基于 Prüfer 序列的 RDF 数据索引与查询

刘翔宇[1)] 吴 刚[2),3)]

[1)](东南大学计算机科学与工程学院 南京 210096)

[2)](医学影像计算教育部重点实验室(东北大学) 沈阳 110819)

[3)](东北大学信息科学与工程学院 沈阳 110819)

摘 要 语义 Web 作为数据之网不断汇集并组织 Web 信息,相关应用因此面临着对语义 Web 所含大规模 RDF 数据高效访问的挑战.建立有效的索引机制是提升 RDF 数据管理和查询性能的一种解决之道.序列式索引既能够支持存储空间压缩又便于采用成熟的序列匹配技术进行数据处理,具有较好的查询处理性能.文中扩展 Prüfer 序列以支持 RDF 数据上的索引和查询,实现了名为 Prig 的原型系统.实验比较并分析了该系统与对比系统在 LUBM 和 SP^{2}Bench 两个测试基准上的实验结果,指出扩展的 Prüfer 索引在大规模 RDF 数据上有着比对比系统更好的查询处理性能.

关键词 RDF;Prüfer 序列;索引;查询

中图法分类号 TP311 **DOI 号**: 10.3724/SP.J.1016.2011.01997

An Indexing and Query Processing Approach of RDF Data Based on Prüfer Sequence

LIU Xiang-Yu[1)] WU Gang[2),3)]

[1)](*School of Computer Science and Engineering, Southeast University, Nanjing* 210096)

[2)](*Key Laboratory of Medical Image Computing of Ministry of Education (Northeastern University), Shenyang* 110819)

[3)](*College of Information Science and Engineering, Northeastern University, Shenyang* 110819)

Abstract As a web of data, the Semantic Web is assembling and organizing web information. Therefore, Semantic Web applications face the challenge of storage and processing of RDF data at a larger and larger scale. An efficient indexing scheme may be one of the solutions. The sequence-based indexing can bring good query performance with mature sequence matching techniques while keeping reasonable space consumption. In this paper, we extend the Prüfer sequence approach to support index and query processing on RDF data, and implement a prototype system called Prig. Performance comparisons with Sesame RDF framework on LUBM benchmark and SP^{2}Bench benchmark are presented. The experimental results illustrate that our approach has a better performance on large-scale RDF data.

Keywords RDF; Prüfer sequence; Index; query

1 引 言

语义 Web[1] 作为下一代 Web 正越来越受到包括学界和工业界在内的研究人员的关注.语义 Web 中的数据是以 RDF(Resource Description Framework)① 数据的形式进行表示的. RDF 提供了灵活的数据模型以用于表达数据之间的关系.基于这种模型的

收稿日期:2011-07-10;最终修改稿收到日期:2011-08-10.本课题得到国家自然科学基金(60903010,61025007,60933001)、国家"九七三"重点基础研究发展规划项目基金(2011CB302206)、江苏省自然科学基金(BK2009268)及北京市"现代信息科学与网络技术"重点实验室开放课题(XDXX1011)资助.**刘翔宇**,男,1986 年生,硕士研究生,主要研究方向为语义 Web 的数据管理. E-mail: xyliu@seu.edu.cn. **吴 刚**(通信作者),男,1978 年生,博士,副教授,中国计算机学会(CCF)会员,研究方向为语义 Web、分布式计算、不确定数据管理. E-mail: wugang@ise.neu.edu.cn.

① http://www.w3.org/TR/rdf-primer/

数据提供了人和机器均易于理解的语义.本质上RDF数据是一个巨大的有向图.在这个图中,每个现实世界中可以标识的实体都是一个结点,两个实体之间的相互关系可以看作是连接这两个实体的带标记的有向边.RDF数据模型在提供这种灵活而强大的能力的同时也带来了数据存储和处理复杂度提高的问题.如何高效地存储和查询大规模的RDF数据一直是语义Web数据管理中的难题.建立有效的索引机制是提升RDF数据管理和查询性能的一种方法.序列式索引既能够支持存储空间压缩又便于采用成熟的序列匹配技术进行处理,具有较好的查询处理性能.考虑到Prüfer序列化①方法能够有效地支持树形结构数据的索引和查询处理[2],但尚不能支持图数据的索引和查询处理,本文提出了一种扩展Prüfer序列化的方法,设计并实现了名为Prig(Prüfer sequences for Indexing Graph)的原型系统以支持对RDF图数据的相应处理.与索引和查询树形结构数据类似,Prig系统将RDF数据解析成RDF图,然后利用扩展的Prüfer序列化方法生成RDF图序列,并进行索引;RDF查询语句被转换为查询子图,同样生成扩展的Prüfer序列;通过匹配查询子图序列与原RDF图序列以及后处理操作,即可得到查询结果.系统架构如图1所示.

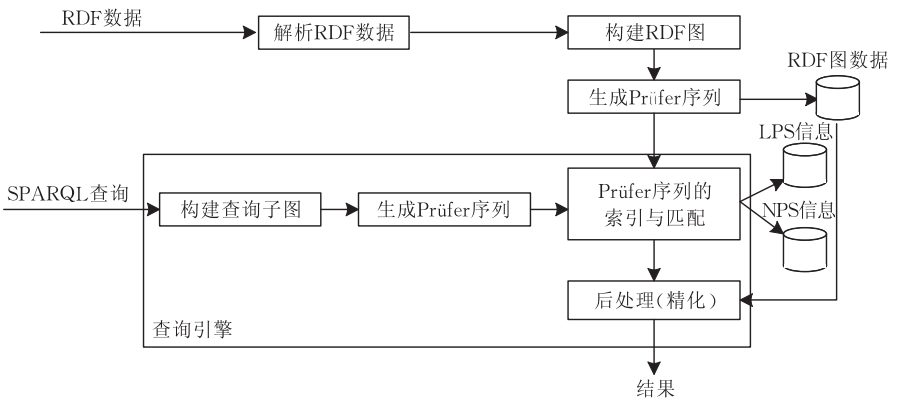

图1 Prig整体架构

本文将首先介绍RDF数据模型、SPARQL查询语言②与Prüfer序列化方法,之后介绍RDF数据管理和序列化相关的工作,第4节详细论述实现数据和查询序列化的方法;第5节论述相应的匹配规则;第6节给出实验结果并加以分析;最后对本文工作总结.

2 RDF数据模型与SPARQL查询语言

2.1 RDF数据模型

RDF的基本构成是以三元组〈subject,predicate,object〉形式存在的对资源的陈述(statement).每一个三元组包括一个主语(subject)、一个谓语(predicate)和一个宾语(object).

三元组的集合称为RDF图.RDF图可以通过带有标签的结点和带有标签的边表示,其中每一个三元组对应为图上的一个"结点-边-结点"的子图,陈述了由谓语表示的在主语和宾语所指的事物之间的关系[3].一个RDF图的结点就是它包含的所有三元组的主语和宾语,而边的方向总是指向宾语.通常可以把RDF图看作一个有向标记图.如图2所示.

图2 包含一个三元组的RDF图

RDF图的含义就是其所有三元组陈述的合取.例如如下三元组集合表示的RDF图的含义是:XXX是一名教授,他在University1工作,他是Department1的成员,Department1是University1的一个下属机构.

〈XXX, rdf: type, Professor〉,
〈XXX, worksFor, University1〉,
〈XXX, memberOf, Department1〉,
〈Department1, subOrgnizationOf, University1〉.

