基于 Spark 的分布式并行推理算法①

叶怡新, 汪璟玢

(福州大学 数学与计算机科学学院, 福州 350108)

摘 要: 现有的 RDF 数据分布式并行推理算法大多需要启动多个 MapReduce 任务, 有些算法对于含有多个实例三元组前件的 OWL 规则的推理效率低下, 使其整体的推理效率不高. 针对这些问题, 文中提出结合 TREAT 的基于 Spark 的分布式并行推理算法(DPRS). 该算法首先结合 RDF 数据本体, 构建模式三元组对应的 alpha 寄存器和规则标记模型; 在 OWL 推理阶段, 结合 MapReduce 实现 TREAT 算法中的 alpha 阶段; 然后对推理结果进行去重处理, 完成一次 OWL 全部规则推理. 实验表明 DPRS 算法能够高效正确地实现大规模数据的并行推理.

关键词: RDF; OWL; 分布式推理; TREAT; Spark

Distributed Parallel Reasoning Algorithm Based on Spark

YE Yi-Xin, WANG Jing-Bin

(College of Mathematics and Computer Science, Fuzhou University, Fuzhou 350108, China)

Abstract: Multiple MapReduce tasks are needed for most of current distributed parallel reasoning algorithm for RDF data; moreover, the reasoning of instances of triple antecedents under OWL rules can't be performed expeditiously by some of these algorithms during the processing of massive RDF data, and so the overall efficiency can't be fulfilled in reasoning process. In order to solve the problems mentioned above, a method named distributed parallel reasoning algorithm based on Spark with TREAT for RDF data is proposed to perform reasoning on distributed systems. First step, alpha registers of schema triples and models for rule markup with the ontology of RDF data are built; then alpha stage of TREAT algorithm is implemented with MapReduce at the phase of OWL reasoning; at last, reasoning results are dereplicated and a whole reasoning procedure within all the OWL rules is executed. Experimental results show that through this algorithm, the results of parallel reasoning for large-scale data can be achieved efficiently and correctly.

Key words: RDF; OWL; distributed reasoning; TREAT; Spark

语义万维网中的 RDF 和 OWL 标准已在各个领域有着广泛的应用, 如一般知识(DBpedia[1])、医疗生命科学(LODD[2])、生物信息学(UniProt[3])、地理信息系统(Linkedgeodata)和语义搜索引擎(Watson)等. 随着语义万维网的应用, 产生了海量的语义信息. 由于数据的复杂性和大规模性, 如何通过语义信息并行推理高效地发现其中隐藏的信息是一个亟待解决的问题. 由于语义网数据的急速增长, 集中式环境的内存限制, 已不适用于大规模数据的推理.

研究 RDFS/OWL 分布式并行推理是目前较新的一个领域. J.Urbani[4-6]等人在 RDFS/OWL 规则集上采用 WebPIE 进行推理, 能够满足大数据的并行推理; 但该算法针对每一条规则启用一个或者多个 MapReduce 任务进行推理, 由于 Job 的启动相对耗时, 因此随着 RDFS/OWL 推理规则的增加, 整体推理的效率受到了限制. 顾荣[7]等人提出了基于 MapReduce 的高效可扩展的语义推理引擎(YARM), 使推理在一次 MapReduce 任务内即可完成 RDFS 规则的推理; 但该算法并不适用于复杂的 OWL 规则的推理. 此外, 当某一规则产生的新三元组重复时, YARM 会存在过多的冗余计算且产生无用数据. 汪璟玢[8]等人提出结合 Rete 的 RDF 数据分布式并行推理算法, 该算法结合

① 基金项目:国家自然科学基金(61300104)

收稿时间:2016-09-21;收到修改稿时间:2016-10-31 [doi:10.15888/j.cnki.csa.005790]

RDF 数据本体, 构建模式三元组列表和规则标记模型; 在 RDFS/OWL 推理阶段, 结合 MapReduce 实现 Rete 算法中的 alpha 阶段和 beta 阶段, 从而实现 Rete 算法的分布式推理; 但该算法在连接 beta 网络推理时需要消耗较多的内存且进行多次迭代时效率低下, 因而此算法受到集群内存和平台的限制. 顾荣[9]等人提出了一种基于 Spark 的高效并行推理引擎(Cichlid), 结合 RDD 的编程模型, 优化了并行推理算法; 但该算法未考虑规则能否被激活, 均需要进行推理, 因而造成了推理性能的浪费和传输的冗余.

