软件学报 ISSN 1000-9825, CODEN RUXUEW
Journal of Software,2019,30(3):498−514 [doi: 10.13328/j.cnki.jos.005696]

E-mail: jos@iscas.ac.cn
http://www.jos.org.cn
Tel: +86-10-62562563

大规模 RDF 图数据上高效率分布式查询处理*

王 鑫[1,2], 徐 强[1,2], 柴乐乐[1,2], 杨雅君[1,2,4], 柴云鹏[3]

[1](天津大学 智能与计算学部,天津 300354)

[2](天津市认知计算与应用重点实验室,天津 300354)

[3](中国人民大学 信息学院,北京 100872)

[4](数字出版技术国家重点实验室,北京 100871)

通讯作者: 杨雅君, E-mail: yjyang@tju.edu.cn

摘 要: 知识图谱是智能数据的主要表现形式,随着知识图谱领域的不断发展,大量的智能图数据以资源描述框架(resource description framework,简称 RDF)形式发布出来.RDF 图上的 SPARQL 查询语义对应于图同态,是一个 NP-完全问题.因此,如何使用分布式方法在大规模 RDF 图上有效回答 SPARQL 查询是一个富有挑战性的问题.目前已有研究使用 MapReduce 计算模型处理大规模 RDF 数据,但其将 SPARQL 查询拆分成单个的查询子句,没有考虑 RDF 数据的丰富语义和自身的图特性,导致 MapReduce 迭代次数过多.首先,利用 RDF 数据内嵌的语义和结构信息作为启发式信息,将查询图分解为星形的集合,可以在更少次迭代内得到查询结果.同时,分解算法给出中间结果较少的星形匹配顺序,基于此顺序,每轮 MapReduce 操作通过连接操作匹配一个新的星形,直至产生最终的答案.最后,在标准合成数据集 WatDiv 和真实数据集 DBpedia 上进行大量的实验评估.实验结果表明:所提基于星形分解的分布式 SPARQL BGP 匹配算法能够高效回答查询,查询时间比 SHARD 和 S2X 算法的查询时间平均提高一个数量级,且优化算法的查询时间与基本算法相比缩短了 49.63%~78.71%.

关键词: 星形分解;分布式;基本图模式匹配;大规模 RDF 图;MapReduce

中图法分类号: TP311

中文引用格式: 王鑫,徐强,柴乐乐,杨雅君,柴云鹏.大规模 RDF 图数据上高效率分布式查询处理.软件学报,2019,30(3):498−514. http://www.jos.org.cn/1000-9825/5696.htm

英文引用格式: Wang X, Xu Q, Chai LL, Yang YJ, Chai YP. Efficient distributed query processing on large scale RDF graph data. Ruan Jian Xue Bao/Journal of Software, 2019,30(3):498−514 (in Chinese). http://www.jos.org.cn/1000-9825/5696.htm

Efficient Distributed Query Processing on Large Scale RDF Graph Data

WANG Xin[1,2], XU Qiang[1,2], CHAI Le-Le[1,2], YANG Ya-Jun[1,2,4], CHAI Yun-Peng[3]

[1](College of Intelligence and Computing, Tianjin University, Tianjin 300354, China)

[2](Tianjin Key Laboratory of Cognitive Computing and Application, Tianjin 300354, China)

[3](School of Information, Renmin University of China, Beijing 100872, China)

[4](State Key Laboratory of Digital Publishing Technology, Beijing 100871, China)

* 基金项目: 国家自然科学基金(61572353, 61402323, 61472427); 天津市自然科学基金(17JCYBJC15400); 数字出版技术国家重点实验室开放课题; 北京自然科学基金(4172031)

Foundation item: National Natural Science Foundation of China (61572353, 61402323, 61472427); Natural Science Foundation of Tianjin (17JCYBJC15400); Opening Project of State Key Laboratory of Digital Publishing Technology; Natural Science Foundation of Beijing (4172031)

本文由智能数据管理与分析技术专刊特约编辑樊文飞教授、王国仁教授、王朝坤副教授推荐.

收稿时间: 2018-07-20; 修改时间: 2018-09-20; 采用时间: 2018-11-01

Abstract: Knowledge graphs are the main representation form of intelligent data. With the development of knowledge graphs, more and more intelligent data has been released in the form of the resource description framework (RDF). It is known that the semantics of SPARQL correspond to graph homomorphism which is an NP-complete problem. Therefore, how to efficiently answer SPARQL queries in parallel over big RDF graphs has been widely recognized as a challenging problem. There are some research works using the MapReduce computational model to process big RDF graph. However, SPARQL queries in these works are decomposed into the set of query clauses without considering any semantics and graph structure embedded in RDF graph, which leads to overmuch MapReduce iterations. This study first decomposes the SPARQL query graph into a set of stars by utilizing the semantic and structural information embedded RDF graphs as heuristics, which can be matched in fewer MapReduce iterations. Meanwhile, a matching order of these stars is given to reduce intermediate results in MapReduce iterations. During the matching phase, each round of MapReduce adds one star with the join operation. The extensive experiments on both synthetic dataset WatDiv, and real-world dataset DBpedia are carried out. The experiments results demonstrate that the proposed star decomposition-based method can answer SPARQL BGP queries efficiently, which outperforms SHARD and S2X by one order of magnitude. Finally, extensive experiments show that the performance of the optimization algorithms is improved by 49.63% and 78.71% than the basic algorithm over both synthetic and real datasets.

Key words: star decomposition; distributed; basic graph pattern matching; large scale RDF graphs; MapReduce

智能数据管理是人工智能发展的重要基石.知识图谱是智能数据的主要表现形式.资源描述框架(resource description framework,简称 RDF)是由国际万维网联盟(W3C)制定的表示知识图谱的推荐标准.随着知识图谱相关技术的不断发展,RDF 三元组数据日益激增,并且被广泛地应用在多个领域,包括科学、生物信息、商业智能和社交网络等[1].在现实世界中,RDF 数据集往往达到数亿条三元组数据.目前,如何有效管理大规模 RDF 智能图数据受到越来越多的关注.与传统的关系型数据库中进行的查询不同,RDF 数据上 SPARQL 的基本图模式(BGP)操作语义等价于数学上的子图同态[2],这是一个著名的 NP 完全问题[3],即 SPARQL 查询可以转化为子图匹配问题.因此,提高 SPARQL 查询性能已成为亟待解决的技术问题.

RDF 数据模型是一种特殊的语义图数据模型,表示为实体-关系或者类关系图,基本构件包括实体、属性和值,通过属性和值描述资源之间的关系.每条 RDF 三元组数据的格式为(s,p,o),是一个陈述,其中,s 是主语,p 是谓语,o 是宾语.(s,p,o)表示资源 s 与资源 o 之间具有联系 p,或表示资源 s 具有属性 p 且其取值为 o.RDF 数据的主语和谓语都是以统一资源标识符(uniform resource identifier,简称 URI)来标识,宾语以 URI 或者字面值(literal)标识.一条三元组数据可以被看做一条有向边,其中,主语和宾语为顶点.一个三元组的宾语可作为另一个三元组中的主语,于是,RDF 数据形成一种有向标签图.

RDF 智能图数据上的查询语言随着 RDF 数据的发展不断发展,早期查询语言包括 RQL,RDQL,SquishQLD 等[4].目前被广泛使用的是 SPARQL(simple protocol and RDF query language),该查询语言是 W3C 提出的 RDF 图上的标准查询语言.基本图模式(basic graph pattern,简称 BGP)是 SPARQL 中最常用、最基本的查询模式.实际上,每个 SPARQL 查询的核心部分是一个 BGP 查询.例如:在维基百科数据集 DBpedia 中查询内陆国家及其人口数,SPARQL BGP 查询如图 1 所示.

```
PREFIX yago:⟨http://dbpediaorg/class/yago/⟩
PREFIX dbp:⟨http://dbpediaorg/property/⟩
SELECT ?country ?population
WHERE{
  ?country a yago:LandlockedCountries.
  ?country dbp:populationEstimate ?population.}
```

Fig.1 An example SPARQL BGP query Q

图 1 示例 SPARQL BGP 查询 Q

目前,处理大规模 RDF 智能图数据查询的研究工作分为两大类——单机和分布式.单机 SPARQL 查询引擎,如 RDF-3X[5]和 gStore[6],采用基于空间换时间策略构建不同的索引方案,利用索引减少查询匹配的搜索空间;分布式 SPARQL 查询处理系统以是否采用分解查询图策略分为两类:一类方法将 SPARQL 查询图分解为三元组模式(triple pattern,至少包含一个变量的三元组)[7−11]或者带有局部结构信息子图的集合,先对分解后的查询子图进行匹配,之后对子图的匹配结果进行连接操作获得最终匹配结果;另一类方法直接处理完整查询图[12−14],大

部分通过图探索减少连接操作的较高代价,通过优化探索计划提高查询性能.

本文将 RDF 图内嵌的语义和结构作为启发式信息(*h*-值),并利用 *h*-值将 RDF 查询图分解为星形结构(高度为 1 的树)的集合进行匹配.Lai 等人[15]指出:无向无标签图上基于星形-连接的算法[16]在评估包含多边的星形时会生成大量中间匹配结果,如:将包含 3 条边的星形结构匹配到最大度为 1 000 的数据图,产生 10^9 个匹配结果.为此,他们提出了基于 MapReduce[17]的 TwinTwigJoin 算法,将查询图分解成 TwinTwig 结构,该结构只包含一条边或两条相连的边.造成上述问题的根本原因是:顶点和边缺乏可区别信息时,星型结构匹配过程可能会产生大量的中间结果.但是 RDF 图中的顶点标签是独一无二的 URIs,同时,边是有方向且带标签的,因此,Lai 等人在文献[15]中提出的问题在 RDF 图中不存在.与分解成 TwinTwig 结构[15]相比,星形分解策略可以保持更多原始查询图的信息,同时,在 MapReduce 计算模型下可以在更少次 MapReduce 迭代操作内结束查询.

本文的主要贡献如下:

(1) 提出了基于 MapReduce 计算模型的有效 SPARQL 基本图模式匹配算法,利用 RDF 智能图数据丰富的语义和自身的图特性设计星形分解策略,并给出中间结果较少的星形匹配顺序来提高查询效率,每轮 MapReduce 操作按序连接一个星形结构直至匹配结束;

(2) 为进一步提高算法的性能,引入两种优化技术:一个通过基于布隆过滤器的邻居信息编码技术过滤 MapReduce 操作中的无用数据输入,另一个技术推迟笛卡尔积操作以提高查询效率;

(3) 在主流大数据处理框架 Spark 下实现了上述算法,并在标准合成数据集 WatDiv[18]和真实数据集 DBpedia 上进行了大量实验,验证了算法的有效性、高效性和可伸缩性,并对查询匹配的时间开销与数据集规模及集群中节点数量的关系进行分析.

