is a possible way to truncate the lineage by Spark's checkpoint mechanism. However, checkpoint stores data in the disk file system, writes many files to the file system every short time, and does not automatically delete the previous checkpoint file, which is detrimental to the file system performance of the entire cluster. Therefore, we use the method of writing and then reading the file in our own program to truncate the lineage. In terms of time efficiency, there is not much difference with Spark's checkpoint mechanism, but it does not generate a lot of files, which improves space efficiency.

If the new triples that are reasoned by a rule are added to the triple library *tb*, and *tb* is applied to the next rule to reason more new triples, then the iterative reasoning process will cause *tb* to generate a large amount of lineage. Since the triple library *tb* may be large, frequently truncating the lineage of *tb* by writing and then reading the file will affect system performance. To resolve this problem, we have improved the algorithm to avoid the large amount of lineage of *tb* by using intermediate results.

Algorithm 2. Reasoning rule base with Spark SQL

```
Input: rulesPlan (reasoning plans of rules), tb0 (original triples)
Output: (derived triples)
def reasoningRules(rulesPlan: List[RulePlan], tb0: DataFrame): DataFrame = {
  var tb = tb0
  var derived = createEmptyTripleDataFrame()
  var tbLoop0 = tb0.union(derived)
  var loopDerived = createEmptyTripleDataFrame()
  var hasDerived = true
  do {
    hasDerived = false
    for (rulePlan <- rulesPlan) {
      val ruleDerived = reasoningRule(rulePlan, tb)
      loopDerived = loopDerived.union(ruleDerived).except(tbLoop0)
      loopDerived = loopDerivedTruncateLineage(loopDerived)
      tb = tbLoop0.union(loopDerived)
    }
    if (!loopDerived.isEmpty()) {
      derived = derived.union(loopDerived).except(tb0)
      derived = derivedTruncateLineage(derived)
      tbLoop0 = tb0.union(derived)
      loopDerived = createEmptyTripleDataFrame()
      hasDerived = true
    }
  } while (hasDerived)
  return derived
}
```

Algorithm 2 shows the iterative reasoning process of the rule base. Where *tb0* is the original triples library before starting the inference, and *derived* is all the new triples that have been reasoned before starting a loop of reasoning on the rule base, *tbLoop0* is the triples library before starting a loop of reasoning on the rule base, *loopDerived* is the new triples reasoned in current loop, *ruleDerived* is the new triples reasoned in current rule. Using *tb0* union *derived* to update *tbLoop0*, using *tbLoop0* union *loopDerived* to update *tb*, using *loopDerived* union *ruleDerived* to update *loopDerived* itself, using *derived* union *loopDerived* to update *derived* itself. To truncate lineage on *loopDerived* and *derived* that avoid the large amount of lineage of *tb*, and the *loopDerived* that is frequently truncated is small, while the larger *derived* (which is a subset of *tb*, smaller than *tb*) is truncated less frequently (once per loop).

3.3. Eliminating Duplicated Facts

It is inevitable that some results of the rules duplicate the facts in the knowledge base during the rule execution process. The duplicated facts not only degrade system performance but also increase management overhead. However, removing duplication also bring additional computation to the reasoner. Existing systems adopt a series of duplication elimination strategies [8][9][10]. We used several strategies to remove all duplicated facts, as follows: (1) removing all duplications before the reasoning process begins; (2) using the DISTINCT clause to remove duplications in the derived data in the ApplySH stage of each rule; (3) after each rule is reasoned, use the *except* method to eliminate the duplications in *loopDerived* and the duplications between *loopDerived* and *tbLoop0*; (4) after each time looping through all the rules, use the *except* method to eliminate the duplications in *derived* and the duplications between *derived* and *tb0*.

3.4. SWRL Reasoning Implementation with RDFS/OWL/SWRL Reasoning Rules

SWRL reasoning is the process of deducing implicit information from existing RDF data by using the RDFS/OWL/SWRL reasoning rules. A rule dependency exists when the evaluation results of a rule trigger another rule execution. Thus improper rule evaluation order will conduct unnecessary job running and bring significant performance degradation [9][10]. We optimize the execution order of RDFS and OWL rules according to the method in [9]. Optimizing the execution order of SWRL rules is what we will do in the future. The SWRL reasoning implementation with the RDFS/OWL/SWRL reasoning rules is shown in Algorithm 3.