为了解决上述问题, 本文针对 OWL Horst 规则, 提出了 DPRS 算法(Distributed parallel reasoning algorithm based on Spark). 该算法结合 TREAT[10]算法和 RDF 数据本体构建模式三元组的 alpha 寄存器 RDD, 预先对规则能否被激活做出判断并标记, 仅对可激活的规则进行推理的处理, 实现在一个 MapReduce 任务中完成 OWL 全部规则的一次推理. 最后, 实时地删除重复的三元组数据和更新冲突集数据到相应的寄存器中, 以进一步提高后续迭代推理的效率. 实验表明, 该算法在数据量动态增加的情况下能够高效地构建 alpha 网络, 并执行正确的推理.

1 基本定义

定义 1. 模式三元组(SchemaTriple), 指三元组的主语谓语和宾语都在本体文件(OntologyFile)中有定义. 即:

$$\forall(S_i, P_j, O_k)(1 \le i,j,k \le n)$$

其中, n 表示模式三元组的总数. 若 $v \in \{S_i, P_j, O_k\}$, $v \in$ OntologyFile, 则:

$$\forall(S_i, P_j, O_k) \in \text{SchemaTriple} \quad (1)$$

定义 2. 实例三元组(InstanceTriple), 指主语谓语和宾语至少有一个在本体文件(OntologyFile)中未定义, 是具体的实例. 即:

$$\forall(S_i, P_j, O_k)\ (1 \le i,j,k \le n)$$

其中, n 表示实例三元组的总数. 若 $v \in \{S_i, P_j, O_k\}$, $\exists v \notin$ OntologyFile, 则:

$$(S_i, P_j, O_k) \in \text{InstanceTriple} \quad (2)$$

定义 3. 三元组类型标记(Flag_TripleType), 用于标识模式三元组与实例三元组, 结合定义 1 和定义 2, 三元组类型标记 Flag_TripleType 定义如下:

$$\forall v \in \{(S_i, P_j, O_k) | 1 \le i,j,k \le n\}$$

其中, n 表示三元组的总数. 则:

$$Flag\_TrileType = \begin{Bmatrix} 1(v \in SchemaTriple) \\ 0(v \notin SchemaTriple) \end{Bmatrix} \quad (3)$$

定义 4. 模式三元组列表(SchemaRDD). 用于获取相同谓语或者宾语的模式三元组集合. 结合定义 1, 模式三元组列表 SchemaRDD 定义如下:

$$\forall(S_i, P_j, O_k) \in \text{SchemaTriple}(1 \le i, j, k \le n)$$

其中, n 表示模式三元组的总数. 则,

$$\text{Schema}RDD = O_m\_RDD \cup P_t\_RDD \quad (4)$$

其中, O_m_RDD 表示满足谓语 $P_j \in$ { rdf:type }且具有相同宾语的三元组集合, 以该宾语命名; P_t_RDD 表示满足谓语 $P_j \notin$ { rdf:type }的所有具有相同谓语的三元组集合, 以该谓语命名. 具体定义如下:

$$O_m\_RDD = \left\{ \left(S_i, P_j, O_k\right) \middle| P_j \in \{rdf:type\} \,\&\, k = m \right\} \quad (5)$$

$$P_t\_RDD = \left\{ \left(S_i, P_j, O_k\right) \middle| P_j \notin \{rdf:type\} \,\&\, j = t \right\} \quad (6)$$

定义 5. 连接变量(LinkVar). 连接变量为在 RDFS/OWL 规则中用于连接两个前件的模式三元组项, 根据规则描述, 连接变量可以不止一个. 本文将每一条规则的连接变量信息以<key,value>的形式存储在 $Rule_m$_RDD, 其中 key 存储该规则所有用于前件连接的模式三元组项, value 存储该规则结论部分的模式三元组项.

DPRS 算法根据连接变量的类型, 对 OWL Horst 规则进行分类. 本文引用 OWL Horst 规则时采用 OWL-规则编号的形式, 例如 OWL-4 表示图 1 中的第 4 条规则. 同时, 给每条规则分配一个规则名称标记, 规则名称标记即为该规则所对应的名称(例如, 规则 OWL-4 的规则名称标记为 OWL-4). 具体的规则分类如下:

1) 类型 1: 只包含一个前件的规则或 SchemaTriple 与 InstanceTriple 组合的规则, 且只有一个 InstanceTriple, 可以在 Map 推理过程中直接输出推理结果(图 1 中规则 OWL-3、OWL-5a、OWL-5b、OWL-6、OWL-8a、OWL-8b、OWL-9、OWL-12a、OWL-12b、OWL-12c、OWL-13a、OWL-13b、OWL-13c、OWL-14a、OWL-14b).