2.2 SPARQL查询语言

SPARQL(Simple Protocol and RDF Query Language)是目前W3C(World Wide Web Consortium)针对RDF查询语言的推荐标准,它定义了RDF查询语言的语法和语义.例如,"查询工作于University1并且是Department1成员的教授",可

① http://en.wikipedia.org/wiki/Pr%C3%BCfer_sequence
② http://www.w3.org/TR/2008/REC-rdfsparql-query-20080115/

以构造如下 SPARQL 语句：

```
SELECT ?x
Where {?x rdf: type Professor.
       ?x worksFor University1.
       ?x memberOf Department1.
       Department1 subOrgnizationOf University1.}
```

以上 SPARQL 查询可用图 3 直观表示. 三元组模式的集合，称为基本图模式(basic graph pattern). 三元组模式类似于 RDF 三元组，不同之处在于主语、谓语和宾语可以是变量. 一个基本图模式匹配 RDF 数据图的一个子图，其条件是子图中的 RDF 术语(统一资源标识符、字面量、空白结点)可以被变量置换，置换结果和原基本图模式等价.

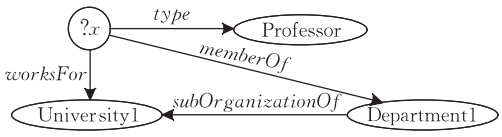

图 3　查询子图

2.3　Prüfer 序列化方法

Prüfer 序列化是一种通过每次删除树上的一个结点来构造一个序列的方法. 构造序列的一般方法是对树进行后根顺序遍历，删除叶节点的同时记录删除结点的双亲结点，从而生成相应序列. Prüfer 序列中的每个元素与树上的每个结点一一对应. 如图 4 所示的树对应的 Prüfer 序列是：(B,3)(B,3)(A,7)(B,6)(B,6)(A,7).

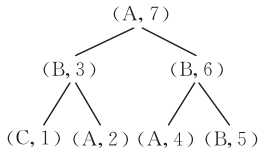

图 4　树形结构

扩展的 Prüfer 序列[2]的方法被应用于 XML 数据的索引语查询，其生成方式与标准的 Prüfer 序列生成方式类似，只是在序列化之前，先为树中的每个叶子结点添加一个空节点作为子节点，然后生成 Prüfer 序列. 这样能够将叶子结点也纳入序列中. 在实现中，序列通常被分为两部分，一部分为标签序列 LPS(Label Prüfer Sequence)，另一部分为数字序列 NPS(Number Prüfer Sequence). 上图中，结点旁标记的字母 A,B,C 为标签名称，1,2,3,4,…是用来唯一标识结点的标记. LPS 就是记录结点的这些信息的序列，NPS 则记录了相应的数字标记信息. 对于大多数查询而言，只需提供待查的标签名称，要求系统返回相应的数据，通过 LPS 可以确定可能的候选结果，进而利用 NPS 剔除错误结果，返回正确结果.

对于上图的例子，采用扩展的 Prüfer 序列化方法生成的 LPS 和 NPS 如下：

LPS：C，B，A，B，A，A，B，B，B，A；

NPS：1，3，2，3，7，4，6，5，6；7.

3　相关工作

3.1　RDF 数据管理

目前 RDF 数据的存储和查询工具主要有 Sesame[4]、Jena[5]、3Store[6] 以及 RDFStore[7] 等. Sesame 是一个独立于存储的支持 SPARQL 查询的系统，采用图匹配的方法，支持语义 Web 上的数据查询. 3Store 在数据存储的层面上采用了 MySQL 数据库，主要关注于交互式的查询性能和数据导入时间. 在 Jena 中，数据的存储可以采用多种的后台数据库，但大多数情况下采用简单的关系数据库：对 Statement、URI 和 Literal 分别建立数据表. 这些工具作为目前 RDF 数据管理的主要工具，为语义 Web 的发展奠定了基础，但是也存在一些问题：

(1) RDF 数据不能直接在关系数据库中存储和查询. 通常需要使用中间层转换工具将 SPARQL 查询转换为 SQL 查询，这样大大地降低了系统的性能.

(2) 一些工具，例如 Jena 采用将 RDF 数据中的元素分类存储的办法，这虽然避免了使用中间层处理，但是带来了大量连接操作，导致系统性能下降. 后续的改进方案中，Jena 在其存储中使用更加详细的表模式，这样又带来了存储开销增加的问题.

在本文所提出的 Prig 系统中将使用 Sesame 作为性能比较的基准. 选择 Sesame 对比的主要原因是考虑到 Sesame 采用的处理方法代表了传统的 RDF 数据处理方式：基于 Triple store，使用大量的三元组选择/连接操作完成 SPARQL 查询，可能会导致系统性能较低，而 Prig 则针对这一问题进行了另一种方式的探索.

3.2　Prüfer 序列在文档匹配中的应用

采用 Prüfer 序列化方法进行文档匹配的研究工作主要集中于对树形结构的 XML 文档的匹配. ViST[8] 是一种将 XML 文档以及 twig 查询编码为一个序列的方法. 该序列是一个二维结构：{($a1$, $p1$),($a2$,$p2$)……}，其中 ai 是结点元素，pi 是到 ai 的路径信息，通过路径的信息实现对 XML 数据的查询匹配. 在最坏情况下，存储空间复杂度为 $O(n^2)$，且查询结果集中存在错误结果. PRIX[2] 通过将树形结构的 XML 文档生成 LPS(Label Prüfer Sequence)和 NPS(Number Prüfer Sequence)，利用

序列中元素之间的一致性和间距等特性进行查询和结果集过滤，返回最终结果. 基于 PRIX，Prasad 等人[9]提出了一种变化的 Prüfer 序列化方法，生成相应序列，并对原有的树形 XML 文档进行分层，进一步优化查询效率，一次性返回正确的结果集. Prüfer 序列还应用于本体匹配领域，将本体语义信息采用 Prüfer 序列化方法表示，进而进行相应的匹配[10]. 现有的方法不能直接应用于图数据的处理，当然也无法直接用于处理 RDF 图数据. 这主要是由于以下问题决定的：

(1) 对于图中的边含有标记信息的情况下匹配难以处理.

(2) 匹配算法对于 RDF 图的特点缺少具有针对性的解决策略，造成匹配算法运行时间开销的增加. 在对 RDF 图进行的查询中，经常有一些查询包含了对结点的约束信息，这些查询如果用一般的图模式匹配方法会造成较大的开销，进而影响系统的性能.

在 Prig 中，根据 RDF 图的特点改进 Prüfer 序列方法生成并匹配序列，可以有效降低系统开销，提高系统性能. 有关图的管理和挖掘的相关知识，可以参见文献[11].

4 RDF 图和查询子图的序列化

4.1 问题定义

在 Prig 系统中，RDF 图和查询子图被分别转换为 Prüfer 序列，然后通过采用序列匹配为方法完成图的查询匹配. 为此我们给出如下定义.

定义 1. 全图 Prüfer 序列. RDF 图 G 经过 Prüfer 序列化方法生成的 Prüfer 序列称为全图 Prüfer 序列，记为 S_G. $|S_G|$ 表示序列的长度，即序列中元素的数量. 用 S_{G_i} 表示序列中第 i 个元素，其中 $1\leq i\leq |S_G|$. 在本文中如果不特别指明，将全图 Prüfer 序列简称为全图序列.

定义 2. 查询子图 Prüfer 序列. 由查询子图经过 Prüfer 序列化方法生成的 Prüfer 序列称为查询子图 Prüfer 序列，记为 S_Q. $|S_Q|$ 表示序列的长度，即序列中元素的数量. 用 S_{Q_i} 表示序列中第 i 个元素，其中 $1\leq i\leq |S_Q|$. 在本文中如果不特别指明，将查询子图 Prüfer 序列简称为查询序列.

