SparkSRE

基于 Spark 的语义推理引擎设计文档

v0.1.0

目录

第1章 推理引擎总体架构与设计

1.1 总体功能与要求

设计实现面向知识图谱及 RDF 数据的语义推理引擎，首先要确定的一个关键点是所能支持的推理语言级别和规则集，因为不同的 OWL 语言级别对应了不同的表达能力和推理计算复杂度。应当根据相关应用的特点选择适当的语言级别和规则，避免一味使用完备的推理逻辑导致生成一些对最终应用价值不大的结果，造成计算和存储资源的浪费。随着大数据时代数据量的迅速增长，推理引擎需要解决的另一个关键点是如何提高推理效率和可扩展性，尤其是在单机推理引擎普遍无法处理海量数据的现状下，更应该思考分布式环境下推理引擎的可扩展性。

1.1.1 功能性需求

本文设计的语义推理引擎主要包括以下几个功能性需求：

1. 支持前向链推理。目前基于 MapReduce 或是基于 Spark 实现的分布式推理引擎中，占主流地位的是针对海量 RDF 数据的前向链推理，并且在进一步提供服务的查询环节，预先的前向链推理使其响应时间明显优于后向链推理。因此本文所设计实现的推理引擎均面向前向链推理。
2. 支持 RDFS 和 OWL Horst 规则集的推理。RDFS 形式简单，是语义 Web 技术中表达能力最弱的一层，但其表达的包含、层次等关系是表达更复杂关系的基础，广泛应用于各种语义 Web 实现中，再加上其良好的可判定性，几乎所有语义推理系统都是支持的。OWL Horst 规则集在可判定性和表达能力之间找到了一个平衡，既能实现 OWL 中一些比较有价值的推理，又具有较合适的计算复杂度，是很多面向 OWL 推理的研究工作的研究重点，因此本文设计的系统予以支持。
3. 支持通用规则的推理。由于 OWL 及其子语言构成的本体描述语言需要在

推理上保证可判定性，表达能力有一定的限制，对于一般通用形式的规则无法进行描述。为了提升推理引擎的通用性，拓宽其适用范围，需要使其能够支持一般通用形式描述的规则推理。支持的规则可以是 DL-Safe 限制下的 SWRL，或者是 Horn 子句等简单形式。

4. 支持推理规则执行可配置。有时为了缩短推理花费的时间，尽快获得推理结果，可能希望推理引擎仅执行规则库中所有规则的一个子集。因此规则库中规则的执行与否，应当是在推理过程执行前可配置的。可配置的规则集使得推理引擎在基于规则的推理上具有更高的通用性。

1.1.2 非功能性需求

在考虑了功能性需求后，本文在系统设计实现过程中还需要满足以下几点非功能性需求：

1. 可扩展性。由于待处理数据量呈爆炸式增长，海量数据远远超过单机所能处理的最大能力，因此一个有效的语义推理引擎应当能够在计算处理能力上实现弹性可扩展。目前各种十分流行的分布式集群计算框架为实现处理能力的可扩展性提供了很好的思路，即通过扩增集群中计算结点的方式来达到可扩展性。
2. 容错性。在分布式环境下，由于结点数众多、网络环境复杂，推理计算过程中遭遇错误在所难免，因此推理引擎的容错和恢复能力就显得非常重要。容错策略和机制应当能保证单个结点上或结点内局部的错误不会导致全局的错误，局部结果通过重计算等方式最终能够合并到全局结果中。

1.2 系统总体架构及分层

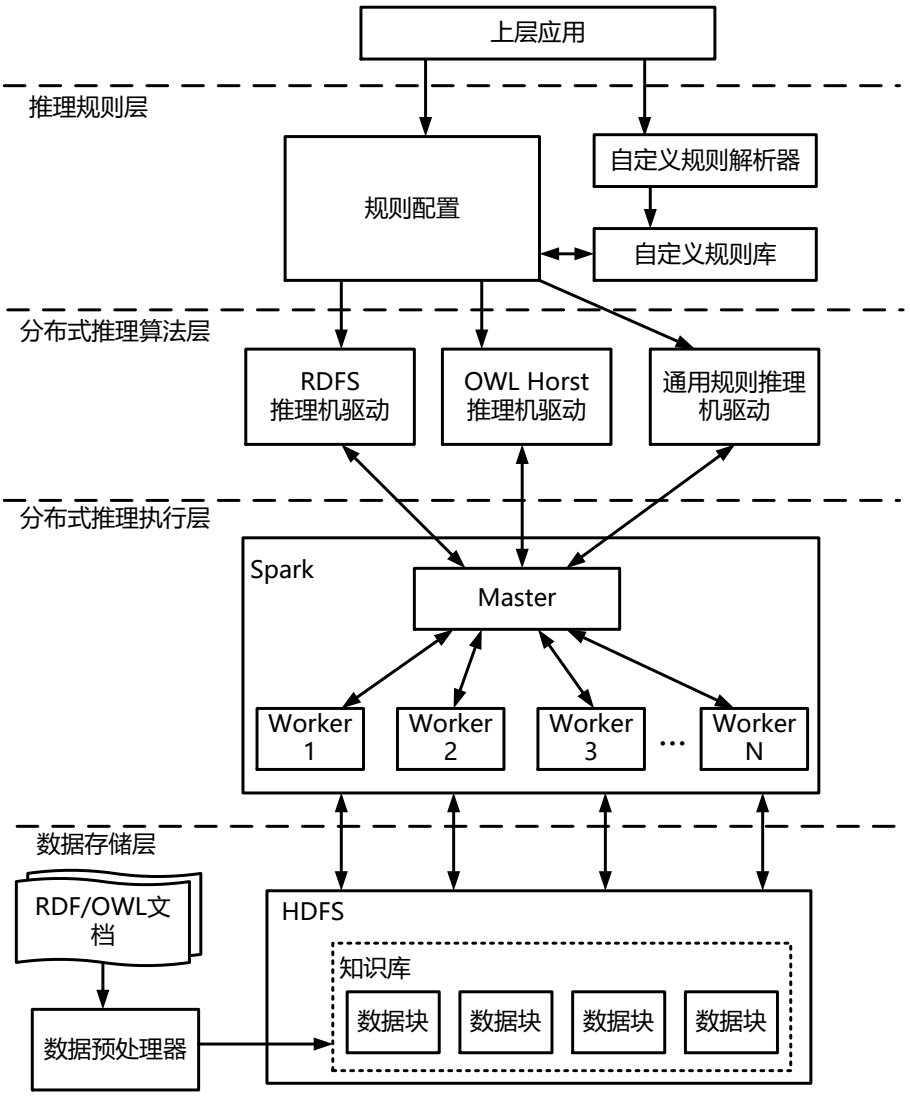

图 1.1 推理引擎系统总体架构

本文设计的推理引擎系统总体架构如图 1.1 所示。从整体上看，系统主要分为 4 层，从上到下依次为推理规则层、分布式推理算法层、分布式推理执行层、数据存储层。

1.2.1 推理规则层

推理规则层衔接了上层应用与分布式推理算法层，为上层应用提供配置可运

行规则、输入自定义规则的接口，为下层推理算法的组合提供依据。推理规则层包括的子模块有规则配置、自定义规则解析器、自定义规则库。

规则配置模块用于对推理引擎中所包含的规则集进行配置，使得每次推理任务执行时可以根据配置按需地进行一个规则子集的推理，不必对所有规则运行推理，满足推理规则执行可配置的功能性需求。自定义规则解析器模块根据预置的规则文法，对从上层应用输入的自定义规则进行解析。首先进行规则的语法检查，对语法正确的规则生成抽象语法树，对语法错误的规则给出错误提示；然后根据抽象语法树进一步生成特定格式的规则，存入自定义规则库中。自定义规则库模块用于保存所有来自上层的经过语法检查正确的自定义规则，根据规则配置动态地生成通用规则推理机驱动。