2) 类型 2: SchemaTriple 与 InstanceTriple 组合的规则, 且有多个 InstanceTriple 的, 需要结合 Map 和 ReduceByKey 两个阶段推理(图 1 中规则 OWL-1、OWL-2、OWL-4、OWL-7、OWL-15、OWL-16).

定义 6. 设 C_{mn} 为第 m 条规则的第 n 个模式三元组前件, 定义规则前件模式标记 $Index_{mn}$, 用于标识是否有符合该前件的模式三元组存在, 即以该模式三元组前件 C_{mn} 所命名的 SchemaRDD 是否为空. 结合定义 4, 规则前件模式标记 $Index_{mn}$ 定义如下:

$$\text{Index}_{mn}=\begin{cases}1\left(C_{mn}\in SchemaRDD\right)\\0\left(C_{mn}\notin SchemaRDD\right)\end{cases}\tag{7}$$

定义 7. 规则标记 Flag_Rule_m, 用于标记该规则是否为不可能激活的规则. 结合定义 6 进行定义规则标记 $Flag\_Rule_m$ 如下:

$$Flag\_Rule_m = \{0,1,2\}$$

其中, 规则不能激活时, Flag_Rulem=0; 规则激活且为类型 1 时, Flag_Rulem=1; 规则激活且为类型 2 时, Flag_Rulem=2.

由于图 1 中 OWL 规则 5a、5b 不影响推理的并行化, 因而, 本文所述推理不考虑这两条规则.

OWL Horst rule

```
1  p rdf:type owl:FunctionalProperty, u p v, u p w => v owl:sameAs w
2  p rdfs:type owl:InverseFunctionalProperty,v p u,w p u => v owl:sameAs w
3  p rdf:type owl:SymmetricProperty, v p u => u p v
4  p rdf:type owl:TransitiveProperty, u p w, w p v => u p v
5a u p v => u owl:sameAs u
5b u p v => v owl:sameAs v
6  v owl:sameAs w => w owl:sameAs v
7  v owl:sameAs w, w owl:sameAs u => v owl:sameAs u
8a p owl:inverseOf q,v p w => w q v
8b p owl:inverseOf q,v p w => w p v
9  v rdf:type owl:Class, v owl:sameAs q => p rdfs:subPropertyOf q
10 p rdf:type owl:Property, p owl:sameAs q => p rdfs:subPropertyOf q
11 u p v, u owl:sameAs x, v owl:sameAs y => x p y
12a v owl:equivalentClass w => v rdfs:subClassOf w
12b v owl:equivalentClass w => w rdfs:subClassOf v
12c v rdfs:subClassOf w, w rdfs:subClassOf v => v rdfs:equivalentClass w
13a v owl:equivalentProperty w => v rdfs:subPropertyOf w
13b v owl:equivalentProperty w => w rdfs:subPropertyOf v
13c v rdfs:subPropertyOf w, w rdfs:subPropertyOf v => v rdfs:equivalentProperty w
14a v owl:hasValue w, v owl:onProperty p, u p v => u rdf:type v
14b v owl:hasValue w, v owl:onProperty p, u rdf:type v => u p v
15 v owl:someValuesFrom w, v owl:onProperty p, u p x, x rdf:type w => u rdf:type v
16 v owl:allValuesFrom u, v owl:onProperty p, w rdf:type v, w p x => x rdf:type u
```

图 1 OWL Horst 规则

2 DPRS算法

在 Rete 算法中, 同一规则连接结点上的寄存器保留了大量的冗余结果. 实际上, 寄存器中大部分信息已经体现在冲突集的规则实例中. 因此, 如果在部分匹配过程中直接使用冲突集来限制模式之间的变量约束, 不仅可以减少寄存器的数量, 而且能够加快匹配处理效率. 这一思想称为冲突集支撑策略. 基于冲突集支撑策略, TREAT[10]算法放弃了 Rete 算法中利用 β 寄存器保存模式之间变量约束中间结果的思想.

DPRS 算法根据 Spark RDD 的特点, 结合 TREAT 算法的原理, 首先根据 RDF 本体数据构建模式三元组对应的 alpha 寄存器 O_m_RDD 或 P_t_RDD 并广播, 然后对每条规则的模式前件进行连接并生成对应的连接模式三元组集合 $Rule_m$_linkvar_RDD, 从而加快推理过程中的匹配速度, 能够实现多条规则的分布式并行推理. DPRS 算法主要包括以下几个步骤:

Step1. 加载模式三元组集合 P_t_RDD、O_m_RDD 和 $Rule_m$_linkvar_RDD 并广播.

Step2. 构建规则标记模型 $Flag\_Rule_m$ 并广播.