定义 3. 基于 Prüfer 序列匹配的查询. 依次从 S_Q 中自左至右选取元素 S_{Q_i}，根据匹配规则从 S_G 中选取相应元素，完成整个序列匹配的过程. 对于 S_{Q_i} $(1\leq i<|S_Q|)$，若存在 S_{G_j} 与之匹配，则 $S_{Q_{i+1}}$ 其候选结点自 S_{G_j} 之后继续开始查找.

定义 4. 查询结果 Prüfer 序列. 由满足查询条件的 S_G 中的结点构成的序列称为结果 Prüfer 序列，记为 S_A. $|S_A|$ 表示序列的长度，即序列中元素的数量. 用 S_{A_i} 表示序列中第 i 个元素，其中 $1\leq i\leq |S_A|$. 在本文中如不特别指明，将查询结果 Prüfer 序列简称为结果序列. S_A 中的所有元素均由 S_G 中的元素组成并保持了其在 S_G 中的相对位置，因此 S_A 是 S_G 的子序列.

除以上定义外，我们还做如下规定：若没有指明具体使用哪一种序列，则将直接使用 S 表示以上任意一种 Prüfer 序列，序列中的元素相应记为 S_i. 对于以上序列中元素所对应的图上结点的入度和出度分别记为 $d_i()$ 和 $d_o()$，例如查询序列中元素 i 对应结点的入度可以记为 $d_i(S_{Q_i})$. 序列对应图的出边集合用 $E_o()$ 表示，例如全图序列 S_{G_i} 对应图的出边集合记为 $E_o(S_{G_i})$.

4.2 RDF 图转化为 Prüfer 序列

将 RDF 图转化为 Prüfer 序列主要面临以下几方面的问题.

首先，在树形结构中生成 Prüfer 序列要求结点上有全局标签(label)信息，用于标识结点所属的类别，多个节点可以具有相同的全局标签. RDF 图中这种类别信息则是通过三元组来表示的. RDF 图中结点上的标签则唯一标识结点的标记，但无法表示结点所属的类别. 为了能够利用 Prüfer 序列方法，需要相应将三元组表示的信息转化为结点上的全局标签信息.

其次，Prüfer 序列起初应用于树形结构. 对于图的一个结点允许有多个双亲结点，其结构较之于树形结构更为复杂. 因此，在序列化过程中，记录结点的双亲结点也存在一些问题. 例如，在图 5 中，结点 course1 存在 graduatestudent1 和 professor2 等 4 个双亲结点，与此同时，graduatestudent1 又是 professor2 的双亲结点，这种层次上的混乱在树中是不会出现的. 针对树形结构构造的 Prüfer 序列所具有的一系列性质，很大程度上是由于单一的双亲结点对孩子结点在序列中的位置进行了分割[2]. 图中多双亲关系的存在，弱化了这些性质，使得后续在剔除错误候选结果时很难利用这些性质.

此外，RDF 图的边具有标签信息. 传统 Prüfer 标记方法中并未考虑树形结构中结点之间边的标签信息.

Prig 提出了一种针对图(特别是 RDF 图)的扩展 Prüfer 序列化方法来解决以上问题.

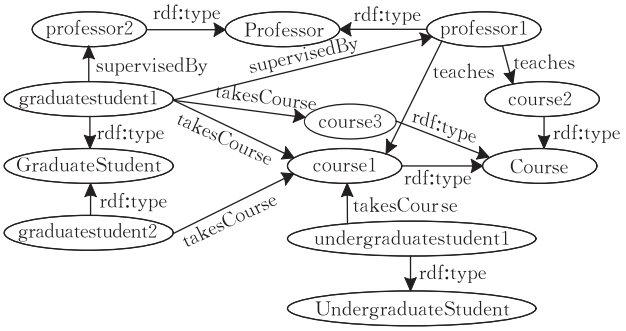

图 5　结构复杂的 RDF 图

针对缺少全局标签的情况，Prig 中采取的方法是将 RDF 图中所有的以 rdf：type 为谓语的三元组中宾语的 URI 作为主语结点的全局标签. 本文称之为“一般化处理”. 为了便于描述，首先将图 5 中的 rdf：type 信息（例如：GraduateStudent）、实体信息（例如：graduate1）以及边信息（例如：takesCourse）等分别规定其简写形式并赋予其 id. 具体如表 1～3 所示.

表 1　全局标签与简称

全局标签	简称
Professor	P
GraduateStudent	G
UndergraduateStudent	U
Course	C

表 2　实体到 Id 的映射

实体名称	Id
graduatestudent1	1
graduatestudent2	6
undergraduatestudent1	8
professor1	2
professor2	5
course1	3
course2	7
course3	4

表 3　边标签、简称与 Id

边标签	简称	Id
supervisedBy	sB	9
takesCourse	tc	11
teaches	t	10

如无特别指明，本文以后提到的全局标签和边标签将均用简称表述.

在 RDF 中，rdf：type 用来说明某个实体是某个类的一个实例. 以图 5 为例，graduateStudent1，professor1 以及 course1 的 rdf：type 分别为 GraduateStudent，Professor 和 Course；经过一般化处理将结点的 rdf：type 信息作为结点的标签. 然而 RDF 图中并非所有结点都有类信息. 对于这部分结点，在查询处理中不得不匹配任何待查询的标签. Prig 将此类结点的标签统一设为 ANY_LABEL. 经过一般化处理的 RDF 图如图 6 所示.

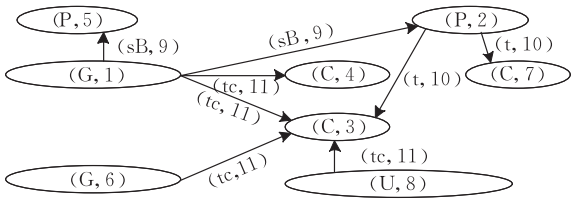

图 6　将 rdf：type 信息作为结点的标签

对于结点可能存在多个双亲结点的问题，则采用合并双亲结点的方法进行处理：记录每个待删除结点，将其所有双亲结点的集合视为一个合并后的结点，并记录在删除结点之后. 为了便于描述，称合并后的双亲结点为一个 bag. 以图 6 为例，为了能够记 RDF 图中的所有结点的信息，扩展的 Prüfer 序列化方法在 1 之前添加一个空节点，采用深度优先拓扑排序的方式删除相应结点记录双亲结点，生成 Prüfer 序列如表 5 中 NPS 栏所示.

为了支持边具有标签信息这一情况，Prig 采用在删除结点记录双亲结点的同时记录边上标签的方法. 这部分信息也记录在 bag 中. 例如，对于图 6 而言，删除结点 1 时，记录为 1(sB：2 tc：3 sB：5 tc：4).

通过以上方法可以将 Prüfer 序列化方法扩展应用于 RDF 图的序列化. 其中，在本文中序列元素的定义不包含所记录的双亲结点信息和边标签信息，即不包含记录在序列中括号内的相关信息. 因此，由定义 1 或 2 可知，序列的长度等于 RDF 全图或查询子图中结点的数量. 同时，序列中的元素与图中的结点之间具有一一对应关系，在本文中不特殊说明的情况下可以替换. 下面给出 RDF 图转化为 Prüfer 序列的一般步骤.

算法 1.　将 RDF 图转化为 Prüfer 序列的算法.