1.2.2 分布式推理算法层

分布式推理算法层衔接了推理规则层与分布式推理规则执行层，根据推理规则层中的规则配置信息和规则库，生成代表不同规则集的用于提交到 Spark 集群中执行的推理机驱动程序。分布式推理算法层包括的子模块有 RDFS 推理机驱动、OWL Horst 推理机驱动、通用规则推理机驱动。

这些所谓的推理机驱动程序实际上是按照 Spark 提供的转换(Transformation)和动作(Action)编程接口所编写的分布式数据处理程序经编译打包后生成的 jar 包，这些 jar 包随后将被提交到 Spark 集群上等待调度执行。

RDFS 推理机驱动是根据规则配置指定的 RDFS 规则子集生成的按照一定执行顺序运行的 Spark 程序。OWL Horst 推理机驱动是根据规则配置指定的 OWL Horst 规则集生成的按照一定执行顺序运行的 Spark 程序。通用规则推理机驱动是根据规则配置指定的自定义规则库中需要运行推理的规则子集而生成的 Spark 程序。相关的规则集和对应的分布式推理算法在第 2 章中详细描述。

1.2.3 分布式推理执行层

分布式推理执行层是推理算法实际执行的地方，对应着一个 Spark 集群。Spark

集群内采用主从架构，由一个集群管理器和若干个工作结点(Worker)组成。Spark 支持的集群管理器有三种，分别是自带的 Spark 独立集群管理器、Mesos[5]和 YARN[6]，本文考虑到研究重点在于分布式推理的算法，因此采用较为简易直接的 Spark 独立集群管理器，对应运行于 Master 结点。

Master 结点的作用主要是任务调度、资源分配、结果汇总、容错处理等。被提交到 Master 结点上的 Spark 驱动程序首先根据对 RDD 的操作生成一张血系图(Lineage Graph)，这是一张有向无环图(Directed Acyclic Graph，DAG)，记录了 RDD 的来源、去向以及相互之间的依赖关系。血系图是 Spark 高容错性的关键，当 RDD 失效或发生计算错误时，Spark 能够根据这张图的依赖关系重计算出所需的 RDD。然后血系图通过 DAG 调度器划分任务阶段，每个阶段准备就绪时提交给集群管理器中的任务调度器。任务调度器具有先进先出和公平调度两种调度算法，将提交上来的任务调度给 Worker 结点执行。

Worker 结点的作用主要是数据分区存放，实际计算的执行。Worker 结点内部有 Executor 进程，Executor 中的内存缓存是 Spark 分布式内存计算的重要保证，由任务调度器调度的任务在 Executor 内以 Task 线程的方式并发执行。

每个 Spark 驱动程序中包含一个核心的 SparkContext 对象，运行一个 Spark 程序时，SparkContext 需要连接到 Master 结点中的集群管理器，然后方可获取到每个 Worker 结点上的 Executor 对象，接着将驱动程序(jar 包)发送给 Executor，最后将计算任务分发给 Executor 执行。每个 Spark 应用都会获得自己的 Executor 执行进程，这些 Executor 在应用的整个生命周期里都维持运行，并且以多线程的方式运行任务。这样带来的好处就是 Spark 应用相互之间能够相互隔离，在调度端每个驱动程序都调度属于自己的任务，在执行端来自不同应用的任务运行于不同的 Java 虚拟机实例中。然而，这就意味着数据无法实现跨 Spark 应用的共享，即不同的 SparkContext 实例之间无法共享数据，除非通过外部存储系统来暂存数据。那么分布式推理算法层中提及的三种推理机驱动程序间便无法有效共享分布式内存中的数据，因此对于需要共享内存数据的推理规则，需要考虑将这些规则整合到同一个 Spark 驱动程序中去。

1.2.4 数据存储层

数据存储层用于为 Spark 集群提供大数据量高可用的分布式数据持久化存储支持，向 Spark 输入待推理的原始数据，接收 Spark 输出的推理结果数据。数据存储层包含的模块有数据预处理器、HDFS。

数据预处理器的作用是将格式各异的 RDF 文档转化为统一的数据格式(如 N-Triples)，然后通过 HDFS 提供的 API 将 RDF 数据导入 HDFS 中，作为待推理的数据，在 HDFS 中使用一个文件夹作为知识库的抽象概念。为了保证高可用的分布式存储，按照数据容错需求配置 HDFS 将同一份数据冗余 3 份存储于不同的数据块中。Spark 集群通过 HDFS 的 API 对其中的三元组数据进行存取。

1.3 系统全局流程

要充分发挥语义推理引擎系统的作用，除了精心设计高效的分布式推理算法之外，还需要重点考虑推理引擎中各子系统之间的相互衔接、数据流入流出关系，使各子系统能够有机地结合，数据流能够形成有效闭环，更好地为最终的应用服务。整个推理引擎中各子系统间全局的流程如图 1.2 所示。

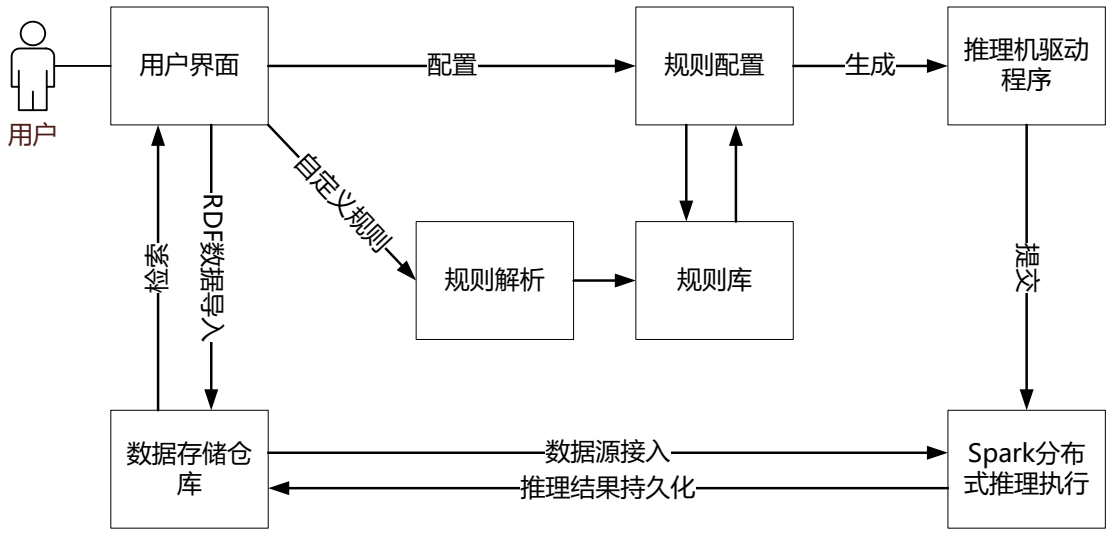

图 1.2 推理引擎系统全局流程

从上图中可以看出，推理引擎系统中各子系统间的数据流关系与系统的总体

架构有一定的对应关系，整体上分为规则配置与解析、分布式推理执行、数据存储等子系统，用户界面不属于本文的讨论范围，下面重点描述规则配置与解析、分布式推理执行的相关流程。