Step3. 并行执行 OWL Horst 规则推理.

Step4. 删除重复三元组.

Step5. 如果产生新的模式三元组数据, 则跳至步骤 Step2, 如果产生新的实例三元组数据, 则跳至步骤 Step3, 否则跳至步骤 Step6.

Step6. 算法结束.

DPRS 算法的总体框架图如图 2 所示.

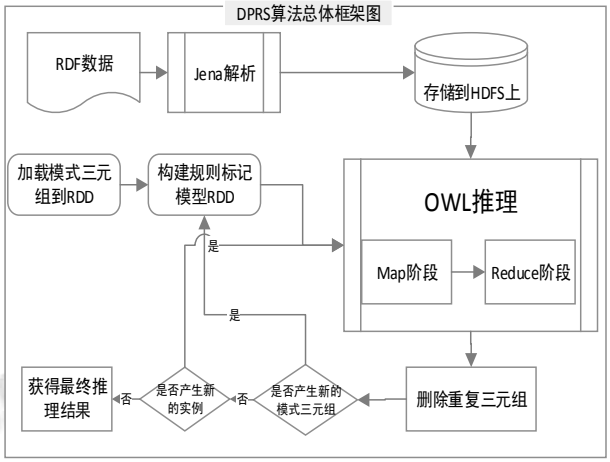

图 2 DPRS 算法总体框架图

2.1 加载模式三元组与构建规则标记模型

由于模式三元组的数量远远少于实例三元组, DPRS 算法将 SchemaTriple 加载到 SchemaRDD 中并广播. 并构建每条规则中的模式三元组或模式三元组连接后的数据 ($Rule_m$_linkvar_RDD 或 O_m_RDD 或 Pt_RDD) 为 alpha 寄存器并广播, 保存对应的 SchemaTriple.

为了尽早判断出不可能被激活的规则, DPRS 算法根据 OWL 规则构建每一条规则内 SchemaTriple 间的关系 O_m_RDD 或 P_t_RDD, 并判断 SchemaRDD 中是否存在规则前件中的 SchemaTriple, 生成对应规则的标

记 $Flag\_Rule_m$, 构建所有规则的标记模型, 将规则标记模型加载到 $Flag\_Rule_m$ 并广播.

通过 SchemaRDD 和构建规则标记模型能够过滤大量 InstanceTriple, 减少 Map 阶段键值对的输出, 从而减少了无效的网络传输, 提高整体推理效率.

2.2 Map 阶段

Map 阶段主要完成数据选择过滤与类型 1 推理, 将过滤的结果以键值对的形式输出, 本文提出的数据分配与过滤算法具体步骤如下:

算法: Map 算法

输入: RDF 三元组格式数据

输出: 以键值对的形式输出 InstanceTriple

Step1. 获取广播变量中的 O_m_RDD、P_t_RDD 和 $Rule_m\_linkvar\_RDD$ 以及规则标记 $Flag\_Rule_m$.

Step2. 对于输入的$\forall(S_i, P_j, O_k)\in$InstanceTriple 判断所有$Flag_Rule_m$的值. 如果值为 0, 则跳至 Step3, 如果值为 1, 则跳至 Step4, 否则跳至 Step5.

Step3. 对(S_i, P_j, O_k)不做任何处理.

Step4. 结合 O_m_RDD 或 P_t_RDD 执行类型 1 的规则推理, 根据规则的结论直接输出对应的三元组 $<Rule_m, \left(S_i^{'}, P_i^{'}, O_k^{'}\right)>$.

Step5 获取对应规则中的模式三元组 alpha 寄存器 O_m_RDD、P_t_RDD 或 $Rule_m\_linkvar\_RDD$, 判断当前的实例三元组是否满足前件连接变量的条件; 满足, 则构建对应的键值对$<Rule_m_linkvar, (S_i, P_j, O_k)>$输出; 不满足, 则不做处理.

以图 1 中规则 8a(inverseOf)为例, 伪码描述如下:

```
Input:   (S1, P1, O1)
Output:  <key, value>
Begin
If (Flag_Rule9 == 2){
  If (P1 equal “rdf:type”){
    yield (Rule9+S1, (“flag=type, resource=”+ S1))
  }
  Else if (P1 equal “owl:sameAs”){
    yield (Rule9+S1,(“flag=sameAs, resource=”+ O1))
  }
}
End
```

类似于规则 8, 推理可以在 Map 阶段就得到规则产生的三元组结果, 那么 reduce 阶段就可以对规则 8 产生的三元组去重并输出.