输入：一个 RDF 图 $G=V\times E$，其中 V 是顶点的集合，E 是边的集合

输出：Prüfer 序列 PS

RDF_to_ PrüferSequence(G)

1. $PS\leftarrow$空序列
2. WHILE $\exists n\in V$ 使得 n.label=NULL
3. IF n 具有类型 t THEN n.label$\leftarrow t$
4. ELSE n.label$\leftarrow$ANY_LABEL
5. WHILE $\exists n\in V$ 使得 $n.id$=NULL
6. $n.id\leftarrow$hash(n)
7. 对 G 进行深度优先拓扑排序，$NS\leftarrow$顶点序列
8. FOR EACH n IN NS
9. $SS\leftarrow\varnothing$　//后继者集合
10. FOR EACH (e,s)使得顶点-边-顶点$(n,e,s)\in G$
11. 将(e,s)加入 SS
12. 将(n,SS)追加到 PS 最后

13. 将 n 从 G 中删除

14. RETURN PS

算法第 1～4 行将 RDF 图抽象为普通图. 这个过程中,将 RDF 图中的结点的类型信息作为全局的标签(label),使 RDF 图抽象为一个 Prüfer 序列化方法可用 Prüfer 序列化方法处理的一般图,对于没有类型信息的结点,将其标签信息设为 ANY_LABEL. 这一过程同时减少了图中的边和结点的数量.

对于上一步中得到的一般化图,采用扩展的 Prüfer 序列化方法构造 RDF 的 Prüfer 序列. 这个过程采用图的深度优先拓扑排序(即基于图的深度优先遍历算法的拓扑排序,参见文献[12])方法完成,其主要原因是由于删除结点并记录其后继结点的操作,本质上与图的拓扑排序是一致的;而且为了能够充分保留并利用图上结点之间的关系来过滤候选集:

算法 5～7 行,对于 RDF 图中的每一个结点,根据其结点标记信息(URI 值或 Literal 值)进行 Hash 操作,对每个结点赋以相应的数值;并将这个值作为相应结点的 Id(全局唯一);

算法 8～13 行,对于上一步中处理过的图进行深度优先拓扑排序,依次记录删除结点以及该结点的后继结点和边信息.

算法同时生成两个序列,一个是基于全局标签的 LPS(Label Prüfer Sequence),另一个是基于结点 Id 的 NPS(Number Prüfer Sequence). 在 LPS 中,每个元素都是结点的全局标签,每个 bag 都是相应的边信息和标签信息. 在 NPS 中,每个元素为结点的 Id,bag 中记录了相应的边信息和 Id 信息.

图 7 中按照相应规则生成的 Prüfer 序列如表 4 所示.

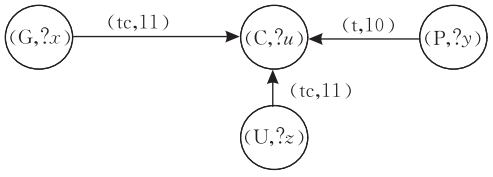

图 7 查询子图

表 4 图 6 的 Prüfer 序列

NPS	LPS
1(9:2 11:3 11:4 9:5)	G(sB:P tc:C tc:C sB:P)
2(10:3 10:7)	B(t:C t:C)
7	C
4	C
5	P
6(11:3)	G(tc:C)
8(11:3)	U(t:C)
3	C

4.3 对 RDF 图的 Prüfer 序列建立索引

为了实现查询子图的快速匹配,需要对 LPS 和 NPS 分别建立索引.

首先对于 LPS 和 NPS,分别为其中的每个元素分配一个名为 LEFTPOS 的值. 这个值记录从序列的左侧起始,该元素在序列中的位置(此处所讲的元素不包括 bag 元素,对于 bag 元素,其 LEFTPOS 值定义为与其相邻的左侧的元素的 LEFTPOS 值). 对于图 7 的 Prüfer 序列与其对应的 LEFTPOS 如表 5 所示.

表 5 带有 LEFTPOS 的 Prüfer 序列

NPS	LPS	LEFTPOS
1(9:2 11:3 11:4 9:5)	G(sB:P tc:C tc:C sB:P)	1
2(10:3 10:7)	B(t:C t:C)	2
7	C	3
4	C	4
5	P	5
6(11:3)	G(tc:C)	6
8(11:3)	U(t:C)	7
3	C	8

第 2 步,对 NPS 建立 B^+ 树索引. NPS 序列作为一棵 B^+ 树存储. 索引的键为 NPS 中元素的 LEFTPOS,其值包含 NPS 中元素的数字 Id 信息.

第 3 步,对 LPS 建立索引,LPS 的索引同样采用 B^+ 树结构实现,与 NPS 索引不同的是 LPS 索引是针对每一种全局标签分别建立一棵 B^+ 树. 以图 6 为例,需要分别对标签 G,P,C 和 U 建立相应的 B^+ 树. B^+ 树的键为相应结点的 LEFTPOS 值,值由以下信息组成:

定义 5. PARENT_LEFTPOS. 此处 PARENT 是指在 Prüfer 序列中位置在当前元素左侧且对应结点有边指向当前元素对应结点的所有元素 LEFTPOS 值中最大者.

以图 6 的 Prüfer 序列为例,NPS 中 Id 为 3 的元素 LEFTPOS 值为 8. 有边指向 Id=3 对应结点的 Id 为 1,2,6,8. 根据定义,标签为 B 的 Id=8 的元素的 LEFTPOS=7,是这 4 个结点中 LEFTPOS 值最大者. 为此,我们认为 Id=8 的元素为 Id=3 的元素的 PARENT. 并将其 LEFTPOS 值作为 Id=3 的元素的 PARENT_LEFTPOS 值. 如果一个结点入度为 0,则其对应元素无PARENT,其PARENT_LEFTPOS 值为 0.

定义 6. MAX_CHILD_LEFTPOS. 对于一个 Prüfer 序列中的元素,其 MAX_CHILD_LEFTPOS 定义为其对应结点有边指向的所有后继结点对应元素中 LEFTPOS 值最大者.

对于图 6 中，G(sB:P tc:C tc:C sB:P)而言，其所指向的结点中 LEFTPOS 值最大的结点是 Id=3 的结点，因此 G 的 MAX_CHILD_LEFTPOS 值为 Id=3 的元素的 LEFTPOS 值，即 8.

定义 7.　MINI_CHILD_LEFTP. 类似于 MAX_CHILD_LEFTPOS 定义，不同的是选取结点对应元素 LEFTPOS 最小值者.

定义 8.　MAX_SIBLING_LEFTPOS. 对于一个三元组$\langle S,P,O\rangle$，我们定义 S 是 O 的直接前驱. RDF 图中，如果一个结点与其它结点有共同的直接前驱，则这个结点与其它结点存在 SIBLING 关系. 从存在此关系的结点中选择对应序列元素具有最大 LEFTPOS 者作为该结点MAX_SIBLING_LEFTPOS.

LPS 在 Prig 的整体设计中具有非常重要的作用. 在 LPS 的索引结构中，以结点的 LEFTPOS 作为 key，以其指向的后继结点的 PARENT_LEFTPOS 等信息作为 value，因此根据以上构建的情况，不难得出其空间复杂度为 $O(m+n)$，其中 n 为图中结点的数量，m 是图中边的数量.

4.4　对于查询子图的处理

为了实现基于 Prüfer 序列匹配的查询，对于查询子图同样需要转化为 Prüfer 序列. 其基本处理流程与 RDF 图序列化的处理方法一致，但是由于查询子图自身的特性，需要解决以下两个问题：

(1) 查询子图与 RDF 全图的同构问题[11]；

(2) 查询子图标签信息的获取.