本文第 1 节介绍相关工作.第 2 节介绍 RDF 图和 SPARQL 基本图模式查询的预备知识.第 3 节描述如何存储 RDF 数据,分解 SPARQL 查询,并在分布式计算框架 MapReduce 下匹配查询.第 4 节提出两种优化技术:一个通过基于布隆过滤器的邻居信息编码过滤掉 MapReduce 迭代中无用的数据输入;另一优化技术将星形匹配过程中的笛卡尔积推迟到 MapReduce 迭代结束来减少该过程中大量无用的笛卡尔积.第 5 节在标准合成数据集和真实数据集上进行实验,并在第 6 节对全文总结.

1 相关工作

近年来,研究者逐步意识到图数据管理的重要性,高效处理 RDF 数据已成为该方向的研究热点之一.本节将对目前已有的大规模 RDF 数据上 SPARQL 查询研究工作进行介绍,主要分为以下两类.

1.1 单机系统上的RDF图查询处理

RDF-3X 查询引擎[5]采用原生态的三元组存储模式,将 RDF 数据和按照 SPO 排列的 15 种 SPO(包含 SPO 中 1,2 或 3 个)索引压缩存储在 B^+-树中,以冗余存储提高查询性能.该方法通过自底向上的动态规划算法选择出代价最优的连接顺序.另外,RDF-3X 收集单个实体的统计信息,大量使用合并连接来换取较高查询效率.gStore[6]是基于图的单机 SPARQL 查询引擎,使用邻接链表存储 RDF 图,并将顶点类和实体用固定长度的比特串编码压缩存储,通过建立 VS*-树索引结构提高查询处理效率并采用剪枝策略减少搜索空间.

随着 Tim Berners-Lee 发起的 Linked Data 运动的持续推进,各个领域发布的 RDF 数据一直持续增长,数据规模达到百亿级.单机已经无法有效处理现有规模 RDF 数据上的 SPARQL 查询[19],分布式/并行技术已成为 RDF 数据管理不可或缺的工具.

1.2 分布式系统上的RDF图查询处理

近年来,大量研究者和实践者提出了若干种方法分布式处理大规模 RDF 图上的 SPARQL 查询问题,分两类叙述.

(1) 分解查询图.

SHARD[7]基于三元组存储处理 RDF 数据集上的 SPARQL 查询,查询图被分解为三元组模式(包含变量的三

元组,即一个查询子句)的集合,通过在三元组模式上迭代将变量绑定到数据图的顶点,该绑定必须同时满足查询中的所有约束.每一轮 MapRedue 操作通过连接操作只添加一个查询子句.类似地,HadoopRDF[8]中查询图的最小分解单位也是三元组模式,并且它也使用 MapReduce 计算框架将 RDF 三元组基于谓语划分到多个小文件中.SPARQL 查询中的一个子句,也就是一个三元组模式,只能同时参与到一个 Hadoop 作业中.这两种方法都没有使用查询图的结构信息,需要过多的 MapReduce 迭代操作,并且大量的连接操作导致很高的查询代价.而本文利用部分图的结构特性,在每次 MapReduce 迭代中添加一个星形,可以在更少的迭代次数内完成查询匹配,从而提高查询效率.同样地,S2X[9]虽然建立在分布式图计算框架 GraphX[20]上实现 SPARQL 的 BGP 匹配,查询图也被分解成查询子句,也就是三元组模式.首先,BGP 查询中所有的三元组模式被匹配;其次,通过迭代计算逐渐舍弃部分中间结果;最后,将剩余的匹配结果进行连接操作,这一阶段可能会导致大量的中间结果.另外,Virtuoso[10]和 S2RDF[11]基于关系数据库处理 RDF 图查询,将 SPARQL 查询子句转化为 SQL 语句,通过关系表的连接操作得到匹配结果.S2RDF[11]引入关系划分模型 ExtVP 存储 RDF 数据,该方法基于 Spark[21]并行计算框架,可以有效最小化查询输入大小.但是 ExtVP 存储方案的预处理需要提前在垂直划分表上作大量的连接操作,代价非常高.本文将 RDF 图分布式存储在多个邻接链表中,可以同时在各个主语顶点的邻接链表上并行进行 MapReduce 计算,加快查询处理.

(2) 完整查询图.

Trinity.RDF[12]将 RDF 数据模型化为原始图的形式,并使用键-值存储 RDF 数据,将顶点标识符作为键,顶点的邻接链表作为值.采用基于代价的方法找到最优探索计划,利用图探索而不是连接操作减少中间结果量,但是最终结果需要在集群中心节点上使用单线程获得.此外,Peng 等人[13]采用局部计算和装配的分布式框架基于 gStore 执行 SPARQL 查询.在局部计算阶段,集群中每个节点匹配完整查询,然后在装配阶段,大量的本地局部匹配结果被发送到中心节点进行连接操作获得最终的结果.当匹配过程的中间结果规模较大时,这两种方法在最后阶段可能是一个性能瓶颈,而本文可以通过 Reduce 函数并行连接星形匹配结果得到答案集.TriAD[14]使用 MPI 协议处理 SPARQL 查询,建立 6 个 SPO 索引,并将 RDF 三元组划分到这些索引中,同时,使用基于本地的概括图加速查询.同样,采用自底向上的动态规划算法确定最小代价估计的三元组模式匹配顺序.但是与本文方法相比,TriAD 中大量的索引导致数据冗余,空间消耗大,如果数据变动需更新大量索引.

不同于以上方法,本文将 SPARQL BGP 查询图分解为星形的集合,利用 RDF 数据丰富语义和图的结构确定中间结果较少的星形匹配顺序.在 MapReduce 并行计算中,Map 函数在分布式邻接链表上并行匹配星形,Reduce 函数并行连接星形与已经生成的查询子图的匹配结果,加快查询速度.

2 预备知识

本节将详细介绍相关背景知识,包括 RDF 图、SPARQL BGP 查询图、BGP 匹配及星形分解等定义,为理解本文后续算法奠定基础.

2.1 RDF图及SPARQL查询

定义 1(RDF 图). 设 U 和 L 分别是统一资源标识符和字面值的有限集合,元组 $(s,p,o)\in U\times U\times(U\cup L)$ 叫做 RDF 三元组,每个三元组 $t=(s,p,o)$ 表示资源 s 和资源 o 有关系 p,或者资源 s 有值为 o 的属性 p,其中,s,p 和 o 分别表示主语、谓语和宾语.一个有限三元组的集合被称为 RDF 图.用 V,E,Σ 分别表示 RDF 图 G 的顶点、边和边标签的集合,正式定义为:$V=\{s|(s,p,o)\in G\}\cup\{o|(s,p,o)\in G\}$,$E\subseteq V\times V$,并且 $\Sigma=\{p|(s,p,o)\in G\}$.函数 $lab:E\to\Sigma$ 返回图 G 中边的标签.

本文定义的 RDF 图不考虑 RDF 标准定义中的空节点(blank node),实际上,本文所研究的问题与空节点的包含是正交的,在 RDF 图定义中省略空节点是为了使表述更简洁.图 2 是从 DBpedia 数据集中摘取的 RDF 图示例 G,为节省空间,对 URI 进行了压缩表示.RDF 数据中的每条三元组可以被看做是 RDF 图中一条有向带标签的边.

定义 2(查询图). 用 Var 表示与 U 和 L 不相交的变量的无限集合,Var 中每个元素的名字按照惯例以字符“?”开始(如 $?x\in Var$).RDF 图 G 上的 SPARQL 基本图模式(basic graph pattern,简称 BGP)查询 Q 被定义为如下形

式:$Q=\{(s_i,p_i,o_i)|(s_i,p_i,o_i)\in(U\cup Var)\times(U\cup Var)\times(U\cup L\cup Var)\}$,其中,$tp_i=(s_i,p_i,o_i)$是三元组模式.SPARQL BGP 查询 Q 也被称为查询图 G_Q.

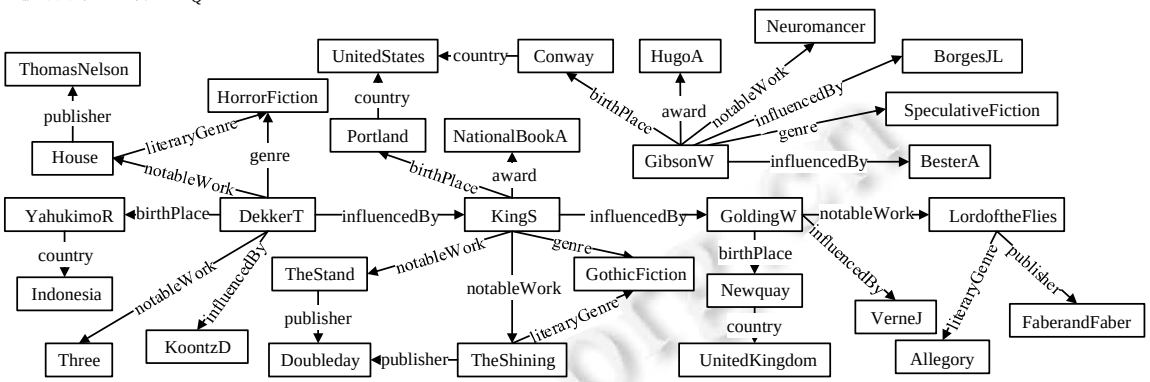

Fig.2 An example RDF graph G

图 2 RDF 示例图 G

用 $V(Q)$,$E(Q)$分别表示查询图 G_Q 中顶点和边的集合,对每一个顶点 $u\in V(Q)$,如果 $u\in Var$,那么 u 可以匹配 RDF 图 G 中的任意顶点 $v\in V$;否则,u 只可以匹配与它标签相同的顶点 $v\in V$.