1.3.1 规则配置与解析流程

在实际运行语义推理计算之前，需要明确实施具体推理的规则集范围，而这些正是规则配置与解析所涉及的工作，相关的流程如图 1.3 所示。

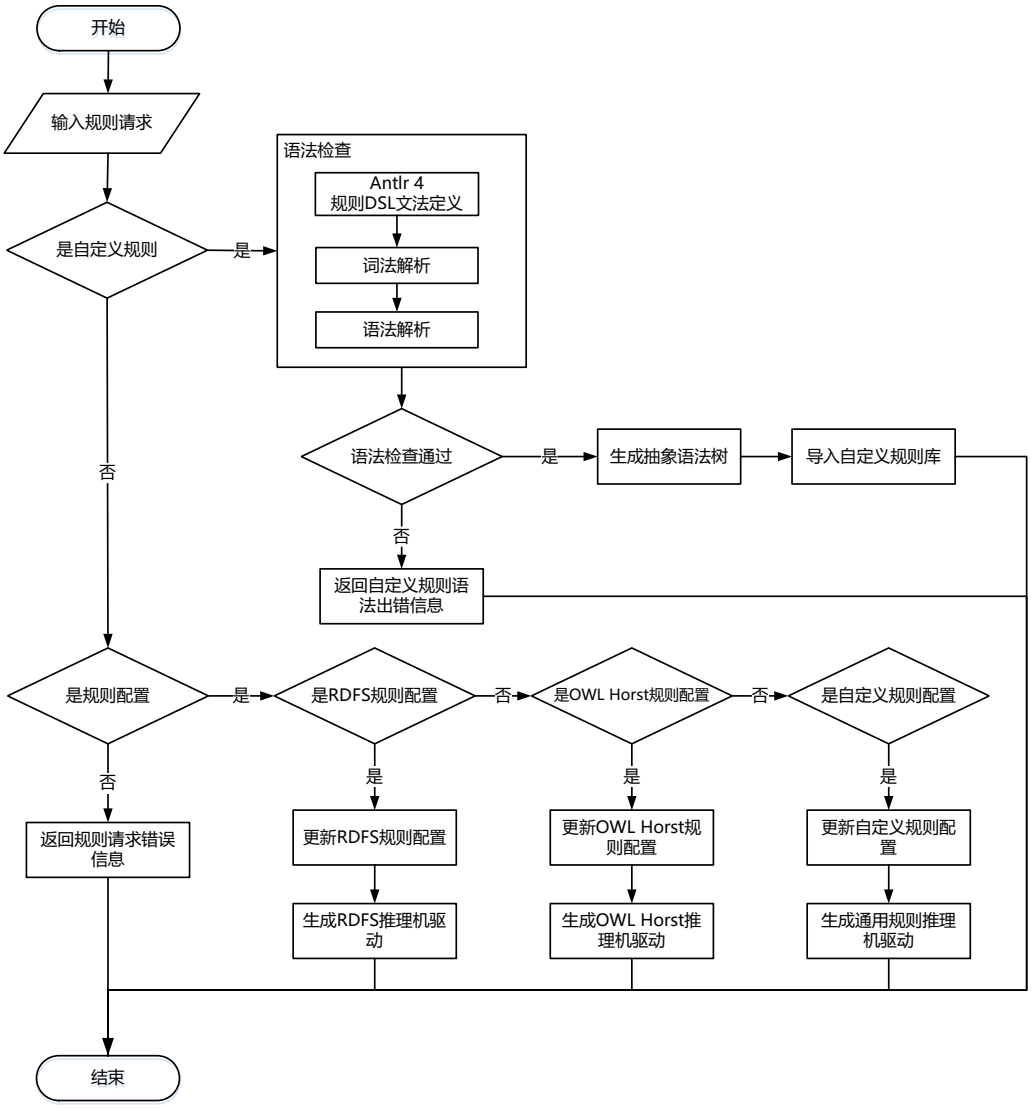

图 1.3 规则配置与解析流程图

首先，上层应用或是用户通过推理规则层提供的接口输入规则请求，判断该

规则请求是自定义规则还是规则配置。若是自定义规则，则进入规则解析流程；若是规则配置，则进入规则配置流程；若都不是，则返回规则请求错误信息，表示该规则请求不符合系统接口规定的输入要求。

在规则解析流程中，核心组件是开源软件 Antlr[7]，它是常用的用于开发语言解释器或编译器的工具框架。首先需要根据 Antlr 的语言文法标准定义规则领域特定语言(Domain Specific Language，DSL)的本法，然后 Antlr 就可以根据该文法生成用于解析规则语言的词法解析器、语法解析器，这些解析器嵌入规则解析模块中作为子模块。语法检查中根据规则 DSL 的文法定义，对输入的自定义规则语句进行词法解析和语法解析。若语法检查不通过，返回规则语句中的出错位置和相关信息；若语法检查通过，则生成该条规则语句的抽象语法树，然后以特定的数据结构导入自定义规则库中。

在规则配置流程中，首先判断该规则请求是否是规则配置，若不是规则配置，则这是一条无效的规则请求，需要返回规则请求出错信息；若这是一条规则配置，则应该根据该规则配置的类型，执行响应的流程，主要是判断这是 RDFS、OWL Horst 还是自定义规则配置。以 RDFS 为例，RDFS 规则集实际上是固定的，RDFS 规则配置的作用是更新指定 RDFS 规则集中应当被执行的子集，然后根据该规则子集生成对应的将会被提交到 Spark 集群上执行的推理机驱动。OWL Horst 规则集以及自定义规则集的相关执行流程也是类似的，区别在于自定义规则集的来源是自定义规则库，其中的规则不是固定的而是可以变动的。

1.3.2 分布式推理执行流程

经过规则配置与解析生成的不同类型的推理机驱动在被提交到 Spark 集群上执行时，也需要按照一定的流程。

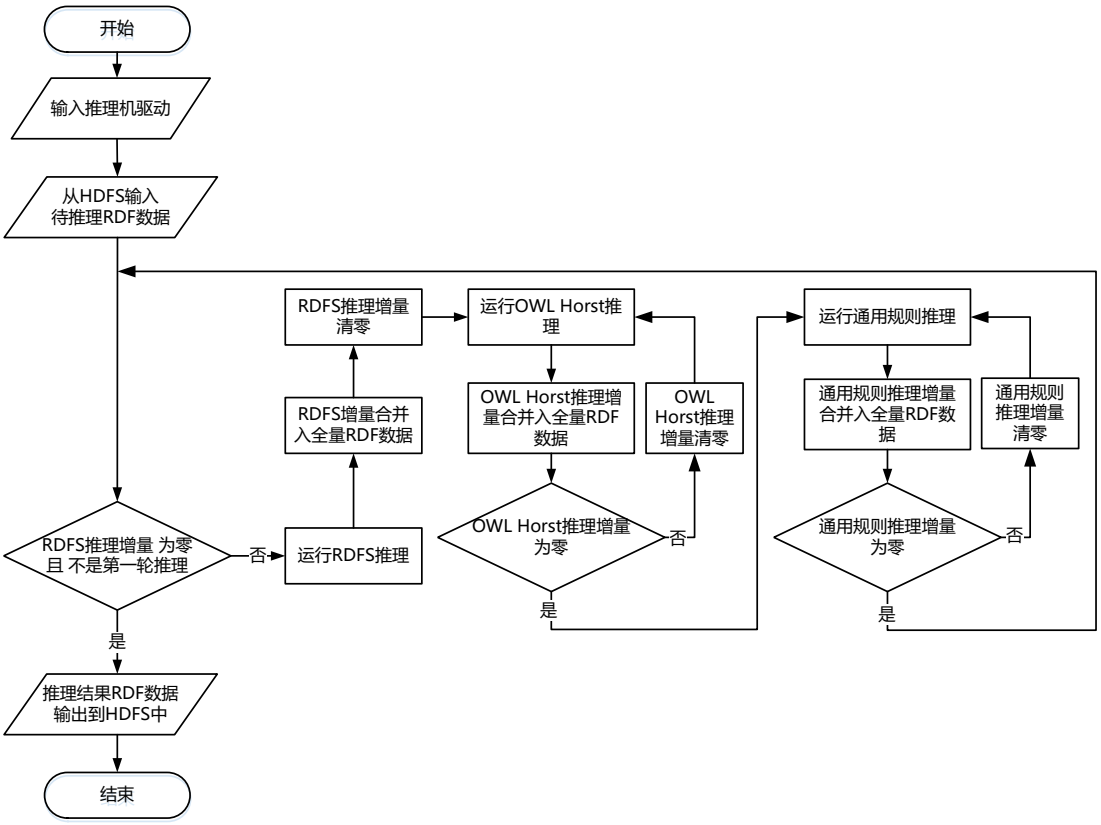

图 1.4 三种推理机驱动在 Spark 中的执行流程

从推理机驱动生成完成开始，到分布式推理执行结束为止，整个执行流程如图 1.4 所示。首先，三种推理机驱动被提交到 Spark 集群等待调度执行，等待推理的 RDF 数据存储于 HDFS 中作为数据输入。接着推理流程进入一组嵌套的循环过程中，即一个外层的 RDFS 推理循环内部嵌套着两个串连的 OWL Horst 推理循环与通用规则推理循环。