4.4.1　对于同构问题

根据 Prüfer 序列的序列化方法，图 7 中的查询子图可以生成以下 LPS：

G(tc:C) U(tc:C) P(t:C) C

我们可以直观看出图 7 是图 6 的子图，然而图 7 的 LPS 却不是图 6 的 LPS. 这是由于查询子图拓扑顺序与 RDF 图拓扑顺序不一致造成的. 用户构造查询的任意性以及变量的存在，都会导致查询子图和原 RDF 图拓扑顺序的不一致，进而导致 Prüfer 序列之间不存在子序列关系.

对于这种情况，考虑到查询子图通常较小，往往只有几个结点，因此可以通过枚举所有拓扑顺序的方法进行解决. 将子图所有拓扑顺序对应生成的 Prüfer 序列分别与原 RDF 图的序列进行匹配即可.

4.4.2　查询子图标签信息的获取

对如下 SPARQL 查询：

```
SELECT ?x, ?y, ?z
WHERE { ?x sB ?y.
        ?y t ?z.
        ?x tc ?z}
```

如果查询中不存在任何全局标签信息，那么将无法生成 LPS 序列. RDF 数据是一种含有丰富语义的数据，采用$\langle S,P,O\rangle$的结构，其中 S 与 O 间的关系是通过 P 定义的. 因此通过 P 可以部分地获取 S 与 O 的信息. 例如一组三元组：⟨a, rdf: type, GraduateStudent⟩，⟨c,rdf:type,Course⟩，⟨a,takeCourse,c⟩表达的信息为 a 是毕业生，c 是课程，a 选修 c 这门课程. 若得知 a 选修 c，则可以猜测 a 的标签信息可能是毕业生，而 c 的标签信息可能是课程. 基于这样一种事实，在 RDF 图的构造过程中建立一个由边到标签信息的索引. 对于每种边，记录其连接的术语的标签信息，并且记录标签位于边的相对位置. 以图 6 为例：sB 连接的标签有(G,P)，t 连接的标签有(P,C)，tc 连接的标签有(U,C)和(G,C). 对于查询中的两个三元组模式⟨?x,sB,?y⟩和⟨?x,tc,?z⟩，通过查找边到标签的索引中 sB 和 tc，并在第一个相对位置上做连接操作可以得到?x 的标签为 G. 依次类推，可以得到?y 的标签为 P，?z 的标签是 C.

通过以上处理可将查询子图转化为相应的 Prüfer 序列.

5　Prig 查询处理

在将 RDF 图和查询子图分别生成相应的 Prüfer 序列的基础上，Prig 的查询处理是通过在 RDF 图的 Prüfer 序列中找到查询子图 Prüfer 序列的匹配来实现的. 与针对树形结构的 Prix 方法类似，可以采用 ViST[8] 的思想，利用 B^+ 树实现一个 Trie 结构，即查询子图 Prüfer 序列中当前元素在全图序列中对应的位置是由查询子图序列中前一个元素的信息以及当前的匹配规则共同决定的. 本节将详细讲述查询匹配处理算法以及匹配规则. 算法流程如下所示.

算法 2.　查询子图 Prüfer 序列与全图序列的匹配.

输入：i：当前元素在查询序列中的位置

　　S_Q：查询序列

　　$info$：查询序列中与前一个元素匹配的信息

输出：是否匹配成功

Query(i,S_Q,$info$)

1. $LPS_Index \leftarrow S_Q[i]$的全局标签所对应的 B^+ 树索引
2. $relation \leftarrow$ cal_relation($S_Q[i-1]$, $S_Q[i]$)
3. $lb \leftarrow$ cal_left_bound($relation$,$info$)
4. $rb \leftarrow$ cal_right_bound($relation$,$info$)
5. $flag \leftarrow$ FALSE

```
6. FOR j=lb TO rb DO
7.    current_info=get_next(LPS_index, j)
8.    IF general_match(current_info) THEN
9.       IF rela_match(relation, info, current_info)
            THEN
10.      IF Query(i+1, S_Q, current_info) THEN
11.         store(S_Q[i], current_info)
12.         flag←TRUE
13. RETURN flag
```

Query 算法第 1～5 行进行初始化. 首先获取查询序列中 S_{Q_i} 处的全局标签所对应的 LPS 索引 B^+ 树. 然后通过 cal_relation() 确定 S_{Q_i} 与 $S_{Q_{i-1}}$ 的关系. 第 3 行和第 4 行根据前一步中得出的关系以及满足 $S_{Q_{i-1}}$ 条件的匹配元素信息 *info*(如 MAX_CHILD_LEFTPOS 等)共同确定 LPS_Index 上的搜索范围(*lb* 和 *rb*).

算法第 6 行开始,在搜索范围内,通过 get_next() 依次获得索引上搜索范围内的各元素. 此后,算法第 8 行首先通过函数 general_match() 进行一般性匹配过滤;对于通过过滤的结果,算法第 9 行再根据 relation、info 和 current_info 进行关系过滤 rela_match(). 对于满足关系过滤条件的结果,则递归调用 Query 函数对 $S_{Q_{i+1}}$ 匹配. 若 $S_{Q_{i+1}}$ 匹配成功,则认为 S_{Q_i} 在当前的匹配是正确的,存储当前匹配作为部分结果,并将匹配成功标志赋值为真.

算法初始调用时,i 初始值为 1,S_Q 为查询序列,*info* 初始值为 NULL.

RDF 全图与查询子图各自的扩展 Prüfer 序列以及序列之间存在着诸多性质. Prig 根据这些性质设计了能够有效过滤错误匹配结点的匹配规则,包括一般性过滤规则和关系过滤规则.

5.1 一般性过滤规则

在不考虑序列中相继元素对应结点间的拓扑结构关系的情况下,序列中元素存在如下可用于过滤的一般性性质.

定理 1. 查询结果序列中元素 S_{A_i} 的 LEFTPOS 值单调递增,其中 $1\leq i\leq |S_A|$.

证明. 采用反证法. 假设查询序列中存在两个元素 S_{A_i} 与 S_{A_j} $(1\leq i<j\leq n)$,满足关系 S_{A_i}.LEFTPOS≥S_{A_j}.LEFTPOS. 由定义 3 以及 LEFTPOS 的定义易知:RDF 图序列中不存在两个 LEFTPOS 相等的相异元素,且如果有如下关系:S_{A_i}.LEFTPOS>S_{A_j}.LEFTPOS,则 $i>j$;这与假设相矛盾,因此假设不成立. 证毕.

例 1. 以查询子图序列?x(sB:?y tc:?z)?y(t:?z)?z 为例,?x,?y 和?z 三者的 LEFTPOS 值是依次递增的. 因此在查询匹配过程中,若已知?x 匹配元素的 LEFTPOS 值,则在查找?y 时,只需从大于 LEFTPOS 值处开始查找. 此时可以过滤索引结构中所有匹配?y 标签且小于?x 的 LEFTPOS 值的元素.

定理 2. 如果 S_{G_j} 是 S_{Q_i} 的候选匹配元素,那么其对应结点的入度满足 $d_i(S_{G_j})\geq d_i(S_{Q_i})$.

证明. 采用反证法. 假设 S_{G_j} 是 S_{Q_i} 的候选匹配元素,且 $d_i(S_{G_j})<d_i(S_{Q_i})$,则 S_{Q_i} 对应结点至少有一条出边是 S_{G_j} 所没有的,这与假设中候选元素条件相矛盾,因此假设错误. 证毕.

例 2. 以图 6 产生的 Prüfer 序列为例,假设存在查询序列?x(sB:?y tc:?z)?y(t:?z)?z. 对于 LEFTPOS=4 的元素,其标签是 C,符合查询子图中?z 的基本要求,但查询子图中?z 的入度是 2,LEFTPOS 值为 3 和 4 的两个元素其入度仅为 1,因此可以将其过滤.