以图 1 中规则 9(type+sameAs)为例, 伪码描述如下:

```
Input:   (S1, P1, O1)
Output:  <key, value>
Begin
If (Flag_Rule9 == 2){
  If (P1 equal “rdf:type”){
    yield (Rule9+S1, (“flag=type, resource=”+ S1))
  }
  Else if (P1 equal “owl:sameAs”){
    yield (Rule9+S1,(“flag=sameAs, resource=”+ O1))
  }
}
End
```

以图 1 中规则 15(someValuesFrom)为例, 伪码描述如下:

```
Input:   (S1, P1, O1)
Output:  <key, value>
Begin
If (Flag_Rule15 == 2){
For (inverse in bc_vpw.value){
  If (P1 equal “rdf:type”){
    If (O1 equal inverse.W){
        yield (Rule15+inverse.P+O1,
(“flag=type, resource=”+inverse.V))
/*输出的 value 中 flag 标识 type 或者 generic, 用于 reduce 中判断为 S 或 O*/
      }
    }
Else if (P1 equal inverse.P){
yield (Rule15+P1+O1,
(“flag=generic, resource=”+S1))
      }
    }
}
End
```

如上所描述的规则 9 和 15, 以规则 9 为例, 在 Map 阶段需要对输入的三元组进行处理, 以“Rule9+连接变量”为 key, 如果谓语为 type, 那么 value 中标记为 type 且资源为连接变量; 如果谓语为 sameAs, 那么 value 中标记为 sameAs 且资源为宾语.

2.3 Reduce 阶段

Reduce 阶段主要完成连接推理. 利用 RDD 的 reduceByKey, 结合 OWL 规则, 根据 SchemaRDD 和 alpha 寄存器以及 Map 阶段的 InstanceTriple 输出结果完成连接推理, 得到推理结果. 本文提出的连接推理算法具体步骤如下:

算法: Reduce 算法

输入: 键值对形式的 InstanceTriple

输出: 输出连接推理生成的三元组

Step1. 获取广播变量中的 O_m_RDD、P_t_RDD 和 $Rule_m$_linkvar_RDD 以及规则标记 Flag_$Rule_m$.

Step2. 获取相同键对应的迭代器; 如果 key 为 $Rule_m$, 则表示为类型 1, 直接将 value 的三元组 $\left(S_i^{'}, P_i^{'}, O_k^{'}\right)$ 输出; 如果 key 为 $Rule_m$_linkvar, 则表示为类型 2, 则根据该 key 对应的 OWL 规则和连接变量, 结合 alpha 寄存器 $Rule_m$_linkvar_RDD 与 value 迭代器完成连接推理, 得到推理结果并输出连接后的三元组 $\left(S_i^{'}, P_i^{'}, O_k^{'}\right)$. 在执行连接推理过程中, 因为符合条件的 SchemaTriple 已经在构建 alpha 寄存器时已连接完毕, 所以只需要执行 SchemaTriple 与 InstanceTriple 或 InstanceTriple 与 InstanceTriple 间的连接即可.

为了更加明确 Reduce 阶段的连接推理, 以图 1 中规则 9(type+sameAs 规则)为例, 伪码描述如下:

```
Input:  <Rule_m_linkvar, Iterator<values>>
Output:  (S, P, O)
Begin:
    Val iter = values
    while (iter.hasNext){
      Var types, sameAs/*用来保存 flag 为 type 或 sameAs 的集合*/
       value = iter.next
       if (value.flag == “type”)
          types.add(value.resource);
       else sameAs.add(value.resource);
    }
    for (v in types)
         for (w in sameAs)
              yield (null, triple(v,”rdf:type”,w));
End
```

以图 1 中规则 15(someValuesFrom 规则)为例, 伪码描述如下:

```
Input:  <Rule_m_linkvar, Iterator<values>>
    或<Rule_m, Iterator<(S, P, O)>>
Output:  (S, P, O)
Begin:
    Val iter = values
    while (iter.hasNext){
      Var types, generic/*用来保存 flag 为 type 或 generic 的集合*/
       value = iter.next
       if (value.flag == “type”)
          types.add(value.resource);
       else generic.add(value.resource);
    }
    for (v in types)
         for (u in generic)
              yield (null, triple(u,”rdf:type”,v));
End
```

由上述的规则 9 和规则 15 的伪码, 以规则 9 为例, 在 Reduce 阶段, 根据输入的 key 和 values, 我们通过 values 中的 flag 值来进行区分并构建输出的三元组.

2.4 删除重复三元组和冲突集更新策略

在执行算法推理的过程中会产生大量重复的三元组数据到冲突集中, 如不删除冲突集中的重复三元组, 则更新 alpha 寄存器时将会产生重复三元组数据, 浪费系统资源, 降低推理效率. 如果每次推理后都能够及时删除冲突集中的重复三元组, 那将会减少很大的网络传输开销. 本文借助 RDD 的 distinct 和 subtract 完成删除重复三元组算法.