在外层的 RDFS 推理循环中，定义一个用于标识 RDFS 推理结果增量的变量，以及一个用于记录推理循环轮次的变量。第一轮推理时增量显然为零，直接进入 RDFS 推理的运行，不是第一轮推理时，则根据 RDFS 推理增量，为零则直接退出循环，不为零则继续循环过程。外层循环中内嵌的两个推理过程其内部逻辑也是类似的，都是先运行推理逻辑，根据当轮次推理结果的增量是否为零来决定是否要退出循环。值得注意的是，每一次运行推理过程后，都有一个将推理结果增量合并入全量 RDF 数据的过程，合并后的结果作为新的全量 RDF 数据输入下一

轮循环的推理过程中。

第2章 基于 Spark 的推理引擎方案实现

2.1 分布式 RDFS 推理

2.1.1 RDFS 规则和依赖顺序

对于语义 Web 领域，RDF 本质上是一个图结构，仅能够表达数据的图结构，而 RDFS 则是如何组织建模 RDF 图的规范，可以使用 RDFS 命名空间下的词汇，表达个体之间的包含等关系。除了建模表达领域对象和数据模型，由于 RDFS 规则蕴含一定语义的同时，可计算性与可判定性也比较良好，因此基于 RDFS 规则的推理广泛应用到各个领域。RDFS 的标准规则集如表 2.1 所示。

表 2.1 RDFS 规则集

编号	前件	后件
1	s p o	_:n rdf:type rdfs:Literal
2	p rdfs:domain x s p o	s rdf:type x
3	p rdfs:range x s p o	o rdf:type x
4a	s p o	s rdf:type rdfs:Resource
4b	s p o	o rdf:type rdfs:Resource
5	p rdfs:subPropertyOf q q rdfs:subPropertyOf r	p rdfs:subPropertyOf r
6	p rdf:type rdf:Property	p rdfs:subPropertyOf p
7	s p o p rdfs:subPropertyOf q	s q o
8	s rdf:type rdfs:subClassOf	s rdfs:subClassOf rdfs:Resource
9	u rdf:type v v rdfs:subClassOf w	u rdf:type w
10	s rdf:type rdfs:Class	s rdfs:subClassOf s
11	u rdfs:subClassOf v v rdfs:subClassOf w	u rdfs:subClassOf w
12	s rdf:type rdfs:ContainerMembershipProperty	s rdfs:subPropertyOf rdfs:member
13	s rdf:type rdfs:Datatype	s rdfs:subClassOf rdfs:Literal

RDFS 推理，本质上是对输入的 RDF 数据不断迭代实施 RDFS 规则，直到没有新的 RDF 三元组生成的一个过程。如果不考虑输入 RDF 数据的分布类型特点，也不考虑推理规则实施的顺序，那么推理过程可能会产生大量的重复计算，因为不同规则的前件和后件之间是存在一定相互依赖关系的。因此分析 RDFS 规则集中各个规则的特点，研究规则之间相互依赖的顺序是实现 RDFS 分布式推理算法的重要前提。

分析表 2.1 中规则的特点可以发现，其中包含一些不是很重要或者说推理价值不大的规则。规则 1、4a、4b、6、8、10 即使排除出去也不失一般性，这些规则前件只含有一个条件，对推理并行性无影响，因此可以在推理过程的任何时候施加一轮推理得到结论，同时，这些规则产生的结论也无法作为相对更重要规则的条件，无法产生更进一步的有用结论。规则 12、13 同样也只有一个条件，考虑到它们前件和后件中的三元组数据在通常的 RDF 数据中不常见，推理价值不大，故可在最后施加一轮推理。因此，本文推理引擎考虑的 RDFS 规则子集为规则 2、3、5、7、9、11、12、13。下面针对这些规则前件和后件中谓词的特点进行一下分类，如表 2.2 和表 2.3 所示。

表 2.2 RDFS 规则子集后件特点归纳

规则后件特点	规则编号
谓词是 rdfs:subPropertyOf 的模式三元组	5、12
谓词是 rdfs:subClassOf 的模式三元组	11、13
谓词是 rdf:type 的实例三元组	2、3、9
其他任意实例三元组	7

表 2.3 RDFS 规则子集前件特点归纳

规则前件特点	规则编号
包含谓词为 rdfs:subPropertyOf 的三元组	5、7
包含谓词为 rdfs:subClassOf 的三元组	9、11
包含谓词为 rdf:type 的三元组	9、12、13
包含其他任意实例三元组	2、3、7

根据上述规则的前件和后件特点，可以得到规则的依赖顺序关系，如图 2.1 所示。该图是一张有向无环图，从图中可以得到规则执行的一个拓扑排序，相对顺序排在前面的规则执行不会依赖排在后面的规则。

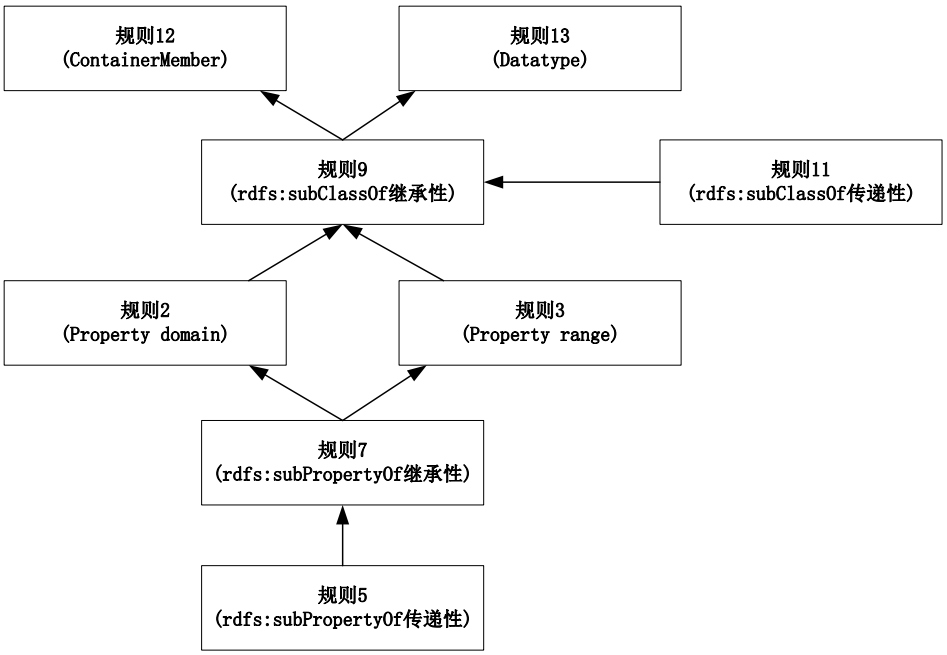

图 2.1 RDFS 规则依赖顺序关系

2.1.2 RDFS 分布式推理算法

在对前件中含有两个及以上条件的规则实施推理时，一般使用 join(连接)操作使得当元组中第一个元素相等时，其他元素能够组合到一起。在面对大规模 RDF 数据时，join 操作除了执行非常频繁以外，还会产生大量的大数据集上的自连接，导致很高的计算开销。另外，设计分布式环境下的推理算法还会面临这样的问题：当 join 操作关联的数据集分布在不同的计算结点上时，大量的数据需要通过网络传输，时间开销巨大。为了减小 join 操作执行时的计算量，需要把整个较大的 RDF 数据集根据规则条件的特点划分成若干相对较小的数据集，这样每次就可以根据条件所需在相对较小的数据集上实施 join 操作。