定理 3. 如果 S_{G_j} 是 S_{Q_i} 的候选匹配元素,那么其对应结点的出度满足 $d_o(S_{G_j})\geq d_o(S_{Q_i})$,且出边集合满足 $E_o(S_{G_j})\supseteq E_o(S_{O_i})$.

证明. 证明同性质 2.

例 3. 以图 6 为例,假设有查询子图的 Prüfer 序列:?y(t:?z),查询变量?y 的标签为 P,在全图的 Prüfer 序列中 LEFTPOS 值为 5 的结点其标签也为 P,但是其边集为空,即查询子图中?y 的边集不是该结点边集的子集. 因此可将其过滤.

以上是一般化的过滤规则,接下来的部分介绍一些关系过滤规则.

5.2 关系过滤规则

除上述一般性性质外,查序子图 Prüfer 序列中相继元素对应结点间的拓扑结构关系也存在一定性质. 查询匹配过程依据 4 种关系:无关系、父子关系、兄弟关系、共同祖先关系,进行相应的过滤.

定义 9. 无关系. 设序列 S 中相邻两元素 S_i 和 S_{i+1} $(1\leq i<|S|)$,如果满足 $d_i(S_{i+1})=0$,则称此相邻元素对应结点间无关系,记为 $N\_R(S_i,S_{i+1})$.

无关系过滤规则. 对于查询序列 S_Q 中相邻元素对应结点 S_{Q_i} 和 $S_{Q_{i+1}}$ $(1\leq i<|S_Q|)$,如果满足关系 $N\_R(S_{Q_i},S_{Q_{i+1}})$,则对于 $S_{Q_{i+1}}$ 只需要按照一般过滤规则过滤候选结点.

例 4. 图 8 Prüfer 序列为?x?y,因为?y 的入度为 0,则称?x 与?y 无关系. ?y 只需按一般规则选取候选匹配结点.

定义 10. 父子关系. 设序列 S 中相邻两元

图 8　查询子图序列中相邻元素对应结点间无关系

素 S_i 和 S_{i+1} ($1 \leq i < |S|$)，如果相应结点间满足 S_i.LEFTPOS=S_{i+1}.PARENT_LEFTPOS，则称 S_i 对应结点是 S_{i+1} 的双亲结点，S_{i+1} 对应结点是 S_i 的孩子结点，这种关系称为父子关系，记为 $P\_R(S_i, S_{i+1})$.

父子关系过滤规则. 对于查询序列 S_Q 中相邻元素对应结点 S_{Q_i} 和 $S_{Q_{i+1}}$ ($1 \leq i < |S_Q|$)，如果满足关系 $P\_R(S_{Q_i}, S_{Q_{i+1}})$，则在 $S_{Q_{i+1}}$ 其标签所属 LPS 索引上的查找范围为 S_{Q_i}.MINI_CHILD_LEFTPOS 和 S_{Q_i}.MAX_CHILD_LEFTPOS 之间，在此范围内同时不满足以下条件的 S_{G_j} 都将被过滤掉(其中 S_{A_i} 为查询结果序列的第 i 个元素)：

(1) $P\_R(S_{A_i}, S_{G_j})$；

(2) S_{A_i}.LEFTPOS<S_{G_j}.PARENT_LEFTPOS.

例 5. 图 9 的 Prüfer 序列为 ?x(?y)，?y，根据 PARENT_LEFTPOS 定义，若 ?y.PARENT_LEFTPOS=?x.LEFTPOS，则 ?y 是 ?x 的孩子. 因此在标签 C 索引中，可将查找范围定为 ?x.MINI_CHILD_LEFTPOS 到 ?x.MAX_CHILD_LEFTPOS 之间. 在此范围内首先按照一般规则过滤候选结点，这样可以有效减小搜索空间. 然后，将 ?y 的候选结点与 ?x 的候选结点做父子关系判断，如果满足父子关系则纳入候选结果集中(对应本规则中的条件 1). 但对于以上匹配可能会漏掉一类正确的候选结果. 如图 10 所示，生成的 Prüfer 序列可能为 G(tc:c) P(t:B) C，此时虽然 C 的双亲之一是 G，但其 PARENT_LEFTPOS 却是 P 的 LEFTPOS 值，为此我们采取如下策略(对应本规则中的条件 2)涵盖这一情况：若 ?y 的候选结点的 PARENT_LEFTPOS 大于 ?x 的候选结点的 LEFTPOS 值，则亦将其纳入候选结果集中. 其余的结果都将被过滤掉.

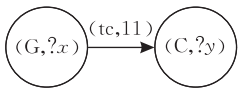

图 9　查询子图中相邻结点父子关系

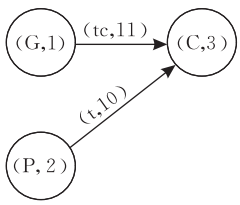

图 10　父子关系时漏配结点的情况

定义 11.　兄弟关系. 设序列 S 中相邻两元素 S_i 和 S_{i+1} ($1 \leq i < |S|$)，如果相应结点间满足 S_i.PARENT_LEFTPOS=S_{i+1}.PARENT_LEFTPOS，则称 S_i 和 S_{i+1} 为兄弟关系，记为 $S\_R(S_i, S_{i+1})$.

兄弟关系过滤规则. 对于查询序列 S_Q 中相邻元素对应结点 S_{Q_i} 和 $S_{Q_{i+1}}$ ($1 \leq i < |S_Q|$)，如果满足关系 $S\_R(S_{Q_i}, S_{Q_{i+1}})$，则在 $S_{Q_{i+1}}$ 其标签所属 LPS 索引上的查找范围应该在：(S_{A_i}.LEFTPOS, S_{A_i}.MAX_SIBLING_LEFTPOS]，在此范围内同时不满足以下条件的 S_{G_j} 都将被过滤掉(其中 S_{A_i} 为查询结果序列的第 i 个元素)：

(1) $S\_R(S_{A_i}, S_{G_j})$.

(2) S_{A_i}.PARENT_LEFTPOS<S_{G_j}.PARENT_LEFTPOS.

例 6. 如图 11 对于 ?y 和 ?z，若不满足无关系和父子关系，则进行兄弟关系的判断. 判断为兄弟关系的条件是 ?y.PARENT_LEFTPOS=?z.PARENT_LEFTPOS. 如果满足该关系，则对于 ?z，根据 LEFTPOS 的单调递增性质，在标签 C 索引中只需搜索 LEFTPOS 值介于 ?y.LEFTPOS 和 ?y.MAX_SIBLING_LEFTPOS 之间的结点，这有效缩小了搜索空间. 将 ?y 的候选结果与之前 ?x 的候选结果做兄弟关系判断，如果满足兄弟关系，则将其纳入候选结果集中(对应本规则中的条件 1). 同样，对于以上匹配可能会漏掉一类正确的候选结果. 如图 12 所示：其生成的 Prüfer 序列可能为 G(sB:P tc:C)P U(tc:C)C. 此时 P 与 C 虽然有共同的双亲 G，但是 C 的 PARENT_LEFTPOS 与 P 的 PARENT_LEFTPOS 不同. 对此我们采取如下策略(对应本规则中的条件 2)涵盖这一情况：若 ?y 的候选结点的 PARENT_LEFTPOS 小于 ?z 候选结点的 PARENT_LEFTPOS，则亦将其纳入候选结果集中，其余的结果都将被过滤掉.

