ISSN 1000-9825, CODEN RUXUEW
Journal of Software, Vol.20, No.9, September 2009, pp.2436−2449
doi: 10.3724/SP.J.1001.2009.03421
© by *Institute of Software*, *the Chinese Academy of Sciences*. All rights reserved.
E-mail: jos@iscas.ac.cn
http://www.jos.org.cn
Tel/Fax: +86-10-62562563

图结构 XML 文档上子图查询的高效处理算法*

王宏志+, 骆吉洲, 李建中

(哈尔滨工业大学 计算机科学与技术学院,黑龙江 哈尔滨 150001)

Efficient Subgraph Query Processing Algorithms on Graph-Structured XML Documents

WANG Hong-Zhi+, LUO Ji-Zhou, LI Jian-Zhong

(School of Computer Science and Technology, Harbin Institute of Technology, Harbin 150001, China)

+ Corresponding author: E-mail: wangzh@hit.edu.cn, http://whongzhi.googlepages.com/

Wang HZ, Luo JZ, Li JZ. Efficient subgraph query processing algorithms on graph-structured XML documents. ***Journal of Software*****, 2009,20(9):2436−2449.** http://www.jos.org.cn/1000-9825/3421.htm

Abstract: Many challenges arise in subgraph query processing of graph-structured XML data. This paper studies the subgraph query processing of graph-structured XML data and proposes. A hash-based structural join algorithm, HGJoin to handle reachability queries on graph-structured XML data. Then, the algorithm is extended to process subgraph queries in form of bipartite graphs. Finally, based on these algorithms and cost model presented in this paper, a method to process subgraph queries in form of general DAGs is proposed. It is notable that all the algorithms above can be slightly modified to process subgraph queries in form of general graphs. Analysis and experiments show that all the algorithms have high performance.
Key words: XML; graph; subgraph query; query processing

摘 要: 研究了图结构 XML 数据上子图查询处理,给出了一系列高效的处理算法.基于可达编码,首先提出基于哈希的结构连接算法(HGJoin)来处理图结构 XML 数据上的可达查询.然后,该算法被扩展来处理特殊的二分图查询.基于这些算法和所给出的代价模型,提出了一般 DAG 子图查询的处理算法和查询优化策略.这些算法经过简单修改即可有效地处理一般的子图查询.理论分析和实验结果表明,算法具有较高的效率.
关键词: XML;图;子图查询;查询处理
中图法分类号: TP311 **文献标识码**: A

XML 数据通常被看成是有标签的树,元素或属性是树的结点,元素或属性间的直接嵌套关系是树的边.然而,在一些常见的 XML 文档中,由于元素间一对多和多对一关系同时存在,XML 文档中元素或属性间的自然对应关系更适于用图结构来表达.很多应用中的 XML 文档均被表达成图结构[1].

在图结构 XML 文档上的查询中,子图查询是一类重要的查询,并逐渐得到了广泛应用.图结构 XML 文档上的子图查询是在图结构 XML 文档上查找与查询中的图匹配的子图.图结构 XML 文档上的子图查询难以表达成树结构 XML 上的 XQuery 查询.现有的树结构 XML 文档上,结构查询的处理方法都使用到了树结构的特性,

* Supported by the National Natural Science Foundation of China under Grant Nos.60773068, 60773063, 60533110, 60703012 (国家自然科学基金); the National Basic Research Program of China under Grant No.2006CB303000 (国家重点基础研究发展计划(973))
Received 2007-12-13; Accepted 2008-07-09

© 中国科学院软件研究所 http://www.jos.org.cn

不适用于图结构 XML 数据上子图查询的处理.

由于子图查询要查询 XML 文档中匹配查询的所有子图,而子图匹配问题是 NP-完全问题.这一固有难度,使得难以找到高效的子图查询处理算法.目前,图结构 XML 文档上子图查询处理方法的研究还很少见.而且,图结构 XML 数据上现有的子图查询处理方法也仅能处理几类特殊形式的子图查询.文献[2]提出的 StackD 算法能够处理 DAG 结构 XML 数据上的树结构查询,这种方法是 TwigStack[3]算法的拓展.然而,StackD 是为 DAG 上树结构查询的处理而设计的,不能高效地处理有环图上的一般子图查询.并且,当图中存在大量边时,StackD 因需要维护数据结构而耗费很大的内存空间,使得这种方法难以适用.文献[4]给出了 TwigStack 的另一种改进,但该方法只能处理 St-平面图 XML 数据上的子图查询,不能处理其他图结构 XML 数据上的子图查询.

本文基于文献[5]中 DAG 上编码的一种扩展,提出了图结构 XML 数据上子图查询处理的高效算法.本文先给出一种基于哈希的算法来处理图结构 XML 数据上可达查询;然后给出该算法的两种扩展,分别用来处理具有多个祖先可达查询和具有多个后代的可达查询;在这两个扩展算法的基础上,再给出处理二分图形式的子图查询的算法;最后,提出了一般 DAG 子图查询的处理策略.该策略的基本思想是,基于本文给出的代价模型和启发式规则,将查询中的 DAG 拆分成一系列二分子图,形成较优的查询计划,再用二分子图查询算法处理的每个子查询,最后连接子查询的查询结果得到最终的查询结果.此外,本文还讨论了如何简单地修改 DAG 子图查询处理算法以处理一般的子图查询.