定义 3(SPARQL BGP 匹配). RDF 图 G 上的 SPARQL BGP 查询 Q 的语义定义为:

(1) μ是从 $V(Q)$中的顶点到 V 中顶点的映射;

(2) $(G,\mu)\models Q$ 当且仅当对任意$(s_i,p_i,o_i)\in Q$ 满足:i) s_i 和 o_i 可以分别匹配$\mu(s_i)$和$\mu(o_i)$;ii) $(\mu(s_i),\mu(o_i))\in E$;iii) $p_i\in Var$ 或 $lab((\mu(s_i),\mu(o_i)))$与 p_i 相同;

(3) $\Omega(Q)$是使$(G,\mu)\models Q$ 满足的μ的集合,也就是 RDF 图 G 上 SPARQL BGP 查询 G_Q 的答案集.

定义 4(星形). 星形是一棵高度为 1 的树,表示为 $T=(r,L)$,其中:(1) r 是 T 的根节点;(2) L 是二元组(p_i,l_i)的集合,也就是说,l_i 是树 T 的叶子,(r,p_i,l_i)是从 r 到 l_i 的边.用 $V(T)$和 $E(T)$分别表示星形 T 中顶点和边的集合.

定义 5(星形分解). SPARQL BGP 查询 $Q=\{tp_1,\ldots,tp_n\}$的星形分解定义为 $D=\{T_1,\ldots,T_m\}$,其中,(1) T_i 是一个星形;(2) $T_i.r\neq T_j.r,T_i.r,T_j.r\in D\wedge i\neq j$;(3) $E(T_i)\cap E(T_j)=\varnothing,T_i.r,T_j.r\in D\wedge i\neq j$;(4) $\bigcup_{1\leqslant i\leqslant m}E(T_i)=E(Q)$.

根据定义 4 可知:本文定义的星形结构中,边的方向都是从根节点指向叶子节点.用 $Sub(Q)$表示查询 Q 中主语的集合,那么 $Sub(Q)$就是查询 Q 的星形分解 D 中星形根节点的集合.如图 3 所示,给出 SPARQL BGP 示例查询图 Q_1 及其星形分解 D_1,包含 3 个星形 T_1,T_2,和 T_3.

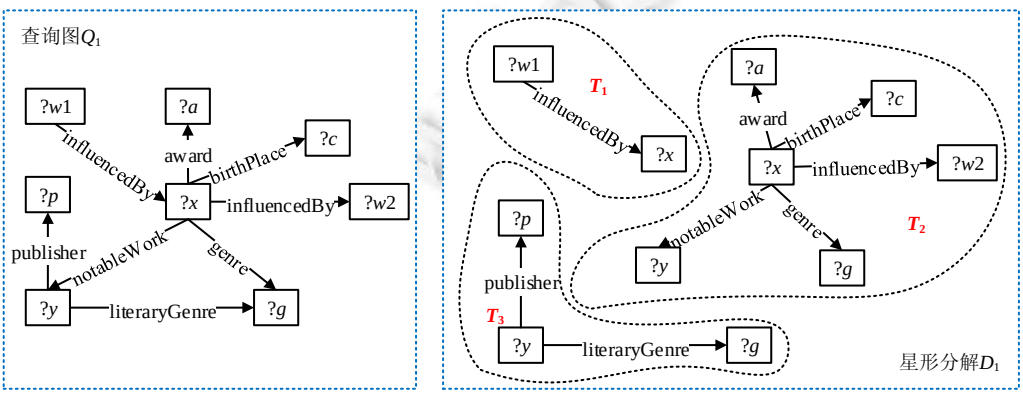

Fig.3 An example SPARQL BGP query graph Q_1 and the corresponding star decomposition D_1

图 3 SPARQL BGP 示例查询图 Q_1 及其星形分解 D_1

2.2 MapReduce并行计算模型

MapReduce 是谷歌提出的用来处理和生成大数据集的并行编程模型,可以运行在由很多普通机器组成的集群上[18].MapReduce 的用户通过自己编写 Map 和 Reduce 两个函数来表达并行计算.MapReduce 计算模型具体定义如下.

定义 6(**MapReduce 计算模型**). MapReduce 计算模型中包括若干轮 Map 和 Reduce 任务,每轮 MapReduce 任务包括一个 Map 函数和一个 Reduce 函数,其中,Map 函数定义为 Map:$(k_1,v_1)\rightarrow[(k_2,v_2)]$,Reduce 函数定义为 Reduce:$(k_2,[v_2])\rightarrow[(k_3,v_3)]$,其中,$[(k_i,v_i)]$表示 key/value 对的列表.在 Map 和 Reduce 之间是一个 shuffle 过程,该过程中,相同 key 的全部 value 值被发送给同一 Reduce 函数.

在 MapReduce 计算中,Map 函数输入一个 key/value 对(k_1,v_1),输出新的 key/value 对集合$[(k_2,v_2)]$作为中间结果,并将此输出发送给 Reduce 函数,Reduce 函数以接收到的一个中间值 key 和对应于该 key 的 value 集合,也就是$(k_2,[v_2])$作为输入,通过归并这些 value,生成新的 key/value 集合$[(k_3,v_3)]$并输出.

3 BGP 查询匹配算法

本节详细介绍本文的 SPARQL BGP 查询匹配方法,具体分为 3 部分:首先,采用分布式邻接链表存储 RDF 图;其次,将给定的 SPARQL BGP 查询分解为星形的集合并确定这些星形的匹配顺序;最后介绍基于 MapReduce 模型的分布式算法.

3.1 存储模式

本文采用分布式邻接链表模式存储 RDF 图,对 RDF 图中的顶点 $v\in V$,用 $N(v)$表示顶点 v 的邻居信息,$N(v)=\{(p_i,v_i)|(v,p_i,v_i)\in G\}$.RDF 示例图 G 的邻接链表存储方案见表 1.RDF 三元组中,URIs 或者字面值通常是很长字符串,如果直接存储大规模的原始数据而不做其他改变,会浪费大量的存储空间.本文采用编码映射方法,用唯一的 ID(整数)代替 RDF 图中顶点和边的标签,也就是 URIs 或者字面值.通过该方法可以压缩 RDF 数据集的存储空间,同时,用整数代替长的字符串可以简化并加速后期的 SPARQL 查询处理.

Table 1 Adjacency list storage schema

表 1 邻接链表存储模式

v	$N(v)$
House	{⟨*publisher*,*ThomasNelson*⟩,⟨*literaryGenre*,*HorrorFiction*⟩}
…	…
KingS	{⟨*notableWork*,*TheStand*⟩,⟨*notableWork*,*TheShining*⟩,⟨*genre*,*GothicFiction*⟩, ⟨*influencedBy*,*GoldingW*⟩,⟨*birthPlace*,*Portland*⟩,⟨*award*,*NationalBookA*⟩}
…	…
Newquay	{⟨*country*,*UnitedKingdom*⟩}

3.2 基于贪心策略的星形分解

这里介绍基于 RDF 图的结构信息和 RDF 内含语义的星形分解算法,该算法利用本文自定义的启发式信息 h-值对 SPARQL BGP 查询图 Q 进行分解,并找到中间结果较少的星形匹配顺序,从而提高 SPARQL BGP 的查询性能.

在本文中,星形是最小的匹配单位,在 RDF 图的邻接链表 $N(v)$上匹配星形 T 时,如果星形 T 的根节点可以匹配邻接链表 $N(v)$的主语顶点 v,对于星形 T 的所有叶子节点 l_i 得到它们在邻接链表 $N(v)$上可以匹配的顶点的集合,本文将此集合定义为叶子节点 l_i 的候选集 $S(l_i)$.接下来具体介绍星形在 RDF 邻接链表上的匹配过程,其匹配算法的伪代码见算法 1.

算法 1. 星形匹配算法.

输入:星形 $T=(r,L)$,其中,$L=\{(p_1,l_1),\ldots,(p_t,l_t)\}$,$v\in V$,邻接链表 $N(v)$;

输出:星形 T 在 $N(v)$上的匹配结果:$\Omega_v(T)=\{\mu_1,\mu_2,\ldots,\mu_n\}$.

1. $\Omega_v(T)\leftarrow\varnothing$;
2. **if** $T.r$ 可以匹配顶点 v **then**
3. **foreach** 每个叶子$(p_i,l_i)\in T.L$ **do** //获得叶子节点 l_i 的候选集合 $S(l_i)$
4. **if** $p_i\notin Var$ **then**
5. $S(l_i)\leftarrow\{v'|(p_i,v')\in N(v)\wedge l_i$ 可以匹配 $v'\}$;
6. **else**
7. $S(l_i)\leftarrow\{v'|(\_,v')\in N(v)\wedge l_i$ 可以匹配 $v'\}$; //_表示通配符
8. $\Omega_v(T)=\{\mu|\mu:V(T)\rightarrow\{v\}\times S(l_1)\times\ldots\times S(l_t)\}$;
9. **return** $\Omega_v(T)\leftarrow\varnothing$;

在邻接链表 $N(v)$上匹配星形 T 的过程包括:(1) 第 2 行匹配星形的根节点 $T.r$ 和 RDF 图中的顶点 v;(2) 第 3 行到第 7 行获得星形 T 的每个叶子节点 l_i 可以匹配的 RDF 图中顶点的候选集合 $S(l_i)$;(3) 第 8 行在星形 T 所有叶子节点的候选集合 $S(l_i)$上做笛卡尔积操作,得到星形 T 在邻接链表 $N(v)$上的匹配结果$\Omega_v(T)$;(4) 第 9 行返回星形 T 在邻接链表 $N(v)$上的匹配结果$\Omega_v(T)$.因此,星形 T 在 RDF 图 G 中的匹配结果$\Omega(T)$为$\Omega_v(T)$的并集,其中,$v\in V$.

例 1:根据算法 1,星形 T_1 在 RDF 示例图 G 上有 6 个匹配结果,包括$\{?w_1\rightarrow DekkerT,?x\rightarrow KoontzD\}$,$\{?w_1\rightarrow KingS,?x\rightarrow GoldingW\}$,$\{?w_1\rightarrow GibsonW,?x\rightarrow SpeculativeFiction\}$,$\{?w_1\rightarrow GibsonW,?x\rightarrow BorgesJL\}$,$\{?w_1\rightarrow GoldingW,?x\rightarrow VerneJ\}$和$\{?w_1\rightarrow DekkerT,?x\rightarrow KingS\}$.星形 T_2 在 RDF 示例图 G 上有 4 个匹配结果,包括$\{?x\rightarrow KingS,?y\rightarrow TheShining,?g\rightarrow GothicFiction,?w_2\rightarrow GoldingW,?c\rightarrow Portland,?a\rightarrow NationalBookA\}$,$\{?x\rightarrow KingS,?y\rightarrow TheStand,?g\rightarrow GothicFiction,?w_2\rightarrow GoldingW,?c\rightarrow Portland,?a\rightarrow NationalBookA\}$等,$T_3$ 有$\{?y\rightarrow House,?p\rightarrow ThomasNelson,?g\rightarrow HorrorFiction\}$,$\{?y\rightarrow TheShining,?p\rightarrow Doubleday,?g\rightarrow GothicFiction\}$,$\{?y\rightarrow LordoftheFiles,?p\rightarrow FaberandFaber,?g\rightarrow Allegory\}$这 3 个匹配结果.