首先定义几个在算法描述中会用到的方法和符号：

1) union：两个集合取并集

2) join：对元组中第一个元素的连接操作

3) map(func)：根据传入 func 函数定义的行为，将元素映射成别的形式

4) filter(func)：对集合元素进行过滤，只留下使得 func 返回值为 true 的元素

5) 下划线_i：对于 N 元组 t，t._i 表示 t 中的第 i 个元素(N 和 i 都是正整数)

算法 4.1：RDF 三元组数据划分

```
输入：RDF 三元组数据 triples
输出：分类后的各三元组集合
begin
   for each triple(s, p, o) in triples
      if p==rdfs:subClassOf then
         subClass ← subClass.union((s, o))
      else if p==rdfs:subPropertyOf then
         subProp ← subProp.union((s, o))
      else if p==rdfs:domain then
         domain ← domain.union((s, o))
      else if p==rdfs:range then
         range ← range.union((s, o))
      else if p==rdf:type then
         types ← types.union((s, o))
      else
         spo ← spo.union((s, p, o))
      end if
   end for
   return subClass, subProp, domain, range, types, spo
end
```

上述算法便是 RDF 三元组数据划分的过程，划分根据三元组中的谓词来进行，最终三元组被划分为 5 类：subClassOf 模式数据集合 subClass 、subPropertyOf 模式数据集合 subProp、主语模式数据集合 domain、宾语模式数据集合 range、类型关系集合 types、一般实例数据集合 spo。其中，模式数据包括 subClass、subProp、domain、range；实例数据包括 type、spo。

下面根据图 2.1 的规则依赖顺序，分别描述每个具体规则的推理算法。

算法 4.2：RDFS 规则 5 subPropertyOf 传递闭包推理

```
输入：subPropertyOf 模式数据集合 subProp
输出：subPropertyOf 传递闭包计算结果
begin
  reverseSubProp ← subProp.map(t => (t._2, t._1))  //(p2,p1)←(p1,p2)
  while true do
    joined ← reverseSubProp.join(subProp)  //(p2,(p1,p3))←(p2,p1)join(p2,p3)
    res ← joined.map(t => t._2)  //(p1,p3)←(p2,(p1,p3))
    subProp ← subProp.union(res)
    if subProp 中元素数量没有增加 then break
  end while
  return subProp
end
```

算法 4.2 描述的 subPropertyOf 传递闭包推理算法首先将 subProp 集合的元组反转，构造反转后的集合 reverseSubProp。在一个循环中对 reverseSubProp 和 subProp 进行 join 连接操作，提取连接后生成的新 subPropertyOf 关系，并入 subProp 集合中，若并入操作以后 subProp 中元素数量没有增加，则跳出循环，返回 subProp。

算法 4.3：RDFS 规则 11 subClassOf 传递闭包推理

```
输入：subClassOf 模式数据集合 subClass
输出：subClassOf 传递闭包计算结果
begin
  reverseSubClass ← subClass.map(t => (t._2, t._1))  //(c2,c1)←(c1,c2)
  while true do
    joined ← reverseSubClass.join(subClass)  //(c2,(c1,c3))←(c2,c1)join(c2,c3)
    res ← joined.map(t => t._2)  //(c1,c3)←(c2,(c1,c3))
    subClass ← subClass.union(res)
    if subClass 中元素数量没有增加 then break
  end while
  return subClass
end
```

算法 4.3 所描述的 subClassOf 传递闭包推理算法与算法 4.2 的思路类似。

算法 4.4：RDFS 规则 7 subPropertyOf 继承性推理

```
输入：实例数据集合 spo，subPropertyOf 模式数据集合 subProp
输出：经过 RDFS 规则 7 推理后的实例数据集合 spo
begin
  pso ← spo.map(t => (t._2, (t._1, t._3)))  //(p1,(s1,o1))←(s1,p1,o1)
  joined ← pso.join(subProp)  //(p1,((s1,o1),q1))←(p1,(s1,o1))join(p1,q1)
  res ← joined.map(t._2._1._1, t._2._2, t._2._1._2)  //(s1,q1,o1)←(p1,((s1,o1),q1))
  spo ← spo.union(res)
end
```

算法 4.4 所描述的 subPropertyOf 继承性推理算法首先将实例数据集合 spo 的谓词提取到 key 的位置，即构造形如(p, (s, o))的集合 pso，然后将 pso 与 subProp 以谓词 p 为 key 做 join 连接操作。连接后的结果映射成新的三元组，结果并入实例数据集合 spo 中。

算法 4.5：RDFS 规则 2 Property domain 推理

```
输入：实例数据集合 spo，模式数据集合 domain
输出：经过 RDFS 规则 2 推理后的类型关系集合 types
begin
  pso ← spo.map(t => (t._2, (t._1, t._3)))  //(p1,(s1,o1))←(s1,p1,o1)
  joined ← pso.join(domain)  //(p1,((s1,o1),c1))←(p1,(s1,o1))join(p1,c1)
  res ← joined.map(t => (t._2._1._1, t._2._2))  //(s1,c1)←(p1,((s1,o1)
  types ← types.union(res)
end
```

算法 4.5 所描述的 Property domain 推理算法也是首相将 spo 集合中的谓词 p 提取到 key 的位置，然后与 domain 集合做连接 join 操作，连接后的结果映射成类型关系元组，并入类型关系集合 types 中。

算法 4.6：RDFS 规则 3 Property range 推理

```
输入：实例数据集合 spo，模式数据集合 range
输出：经过 RDFS 规则 3 推理后的类型关系集合 types
begin
  pso ← spo.map(t => (t._2, (t._1, t._3)))  //(p1,(s1,o1))←(s1,p1,o1)
  joined ← pso.join(range)  //(p1,((s1,o1),c1))←(p1,(s1,o1))join(p1,c1)
  res ← joined.map(t => (t._2._1._2, t._2._2))  //(o1,c1)←(p1,((s1,o1),c1))
  types ← types.union(res)
end
```

算法 4.6 所描述的 Property range 推理与算法 4.5 类似，只是在连接后的结果映射成类型关系元组时，元素位置有所不同。

算法 4.7：RDFS 规则 9 subClassOf 继承性推理

```
输入：subClassOf 模式数据集合 subClass，类型关系集合 types
输出：经过 RDFS 规则 9 推理后的类型关系集合 types
begin
  reverseTypes ← types.map(t => (t._2, t._1))   //(c1,s1)←(s1,c1)
  joined ← subClass.join(reverseTypes)   //(c1,(c2,s1))←(c1,c2)join(c1,s1)
  res ← joined.map(t => (t._2._2, t._2._1))   //(s1,c2)←(c1,(c2,s1))
  types ← types.union(res)
end
```

RDFS 规则 12、13 仅有一个规则前件，推理算法比较简单，在此不再详述。

2.1.3 基于 Spark 广播变量的优化

Spark 提供了一种称为广播变量(broadcast variable)的机制，该机制使得 Spark 可以在集群中的每个计算结点的内存中缓存一个只读的变量，而从开发者的角度来看，可以像使用一个常规变量那样使用该共享变量抽象。于是当结点在计算过程中需要取广播变量的值，可以直接从结点本身的内存中获取，而无需通过任务执行过程中从远程其他结点上拷贝变量的副本。