本文的主要贡献可以归纳为如下 3 点:(1) 基于文献[5]中的可达编码,本文提出了一系列基于哈希的算法来实现子图查询处理中的基本操作——几类特殊形式的子图查询;(2) 本文提出了基于基本操作的一般子图查询处理策略,给出了查询计划的代价模型、查询优化算法和选择查询计划的一组启发式规则;(3) 实现了本文提出的子图查询处理算法,并用 XMark 测试集和人工数据集验证了算法的有效性和高效性.

本文第 1 节给出相关定义及预备知识.第 2 节给出处理可达查询的 HGJoin 算法.第 3 节扩展 HGJoin 以处理二分图查询.第 4 节提出处理一般 DAG 子图查询的算法,给出了查询计划模型及其代价模型,讨论生成查询计划的加速策略.第 5 节讨论实验结果.第 6 节给出本文的结论.

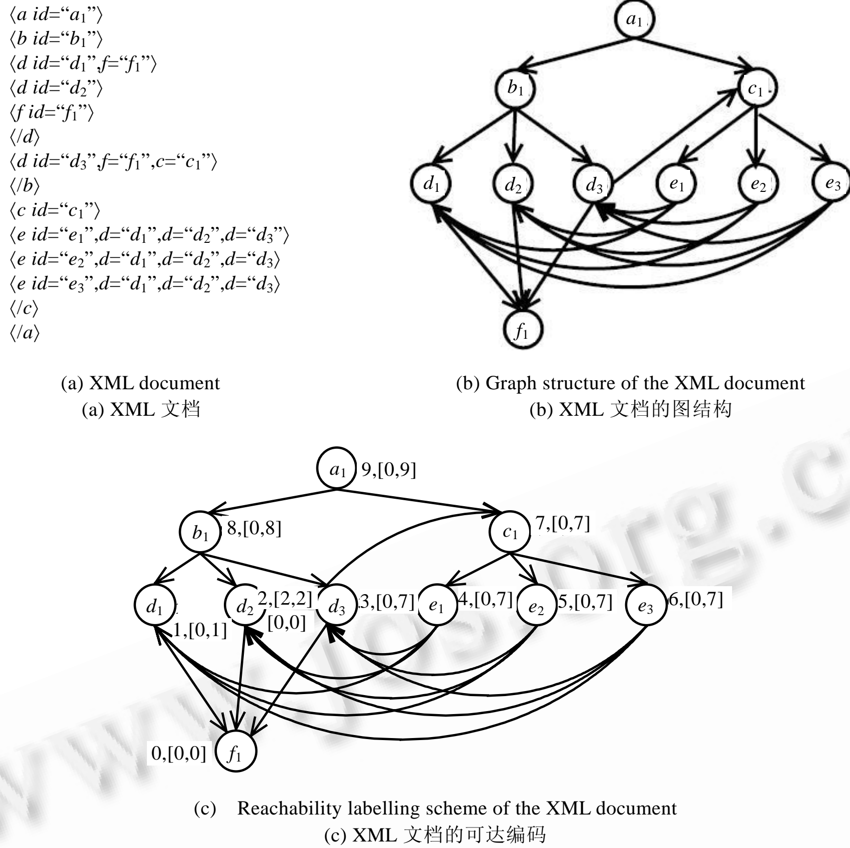
这里介绍 XML 数据编码的相关工作.图上可达编码的研究包括文献[5−11].文献[6]提出了 2-hop 可达编码.文献[7]使用 2-hop 编码来处理图结构 XML 数据上的可达查询.文献[8]通过寻找稠密子图来近似计算 2-hop 编码的方法.文献[9]提出了 HLSS 编码,这种编码首先通过遍历生成树为每个结点求得(preorder,postorder),然后为不在生成树中的边计算 2-hop 编码.与此类似,文献[10]中也先通过遍历生成树得到每个结点的(preorder,postorder),然后为不在生成树中的边计算传递闭包矩阵并利用(preorder,postorder)缩减传递闭包矩阵得到最终的编码.本文算法基于文献[11]中编码的一个扩展,与文献[6−10]中的编码相比,该编码与邻接编码兼容,且利用它来判断可达关系时可以有效避免集合比较与矩阵查找,因此可以用来处理同时带有邻接关系和可达关系的子图查询;此外,在不损失其特征的情况下,该编码可以用来处理有环的图.这正是本文选用这一编码的原因.

1 预备知识

由于 XML 数据的两个结点之间可能存在 IDREF 表示引用关系[12],XML 数据可以看成有标签的有向图.其中,XML 数据中的元素和属性映射为图上的结点,直接的嵌套关系和引用关系映射为图上的边.一个 XML 数据片段如图 1(a)所示,其图结构如图 1(b)所示.

在图结构的 XML 数据中,可以根据结点之间的结构关系定义结构查询.图结构 XML 数据中结点间的结构关系主要有邻接关系和可达关系,在 XML 数据的图结构 G 中,两个结点 a 和 b 满足邻接关系当且仅当 G 中存在从 a 到 b 的边,两个结点 a 和 b 满足可达关系当且仅当 G 中存在从 a 到 b 的路径.可达查询 $a\rightarrow d$ 是查找 G 中所有结点序对$\langle n_a,n_d\rangle$,其中,n_a 的标签为 a,n_d 的标签为 d,且在 G 中 n_a 和 n_d 满足可达关系.例如,在图 2(b)所示的图上执行可达查询 $a\rightarrow e$ 得到的结果包含$\langle a,e_1\rangle,\langle a,e_2\rangle,\langle a,e_3\rangle$.类似地,可以定义邻接查询.可达查询和邻接查询的复合构成了子图查询.下面给出子图查询的正式定义.