考虑匹配顺序 $T_1T_3T_2$,因为星形 T_1 和 T_3 没有共享的顶点,所以对星形 T_1 和 T_3 的匹配结果进行连接操作时生成 6×3=18 个中间结果;另一匹配顺序 $T_2T_3T_1$ 只生成 1 个中间结果$\{?x\rightarrow KingS,?y\rightarrow TheShining,?g\rightarrow GothicFiction,?w_2\rightarrow GoldingW,?c\rightarrow Portland,?a\rightarrow NationalBookA,?p\rightarrow Doubleday,?g\rightarrow GothicFiction\}$.基于星形分解进行 SPARQL BGP 查询匹配时,不同的星形匹配顺序产生的中间结果有明显差异.

对 RDF 图 G 上的查询 Q,用 $Pred(u)$表示顶点 u 的属性(也就是谓语)集合,$Pred(u)=\{p_i|(u,p_i,u_i)\in Q\}$.函数 $freq:\Sigma\rightarrow\mathbb{N}$返回谓语 p 在 RDF 图 G 中的频率,其中 $freq(p)=|\{(s_i,p,o_i)|(s_i,p,o_i)\in G\}|$,$\mathbb{N}$表示自然数的集合.本文的启发式信息 h-值由两个因子决定:(1) 在查询图中顶点的出度越大,该顶点的邻居节点中变量个数可能越多,那么以该顶点为根的星形被匹配时会绑定更多的变量;(2) 查询图中,顶点的出边标签在 RDF 数据图中越稀有,该顶点的选择度越高,也就是当匹配以该点为根节点的星形时会产生相对少的匹配结果.如果顶点 u 的所有属性都是变量,则 $h(u)=0$;否则,查询图中顶点 u 的 h-值的具体定义如下:

$$h(u)=\frac{|outDeg|}{\min_{p\in Pred(u)}freq(p)} \tag{1}$$

由例 1 可知,对相同的星形分解结果,不同的星形匹配顺序在查询执行过程中产生的中间结果数目明显不同,基于此查询性能有非常显著的差异.因此,本文以基于 RDF 图的结构信息和丰富语义的 h-值作为启发信息,通过贪心策略对 BGP 查询图进行分解,并得到中间结果较少的星形匹配顺序.该分解策略的算法伪代码见算法 2.

算法 2. 星形分解算法.

输入:SPARQL BGP 查询 Q:$\{tp_1,\ldots,tp_n\}$;

输出:星形分解队列 D:$\{T_1,\ldots,T_m\}$.

1. 星形分解队列 $D\leftarrow\varnothing$;
2. 查询 Q 中常量集合 $Q_c\leftarrow\{s|s\in Sub(Q)\wedge s\notin Var\}$;

3. **if** $Q_c\neq\varnothing$ **then** //查询图中包含主语常量
4. $r\leftarrow \arg\max_{v\in Q_c} h(v)$; //选取 h-值最大的顶点作为第一个星形的根节点
5. **else**
6. $r\leftarrow \arg\max_{v\in Sub(Q)} h(v)$;
7. *genStar*(r,Q,D);
8. **while** $Q\neq\varnothing$ **do**
9. $M_v\leftarrow\{s|s\in Sub(Q)\wedge s\in V(D)\}\cup\{s|(s,p,o)\in Q\wedge o\in V(D)\}$; //得到根的候选集合
10. $r\leftarrow \arg\max_{v\in M_v} h(v)$;
11. *genStar*(r,Q,D)
12. **return** 星形分解队列 D:$\{T_1,\ldots,T_m\}$;
13. **function** *genStar*(r,Q,D) //根据根节点生成星形
14. $T.r\leftarrow r$, $T.L\leftarrow\{(p_i,l_i)|(r,p_i,l_i)\in Q\}$;
15. D.*enqueue*(T);
16. $Q\leftarrow Q\backslash\{(r,p_i,l_i)|(p_i,l_i)\in T.L\}$;

算法 2 根据 h-值将 RDF 查询图分解为星形的集合并给出匹配顺序,具体过程如下:第 3 行、第 4 行将 Q_c 中 h-值最大的常量顶点作为第 1 个星形的根节点,这样只在与该查询图中常量顶点匹配的数据顶点的邻接链表上进行匹配,可以很大程度减少中间结果.如果查询图中的主语都是变量,第 5 行、第 6 行将 $Sub(Q)$中 h-值最大的顶点作为第 1 个星形的根节点.然后,第 7 行调用 *genStar*(r,Q,D)函数生成以 r 为根节点的星形 T,并加入星形分解队列 D 中(第 14 行、第 15 行),第 16 行删除查询 Q 中已加入到 D 中的三元组模式.第 8 行~第 11 行迭代生成新的星形,直至 Q 中的三元组模式(也就是查询图中的边)被分解完,其中,第 9 行得到新生成星形根节点的候选顶点的集合 M_v,这些候选的根节点与已经生成的星形分解队列 D 中至少分享一个相同的顶点.第 10 行、第 11 行在候选集中选择 h-值最大的顶点作为根节点并生成新的星形.

例 2:考虑上文提及的示例查询 Q_1,$h(?w_1)=1/6$,$h(?x)=5/2$,$h(?y)=2/3$.根据算法 2,第 1 个选择的根节点是顶点?x,生成星形 T_1',也就是图 2 中的星形 T_2.然后,根据 h-值按序生成星形 T_2',T_3',即图 2 中的星形 T_3,T_1.

3.3 基于MapReduce的SPARQL BGP匹配算法

上节算法 2 将给定的 BGP 查询图分解为星形的集合,同时给定这些星形的匹配顺序,该匹配顺序利用启发式信息 h-值生成,使得以此顺序匹配这些星形时可以产生较少的中间结果,提高查询性能.本节将详细介绍如何按算法 2 产生的星形匹配顺序在左深连接(left-deep join)框架下基于 MapReduce 模型回答 SPARQL BGP 查询.

定义 7(局部查询图). 给定 BGP 查询图 G_Q,算法 2 生成星形分解 D 并给出星形的匹配顺序 $T_1\ldots T_m$,满足 $\bigcup_{1\leqslant i<j} V(T_i)\cap V(T_j)\neq\varnothing, 1\leqslant j\leqslant m$.局部查询图 P_j,$1\leqslant j\leqslant m$ 是查询图 G_Q 的子图,满足:(1) $V(P_j)=\bigcup_{1\leqslant i\leqslant j} V(T_i)$;(2) $E(P_j)=\bigcup_{1\leqslant i\leqslant j} E(T_i)$.

由定义 7 可知,$P_1=T_1$,$P_m=G_Q$.用 $\Omega(T_i)$和 $\Omega(P_i)$分别表示星形 T_i 和局部查询图 P_i 的匹配结果.将局部查询图 P_{t-1} 与星形 T_t 顶点交集的匹配结果作为连接的键 μ_{key}:$V(P_{t-1})\cap V(T_t)\rightarrow V$,连接 $\Omega(P_{t-1})$与 $\Omega(T_t)$可得到 $\Omega(P_t)$.每次 MapReduce 迭代操作新匹配一个星形,并且从第 2 次迭代开始,将已经匹配的局部查询图的结果与新匹配的星形结果进行连接操作.图 4 为处理 SPARQL BGP 查询方法的 MapReduce 总体框架图.并行 SPARQL BGP 查询匹配算法 SDec 通过多次 MapReduce 迭代计算得到查询结果,具体算法伪代码见算法 3.

算法 3. SDec 算法.

输入:RDF 图 $G=(V,E,\Sigma)$,星形分解队列 D:$\{T_1,\ldots,T_m\}$;

输出:答案集:$\Omega(Q)$.

1. $\Omega(Q)\leftarrow\varnothing$, $t\leftarrow 1$;
2. **while** D 不为空 **do**

3. T_t←D.*enqueue*(·);
4. *Map*(∅,$N(V)$); //在邻接链表 $N(v)$上并行匹配星形 T_t
5. **if** t>1 **then**
6. *Map*(∅,μ),其中,$\mu\in\Omega(P_{t-1})$; //将局部查询图 P_{t-1} 的匹配结果映射为键值对
7. *Reduce*(μ_{key},(Ω_1,Ω_2)),其中,$\Omega_1\subseteq\Omega(P_{t-1})\wedge\Omega_2\subseteq\Omega(T_t)$;
8. t←t+1;
9. **return** 答案集:$\Omega(Q)$;
10. **function** *Map*(∅,$N(V)$或者μ)
11. **if** 输入值是邻接链表 $N(v)$ **then**
12. 调用星形匹配算法得到星形 T_t 的匹配结果$\Omega_v(T_t)$;
13. **if** t=1 **then**
14. **foreach** $\mu\in\Omega_v(T_1)$ **then**
15. **return** (∅,μ);
16. **else**
17. **foreach** $\mu\in\Omega_v(T_t)$ **then**
18. μ_{key}←{(u_k,$\mu(u_k)$)|$u_k\in V(P_{t-1})\cap V(T_t)$}; //得到 Reduce 中连接的键
19. **return** (μ_{key},μ);
20. **else**
21. μ_{key}←{(u_k,$\mu(u_k)$)|$u_k\in V(P_{t-1})\cap V(T_t)$};
22. **return** (μ_{key},μ);
23. **function** *Reduce*(μ_{key},(Ω_1,Ω_2))
24. **foreach** (μ,μ')$\in\Omega_1\times\Omega_2$ **do**
25. **return** (∅,$\mu\cup\mu'$);