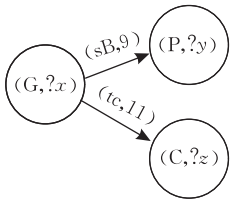

图 11　查询子图中相邻节点为兄弟关系

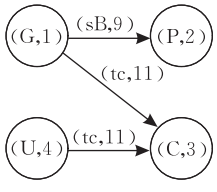

图 12　查询子图兄弟关系漏配

定义 12. 共同祖先关系. 设序列 S 中相邻两元素 S_i 和 S_{i+1}($1 \leq i < |S|$),如果相应结点间不存在无关系、父子关系和兄弟关系,且满足 S_i.PARENT_LEFTPOS $< S_{i+1}$.PARENT_LEFTPOS,则称二者具有共同祖先关系,记为 $C\_R(S_i, S_{i+1})$.

共同祖先关系过滤规则. 对于查询序列 S_Q 中相邻元素对应结点 S_{Q_i} 和 $S_{Q_{i+1}}$($1 \leq i < |S_Q|$),如果满足 $C\_R(S_{Q_i}, S_{Q_{i+1}})$,则在 $S_{Q_{i+1}}$ 其标签所属 LPS 索引上查找范围是索引中所有满足如下关系的元素:S_{G_j}.LEFTPOS $> S_{A_i}$.LEFTPOS,在此范围内的 S_{G_j} 都可作为候选 $S_{A_{i+1}}$,否则予以过滤.

例 7. 如图 13 中所示的查询子图的 Prüfer 序列为 $?x(\text{sB}:?y, \text{tc}:?u)?y(\text{t}:z)?z?u$ 时,则序列中 $?u$ 与其左侧相邻元素对应结点既非兄弟关系,也非父子关系,而是与 $?z$ 为共同祖先关系,其判断条件 $?z$.PARENT_LEFTPOS $>$ $?u$.PARENT_LEFTPOS. 此时,我们可以按照共同祖先关系过滤规则进行候选结点的选取和过滤.

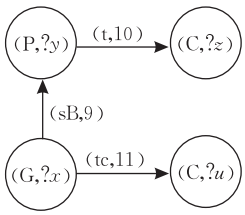

图 13 查询子图中共同祖先关系

5.3 最终结果过滤

最后,对于生成的候选结果集采用传统的图匹配算法进行相应的剔除工作,去除错误的结果,返回正确的结果集.

6 实验结果

实验所使用的硬件为 Intel(R) Pentium(R) 4 CPU 2.80 GHz,内存 DDR 2 GB,磁盘 320 GB,转速 7200 rpm. 软件环境中操作系统为 Linux CentOS Release 5.6,采用 C++作为编程语言,开发环境为 NetBeans IDE 6.9.1. 系统采用 Raptor① 作为 RDF 数据解析工具,使用 Berkeley DB② 作为后台数据存储工具.

实验中以采用 Native 存储方式的 Sesame 2.2.4 作为对比系统,分别采用 LUBM 基准数据集和 SP²Bench 基准数据集进行了对比实验.

对于 LUBM 数据集中 14 个标准查询分别在 LUBM(10),LUBM(100),LUBM(1000)上进行了测试,实验结果如图 14 所示. 测试结果显示在绝大多数情况下 Prig 在该数据集上的查询响应时间低于 Sesame,尤其是在数据规模较大(LUBM(1000))的情况下. 但当数据规模较小时(LUBM(10),LUBM(100)), Prig 在 Q3,Q4,Q5,Q6,Q9,Q10,Q12,Q13 等查询中表现出性能稍落后于 Sesame. 其主要由以下两方面原因造成:

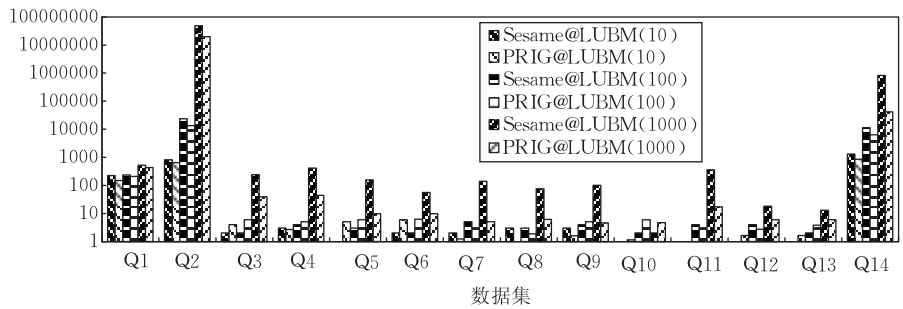

图 14 Sesame 与 Prig 在 LUBM 数据集的查询测试实验结果

首先,Prig 与 Sesame 的查询处理方法不同. Sesame 以主要通过垂直表(vertical table)的形式组织三元组,三元组中的主语、谓语、宾语分别对应元组中的一项. 通过调整主语、谓语、宾语上建立索引的顺序可以提高三元组访问的性能. 面对复杂查询 Sesame 要在垂直表上进行自连接操作. 而 Prig 则在获取查询中变量的候选结点过程中,使用一般性过滤和关系过滤等多种过滤规则缩小候选集合,从而提高查询的处理效率. 在数据规模较小的情况下,Prig 过滤过程所需的时间开销相对于缩小候选结果集所带来的性能提高要更突出,因此表现出 Prig 的查询响应时间高于 Sesame. 随着数据规模的增大,Prig 缩小候选结果集所带来的查询性能改善随之显现. 而此时, Sesame 则由于缺少有效的过滤方法,面临大量的连接操作.

另一方面,以查询 Q4 为例,由于存在图同构问题,查询子图生成的Prüfer序列并不唯一. 在Prig

① http://librdf.org/raptor/
② http://www.oracle.com/technetwork/database/berkeley-db/overview/index.html

系统中采取枚举所有可能的查询子序列的方法，并依次进行序列匹配查询，进而可能造成查询响应时间超过 Sesame 的现象. 但由于 Prig 所具有的过滤功能，使得查询性能在数据规模增大的情况下优于 Sesame.

总体而言，随着 LUBM 数据规模不断增大，Prig 相对于 Sesame 有着更好的查询性能. 如图 15 所示，我们统计了在 LUBM(10)，LUBM(100)以及 LUBM(1000) 3 个数据集下，Sesame 和 Prig 两个系统分别执行 14 个查询的时间总和. 结果显示，相较于 Sesame，Prig 分别依次提高了 28.9%、43.3% 和 58.9%. 因此在数据规模增长的趋势下，Prig 有着较好的可伸缩性.

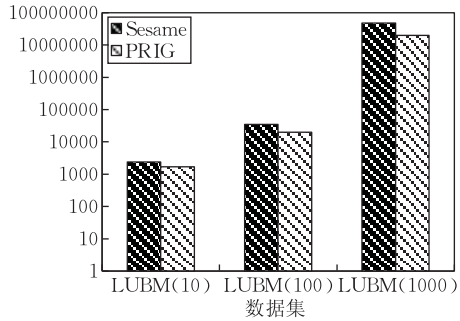

图 15　Sesame 与 Prig 在 LUBM 数据集的查询时间总和比较

与此同时，我们使用 SP² Bench 数据集，对测试基准中 7 个标准查询的简单查询部分[①]分别在 SP² Bench(2.5M)，SP² Bench(10M)，SP² Bench(40M) 上进行了相关的测试，其测试结果如图 16 所示.