© 中国科学院软件研究所 http://www.jos.org.cn



(a) XML document
(a) XML 文档

(b) Graph structure of the XML document
(b) XML 文档的图结构

(c) Reachability labelling scheme of the XML document
(c) XML 文档的可达编码

Fig.1 An example of graph-structured XML data and its coding
图 1 图结构 XML 数据及其编码的例子

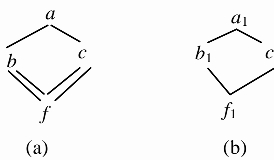


Fig.2 Example queries
图 2 示例查询

定义 1. 子图查询是一个有标签的有向图 $Q=(V,E,tag,rel)$,其中,V 是有向图的结点集,E 是有向图的边集,函数 $tag:V\rightarrow TAG$ 是标签函数(TAG 是标签集),函数 $rel:E\rightarrow AXIS$ 指明查询中结点间的结构关系($AXIS$ 是结点之间关系类型的集合,如 $PC\in AXIS$ 是邻接关系,$AD\in AXIS$ 是可达关系).有向图(V,E)称为查询图.若 $ab\in E$ 且 $rel(ab)=PC$,则称 ab 为邻接边.若 $ab\in E$ 且 $rel(ab)=AD$,则称 ab 为可达边.V 中入度为 0 的结点称为源,V 中出度为 0 的结点称为汇.为简单计,子图查询记为 $Q=(V,E)$.

定义 2. 图 $G=(V_G,E_G)$上的子图查询 $Q=(V_Q,E_Q)$的结果是 $R_{\{G,Q\}}=\{g=(V_g,E_g)|V_g\subseteq V_G$,且存在双射 $f_g:V_g\rightarrow V_Q$ 和单射 $p_g:V_g\rightarrow V_G$ 使得:$\forall n\in V_g,tag(n)=tag(f_g(n))=tag(p_g(n));\forall e=(n_1,n_2)\in E_g,p_g(n_1),p_g(n_2)$满足关系 $rel(f(n_1),f(n_2))\}$.

例如,图 2(a)给出了一个子图查询,其中单线边是邻接边,双线边是可达边.在如图 2(b)所示的图结构 XML 文档上执行该查询的结果如图 2(b)所示.

对图结构 XML 数据进行恰当编码可以快速判定 XML 数据中任意两个结点间的可达关系,进而高效处理子图查询.本文使用文献[11]给出的可达编码的一种扩展[5].此编码赋予图中的任一结点 n 一个 $n.id$ 和一系列区间 I_n.文献[5]证明了图中的两个结点 a 可达 b 当且仅当 $b.id$ 属于 a 的某个区间.例如,图 1(b)中的可达编码如图 1(c)所示,由于结点 f_1 的 id 属于 d_2 对应的区间[0,0]中,可以断定,d_2 和 f_1 满足可达关系.

为了根据可达编码快速选出满足指定可达关系的结点对,我们对可达编码采取了如下的存储策略:对于 XML 文档的标签 TAG 中的每个标签 t,存储两个序列 $t.Alist$ 和 $t.Dlist$.设 N_t 表示标签为 t 的结点构成的集合,$t.Dlist$

© 中国科学院软件研究所 http://www.jos.org.cn

是集合$\{n.id|n\in N_t\}$中的元素递增排序后得到的序列,$t.Alist$ 是集合$\{\langle val,n.id\rangle|n\in N_t,[x,y]\in I_n,val=x$ 或 $val=y\}$中的元素按照第 1 个分量递增排序后得到的序列.由编码性质,任意结点 n 的区间集合 I_n 中没有相交的区间,故 $tag(n).Alist$ 有如下性质,它是本文算法的基础.经过预处理的 Alist 和 Dlist 都有序,在处理过程中无需再排序.

性质 1. 设 n 是的一个结点,$tag(n).Alist$ 中第 2 分量等于 $n.id$ 的所有元组构成子序列 $(val_{i_1},n.id),\ldots,(val_{i_k},n.id)$,则不存在 $1\le s\le u\le r\le v\le k$,使得$[val_{i_s},val_{i_r}]$和$[val_{i_u},val_{i_v}]$或$[val_{i_s},val_{i_v}]$和$[val_{i_u},val_{i_r}]$同属于 I_n.

为了在查询处理算法中判断同一个结点的区间是否处理结束,对于每个结点 n,在 $tag(n).Alist$ 中第 2 分量等于 $n.id$ 的最后一个元组之后添加元组$(null,id)$,其中 $null$ 表示空.为了分析算法的复杂性,令 $N=\max_{t\in TAG}|\{n\in V_G|n$ 的标签为 $t\}|$.任 Alist 中元组的第 2 分量构成的集合记为 ID_A.显然,$|ID_A|\le N,|Dlist|\le N$ 对任意 Alist 和 Dlist 均成立.