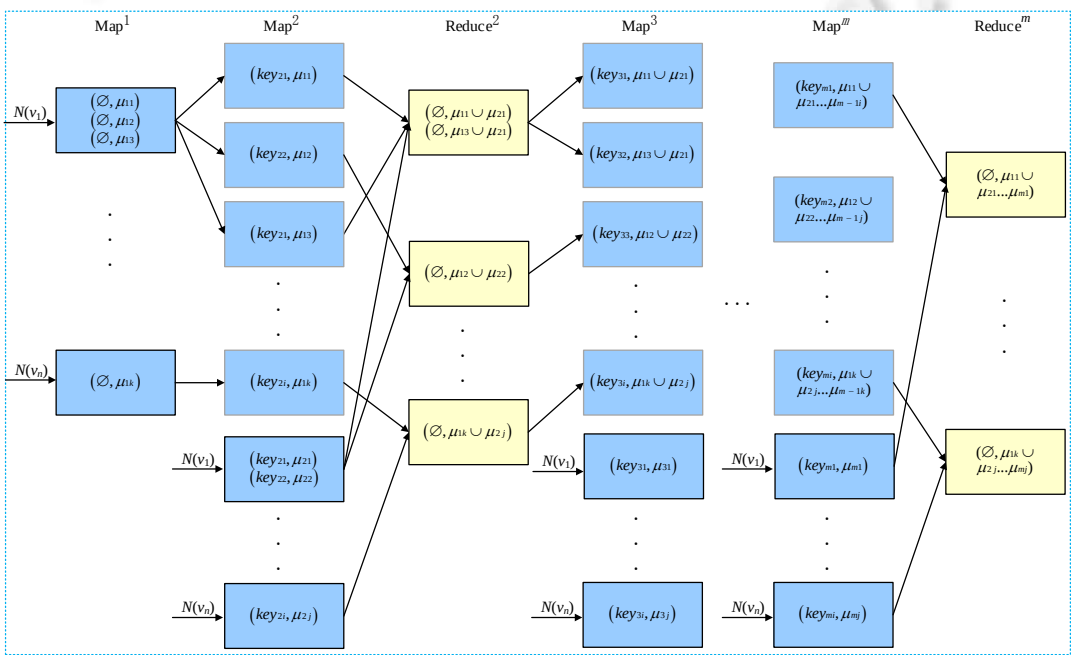

Fig.4 MapReduce framework of our SPARQL BGP matching algorithm

图 4 SPARQL BGP 匹配算法的 MapReduce 框架图

算法3第2行~第7行通过MapReduce操作,按算法2给定的顺序匹配星形,每轮MapReduce迭代连接局部子图和星形的匹配结果,直到所有星形被匹配.Map函数由两部分组成:(1) Map函数输入值为$N(v)$时,算法3在RDF图的邻居信息$N(v)$上并行匹配星形T_t,并且将匹配结果μ以(μ_{key},μ)对输出(第11行~第19行);(2) 输入值为$\Omega(P_{t-1})$中的匹配结果μ时,输出(μ_{key},μ)对(第20行~第22行).第23行~第25行,Reduce函数以映射μ_{key}为键,连接局部查询图P_{t-1}和星形T_t的匹配结果,生成P_t的匹配结果.

定理1. 给定RDF图G和查询图G_Q,假定算法2分解G_Q为星形队列$D=\{T_1,\ldots,T_m\}$.算法3给出正确查询结果并且MapReduce迭代次数为m.

证明:算法3在第1轮MapReduce迭代的Map函数中调用算法1匹配星形T_1,得到答案集$\Omega_v(T_1)$,也就是$\Omega(P_1)$;从第2轮开始,算法3按照算法2给定的匹配顺序,在每轮MapReduce迭代的Map函数中匹配一个新的星形T_t并得到查询结果$\Omega(T_t)$,在Reduce函数中,将上轮得到的局部查询图P_{t-1}的匹配结果$\Omega(P_{t-1})$与星形T_t匹配结果$\Omega(T_t)$进行连接得到$\Omega(P_t)$,从而在第m轮MapReduce中得到$\Omega(P_m)$,也就是查询Q的答案集.证毕. □

定理2. 算法3的时间复杂度为$O(|V|^m|N_{\max}|^{m|L_{\max}|})$,其中,$m$为星形个数,$|V|$为RDF图的顶点大小,$|N_{\max}|$和$|L_{\max}|$分别是RDF图$G$和查询图$G_Q$中最大的出度.

证明:算法3在第m轮MapReduce迭代计算后得到查询Q的答案集,其时间复杂度由两部分组成:(1) 星形匹配的时间复杂度为$\sum_{1\leqslant t\leqslant m}\sum_{v\in V}(|N(v)|+|\Omega(T_t)|)$;(2) 连接操作的时间复杂度为$\sum_{1\leqslant t\leqslant m}|\Omega(P_{t-1})|\times|\Omega(T_t)|$.最坏情况下,星形$T_t$的每个叶子可以匹配顶点$v\in V$的所有邻居顶点,也就是$|\Omega(T_t)|=|N(v)|^{|T_{t,L}|}$.因此,算法3的时间复杂度为$O(|V|^m|N_{\max}|^{m|L_{\max}|})$.证毕. □

4 优化技术

本节提出两个优化技术,提高基本算法的查询效率.虽然不能降低算法的复杂度,但是经过实验验证,可以明显提高查询效率.其中,一个技术通过布隆过滤器编码顶点的邻居信息减少数据输入,另一技术通过推迟笛卡尔积提高查询性能.

4.1 基于布隆过滤器的邻居信息编码

布隆过滤器(Bloom filter)利用位数组简洁地表示一个集合,并且可以快速检索一个元素是否在这个集合中,是一种空间效率很高的随机数据结构.布隆过滤器有可能会出现错误判断,但不会漏掉判断.它的核心思想是,利用多个不同的哈希函数来解决冲突.

布隆过滤器BF初始时是一个包含m位的位数组,每一位都置为0,即,BF整个数组的元素都设置为0.随后,每添加一个元素x,BF使用k个相互独立的哈希函数,分别将集合中的每个元素映射到$\{1,\ldots,m\}$的范围中,将数组中对应的比特位置为1.最后,判断y是否属于BF时,只需要对y使用k个哈希函数得到k个哈希值:如果所有$hash_i(y)$的位置都是1($1\leqslant i\leqslant k$),即k个位置都被设置为1了,那么y是集合中的元素;否则,就认为y不是集合中的元素.

在布隆过滤器中,根据输入元素个数n和BF数组位数m,可得到使误判(flase positive)率f最小的哈希函数的个数k:$k=\ln 2\,m/n$.此外,哈希函数个数k与误判率f满足等式$k=-\log_2 f$.由该等式可知:k越大,误判率f小.本文借鉴布隆过滤器中的参数设置对RDF图中主语的邻居边及顶点进行编码时,但是不再判断一个元素是否在BF集合中,而是判断一个布隆过滤器集合是否是另一个布隆过滤器集合的子集.形式化定义为:如果$BF_1\&\&BF_2=BF_1$,那么布隆过滤器BF_1是BF_2的子集;否则,就认为不是.即:判断布隆过滤器BF_1是否是布隆过滤器BF_2的子集时,如果所有BF_1为1的位置,BF_2对应的位置都被设置为1了,那么布隆过滤器BF_1是BF_2的子集;否则就认为不是.

对RDF数据图中的每个主语s的邻居信息基于布隆过滤器进行编码,并将此编码作为它的签名,表示为$Sig(s)$.把主语s的每条出边$e=(s,o)\in V\times V$用k个哈希函数映射成长度为$M+N$的位数组$ecd(e)$.前M位表示边标签的编码,用k个哈希函数$h_i(i=1,\ldots,k)$将M位中的k位置为"1".本文选取经典的字符串哈希函数,如BKDRHash,APHash,DJBHash等.对第i个哈希函数h_i,设置前M位中的第$h_i(lab(e)) \bmod M$位为"1",其中,$h_i(lab(e))$表示边标

签 *lab*(*e*)的哈希值.后 *N* 位表示顶点的编码,其方式与上述类似,将后 *N* 位中的 *k* 位采用不同的哈希函数置为“1”.将主语 *s* 所有出边的位数组 *ecd*(*e*)按位或操作后得到它的签名 *Sig*(*s*)=*ecd*(e_1)|…|*ecd*(e_t),其中,e_i 为主语 *s* 的出边.

如图 5 所示,对 RDF 图 *G* 的顶点 GoldingW 及图 3 中的星形 T_3 进行邻居信息编码.首先对顶点 *GoldingW* 的出边(*GoldingW*,*LordoftheFlies*)进行编码,将边标签 *notableWork* 映射成长度为 8 的位数组,具体操作如下:首先求哈希值 *APHash*(*notableWork*)和 *DJBHash*(*notableWork*),并对这两个哈希值 mod 8 得到余数 2 和 7;然后将位数组从左到右的第 2 位和第 7 位置为 1(从第 0 位开始),其余位置为 0,如图 5 所示;类似地,将相邻顶点 *LordoftheFlies* 映射到长度为 12 的位数组,取 *APHash* 和 *DJBHash* 函数的值并 mod 12 得到余数进行编码,将边标签的位数组和顶点的位数组连接得到边(*GoldingW*,*LordoftheFlies*)的编码 *ecd*((*GoldingW*,*LordoftheFlies*));最后,将不同出边的编码按位或操作得到 *GoldingW* 顶点的签名 *Sig*(*GoldingW*).值得注意的是:没有出边的顶点,其签名的所有位默认为 0.同样地,对星形 T_3 进行邻居信息编码,边标签 *literaryGenre* 的两个哈希值 mod 8 得到余数 3 和 5 后编码位数组,如图 5 所示.因星形中的顶点?*g* 为变量,所以它的位数组 12 位都置为 0,将这两个位数组连接后得到编码 *ecd*((*literaryGenre*,?*g*)),计算其他边的编码进行或操作后,得到星形 T_3 的邻居信息编码 *Sig*(T_3).