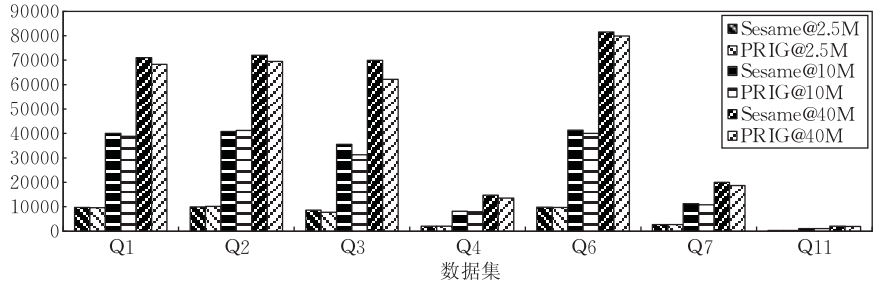

图 16　简单图模式下 Sesame 与 Prig 在 SP2Bench 数据集查询性能比较

在 SP² Bench 数据集中 Prig 的整体性能同比仍优于 Sesame. 但也存在与 LUBM 数据集中类似的情况，如图 17 所示，在数据规模较小的情况下，SP² Bench(2.5M) 和 SP² Bench(10M) 时 Q2，Q4，Q7，Q11 相对于 Sesame 在查询性能上存在着一定的差距，但这一问题随着数据规模的增长得到改善. 总体而言，Prig 在 SP² Bench 数据集上仍然有着比较好的伸缩性，性能依次提高了 1.19%，3.89%，5.21%.

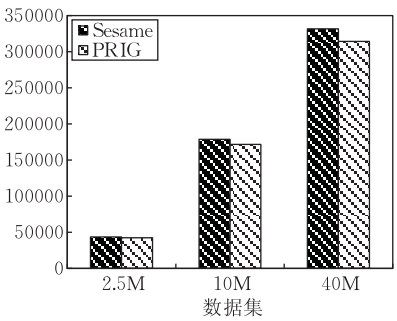

图 17　Sesame 与 Prig 在 SP2 Bench 的查询时间总和比较

对比图 15 与图 17，Prig 在 SP² Bench 数据集中所表现出的性能相较于其在 LUBM 数据集中所体现的性能有所下降，主要有以下两方面原因：

(1) SP² Bench 数据集中 RDF 图结构要比 LUBM 数据集的更复杂，因此在序列中搜索和过滤候选结点的开销更大，造成了相对较高的时间开销.

(2) SP² Bench 测试基准中查询子图所对应的 Prüfer 序列大多不唯一. Prig 不得不枚举所有可能的序列，然后依次进行查询. 这也造成了很大的时间开销.

通过以上在 LUBM 和 SP² Bench 测试基准上的实验结果显示 Prig 在整体上优于Sesame，并且在数据规模增长的趋势下，有着较好的可伸缩性. 这表明 Prig 所采用的扩展 Prüfer 序列化方法以及提出的若干过滤规则对 RDF 数据的索引和查询具有较好效果.

7　结束语

本文首次将 Prüfer 序列化方法引入 RDF 图的查询和处理工作中. 扩展了 Prüfer 从一般树形结构查询匹配到图结构查询匹配的转变，提出了若干有效的过滤规则来提高查询处理性能. 实现了基于这一方法的 Prig 系统. 在两个测试基准上的实验结果表明 Prig 能够提升 RDF 数据索引和管理性能.

① 由于目前 Prig 原型系统仅支持简单查询，因此选取了查询中可以提取简单查询的 7 个标准查询进行相关的测试.

参 考 文 献

[1] Berners-Lee T, Fischetti M, Dertouzos M L. Weaving the Web: The Original Design and Ultimate Destiny of the World Wide Web. Harper, San Francisco, 1999

[2] Rao P, Moon B. PRIX: Indexing and querying XML using prfer sequences//Proceedings of the IEEE 23rd International Conference on Data Engineering (ICDE2007). Boston, MA, United States, 2004: 288-299

[3] Gao Zhi-Qiang, Pan Yue, Ma Li et al. Principle and Application of the Semantic Web. Beijing: China Machine Press, 2009(in Chinese)
(高志强, 潘越, 马力等. 语义 Web 原理及应用. 北京: 机械工业出版社, 2009

[4] Broekstra J, Kampman A, van Harmelen F. Sesame: An architecture for storing and querying RDF data and schema information//Proceedings of the Spinning the Semantic Web. Cambridge, MA: MIT Press, 2003: 197-222

[5] Wilkinson K, Sayers C, Kuno H A, Reynolds D. Efficient RDF storage and retrieval in jena2//Proceedings of the SWDB'03, The first International Workshop on Semantic Web and Databases, Co-Located with VLDB 2003, 2003: 131-150

[6] Harris S. SPARQL query processing with conventional relational database systems//Proceedings of the International Workshop on Scalable Semantic Web Knowledge Base Systems. New York, 2005: 235-244

[7] Bjelogrlic Z, van Gulik D W, Reggiori A. Indexing and retrieving semantic Web resources: The RDFStore model. http://www.w3.org/2001/sw/Europe/events/20031113-storage/positions/asemantics.pdf

[8] Wang H, Park S, Fan W, Yu P S. ViST: A dynamic index method for querying XML data by tree structures//Proceedings of the ACM SIGMOD. San Diego, CA, United States, 2003: 110-121

[9] Prasad K Hima, Kumar P Sreenivasa. Efficient indexing and querying of XML data using modified prüfer sequences//Proceedings of the ACM CIKM. Bremen, Germany, 2005: 397-404

[10] Algergawy A, Schallehn E, Saake G. A sequence-based ontology matching approach//Proceedings of the 18th European Conference on Artificial Intelligence Workshops. Patras, Greece, 2008: 26-30

[11] Aggarwal C C, Wang H. Managing and Mining Graph Data. NY, USA: Springer, 2010

[12] Cormen Thomas H, Leiserson Charles E, Rivest Ronald L, Stein Clifford. Introduction to Algorithms. 2nd Edition. Cambridge, Massachusetts, USA: The MIT Press, 2001

LIU Xiang-Yu, born in 1986, M. S. candidate. His research interests focus on semantic Web data management.

WU Gang, born in 1978, Ph. D., associate professor. His research interests include semantic Web, distributed computing, and uncertain data management.

Background

As the next generation of Web, the semantic Web has received considerable attentions from the industrial and academic communities. The Resource Description Framework (RDF) is the basic data model of representing resources and their relations on the semantic Web. An RDF model is a labeled, directed graph. Although the model is flexible, it increases the complexity of storing and querying RDF data. The situation becomes even worse when dealing with large scale RDF data like Linking Open Data (LOD). Developing effective index structures is one of the solutions to improve the performance of RDF data management. As we know, sequence-based index structures are space efficient and widely supported by existing sequence matching algorithms. Hence, they usually have better query performance. Considering that Prüfer sequence has been successfully applied in indexing and querying the tree structured data but the graph structured data, we propose an extended Prüfer sequence approach, called Prig, to support indexing and querying RDF graph data. The approach first parses RDF documents into RDF graphs. Extended Prüfer sequences are then generated from the RDF graphs and indexed. Queries are converted into query graphs and hence to extended Prüfer sequences similarly. The sequence matching between sequences generated from RDF graphs and query graphs is performed to achieve the answer. We also provide several effective filtering schemes to further improve the matching efficiency. To the best of our knowledge, Prig is the first RDF graph indexing and querying approach based on Prüfer sequences. The evaluation results on the LUBM benchmark and the SP^2 Bench benchmark show that Prig has a better performance on large-scale RDF data comparing with the well-known Sesame RDF framework.

This research is supported by the National Natural Science Foundation of China (NSFC) under grants No. 60903010, No. 61025007 and No. 60933001, the National Basic Research Program of China under grant No. 2011CB302206, the Natural Science Foundation of Jiangsu Province under grant No. BK2009268, and the Key Laboratory of Advanced Information Science and Network Technology of Beijing under grant No. XDXX1011.