通常情况下，对于一个实际 RDF 三元组集合来说，实例三元组数据的数量往往是远远大于模式三元组数据的。因此在算法 4.1 的数据划分过程之后，所产生的模式数据 subClass、subProp、domain、range 均可设置为 Spark 的广播变量，以备后续的计算过程。

2.2 分布式 OWL Horst 推理

上一节讨论了基于 Spark 特性实现的 RDFS 语义推理算法，这一节探讨更复杂的基于 OWL Horst 规则集[8]的推理。OWL Horst 规则集是事实上的可扩展 OWL 推理标准，在工业界有 OWLIM 等三元组存储引擎支持，同时它在 OWL Full 的不可判定性与 RDFS 的有限表达能力中间达到了一个平衡，既降低了计算复杂度，

又维持了相当不错的表达性，因此基于 OWL Horst 规则集的推理是很多分布式推理研究工作都在考察的重点[1][2][3]。

2.2.1 OWL Horst 规则与划分

基于 OWL Horst 规则集的推理也被称作 pD*推理，相关的规则集是表 2.1 和表 2.4 所示规则集的并集。

表 2.4 OWL Horst 规则集

编号	前件	后件
1	p rdf:type owl:FunctionalProperty u p v u p w	v owl:sameAs w
2	p rdf: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 q w	w p v
9	v rdf:type owl:Class v owl:sameAs w	v rdfs:subClassOf w
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

续表 2.4

编号	前件	后件
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 w v owl:onProperty p u rdf:type v u p x	x rdf:type w

分析表 2.4 中规则的特点，根据规则前件中不同类别条件的数量，可以对 OWL Horst 规则做如下划分：

1) 规则前件只有一个三元组

规则 5a、5b、6、12a、12b、13a、13b 就是这样的规则，正如 2.1.1 节分析 RDFS 规则时所述，因为这些规则前件只有一个三元组，在推理过程的任意阶段实施一轮推理均可，对推理并行性无影响。

2) 规则前件有两个及两个以上三元组，但只含至多一个实例三元组

规则 3、8a、8b、9、10、12c、13c、14a、14b 就是这样的规则，对于这类规则的推理，计算开销最大的部分就在于一次实例三元组与模式三元组之间的 join 操作。由于与条件中含有两个三元组的 RDFS 规则类似，这类规则的推理也可以沿用相同的策略，即全量模式三元组通过 Spark 广播变量机制缓存到每一个结点上，实例三元组划分为分区分布到各个结点，于是在每个结点上各分区中的实例数据便可以直接与本结点上的模式三元组实施 join 操作。

3) 规则前件有两个实例三元组

规则 1、2、4、7、11、15、16 就是这样的规则，这类规则涉及到实例三

元组之间的 join 操作。但这种单纯的实例数据之间的 join 操作面临着一个问题，即将 join 一侧所需数据完全载入内存而 join 另一侧数据以分块或逐条流式读取的方式来实现连接运算便不再可行，因为实例三元组的数量远大于模式三元组而单个计算结点的内存根本不足以完全加载实例三元组。

规则1、2需要一次模式三元组与实例三元组之间以谓词p为关键字的join操作，以及一次实例三元组之间的 join 操作。规则 4 是一个模式三元组和两个实例三元组之间的三方 join 操作，乍看之下与规则 1、2 相似，但是两者其实有着显著不同，即规则 4 的结论可以作为它本身的条件进一步推理，形成了传递性关系，需要使用传递闭包计算的方式来处理。规则 7、11 可归为 sameAs 类。规则 15、16 则需要至少 3 次 join 操作。

4) owl:sameAs 规则

规则 5、6、7、9、10、11 就属于这类规则。

2.2.2 传递性规则推理算法

对传递性规则(规则 4)实施推理实际上就是在相关的 RDF 图上计算传递闭包，从关系代数的角度考虑，在一个有向图的顶点对集合上求传递闭包的通常做法是在该顶点对集合上实施自连接操作。但是这种方法的缺点也很明显，即迭代执行次数会很多，一次迭代计算至多只能使传递链的长度加 1，长度为 n 的隐式传递链需要至多 n 次迭代计算才能完全求出。另外，这种方法会产生大量的重复计算，因为如果不加限制的话每轮迭代都会将之前已经推理出的结论元组再推理一遍，而且同一个结论可能来自于不同条件的组合。举例来说，假设有三元组集合{(a p b), (b p d), (a p c), (c p d)}且谓词 p 具有传递性，那么由(a p b)与(b p d)可以得到(a p d)，由(a p c)与(c p d)也可以得到(a p d)，这样就产生了重复计算。

为了克服上述缺点，本文采用文献[9]中所述的传递闭包算法来执行传递性规则的推理。具体算法过程如算法 4.8 所示：

算法 4.8：分布式 OWL Horst 规则 4 传递闭包推理算法
输入：模式三元组集合 schema，一般实例三元组集合 spo 输出：更新后的一般实例三元组集合 spo

```
begin
  transProp ← schema 中宾语是 owl:TransitiveProperty 的三元组的主语
  transSPO ← spo 中谓词存在于 transProp 中的实例三元组
  result ← transSPO
  inc ← result   //inc 每轮迭代新推理出来的三元组
  left ← inc.map(t => ((t._2, t._3), t._1))
  do
    right ← inc.map(t => ((t._2, t._1), t._3))
    phase1 ← left.join(right).map(t => (t._2._1, t._1._1, t._2._2))
    inc ← phase1.subtract(result)
    incCount ← inc 中三元组数量
    if incCount!=0 then
      left ← inc.map(t => ((t._2, t._3), t._1))
      right ← result.map(t => ((t._2, t._1), t._3))
      phase2 ← left.join(right).map(t => (t._2._1, t._1._1, t._2._2))
      result ← phase2.union(inc).union(result)
    end if
  while incCount != 0
  transSPO ← result
  spo ← spo.union(result)
end
```

上述算法首先从所有模式三元组 schema 中过滤出所有具有传递性的属性得到 transProp，再从实例三元组集合 spo 中过滤出谓词属于传递性属性的实例三元组得到 transSPO。使用一个 inc 变量来记录每轮迭代过程中新推理出来的三元组增量，初始化为 transSPO。迭代过程在一个 while 循环中进行，每一轮迭代分为两个阶段。阶段一中，首先是增量与增量自身的 join 操作，join 的结果 phase1 对已推理出结果 result 求差就得到当前轮迭代的增量 inc，求差集的目的就是为了避免对已推理出结果的重复计算。若 inc 中三元组数量 incCount 不为 0，即该轮迭代还能够进一步推理出新的结果就进入阶段二。阶段二首先对 inc 和 result 实施 join 操作得到 phase2，然后把 phase2 和 inc 的三元组并入已推理结果集 result 中。只要 incCount 不为 0，while 循环就一直迭代进行下去。

2.2.3 两个实例三元组连接类规则推理算法

规则前件中含有两个实例三元组的 OWL Horst 规则有规则 1、2、4、7、11、

15、16，由 2.1.1 节的讨论可知，规则 4 可归为传递性规则，规则 7、11 可归为 sameAs 类规则。因此本节只讨论规则 1、2、15、16，而这四条规则按照所需连接次数又可以分为 2 次 join 的规则 1、2，3 次 join 的规则 15、16。

对于规则 1、2 来说，其推理模式比较相似，都是使用模式三元组中提供的谓词 p 来过滤一般实例三元组，然后针对过滤出来的实例三元组，使用两个资源的组合作为 join 操作的关键字。之所以使用两个资源的组合做连接而不是一个资源，是因为所有三元组数据对于单个主语或宾语资源有可能非常不均匀地分布在集群结点上，造成数据倾斜和负载不均衡问题，而使用主谓或谓宾的组合作为数据分区标准则可以大大减少数据倾斜的可能。对于规则 1，join 关键字是实例三元组中主语和谓词(subject, predicate)的组合；对于规则 2，join 关键字是实例三元组中谓词和宾语(predicate, object)的组合。规则 1 的推理如算法 4.9 所示，规则 2 的推理算法类似，不再赘述。