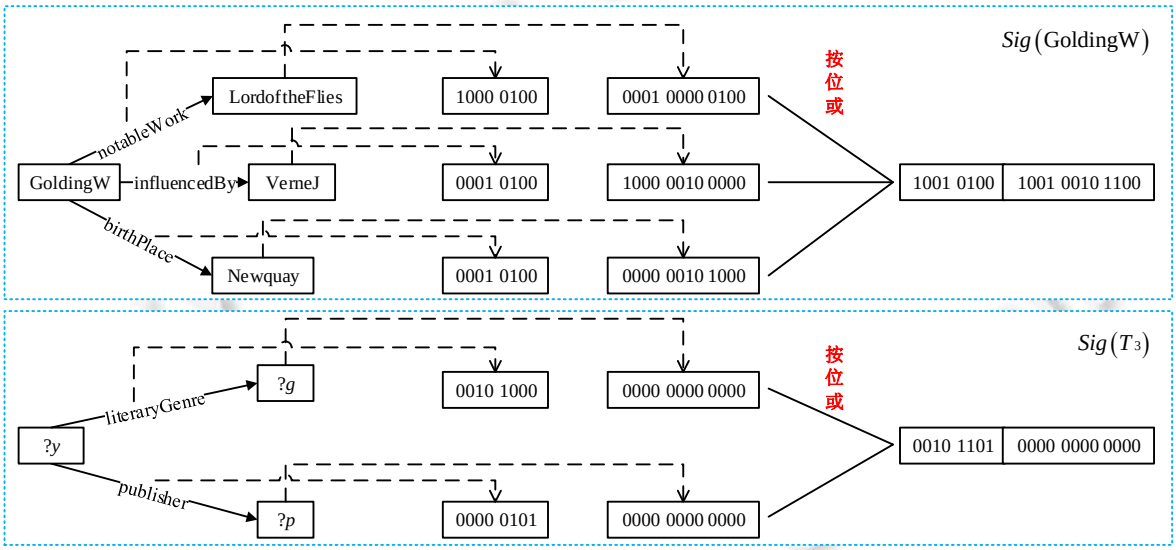

Fig.5 Procedure of the signature generation of vertex labeled *GoldingW* and star T_3

图 5 主语 *GoldingW* 和星形 T_3 签名的生成过程

本文第 3 节将 SPARQL BGP 查询图分解为星形结构的集合,对星形 *T* 采用相同的方式进行编码,每个星形结构也生成一个长度固定的位数组做为签名,表示为 *Sig*(*T*).当星形 *T* 中的边或顶点为变量时,它的所有编码位都置为 0.

剪枝规则. 星形 *T* 在邻居信息 *N*(*v*)上匹配时:如果 *Sig*(*T*)&*Sig*(*v*)≠*Sig*(*T*),在 *N*(*v*)上匹配星形 *T* 的过程被剪掉.

在每轮 MapReduce 操作中,Map 函数在每个顶点的邻接链表上并行匹配一个新的星形 *T*.如果在顶点 *v* 的邻接链表上匹配星形 *T* 时满足剪枝规则,也就是星形 *T* 的邻居边标签集合不是顶点 *v* 邻居边标签的子集,或者星形的叶子节点的集合不是顶点 *v* 邻居顶点的子集,那么星形 *T* 在顶点 *v* 的邻接链表上必然没有匹配结果,因此不再进行算法 1 的星形匹配.本文通过上述剪枝规则减少 MapReduce 迭代过程中无用的数据输入,同时减少大量没必要的星形匹配过程中涉及的计算,以此优化 BGP 查询.

例 3:如图 5 所示,当在邻接链表 *N*(*GoldingW*)上匹配星形 T_3 时,因为 *Sig*(T_3)&*Sig*(*GoldingW*)≠*Sig*(T_3)所以在邻接链表 *N*(*GoldingW*)上的星形 T_3 的匹配过程被减掉,从而可以减少中间结果.

4.2 推迟笛卡尔积操作

在本文的基本算法中,初始星形匹配阶段在匹配星形 $T=(r,L)$,其中,$L=\{(p_1,l_1),\dots,(p_t,l_t)\}$,$v\in V$ 时,首先根据叶子节点的标签在各个主语 v 的邻接链表 $N(v)$上得到各个叶子节点 l_i 的候选集合 $S(l_i)$,然后通过叶子节点候选集合的笛卡尔积$\{v\}\times S(l_1)\times\dots\times S(l_t)$,得到星形 T 的所有匹配结果.但是大部分星形匹配结果无法形成最终的查询答案,从而导致在星形匹配阶段做了很多无用的笛卡尔积操作,在 Reduce 阶段连接代价过高.

本文的优化算法在星形匹配阶段仅仅得到各叶子节点的候选集合即可.在 Map 函数中计算局部查询图 P_{t-1} 和星形 T_t 的顶点交集 $V(P_{t-1})\cap V(T_t)$中的各个顶点的候选集,并通过这些顶点候选集的笛卡尔积得到 Reduce 函数连接的键.MapReduce 操作结束后,通过剩余顶点候选集的笛卡尔积得到最终的查询结果.将各个星形匹配过程中叶子节点候选集的笛卡尔积推迟到 MapReduce 操作结束的时候,做过滤部分无用的计算.

例 4:匹配查询图 Q_1 时,在前 3 轮的 Map 函数中,仅需要得到每个星形叶子节点的候选集.如图 6 所示,第 2 轮 MapReduce 中,通过 $V(P_1)\cap V(T_3)=\{?y,?g\}$中顶点的候选集合的笛卡尔积,得到星形 T_3 中$\{?y,?g\}$的匹配结果$\{?y\rightarrow House,?g\rightarrow HorrorFiction\}$,$\{?y\rightarrow TheStand,?g\rightarrow GothicFiction\}$ 和 $\{?y\rightarrow LordoftheFlies,?g\rightarrow Allegory\}$;$P_1$ 中$\{?y,?g\}$的匹配结果$\{?y\rightarrow TheStand,?g\rightarrow GothicFiction\}$和$\{?y\rightarrow TheShining,?g\rightarrow GothicFiction\}$.将这些匹配结果作为键,在 Reduce 过程中进行连接操作,可以观察到,仅当$\{?y\rightarrow TheStand,?g\rightarrow GothicFiction\}$可以得到局部查询图 P_2 的匹配结果.类似地,在第 3 轮 MapReduce 中,通过 $V(P_2)\cap V(T_1)=\{?x\}$中顶点候选集合的笛卡尔积,得到星形 T_1 和局部查询图 P_2 中$\{?x\}$的匹配结果集合,以这些匹配结果作为键值,在 Reduce 中进行连接操作.最后,通过剩余顶点集合$\{?a,?c,?w_2,?p,?w_1\}$中各顶点的候选集的笛卡尔积,得到查询 Q_1 的最终匹配结果$\{?y\rightarrow TheShining,?g\rightarrow GothicFiction,?x\rightarrow KingS,?a\rightarrow NationalBookA,?c\rightarrow Portland,?w_2\rightarrow GoldingW,?p\rightarrow Doubleday,?w_2\rightarrow DekkerT\}$.在基本算法 SDec 中,Map 函数在邻接链表 $N(GibsonW)$上匹配星形 T_2 时需要各个叶子节点的候选集的笛卡尔积 $S(?a)\times S(?c)\times S(?w_2)\times S(?g)\times S(?y)$,然而查询图中顶点$?x$ 匹配 RDF 数据图中顶点 $GibsonW$ 并不能形成最终答案,因此,基本算法做了无用的笛卡尔积,代价较高.相反,优化算法通过推迟笛卡尔积提前过滤掉在邻接链表 $N(GibsonW)$上的笛卡尔积操作.

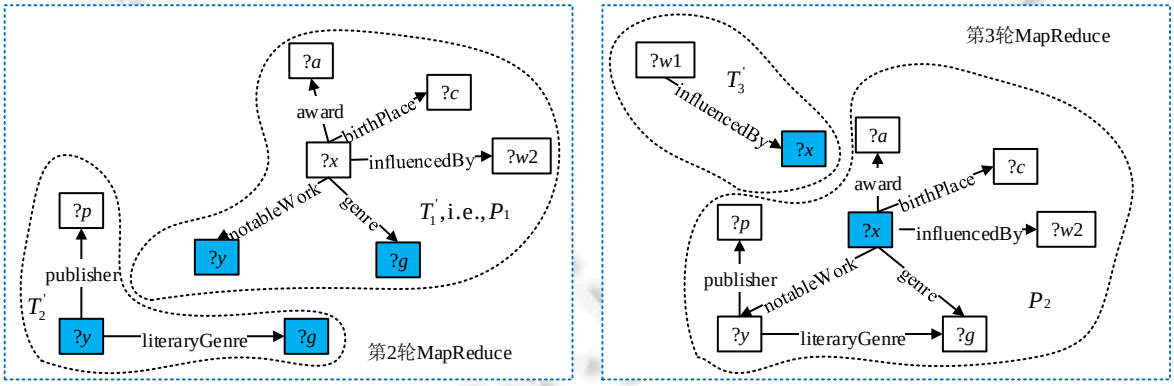

Fig.6 Matching process of the example SPARQL BGP query graph Q_1

图 6 SPARQL BGP 示例查询图 Q_1 查询匹配过程

5 实 验

本节在合成数据集和真实数据集上验证算法的有效性和可扩展性,并且与 SHARD,S2X 和 SDec 的优化版本的算法比较.不与 S2RDF 和 TriAD 方法作比较的原因如下.

(1) S2RDF 使用 ExtVP 索引提高查询效率,但是预处理时间代价过高,严重限制了该系统对大规模数据集的适用性,最新的文献[22]中通过实验也验证了这一点.例如:在相同的实验环境下,S2RDF 预处理 DBpedia 数据集的时间超过 2.5 个小时,远大于本文 SDec 系统的查询执行时间;

(2) 如文献[22]所述,TriAD 假设数据必须装载到集群的内存中,如果数据量超过了集群内存大小,TriAD

将不再适用.而本文没有这个限制条件,而且 TriAD 借助了大量的索引加快查询.

因此,与这两种方法进行比较实验是不公平的.

本文的 SDec 算法与图划分策略和分布式集群环境上的存储策略是独立无关的.实际上,本文使用分布式文件系统 HDFS(Hadoop distributed file system)的默认划分策略.

5.1 实验环境和数据集

本文实验平台使用的分布式集群包括 8 个相同的计算节点,每个节点使用主频为 3.60GHz 的 Intel(R) Core(TM) i7-7700 四核处理器,其内存大小为 16GB,硬盘大小为 500GB.节点间通信使用 1000Mbps 以太网.实验平台所用集群的所有节点均使用 Linux 64-bit CentOS 操作系统,其使用的 Hadoop 版本号为 2.7.4,Spark 版本号为 2.2.0.本文原型系统的所有代码均用 Scala 语言实现.