Algorithm 3. Reasoning RDFS/OWL/SWRL rule base with Spark SQL

```
Input: rdfsRulesPlan (reasoning plans of RDFS rules), owlRulesPlan (reasoning plans of OWL
        rules), swrlRulesPlan (reasoning plans of SWRL rules), tb0 (original triples)
Output: (derived triples)
def reasoningMixedRules(rdfsRulesPlan: Dataset[RulePlan], owlRulesPlan: Dataset[RulePlan],
        swrlRulesPlan: Dataset[RulePlan], tb0: DataFrame): DataFrame = {
  val rulesPlan = rdfsRulesPlan.union(owlRulesPlan).union(swrlRulesPlan).collect.toList
  return reasoningRules(rulesPlan, tb0)
}
```

4. Conclusion and Future Work

In our previous work, based on the analysis of existing semantic reasoning algorithms, combined with Spark SQL which is a newer and higher level parallel computing platform with structured data processing capabilities, we proposed a SWRL parallel reasoning method based on Spark SQL. This paper is about the specific implementation of the method. We implemented SWRL parallel reasoning based on Spark SQL and optimized the parallel reasoning algorithm to improve performance. In the future work, we intend to further improve the reasoning system and use large data sets to evaluate our system.

5. Acknowledgment

This work was supported in part by the Beijing Municipal Education Commission Science and Technology Plan Project (KM201710857001).

6. References

[1] Tim Berners-Lee. Semantic Web Road Map. September 1998. http://www.w3.org/DesignIssues/Semantic.html.
[2] Tim Berners-Lee, James Hendler, and Ora Lassila. The Semantic Web. Scientific American, 2001, 284(5):35-43.
[3] W3C. Resource Description Framework (RDF): Concepts and Abstract Syntax. http://www.w3.org/TR/rdf-concepts/.
[4] W3C. RDF Schema. http://www.w3.org/TR/rdf-schema/.
[5] W3C. OWL 2 Web Ontology Language: Profiles. https://www.w3.org/TR/owl2-profiles/.
[6] W3C. SWRL: A Semantic Web Rule Language Combining OWL and RuleML. http://www.w3.org/Submission/SWRL/.
[7] Li Ren. Research on Key Technologies of Large-Scaled Semantic Web Ontologies Querying and Reasoning based on Hadoop. Chongqing University PhD thesis. 2013.
[8] Urbani, J., Kotoulas, S., Maassen, J., et al.:WebPIE: a web-scale parallel inference engine using MapReduce. J. Web Semant. 17(44), 59-75 (2012).
[9] Gu, R., Wang, S., Wang, F., et al.: Cichlid: efficient large scale RDFS/OWL reasoning with Spark. In: IPDPS, pp. 700-709 (2015).
[10] Haijiang Wu, Jie Liu, Tao Wang, Dan Ye, Jun Wei, Hua Zhong. Parallel Materialization of Datalog Programs with Spark for Scalable Reasoning. 17th International Conference on Web Information System Engineering (WISE 2016), pp. 363-379.
[11] Wan Li, Huaai Kang, Dongbo Ma and Weiwei Wei. SWRL Parallel Reasoning Method with Spark SQL. 18th IEEE/ACIS International Conference on Computer and Information Science (ICIS 2019), pp. 270-273.
[12] Spark SQL, DataFrames and Datasets Guide. http://spark.apache.org/docs/latest/sql-programming-guide.html
[13] J. Lu, L. Ma, L. Zhang, J.-S. Brunner, C. Wang, Y. Pan, and Y. Yu, “Sor: a practical system for ontology storage, reasoning and search,” in Proceedings of the 33rd international conference on Very large data bases. VLDB Endowment, 2007, pp. 1402–1405.
[14] Abiteboul, S., Hull, R., Vianu, V.: Foundations of Databases. Addison Wesley, Boston (1995).
[15] Ullman, J.D.: Principles of Database and Knowledge-base Systems, vol. I. Computer Science Press, New York (1988).