通过上述的删除重复三元组后, 冲突集中的模式三元组分别更新到对应的 alpha 寄存器中, 实例三元组合则并到实例文件中.

2.5 算法的复杂度与完备性

复杂性分析是算法分析的核心, DPRS 算法的复杂性与集中式算法复杂性的分析不太相同, 将 DPRS 算法的最坏情况下的时间复杂性分为 Map 阶段的时间复杂性和 Reduce 阶段的时间复杂性. 假设数据集的规模大小为 N 个三元组, 其中模式三元组为 n 个, 在 MapReduce 中 Map 阶段的并行数为 k, Reduce 阶段传入的实例三元组个数为 m, Reduce 阶段的并行数为 t.

由于DPRS算法在Map阶段对每个输入的三元组, 结合 SchemaList、$Flag\_Rule_m$ 扫描一次, 即可判断该三元组是该舍弃或是能参与某些规则推理, 如能参与后续规则推理, 则以该规则名称为 key 结合此三元组输出. 因此, Map 阶段的时间复杂性为: O(n*N/k).

由于图 1 OWL 规则中, 规则 1、2、3、4、15、16 都含有两个实例三元组前件, 将上述规则称作多实例变量规则, 多实例变量规则的 Reduce 阶段则需要遍历两次输入的实例三元组与模式三元组连接, 才能得到推理结果. 因此在 Reduce 阶段的时间复杂性分为单变量和多变量进行分析.

Reduce 阶段多变量的时间复杂性为: O(n*m/t). 由于 n 的数目非常少, 可以认为其量级为常数.

DPRS 算法首先将数据集中的模式三元组载入内存并广播, 根据定义 7 和 OWL 规则的描述构建各个规则的 $Flag\_Rule_m$, 从而过滤掉不可能激活的规则. 在能被激活的规则并行推理过程中的 Map 阶段, 对于输入的一个三元组, DPRS 判断其是否满足某个规则前件, 只要满足, 就将此规则名称作为键(key), 值(value)为该三元组输出; 若一个三元组数据满足多个规则前件, 我们也将据此方法产生多个不同键(key)的输出, 以保障 Reduce 阶段推理连接的正确性和数据完整性. 如果 Reduce 阶段产生的三元组去重后, 有产生新的模式三元组, 那么 DPRS 算法将重新计算各个规则的 Flag_Rulem, 再执行规则的并行推理迭代; 如果 Reduce 阶段产生的三元组去重后产生的是实例三元组, 那么 DPRS 算法直接执行规则的并行推理迭代, 直到没有新的三元组数据产生为止. 因而 DPRS 算法所得到的推理结果是完备的.

3 实验与结果分析

实验所使用的软件环境为操作系统 Linux Ubuntu, 采用 scala 作为编程语言, 开发环境为 IntelliJIDEA. 在实验环境中, 用表 1 所示配置作为本系统 Spark 集群的配置, 共计 8 台, 其 Hadoop 集群中 1 台作为 HDFS 的名称节点, 1 台作为 JobTracker 节点, 6 台为 HDFS 的数据节点和 TaskTracker 节点, Spark 集群中 1 台作为 Master 兼 Worker 节点, 7 台作为 Worker 节点. 集群工作站的基本配置如表 1 所列.

表 1　Hadoop 集群工作站的基本配置

名称	规格
CPU	Intel(R) Core(TM)　i5-3570M
内存	8G
硬盘	500 GB SATA
Java	JDK1.6
Hadoop	Hadoop 1.0.4
Spark	Spark 1.4.0

本文将 DPRS 算法与 DRRM[4]和 Cichlid-OWL [9] 在相同的实验环境下针对不同的数据集进行对比实验. 本实验采用 LUBM[11] (Lehigh University Benchmark)数据集和 DBpedia[1]数据集进行测试. 数据集的基本参数说明如表 2 所列.

表 2　数据集的基本参数说明

数据集名称	三元组数(个)	属性数(个)	类数(个)	模式三元组数(个)
LUBM50	6,865,225	51	43	176
LUBM100	13,828,451	51	43	176
LUBM200	27,541,125	51	43	176
DBpedia3.7	14,339,055	1662	336	5314
DBpedia3.8	16,968,652	1794	376	5795
DBpedia3.9	21,884,910	2352	570	7766

我们将实验数据集中的模式三元组数进行统计如表 2 所示, 与整个数据集的大小相比, 模式三元组的数量非常少, 在所测试的数据集范围内, 模式三元组数目最高仅仅达到了整个数据集的 0.04%.