算法 4.9：分布式 OWL Horst 规则 1 推理算法

```
输入：模式三元组集合 schema，三元组集合 triples
输出：经过推理更新后的相同类集合 same
begin
  funcProp ← schema
    .filter(t => t._2==rdf:type && t._3==owl:FunctionalProperty)
    .map(t => t._1)
  funcPropSet ← funcProp 生成集合 set 然后构造广播变量
  funcTriple ← triples.filter(t => funcPropSet.value.contains(t._2))
  func1 ← funcTriple.map(t => ((t._1, t._2), t._3))
  func1 中的元素((s, p), o)以(s, p)为关键字进行分区
  result ← func1.join(func1).map(t => (t._2._1, t._2._2)).filter(t => t._1!=t._2)
  same ← same.union(result)
end
```

上述规则 1 的算法首先从模式三元组集合 schema 中过滤出所有函数式属性，构造成集合后作为广播变量得到 funcPropSet。然后从三元组集合 triples 中过滤出谓词存在于 funcPropSet 中的三元组，再将三元组(s, p, o)的主谓相组合，得到形如((s, p), o)的三元组集合 func1。接着 func1 中元素以(s, p)为关键字进行数据分区，(s, p)相同的三元组将会被划分到同一个计算结点上。func1 自身做 join 操作，得

益于上一步的分区，这些 join 都是在计算结点本地进行，最后过滤掉资源相同的元组得到结果 result。

对于规则 15、16 来说，涉及到多个 join 操作。形如(v owl:someValuesFrom w)和(v owl:onProperty p)的模式三元组间的 join 操作，由于模式数据远小于实例数据，模式数据可以完全加载到结点内存中，在内存中完成 join 操作。而大规模实例数据形如(u p x)匹配所有三元组，(x rdf:type w)匹配很大的一个子集，无法完全加载到单台机器的内存中，需要考虑一种有效的数据分区方法，使得被划分到每个计算结点上的待连接实例数据尽可能地均匀且尽可能地小。

算法 4.10：分布式 OWL Horst 规则 15 推理算法

```
输入：三元组集合 triples，类型关系集合 types(x, w)
输出：推理增加后的类型关系集合 types
begin
  someValSO← triples.filter(t => t._2==owl:someValuesFrom).map(t => (t._1, t._3))
  onPropSO ← triples.filter(t => t._2==owl:onProperty).map(t => (t._1, t._3))
  schemaJoin ← someValSO.join(onPropSO)                          //(v,(w,p))
  schemaJoinW ← schemaJoin.map(t => (t._2._1, (t._2._2, t._1)))  //(w,(p, v))
  schemaJoinP ← schemaJoin.map(t => (t._2._2, null))             //(p, null)
  typesReverse ← types.map(t => (t._2, t._1))                    //(w,x)
  types_join_schema ← typesReverse
    .join(schemaJoinW).map(t => ((t._2._1, t._2._2._1), t._2._2._2))  //((x,p),v)
  triplesPSO ← triples.map(t=> (t._2,(t._1, t._3)))              //(p,(u,x))
  spo_join_schema ← triplesPSO
    .join(schemaJoinP).map(t => ((t._2._1._2, t._1), t._2._1._1))  //((x,p),u)
  result←spo_join_schema.join(types_join_schema).map(t=>(t._2._1, t._2._2)) //(u,v)
  types ← types.union(result)
end
```

本文采用与文献[1]中类似的方法，首先对模式数据(v owl:someValuesFrom w)和(v owl:onProperty p)进行 join 操作，得到的结果再分别与(u p x)、(x rdf:type w)形式的三元组实施 join，这样得到两个 join 结果。这两个结果中均含有变量 x 和 p，于是就可以根据 x 与 p 的组合(x, p)对上一步的两个 join 结果实例数据分区，使含有相同(x, p)关键字的元组被划分到集群中同一个计算结点上。又因为 x 可以匹配三元组主语或谓语，p 匹配三元组的谓词，(x, p)组合在一起作为关键字分区

可以有效避免只对单一资源(主语或谓语)进行分区导致的数据倾斜，每个结点分得的实例数据相对更均匀。具体的规则 15 推理如算法 4.10 所示，规则 16 类似。

2.2.4 sameAs 类规则推理算法

sameAs 类规则就是三元组谓词部分为 owl:sameAs 的规则，包括规则 5、6、7、9、10、11。该类规则是 OWL Horst 规则集中计算和空间开销最大的一类，因为按照传统的推理方法完全实施 sameAs 规则推理会急剧地产生大量对于进一步推理价值很低甚至无用的新三元组。

为了减少无价值三元组的大量产生，降低计算和存储资源开销，已有的一些分布式推理引擎采用的主流处理办法是不直接显式地推理出 sameAs 结论三元组，而是采用并查集思路，构建一种存储上更紧凑的映射表的数据结构 sameAs 表[1]。

算法 4.11：分布式 OWL Horst 规则 7 推理算法

```
输入：以 owl:sameAs 为谓词的三元组预处理后，(主语, 宾语)二元组集合 same
输出：组号到资源映射集合 idToResource(groupId, {resource1, resource2, … resourceN})，资源到组号映射集合 resourceToId(resource, groupId)
begin
  do
    sameIter ← same
    same ← sameIter.map(t => (t._2, t._1)).union(sameIter).groupByKey()
      .flatMap(t => {
        minSecond = t._2 中的最小值
        if(t._1 < minSecond) then
          foreach resource in t._2
            生成二元组(resource, t._1)
          end for
        else
          foreach resource in t._2
            生成二元组(resource, minSecond)
          end for
        end if
      })
    same ← same 中去掉满足 t._1 与 t._2 相等的二元组
  while same 中内容发生改变
  resourceToId ← same 中元素经过去重后
  idToResource ← resourceToId.map(t => (t._2, t._1)).groupByKey()
end
```

本文采用类似于 sameAs 表的推理算法设计思想，首先以规则 7、11 为例，它们的依赖关系是 7→11，故先处理规则 7。规则 7 以谓词为 owl:sameAs 的三元组的(主语, 宾语)二元组集合为输入，生成两组映射集合：一个是组号 groupId 到资源的映射，便于根据 groupId 遍历该组下面的所有等价资源，可作为规则 11 推理算法的输入；另一个是资源到组号 groupId 的映射，便于查询某一给定资源所属组号。相关算法如算法 4.11 所示。

规则 11 的推理以一般实例三元组集合和规则 7 推理的输出为输入，对于每一个一般实例三元组，若其主语能够通过 resourceToId 查找到一个对应组号 groupId，那么就再用该 groupId 在 idToResource 集合中取出该组下面的所有资源，替换到原三元组的主语位置，构造出新的结论三元组。对于该实例三元组的宾语也实施类似的处理过程。具体的推理算法过程如算法 4.12 所示。

算法 4.12：分布式 OWL Horst 规则 11 推理算法

```
输入：一般实例三元组集合 spo，组号到资源映射集合 idToResource，资源到组号映射集合 resourceToId
输出：经规则 11 推理更新后的一般实例三元组集合 spo
begin
   foreach triple t in spo
      if resourceToId 包含 t._1 和 t._3 then
         sameSubjects <- idToResource.get(resourceToId.get(t._1))
         sameObjects <- idToResource.get(resourceToId.get(t._3))
         foreach s in sameSubjects
            foreach o in sameObjects
               生成三元组(s, t._2, o)加入 spo
            end for
         end for
      end if
   end for
end
```