2 基于哈希的连接算法(HGJoin 算法)

本节提出了基于哈希的连接算法来利用可达编码处理图结构 XML 数据上形如 $a\rightarrow d$ 的可达查询.

根据存储策略,令 $Alist=tag(a).Alist$ 且 $Dlist=tag(d).Dlist$.直观上,对于 $Dlist$ 中的任意元素 id_i,如果 $Alist$ 中存在两个元组$(x,id_i),(y,id_i)$使得 $x\le id_i\le y$ 且$[x,y]$是某个结点的编码区间,则(id_i,id_j)属于查询结果集.性质 1 保证,只需对 $Alist$ 和 $Dlist$ 交替扫描一遍即可求得查询结果集,无需对这两个序列进行多遍扫描.在扫描过程中,利用哈希表 H 来存储满足如下条件的结点 n 的 id:I_n 中某个区间的开始位置 x 已经被扫描,但尚未遇到相应区间的结束位置 y.性质 1 保证,在遇到相应的 y 值之前,不会遇到 I_n 中其他区间的开始位置;在遇到相应的 y 值时,$n.id$ 将从 H 中移除.具体作法是,为 $Alist$ 和 $Dlist$ 分别设置扫描指针 a 和 d.在扫描 $Alist$ 的过程中,如果当前元组(val_a,id_a)满足 $val_a\le id_d$ 且 $id_a\notin H$,则说明扫描到可能包含 id_d 的某区间的开始位置,将 id_a 添加到 H,更新 a;如果 $val_a<id_d$ 且 $id_a\in H$,则说明扫描到不可能包含 id_d 的某区间的结束位置,由于 $Dlist$ 中的元素递增有序,该区间也不可能包含 $Dlist$ 中其他未被扫描的元素,故从 H 中移除 id_a,更新 a;如果 $val_a=id_d$ 且 $id_a\in H$,或 $val_a>id_d$,则说明 id_d 含于以 val_a 为结束位置的区间中,转而扫描 $Dlist$ 输出部分结果.在扫描 $Dlist$ 的过程中,如果 $id_d<val_a$,或 $val_a=id_d$ 且 $id_a\in H$,根据 H 存储的元素性质对$\forall id\in H$ 输出(id,id_d),更新 d;如果 $id_d>val_a$,或 $val_a=id_d$ 但 $id_a\notin H$,则说明 val_a 可能是包含 id_d 的一个区间的起始位置,转而扫描 $Alist$.扫描过程见算法 1.

算法 1. HGJoin 算法.

输入:$Alist$:祖先结点编码序列,$Dlist$:后代结点编码序列;

输出:$OutputList$:满足可达关系的序对集合.

```
a←Alist.begin,d←Dlist.begin;
while a≠Alist.end AND d≠Dlist.end do
if val_a≤id_d then
    if id_a∉H then
        H.insert(id_a);
        a=a.nextNode;
    else if val_a<id_d then
        H.delete(id_a);
        a=a.nextNode;
    else
        for ∀id∈H do
            把(id,id_d)加入 OutputList;
        d=d.nextNode;
else            /* id_d<val_a */
    for ∀id∈H do
        把(id,id_d)加入 OutputList;
```

www.jos.org.cn

$d=d.nextNode$.

下面分析 HGJoin 算法的时间复杂性和空间复杂性.HGJoin 算法的时间开销 $cost$ 由 3 部分构成:操作 H 的代价、磁盘 I/O 的代价和输出结果的代价.对 $Alist$ 中的每个区间起始位置,对 H 进行一次插入,其代价记为 $cost_I$;在每个区间结束位置,对 H 进行一次删除,其代价记 $cost_D$.故操作 H 的代价为$(|Alist|/2)\cdot cost_H$,其中,$cost_H=cost_I+cost_D$.设$|e_A|$和$|e_D|$分别是 $Alist$ 和 $Dlist$ 中每个元组的大小,$|B|$是磁盘块大小,$cost_{I/O}$ 是存取每个磁盘块的代价,则磁盘 I/O 代价为$(|Alist|\cdot|e_A|/|B|+|Dlist|\cdot|e_D|/|B|)\cdot cost_{I/O}$.输出结果的代价表示为$|result|\cdot cost_o$,其中,$cost_o$ 是输出一个元组的代价.故 $Cost=\frac{|Alist|}{2}\cdot cost_H+|result|\cdot cost_o+\left(\frac{|Alist|\cdot|e_A|}{|B|}+\frac{|Dlist|\cdot|e_D|}{|B|}\right)\cdot cost_{I/O}$.

HGJoin 算法的空间开销主要是哈希表 H 的开销,因此空间复杂性即为算法运行过程中 H 的最大体积.最坏情况下,ID_{Alist} 中的元素同时存在于 H 中;此时,$Dlist$ 中某个 id 对应的结点是 $Alist$ 中所有不同 id 对应结点的共同后代.因此,HGJoin 算法的空间开销不超过 N.

3 HGJoin 算法的扩展

本节给出 HGJoin 算法的两种扩展——IT-HGJoin 和 T-HGJoin.IT-HGJoin 用来处理具有多个祖先和一个后代的子图查询(如图 3(a)所示),T-HGJoin 用来处理有一个祖先和多个后代的子图查询(如图 3(b)所示).