表3　不同数据集三种算法在OWL推理产生的三元组数对比

数据集名称	DPRS 推理产生的三元组数	DRRM 推理产生的三元组数	Cichlid-OWL 推理产生的三元组数
LUBM50	1899433	1899433	1899433
LUBM100	3810426	3810426	3810426
LUBM200	7603597	7603597	7603597
DBpedia3.7	74879640	74879640	74879640
DBpedia3.8	95926101	95926101	95926101
DBpedia3.9	146249474	146249474	146249474

表 4　不同数据集三种算法在 OWL 推理时间对比

数据集名称	DPRS 推理时间(s)	DRRM 推理时间(s)	Cichlid-OWL 推理时间(s)
LUBM50	124.941	252.423	130.580
LUBM100	293.608	467.376	295.464
LUBM200	525.120	936.425	530.063
DBpedia3.7	574.283	2393.791	678.843
DBpedia3.8	644.527	2541.000	796.820
DBpedia3.9	776.646	2969.073	982.567

表 5 DPRS 算法在不同数据集上执行推理的数据

数据集名称	模式三元组数 n	实例三元组个数 N	Map 并行数 k	Reduce 并行数 t	传入 Reduce 的实例三元组数 m
LUBM50	176	6,865,225	32	4	683,799
LUBM100	176	13,828,451	32	4	1,295,544
LUBM200	176	27,541,125	32	4	2,661,258
DBpedia3.7	5314	14,339,055	32	4	28,454,263
DBpedia3.8	5795	16,968,652	32	4	34,533,396
DBpedia3.9	7766	21,884,910	32	4	40,187,315

从表 3 和表 4 可知, 在 OWL 规则推理结果一致的情况下, DPRS 比 Cichlid-OWL 具有优势. 其中, 由于 LUBM 数据集本体比较简单, OWL Horst 中的许多规则无法被激活, 所以 DPRS 相比 Cichlid-OWL 的优势比较微弱; 对于比较复杂的 DBpedia 本体而言, OWL 的大部分规则都可被激活, 由于本文使用了 alpha 寄存器广播、连接变量、规则标记和冲突集更新策略, 使得 DPRS 算法的推理时间相对 Cichlid-OWL 算法最大缩短了 21%的时间.

另外, DPRS 与 DRRM 相比均有较大的优势. 首先, DPRS 使用 Spark 平台比 DRRM 使用的 Hadoop 具有迭代性能优势; 再者, DPRS 采用冲突集更新策略, 避免了 beta 网络的开销, 大大减少了传输冗余造成的浪费. 使得 DPRS 算法的推理时间相对 DRRM 算法最大缩短了 73.8%的时间.

根据 2.5 节的复杂度分析, 其中 k 和 t 为常数, 所以推理时间的复杂度与 N 和 m 成线性关系. 结合表 4 和表 5, 考察数据集 LUBM50 和 LUBM200, 实例三元组个数 N 的比例为 1:4.01, 传入 Reduce 的实例三元组数 m 的比例为 1:3.89, 推理时间的比例为 1:4.20; 考察数据集 DBpedia3.7 和 DBpedia3.9, 实例三元组个数 N 的比例为 1 : 1.53, 传入 Reduce 的实例三元组数 m 的比例为 1:1.41, 推理时间的比例为 1:1.35. 可以发现, 我们的推理时间基本是与 N 和 m 成线性关系. 从实验结果上符合了理论的分析, 证明了算法的正确性.

从图 3 和图 4 可知, 在执行 OWL 规则推理时, 虽然两种算法都需要多次迭代才能使得推理最终停止, 但是 DPRS 在推理前构建并广播了模式三元组的 alpha 寄存器, 并且在每次迭代中采用高效的过滤机制, 过滤掉大量的实例三元组数据, 减少了并行计算量和网络传输的开销, 使得 DPRS 算法在最终的推理时间较 Cichlid-OWL 略占优势, 尤其是在 DBpedia 数据集下, 从表 2 中可以看出, Dbpedia 的模式三元组占比相对 LUBM 高, 且数据集较为复杂, 其优势更加明显.