本实验使用的数据包括 WatDiv 标准合成数据集以及 DBpedia 真实数据集,这两个数据集都是 RDF 数据集.WatDiv 是一个标准的 RDF 合成数据集,允许用户定义自己的数据集并生成不同大小的数据集,本文使用了 3 个不同规模的 WatDiv 数据集(也就是 WatDiv1M,WatDiv10M 和 WatDiv100M)进行实验测试和比较;DBpedia 数据集是从维基百科中提取的一个真实数据集.本次实验中,所有数据集均为 N-Triple 格式,具体统计信息见表 2.本文根据 RDF 查询图的形状将它们分为 4 类,包括线性查询(linear queries,简称 L)、星形查询(star queries,简称 S)、雪花形查询(snowflake queries,简称 F)和复杂查询(complex queries,简称 C),见表 3.本文使用 WatDiv 数据集给定的 20 个基本查询模板进行实验测试.因为缺乏 DBpedia 数据集上的查询模板,本文设计 8 个查询覆盖上文提及的 4 个查询种类.

Table 2 Datasets

表 2 数据集

数据集	顶点个数	边个数
WatDiv1M	158 118	1 109 678
WatDiv10M	1 052 571	10 916 457
WatDiv100M	10 250 571	108 997 714
DBpedia	6 060 648	23 509 250

Table 3 Queries

表 3 查询

查询	WatDiv	DBpedia
L	L1, L2, L3, L4, L5	L1, L2
S	S1, S2, S3, S4, S5, S6, S7	S1, S2
F	F1, F2, F3, F4, F5	F1, F2
C	C1, C2, C3	C1, C2

5.2 实验结果

本节分 4 组实验进行详细介绍,包括改变优化算法中哈希函数个数 k、改变数据集规模及改变集群中节点个数来验证优化算法 $SDec_{opt}$、基本算法 Sdec,SHARD 和 S2X 这 4 种方法的效率和可伸缩性.以下实验结果中,每个查询测试 3 次取平均值,避免随机误差.

(1) 改变编码优化技术中参数 k 测试查询性能.

本组实验分别设置哈希函数个数 k=1,2,3,在 WatDiv100M 数据集上测试 20 个模板查询.随着 k 从 1 增加到 3,误判率 f 减小,可以在星形匹配阶段过滤更多的无用数据输入,加快数据查询;同时,随着 RDF 数据图中主语签名的位数组长度变长,占用内存空间越多,影响查询效率.

如图 7 所示,当 k=3 时,编码邻居信息增加查询运行时内存带来的影响占主要因素,导致查询时间增加;k=2 时,在绝大部分查询上可以取得最好的查询性能.随着哈希函数个数从 1 到 3,邻居信息 BF 编码的运行时内存变化.本文以下的所有实验中,优化方法中的哈希函数个数 k 取值为 2.

(2) SPARQL BGP 查询性能测试.

本组实验分别在合成数据集 WatDiv100M 和真实数据集 DBpedia 上验证集群节点个数为 8 时本文所提方法的查询效率.如图 8 所示,在合成数据集 WatDiv 上,$SDec_{opt}$ 在 20 个模板查询上都可以获得最高的查询效率;同时,SDec 也优于 SHARD 和 S2X.在本文中,运行时间超过 1×10^4s 用 INF 表示.如图 8 所示:对查询 F3 和 C2,S2X 不能在该时间限制内结束,而本文的优化算法仅需要 19.24s 和 36.92s 就可以得到查询结果.对剩余的 18 个查询,本文优化算法的查询时间与 S2X 相比提高了 7 倍~40 倍,平均查询时间提高 17 倍.究其原因:S2X 将 BGP 中所有三元组模式的匹配结果进行连接操作会产生大量的中间结果,代价较高;而本文的最小匹配结构为星形,同时考虑 RDF 图的丰富语义和结构信息,减少了大量无用的中间结果.与 SHARD 相比,$SDec_{opt}$ 的查询时间提高了 16 倍~99 倍,平均查询时间提高 40 倍.SHARD 采用三元组存储也将 BGP 查询分解为三元组模式进行匹配,在包含 3 列的数据表上进行连接操作,连接代价高.因此,$SDec_{opt}$ 算法的查询时间比 S2X 和 SHARD 平均提高了一个数量级.

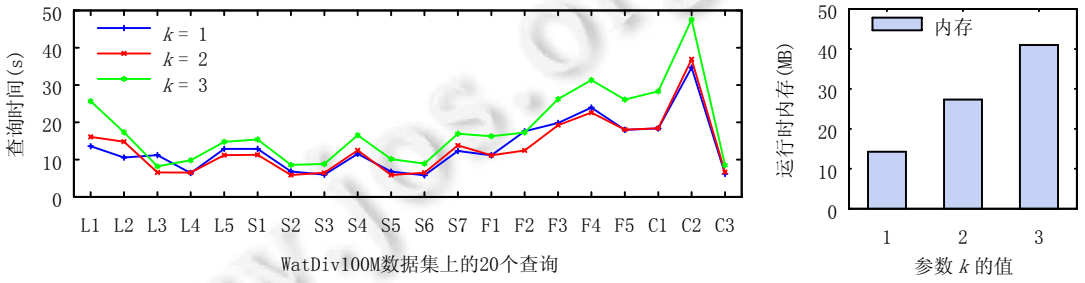

Fig.7 Experiments of changing the number of hash functions k and the run memory

图 7 改变哈希函数个数 k 的实验结果及运行时内存变化

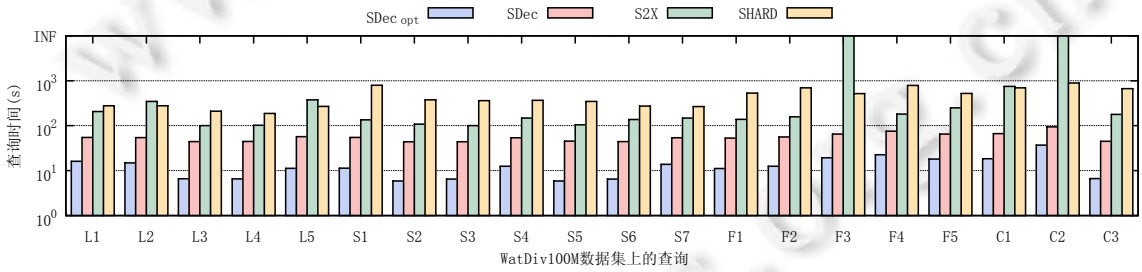

Fig.8 Query efficiency of different SPARQL processors over synthetic dataset

图 8 不同 SPARQL 处理器在合成数据集上的查询效率

此外,与基本算法 SDec 相比,优化算法时间减少了 60.68%~87.00%,即,$SDec_{opt}$ 可以更有效地评估 BGP 查询.该实验结果说明,本文的优化技术效果非常显著.原因包含以下两点:(1) 优化算法对邻居信息进行 BF 编码,在星形匹配阶段,如果星形结构的签名与 RDF 图中主语 s 的签名不匹配,直接将 $N(s)$上的星形匹配剪掉,以此过滤掉大量的无用计算;(2) 基本算法在星形匹配阶段进行了大量的无用笛卡尔积,而优化算法通过推迟笛卡尔积操作减少很多无用的笛卡尔积操作,提高了查询性能.

如图 9 所示,在真实数据集 DBpedia 上,$SDec_{opt}$ 在所有查询上的效率也是最高的.回答查询 C2 时 S2X 报错,导致时间缺失.这说明 S2X 无法有效回答涉及大量中间结果的复杂查询.对于剩余的 7 个查询,优化算法 $SDec_{opt}$ 查询时间比 S2X 快 7 倍~497 倍,平均查询时间比 S2X 快 120 倍;同时,与 SHARD 相比,优化算法 $SDec_{opt}$ 查询时间快 7 倍~26 倍,平均查询时间快 15 倍.即:在真实数据集上,本文算法的平均时间也比 SHARD 和 S2X 快一个数量级.另外,由图 9 观察到:优化算法的查询时间明显提高,比基本算法 SDec 查询时间减少 49.63%~78.71%.

(3) 改变数据集规模的查询性能测试.

本组实验通过改变 WatDiv 数据集规模从 1M~100M 在 8 个节点的集群上测试 18 个查询.本文将这 18 个

查询分为 4 个查询种类,分别计算 SDec,SDec$_{opt}$,S2X 和 SHARD 的 4 类平均查询时间.如图 10 所示,随着数据规模增大,所有 SPARQL 查询处理器的查询时间都增加,SHARD 和 S2X 的查询时间增加幅度远大于本文方法.观察图 10 可知:数据集规模从 10M 个三元组增加到 100M 个三元组,S2X 和 SHARD 的查询性能严重下降.如:对雪花形查询,SHARD 和 S2X 在 WatDiv100M 上需要将近 1000s 的时间才可以得到查询结果,而本文的优化算法仅需 10s 左右.

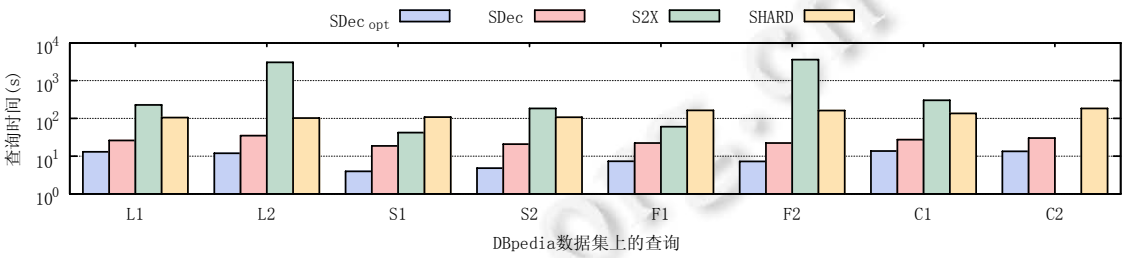

Fig.9 Query efficiency of different SPARQL processors over real-world dataset

图 9 不同 SPARQL 查询处理器在真实数据集上的查询效率

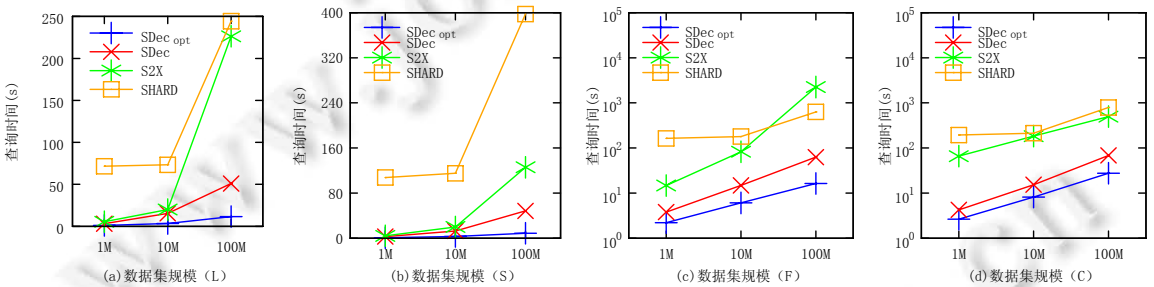

Fig.10 Experiment results of changing the size of datasets

图 10 改变数据集规模的实验结果

(4) 算法可伸缩性测试.