(a) IT-Query
(a) IT-查询
(b) T-Query
(b) T-查询

Fig.3 Two special subgraph queriess
图 3 两类特殊的子图查询

定义 3. 对于子图查询 $Q=(V,E,tag,rel)$,如果 $V=V_s\cup\{d\},d\notin V_s$,且 $E=\{(a,d)|a\in V_s\}$,则称 Q 为一个 IT-查询.如果 $V=\{a\}\cup V_s,a\notin V_s$,且 $E=\{(a,d)|d\in V_s\}$,则称 Q 为一个 T-查询.

3.1 IT-HGJoin算法

设 IT-查询 Q 的源结点分别为 $a_1,\ldots,a_k$,汇结点为 d,令 $Alist_i=tag(a_i).Alist,Dlist=tag(d).Dlist,i\in\{1,2,\ldots,k\}$.与 HGJoin 算法类似,IT-HGJoin 算法利用性质 1 交替地扫描 $Alist_1,\ldots,Alist_k$ 和 $Dlist$ 一遍,得到 IT-查询的查询结果;扫描过程中,每个 $Alist_i$ 使用各自的哈希表 H_i,其功能同 HGJoin 算法中的 H.在算法中,游标 $l_1,l_2,\ldots,l_k$ 分别指向 $Alist_1,Alist_2,\ldots,Alist_k$ 中的当前扫描位置,l 指向 $Dlist$ 中的当前扫描位置.算法依次扫描 $Alist_1,Alist_2,\ldots,Alist_k$,将其中所有满足 $val_{l_i}\le id_l$ 且 $id_{l_i}\notin H_i$ 的序对(val_{l_i},id_{l_i})中的 id_{l_i} 插入 H_i 中,并将满足 $v_{l_i}<id_{l_i}$ 且 $id_{l_i}\in H_i$ 的序对(val_{l_i},id_{l_i})中的 id_{l_i} 从 H_i 移出.处理完 $Alist_k$ 之后,转而扫描 $Dlist$,其过程类似于 HGJoin 扫描 $Dlist$ 的过程;此时,每个 H_i 中的 id 对应的结点都是 id_l 对应结点的祖先,若所有 H_i 都不为空,则说明 id_l 对应结点及相应的祖先结点构成满足 IT-查询的子图,故需要输出这些子图;由于 $H_1\times H_2\times\ldots\times H_k$ 中每个元组对应着 id_l 的一组不同祖先,输出 $H_1\times H_2\times\ldots\times H_k\times\{id_l\}$ 中的所有元组;输出这部分结果之后,再转而依次扫描 $Alist_1,\ldots,Alist_k$.实现 IT-HGJoin 算法时,$|H_1\times H_2\times\ldots\times H_k|$可能很大,为了有效地存储查询结果,可以采用延迟处理策略.即,先分别存储 $H_1,\ldots,H_k$ 和相应的 id_l,直到需要使用查询结果时再进行笛卡尔积.

显然,最坏空间复杂度为 kN.类似于 HGJoin 算法的分析,可以得到 IT-HGJoin 算法的时间复杂度为

$$Cost=\sum\frac{|Alist_i|}{2}\cdot cost_H+|result|\cdot cost_{output}+\left(\frac{\sum|Alist_i|\cdot|e_A|}{|B|}+\frac{|Dlist|\cdot|e_D|}{|B|}\right)\cdot cost_{I/O}.$$

3.2 T-Join算法

设 T-查询的源结点为 a,汇结点分别为 $d_1,\ldots,d_k$.令 $Alist=tag(a).Alist,Dlist_i=tag(d_i).Dlist,i\in\{1,2,\ldots,k\}$.T-查询中

© 中国科学院软件研究所 http://www.jos.org.cn

源结点 a 有多个后代结点 $d_1,\dots,d_k$,然而在可达编码中,标签分别为 $tag(d_1),\dots,tag(d_k)$的那些结点的 id 不一定同时属于标签为 $tag(a)$的结点的同一区间.故为了得到 T-查询的查询结果,必须先获得所有可达查询 $a\rightarrow d_i$ 的结果(其中,$i\in\{1,\dots,k\}$),再在这些中间结果上进行连接操作.

处理可达查询 $a\rightarrow d_i$ 时,为了提高效率,可将 k 个扫描过程复合在一起,即在扫描 Alist 的过程中同时处理 $Dlist_1,\dots,Dlist_k$,这样,扫描所有序列一遍即可获得所需的所有中间结果.为了提高连接操作的效率,为每个 $Dlist_i$ 建立一个存储中间结果的哈希表 IHT_i.当 IHT_i 中某个桶满,则将该桶内的中间结果按照第 1 分量排序后写到磁盘上.扫描得到中间结果后,将 $IHT_1,\dots,IHT_k$ 中所有形如$(id,id_1)\in IHT_1,\dots,(id,id_k)\in IHT_k$ 的元组连接得到形如$(id,id_1,\dots,id_k)$的元组输出即得到 IT-查询的查询结果.显然,连接操作的代价随$|IHT_i|$的增长而迅速增大.为了减小连接操作的代价,应该尽量减小连接操作执行时$|IHT_i|$的大小.