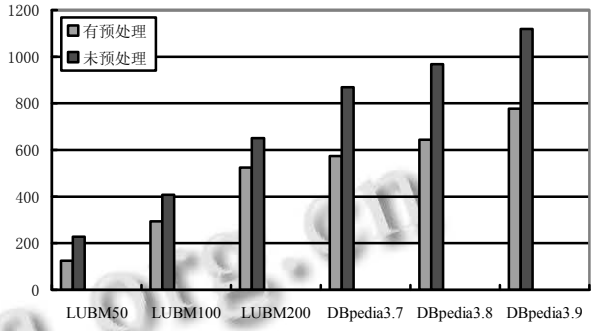

图 3 采用未预处理与预处理算法在 OWL 推理的时间对比

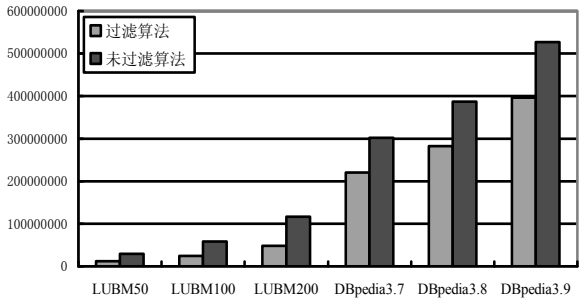

图 4 采用过滤算法与未过滤算法产生的中间结果数目对比

由于在执行推理过程中会产生重复的三元组数据, 重复三元组数据会造成系统资源无谓的浪费并增加网络的开销. 文中 3.4 节提出的删除重复三元组算法, 能够减少重复的三元组数据. 为了评估算法的有效性, 将删除重复三元组前后的数据量进行对比如图 5 所示. 删除重复三元组后的三元组数量少于推理三元组数量, 在所测试的数据范围内.

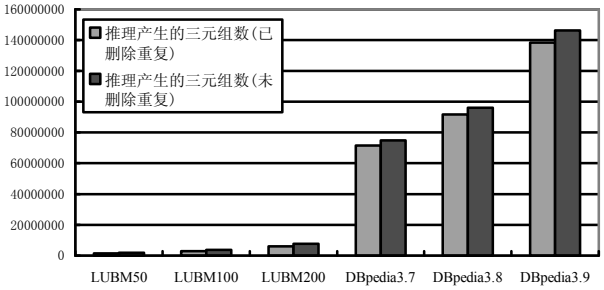

图 5 删除重复三元组前后三元组数量对比

4 结语

本文提出的 DPRS 算法能够通过执行一次 MapReduce 任务就完成 OWL 所有规则的一次推理, 弥补了现有方法大多需要启动多个 MapReduce 任务以及在大规模数据下无法对 OWL 规则中含有实例三元组的规则进行推理的问题. DPRS 算法能够在 MapReduce 计算框架下高效地实现大规模数据的并行推理, 但无法对流式数据进行推理. 下一步将会在此方面进行改进, 且研究更深一步的 OWL DL 推理.

参考文献

1 Auer S, Bizer C, Kobilarov G, et al. Dbpedia: A nucleus for a web of open data. The Semantic Web. Springer Berlin Heidelberg. 2007. 722–735.

2 Jentzsch A, Zhao J, Hassanzadeh O, et al. Linking Open Drug Data. I–SEMANTICS. 2009.

3 Apweiler R, Bairoch A, Wu CH, et al. UniProt: The universal protein knowledgebase. Nucleic Acids Research, 2004, 32(s1): D115–D119.

4 Urbani J, Kotoulas S, Maassen J, et al. WebPIE: A web-scale parallel inference engine using MapReduce. Web Semantics: Science, Services and Agents on the World Wide Web, 2012, (10): 59–75.

5 Urbani J, Kotoulas S, Maassen J, et al. OWL reasoning with WebPIE: Calculating the closure of 100 billion triples. Extended Semantic Web Conference. Springer Berlin Heidelberg. 2010. 213–227.

6 Urbani J. On web-scale reasoning[PhD. dissertation]. Amsterdam, Netherlands: Computer Science Department, Vrije Universiteit, 2013.

7 顾荣,王芳芳,袁春风,等.YARM:基于 MapReduce 的高效可扩展的语义推理引擎.计算机学报,2015,38(1):74–85.

8 汪璟玢,郑翠春.结合 Rete 的 RDF 数据分布式并行推理算法.模式识别与人工智能,2016,(5):5.

9 Gu R, Wang S, Wang F, et al. Cichlid: Efficient large scale RDFS/OWL reasoning with spark. Parallel and Distributed Processing Symposium (IPDPS), 2015 IEEE International. IEEE. 2015. 700–709.

10 Miranker DP. TREAT: A new and efficient match algorithm for AI production system. Morgan Kaufmann, 2014.

11 Guo Y, Pan Z, Heflin J. LUBM: A benchmark for OWL knowledge base systems. Web Semantics: Science, Services and Agents on the World Wide Web, 2005, 3(2): 158–182.