本组实验在 WatDiv100M 和 DBpedia 数据集上进行,改变集群节点个数,从 4~8.从每类查询中随机选取 1 个查询,也就是 L4,S4,F2,C3.如图 11 所示:随着集群节点个数的增加,在合成数据集 WatDiv 上,这 4 个 SPARQL 查询处理器的查询时间减少.原因是随着集群节点个数的增加,CPU 核数增加,查询过程中计算的并行度增加,查询所需的总时间减少.由图 11 可以看出,SDec$_{opt}$ 的查询性能始终是最高的.

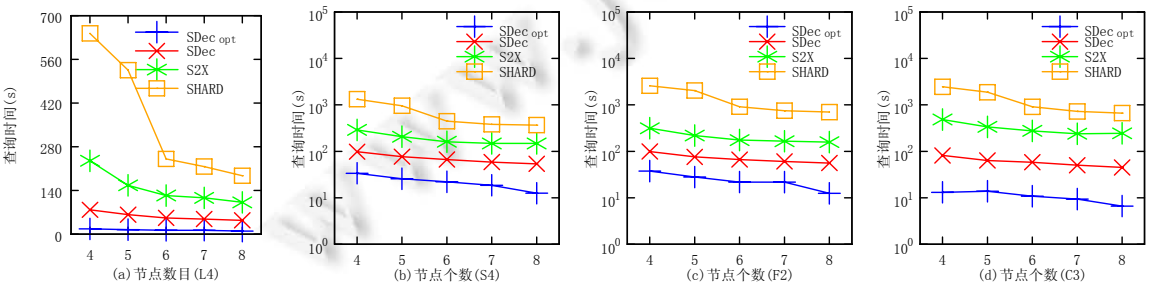

Fig.11 Experiment results of changing the number of sites on synthetic dataset WatDiv100M

图 11 改变集群节点个数在合成数据集 WatDiv100M 上的实验结果

在真实数据集 DBpedia 上,改变集群节点的个数验证算法的可伸缩性.本次实验选取 L1,S1,F1,C1 查询,即,

从每类查询中随机选取 1 个查询进行实验.如图 12 所示:随着集群节点个数从 4 增加到 8,上述 4 个 SPARQL 查询处理方法的查询时间逐渐减少.

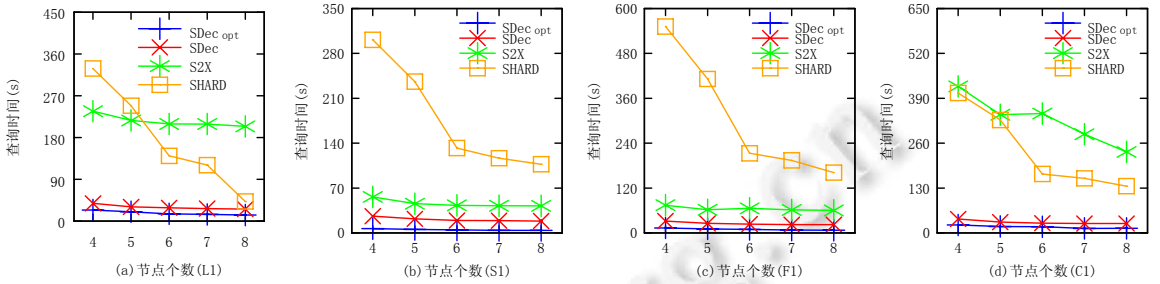

Fig.12 Experiment results of changing the number of sites on real-world dataset DBpedia

图 12 改变集群节点个数在真实数据集 DBpedia 上的实验结果

6 总 结

本文提出了基于 MapReduce 框架的查询处理器 SDec,有效回答大规模 RDF 智能图数据上的 SPARQL BGP 查询.本文的 SDec 算法利用 RDF 图内嵌的丰富语义和 RDF 图结构分解查询图为星形集合,并确定这些星形的匹配顺序,以此减少星形匹配过程中的中间结果.此外,为了进一步提高查询算法的效率,本文设计了两个优化技术,包括对邻居信息编码过滤无用的数据输入和推迟笛卡尔积操作改进基本算法的性能.合成数据集和真实数据集上的大量实验验证了本文所提算法的有效性、高效性和伸缩性,并且实验结果显示,本文的算法优于 SHARD 和 S2X.

References:

[1] Hammoud M, Rabbou DA, Nouri R, Beheshti SMR, Sakr S. DREAM: Distributed RDF engine with adaptive query planner and minimal communication. Proc. of the VLDB Endowment, 2015,8(6):654–665.

[2] Pérez J, Arenas M, Gutierrez C. Semantics and complexity of SPARQL. In: Cruz I, ed. Proc. of the Semantic Web (ISWC 2006). Berlin: Springer-Verlag, 2006. 30–43.

[3] Dyer M, Greenhill C. The complexity of counting graph homomorphisms. Random Structures & Algorithms, 2000,17(3-4): 260–289.

[4] Du F, Chen YG, Du XY. Survey of RDF query processing techniques. Ruan Jian Xue Bao/Journal of Software, 2013,24(6): 1222–1242 (in Chinese with English abstract). http://www.jos.org.cn/1000-9825/4387.htm [doi: 10.3724/SP.J.1001.2013.04387]

[5] Neumann T, Weikum G. RDF-3X: A RISC-style engine for RDF. Proc. of the VLDB Endowment, 2008,1(1):647–659.

[6] Zou L, Chen L, Shen X, Huang R, Zhao D. gStore: A graph-based SPARQL query engine. VLDB Journal—The Int'l Journal on Very Large Data Bases, 2014,23(4):565–590.

[7] Rohloff K, Schantz RE. High-performance, massively scalable distributed systems using the MapReduce software framework: The SHARD triple-store. In: Proc. of the Programming Support Innovations for Emerging Distributed Applications. New York: ACM Press, 2010. Article No.4.

[8] Husain M, Mcglothlin J, Masud MM, Khan LR, Thuraisingham B. Heuristics-based query processing for large RDF graphs using cloud computing. IEEE Trans. on Knowledge & Data Engineering, 2011,23(9):1312–1327.

[9] Schätzle A, Przyjaciel-Zablocki M, Berberich T, Lausen G. S2X: Graph-parallel querying of RDF with GraphX. In: Wang F, ed. Proc. of the Biomedical Data Management and Graph Online Querying. Switzerland: Springer-Verlag, 2016. 155–168.

[10] Erling O, Mikhailov I. Virtuoso: RDF Support in a Native RDBMS. Berlin: Springer-Verlag, 2009. 501–519.

[11] Schatzle A, Przyjacielzablocki M, Skilevic S, Lausen G. S2RDF: RDF querying with SPARQL on spark. Proc. of the VLDB Endowment, 2016,9(10):804–815.

[12] Zeng K, Yang J, Wang H, Shao B, Wang Z. A distributed graph engine for Web scale RDF data. Proc. of the VLDB Endowment, 2013,6(4):265−276.

[13] Peng P, Zou L, Chen L, Zhao D. Processing SPARQL queries over distributed RDF graphs. The VLDB Journal—The Int'l Journal on Very Large Data Bases, 2016,25(2):243−268.

[14] Gurajada S, Seufert S, Miliaraki I, Theobald M. TriAD: A distributed shared-nothing RDF engine based on asynchronous message passing. In: Dyresson C, ed. Proc. of the ACM Conf. on Management of Data. New York: ACM Press, 2014. 289−300.

[15] Lai L, Qin L, Lin X, Chang L. Scalable subgraph enumeration in MapReduce. Proc. of the VLDB Endowment, 2015,8(10): 974−985.

[16] Sun Z, Wang H, Wang H, Shao B, Li J. Efficient subgraph matching on billion node graphs. Proc. of the VLDB Endowment, 2012, 5(9):788−799.

[17] Dean J, Ghemawat S. MapReduce: Simplified data processing on large clusters. Communications of the ACM, 2008,51(1): 107−113.

[18] Aluc G, Hartig O, Ozsu M, Daudjee K. Diversied stress testing of RDF data management systems. In: Mika P, ed. Proc. of the Semantic Web (ISWC 2014). Berlin: Springer-Verlag, 2014. 197−212.

[19] Zou L, Peng P. A survey of distributed RDF data management. Journal of Computer Research and Development, 2017,54(6): 1213−1224 (in Chinese with English abstract).

[20] Gonzalez JE, Xin RS, Dave A, Crankshaw D, Franklin MJ, Stoica I. GraphX: Graph processing in a distributed dataflow framework. OSDI, 2014,14(2):599−613.

[21] Zaharia M, Chowdhury M, Franklin MJ, Shenker S, Stoica I. Spark: Cluster computing with working sets. HotCloud, 2010, 10(10-10):95.

[22] Abdelaziz I, Harbi R, Khayyat Z, Kalnis P. A survey and experimental comparison of distributed SPARQL engines for very large RDF data. Proc. of the VLDB Endowment, 2017,10(13):2049−2060.

附中文参考文献:

[4] 杜方,陈跃国,杜小勇.RDF 数据查询处理技术综述.软件学报,2013(6):1222−1242. http://www.jos.org.cn/1000-9825/4387.htm [doi: 10.3724/SP.J.1001.2013.04387]

[19] 邹磊,彭鹏.分布式 RDF 数据管理综述.计算机研究与发展,2017,54(6):1213−1224.

王鑫(1981－),男,天津人,博士,副教授,CCF 高级会员,主要研究领域为知识图谱数据管理,图数据库,大规模知识处理.

杨雅君(1983－),男,博士,讲师,CCF 专业会员,主要研究领域为图数据管理,图挖掘.

徐强(1993－),女,硕士生,主要研究领域为知识图谱数据管理,图数据库.

柴云鹏(1983－),男,博士,副教授,博士生导师,CCF 专业会员,主要研究领域为分布式系统,云计算,存储系统.

柴乐乐(1995－),女,硕士生,主要研究领域为知识图谱表示学习.