T-Join 算法采取的策略是,在扫描 $Alist$ 的过程中,如果遇到 id 的结束标记$(null,id)$,见第 2.3 节,则表明在以后的扫描中不可能再产生形如$(id,*)$的中间结果插入 IHT_i 中,故可以在当前 $IHT_1,\dots,IHT_k$ 中进行连接产生所有形如$(id,id_1,\dots,id_k)$的查询结果,然后将 $IHT_1,\dots,IHT_k$ 中形如$(id,*)$的中间结果删除,合并相应磁盘块.

尽管上述策略使得连接操作时$|IHT_i|$最小,但频繁的连接操作仍使得算法的时间代价比较大,并且每次连接操作后可以合并的磁盘块非常有限.为了进一步提高算法性能,T-Join 算法还使用“延迟连接”策略.即, T-Join 算法维护一个固定大小的祖先表 A,在扫描 $Alist$ 的过程中,如果遇到 id 的结束标记$(null,id)$,则将 id 插入 A 中;直到 A 满或 Alist 扫描完成时,才在当前 $IHT_1,\dots,IHT_k$ 中进行连接产生所有形如$(id,*,\dots,*)$的查询结果.其中,id 是 A 中任意的元素.然后,将 $IHT_1,\dots,IHT_k$ 中形如$(id,*)$的中间结果删除,并对任意 IHT_i 的每个桶合并可以合并的磁盘块.当然,实现时为了减小存储查询结果的空间开销,还可以采取 IT-HGJoin 类似的延迟处理策略.

下面分析 T-HGJoin 算法的时间复杂性和空间开销.T-HGJoin 算法的时间开销包含 4 个部分:操作 H 的代价、访问 $Alist$ 和所有 $Dlist_i$ 的磁盘 I/O 代价、处理中间结果的代价和输出最终结果的代价.前两部分及最后一个部分的代价分析类似于 HGJoin 算法的分析.在最坏情况下,$Alist$ 中的元素均为每个 $Dlist_i$ 中任意元素的祖先且 $Alist$ 中的元素也存在祖先-后代关系,此时$|IHT_i|\le N^2$.在中间结果上进行操作的代价包括存储中间结果的代价、连接中间结果的代价和删除中间结果的代价.设哈希函数均匀,IHT_i 中哈希桶个数为 n_b,IHT_i 中每个元组的大小为$|entry_{IHT}|$,则存储中间结果的代价是 $k\cdot N^2\cdot cost_I+k\cdot(|entry_{IHT}|\cdot N^2/|B|-n_b)\cdot cost_{I/O}$,记作 C_{in};删除中间结果的代价和存储中间结果的 I/O 代价相同.在哈希函数均匀的条件下,连同内存中的块,IHT_i 中的每个桶有$|entry_{IHT}|\cdot N^2/(|B|\cdot n_b)$个磁盘块.对于祖先表 A 中的每个 id,连接操作需要访问每个 IHT_i 中与 $hash(id)$关联的桶的所有磁盘块,故每次连接操作的代价为$\{[|entry_{IHT}|\cdot N^2/(|B|\cdot n_b)-1]^k\cdot cost_{I/O}+[N^2/n_b]^k\cdot cost_{join}\}\cdot|A|$,记作 C_J.其中,$cost_{join}$ 是产生一个输出的代价.连接操作执行的次数为$|ID_{Alist}|/|A|$,不超过 $N/|A|$.综上所述,T-HGJoin 有的最坏运行时间 k 开销为 $Cost=\dfrac{|Alist|}{2}\cdot cost_H+\left(\dfrac{|Alist|\cdot|e_A|}{|B|}+\dfrac{\sum|Dlist_i|\cdot|e_D|}{|B|}\right)\cdot cost_{I/O}++k\cdot N^2\cdot cost_I+C_{in}+C_J+|result_F|\cdot cost_o$.

T-HGJoin 算法的内存开销,包括扫描过程中为 $Alist$ 分配的哈希表 H 的开销和存储中间结果的所有 IHT_i 的内存开销.使用前面的记号,T-HGJoin 算法的内存开销为 $N+k\cdot n_b\cdot|B|$.

3.3 Bi-HGJoin算法

本节给出二分图形式查询的处理算法,它是 T-HGJoin 和 IT-HGJoin 的组合.首先,我们给出二分图中所有后代具有相同祖先这种特殊查询(简称 CBi 查询)的处理算法,然后讨论一般二分图查询的处理策略.

设 CBi 查询的源分别为 $a_1,\dots,a_m$,汇分别为 $d_1,\dots,d_n$.令 $Alist_i=tag(a_i).Alist,Dlist_j=tag(d_j).Dlist,i\in\{1,2,\dots,m\}$, $j\in\{1,2,\dots,n\}$.在算法中,游标 $l_1,l_2,\dots,l_m$ 分别指向 $Alist_1,Alist_2,\dots,Alist_m$ 的当前扫描位置,$t_1,t_2,\dots,t_n$ 分别指向 $Dlist_1$, $Dlist_2,\dots,Dlist_n$ 的当前扫描位置.与 IT-HGJoin 类似,算法为每个源结点 a_i 分配一个哈希表 H_i.与 T-HGJoin 类似,算法为查询图中的每个汇 d_j 对应的中间结果分配一个哈希表 IHT_j.