因为采用了 sameAs 表的思想来解决 owl:sameAs 相关的推理，规则 7 推理执行后，规则 6、9、10 的推理算法便没有必要再专门实现了。

2.3 分布式通用规则的推理

上述的相关分布式语义推理算法均是面向特定的规则集如 RDFS 和 OWL Horst 来实现的，对于不同形式规则的特点，提出有针对性的优化策略。这些推理算法的缺点是不具有一般性，每一个规则都需要专门实现对应分布式推理算法，无法支持灵活多变的推理规则需求。基于上述原因，本节探究讨论一种基于 Spark 实现的采用 SWRL 规则模式的分布式通用规则推理算法。

2.3.1 通用规则的表示模型

SWRL 规则由规则体(body)和规则头(head)构成，规则体即前件部分，规则头即后件部分，SWRL 规则的含义是当前件部分所有三元组在给定的知识库上满足时，后件部分得到的结论三元组也一定满足。一条 SWRL 规则可以采用 Horn 子句的形式表示，规则前件部分和后件部分由一到多个 RDF 三元组表示的规则原子合取构成，如式(4.1)所示，其中左侧是规则体，右侧是规则头。

$$B_1 \wedge B_2 \wedge \cdots \wedge B_n \rightarrow H \tag{4.1}$$

对于每个规则原子三元组，主语、谓词、宾语除了满足一般三元组可取值之外，还可以是以问号?开头的变量，对于知识库中匹配规则原子模式的三元组，变量可以由对应的资源或是数值常量替换，例如对以下规则 r：

$$B_1 :< ?x \ \ father ? y > \wedge B_2 :< ? y \ brother ? z > \rightarrow H :< ? x \ uncle ? z > \tag{4.2}$$

其中?x、?y、?z 都是变量，式(4.2)规则表达的含义是若存在?x 的父亲?y 且?y 的兄弟是?z，那么根据规则可得到结论?x 的舅舅是?z。给定知识库 K，包含三元组 t_1:<p_1 *father* q_1>，t_2:<p_2 *father* q_1>，t_3:<q_1 *brother* u_1>。则变量?x 由 p_1 和 p_2 替换，变量?y 由 q_1 替换，变量?z 由 u_1 替换，那么在知识库 K 应用规则 r 的推理结果就是 t_4:<p_1 *uncle* u_1>，t_5:<p_2 *uncle* u_1>。

2.3.2 通用规则分布式推理在 Spark 上的实现

对于在 Spark 上实现通用 SWRL 规则的分布式推理，文献[10]中提出了一种将 Datalog 规则转成 Spark RDD 变换操作来部分实现分布式 SWRL 推理能力的方法，

本文根据这些相关方法，做出一些改进。

算法 4.13：在 Spark 上通过 Datalog 实现类 SWRL 推理的分布式算法

```
输入：知识库三元组集合 triples，规则 r
输出：推理结果三元组集合 result
begin
  commonVars ← 提取 r 的规则体中所有公共变量及相关信息                    // 1.
  for each bodyAtom in r 的规则体中的所有规则原子                          // 2.
    matched ← triples.filter(t => 匹配 bodyAtom 模式的三元组 t)
    matchedMap.put(bodyAtom.index, matched)
  end for
  joinResult ← 初始化为第一个 commonVar 中的第一个 bodyAtom 对应的 matched 集合
  for each commonVar in commonVars                                        // 3.
    for each bodyAtomIndex in commonVar.bodyAtomIndexes
      matched ← matchedMap.get(bodyAtomIndex)
      rHS ← matched.map(t => adjustKey(t, commonVar.getVarIndex(bodyAtomIndex)))
      joinResult ← joinResult.join(rHS)
    end for
  end for
  result ← joinResult.map(t => derivedByHead(t, r 的规则头))              // 4.
end
```

1. 在上述算法中，首先对规则 r 中的所有公共变量进行预处理。所谓公共变量就是指规则体中，在两个及两个以上规则原子里出现过的变量，这些公共变量是对不同规则原子所匹配的三元组进行连接操作的重要基础。规则 r 的规则体中所有公共变量及其相关信息被提取然后存放到集合 commonVars 中，其中每个元素 commonVar 包含的信息有公共变量名、变量所在规则原子的下标 bodyAtomIndex、变量在所处三元组中位置下标 varIndex。
2. 接下来在一个循环中对 r 的规则体中每一个规则原子 bodyAtom 进行处理。先从知识库三元组全集 triples 中过滤出能够与 bodyAtom 匹配的三元组，放入 matched 集合中。所谓匹配，就是指对于给定的 bodyAtom 规则原子，其中非变量位置的资源或数值常量能够与当前三元组中对应位置的资源或数值常量完全相同，而变量位置可以匹配到任意资源。然后构造规则原子的下标到 matched 集合的映射 matchedMap，便于在后续步骤

中快速查找。

3. 在一个二重循环中，对于每一个 commonVar 中的每一个规则体原子下标 bodyAtomIndex，从 matchedMap 获取对应的匹配三元组集合 matched。对 matched 实施变换操作，在 adjustKey 方法中，根据公共变量在所处三元组中位置下标 varIndex，调整公共变量到连接 key 的位置，构造连接右手边变量 rHS，便于下一步的 join 操作。再将连接结果 joinResult 与 rHS 做 join 操作，更新 joinResult。
4. 最后，根据 r 的规则头的模式及变量，从 joinResult 中提取所需的变量构造推理结果 result。

参考文献

[1] Urbani J, Kotoulas S, Maassen J, et al. OWL reasoning with WebPIE: calculating the closure of 100 billion triples[C]//Extended Semantic Web Conference. Springer Berlin Heidelberg, 2010: 213-227.

[2] Kim J M, Park Y T. Scalable OWL-Horst ontology reasoning using SPARK[C]//2015 International Conference on Big Data and Smart Computing (BIGCOMP). IEEE, 2015: 79-86.

[3] Gu R, Wang S, Wang F, et al. Cichlid: efficient large scale RDFS/OWL reasoning with spark[C]//Parallel and Distributed Processing Symposium (IPDPS), 2015 IEEE International. IEEE, 2015: 700-709.

[4] Liu Z, Feng Z, Zhang X, et al. RORS: Enhanced Rule-based OWL Reasoning on Spark[J]. arXiv preprint arXiv:1605.02824, 2016.

[5] Hindman B, Konwinski A, Zaharia M, et al. Mesos: A Platform for Fine-Grained Resource Sharing in the Data Center[C]//NSDI. 2011, 11: 22-22.

[6] Vavilapalli V K, Murthy A C, Douglas C, et al. Apache hadoop yarn: Yet another resource negotiator[C]//Proceedings of the 4th annual Symposium on Cloud Computing. ACM, 2013: 5.

[7] Antlr. http://www.antlr.org/.

[8] ter Horst H J. Completeness, decidability and complexity of entailment for RDF Schema and a semantic extension involving the OWL vocabulary[J]. Web Semantics: Science, Services and Agents on the World Wide Web, 2005, 3(2): 79-115.

[9] Leskovec J, Rajaraman A, Ullman J D. Mining of massive datasets[M]. Cambridge University Press, 2014.

[10] Wu H, Liu J, Wang T, et al. Parallel Materialization of Datalog Programs with Spark for Scalable Reasoning[C]//International Conference on Web Information Systems Engineering. Springer International Publishing, 2016: 363-379.

[11] Guo Y, Pan Z, Heflin J. LUBM: A benchmark for OWL knowledge base

systems[J]. Web Semantics: Science, Services and Agents on the World Wide Web, 2005, 3(2): 158-182.

[12] Motik B, Nenov Y, Piro R, et al. Parallel Materialisation of Datalog Programs in Centralised, Main-Memory RDF Systems[C]//AAAI. 2014: 129-137.