Bi-HGJoin 算法包括两个交替的步骤:在第 1 个步骤中,与 IT-HGJoin 类似,算法依次扫描 $Alist_1,\dots,Alist_m$,将满足 $val_{l_i}\le\min(id_{t_j})(1\le j\le n)$且 $id_{l_i}\notin H_i$ 的序对(val_{l_i},id_{l_i})中的 id_{l_i} 插入 H_i 中.处理完 $Alist_m$ 之后,转而处理最小 id_{t_j}

值对应的 $Dlist_j$,若哈希表 $H_1,\ldots,H_m$ 非空,则将 $H_1\times H_2\times\ldots\times H_m\times\{id_{t_j}\}$中的每个元组$(h_1,h_2,\ldots,h_m,id_{t_j})$插入 IHT_j 对应哈希值 $hash(h_1,h_2,\ldots,h_m)$的桶中;第 2 个步骤和 T-HGJoin 的中间结果连接类似,当所有 $Alist_i$ 均已扫描到 h_i 的结束标记$(null,h_i)$后(其中,$1\le i\le m$),祖先组合$(h_1,h_2,\ldots,h_m)$被插入祖先表 A 中;当 A 满或所有 $Alist_i$ 均被扫描完时,则对 A 中的每个组合$(h_1,h_2,\ldots,h_m)$,在 $IHT_1,\ldots,IHT_n$ 中哈希值 $hash(h_1,h_2,\ldots,h_m)$对应的桶上做连接操作,得到形如$(h_1,h_2,\ldots,h_m,id_1,\ldots,id_n)$的查询结果,然后删除相应的中间结果并合并这些桶的磁盘块.连接操作完成后,继续执行第 1 个步骤.重复上述过程,直到所有 $Alist_i$ 被扫描完.

该算法不能处理一般的二分图查询,由于各个汇对应的源可能不同,一般二分图的处理将在下一节讨论.

4 一般 DAG 子图查询的处理算法

直接处理一般 DAG 子图查询不仅需要大量的内存空间,其间生成的大量中间结果还要使用大量的磁盘空间,其效率很低.因此,本节给出的处理策略不是直接处理一般 DAG 子图查询,而是将一般 DAG 子图查询分解为若干 CBi 子图查询,通过分别处理每个 CBi 子查询并对结果进行过滤,得到较小的中间结果,然后在中间结果上执行连接操作得到最终查询结果,连接操作用 sort-merge 算法实现.例如,处理图 4(a)中的查询时,可以先将查询分解成图 4(b)中的查询 q_{11}、图 4(c)中的查询 q_{12} 和可达查询 $c\rightarrow e$.然后,利用 q_{11} 生成的中间结果对标签 $tag(b)$的编码数据和标签 $tag(c)$的编码数据进行过滤.再在过滤后的编码数据上处理子查询 q_{12} 得到相应的中间结果,在过滤后的编码数据上处理可达查询得到相应的中间结果.最后,在 3 个中间结果上进行连接得到最终结果.

显然,上述策略的关键是如何分解查询并构建形如图 4(d)所示的查询计划.查询计划可以表示成一个 DAG $D=(V,E)$,V 中的每个顶点表示一种操作(可能出现的操作包括 HGJoin,IT-HGJoin,T-HGJoin 和 Bi-HGJoin,$Filter_x$ 和 sort-merge 操作),箭尾处的操作结果是相应箭头指向的操作的输入.例如在图 4(d)中,$T\text{-}HGJoin_{\{(a,b),(a,c)\}}$表示在标签 $tag(a)$,$tag(b)$,$tag(c)$的编码数据上执行 T-HGJoin 算法,$Filter_c$ 表示在标签 $tag(c)$的编码数据上过滤掉未出现在 $T\text{-}HGJoin_{\{(a,b),(a,c)\}}$结果中的编码数据,其他操作的意义类似.

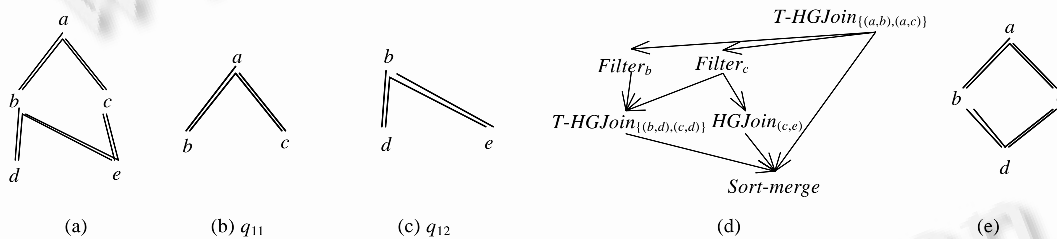


Fig.4 An example of query plan

图 4 查询计划的例子

4.1 代价模型

子图查询的查询计划有多种选择.为生成高效的查询计划,需要进行查询优化.作为查询优化的基础,本节给出查询计划的代价模型.对查询计划中的每个操作,其代价由执行时间和所需内存的大小两部分构成.前者考虑了查询计划的执行效率,后者考虑执行查询计划所需的内存空间.由于 sort-merge 操作的代价估计在关系数据库中已被广泛深入研究,HGJoin 和 IT-HGJoin 的时间代价估计按照第 2 节和第 3.1 节的时间复杂性分析进行,这里仅给出 T-HGJoin 和 Bi-HGJoin 的代价模型.

T-HGJoin 的代价模型和第 3.2 节中的时间复杂性分析类似.利用中间结果大小估计技术[13,14]可以对 $\forall a\in ID_{Alist}$ 和$\forall i\in\{1,2,\ldots,k\}$估计出 $Alist$ 和 $Dlist_i$ 连接后中间结果中与 a 相关的元组个数 $P_{a,i}$,进而得到 $Alist$ 和 $Dlist_i$ 连接后中间结果中元组的个数的估计 $sel(A,D_i)=\sum_{a\in ID_{Alist}}P_{a,i}$.此外,用该技术还可以估计在 $Alist$ 中访问到$(null,a)$的时刻 IHT_i 中含有与 a 相关的中间结果的磁盘块数量 $S_{a,i}$.因此,与 a 相关的所有中间结果的个数可以估计为 $S_a=\sum_{i\in\{1,\ldots,k\}}S_{a,i}$;在连接与 a 相关的中间结果时,需要以 Nest_loop 的方式访问分别分布于 $IHT_1,\ldots,IHT_k$

© 中国科学院软件研究所 http://www.jos.org.cn

中的 $S_{a,1},\dots,S_{a,k}$ 个磁盘块,其间需要访问的磁盘块数量可以估计为 $NL_a=\prod_{i\in\{1,\dots,k\}}S_{a,i}$.基于这些估计,T-HGJoin 操作的时间代价可以估计为

$$Cost=\frac{|Alist|}{2}\cdot cost_H+\left(\frac{|Alist|\cdot|e_A|}{|B|}+\frac{\sum|Dlist_i|\cdot|e_D|}{|B|}\right)\cdot cost_{I/O}+$$
$$cost_I\cdot\sum_{i=1}^{k}sel(A,D_i)+2\cdot\sum_{a\in ID_{Alist}}S_a\cdot cost_{I/O}+\sum_{a\in A}NL_a+|result_F|\cdot cost_o.$$

对 Bi-HGJoin 操作时间代价的估计与 T-HGJoin 类似.将 $Alist_1,\dots,Alist_m$ 和 $Dlist$ 进行连接生成的中间结果代价记为 $selectivity(A_1,A_2,\dots,A_m,D_i)$,与结点组合 $a_1,a_2,\dots,a_m(a_i\in ID_{Alist_i})$相关的所有中间结果个数记为 $S_{a_1,a_2,\dots,a_m}$,对包含结点组合 $a_1,a_2,\dots,a_m$ 的中间结果进行连接时需要访问的磁盘块数记为 $NL_{a_1,a_2,\dots,a_m}$.这些量的估计方法同 T-HGJoin 代价模型中对应量的估计方法类似.则 Bi-HGJoin 操作的时间可以估计为

$$Cost=\frac{\sum|Alist_i|}{2}\cdot cost_H+\left(\frac{\sum|Alist|\cdot|e_A|}{|B|}+\frac{\sum|Dlist_i|\cdot|e_D|}{|B|}\right)\cdot cost_{I/O}+cost_I\cdot\sum_{i=1}^{n}sel(A_1,\dots,A_m,D_i)+$$
$$2\cdot\sum_{a_1\in A_1,\dots,a_m\in A_m}S_{a_1,\dots,a_m}\cdot cost_{I/O}+\sum_{a_1\in A_{j1},\dots,a_m\in A_{jm}}NL_{a_1,\dots,a_m}+|result_F|\cdot cost_o.$$

操作所需的内存开销包括固定内存和可变内存两部分:固定内存是查询处理中间结果的缓存,其大小等于系统中设定的值 fixed_mm;可变内存是查询处理过程中 Alist 对应的哈希表所用的内存的最大值,其大小与查询处理过程中后代的祖先的最大个数之和呈正比.设 $ancestor_d$ 表示 $d\in Dlist$ 在 Alist 中的祖先个数,$|entry_H|$表示哈希表 H 中数据元素的大小,操作的空间代价为 $cost_{mm}=fixed_mm+\max_{d\in Dlist}(|ancestor_d|)\cdot|entry_H|$.

4.2 查询计划生成算法

本节首先用查询图来表示查询执行的过程,然后在查询图上用最短路径算法得到相应的最优查询计划.

设(V_Q,E_Q)是子图查询 Q 的查询图(记 $m=|E_Q|$),称有向图 $G^*=(V_{G^*},E_{G^*})$是子图查询 Q 的状态图,其中,$V_{G^*}=\{g|g=(V_g,E_g)$且 $E_g\subseteq E_Q\}$,从 $g\in V_{G^*}$ 到 $g'\in V_{G^*}$ 存在一条有向边当且仅当存在一个子查询 $Q_{gg'}=(V_{gg'},E_{gg'})$使得 Q'是可达查询、T-查询、IT-查询和 CBi-查询中的一种且 $E_g-E_{gg'}=E_{g'}$;若$|E_{gg'}-E_g|=i$,则称 g 位于 G^*的第 i 层.显然,G^*中第 0 层只有 1 个顶点,即查询图(V_Q,E_Q)本身.称该顶点为 G^*的初始状态,记为 g_0;G^*的第 m 层也仅含唯一由孤立点构成的图$(V_Q,\varnothing)$,称该顶点为 G^*的终止状态,记为 g_m;V_{G^*} 中的其他元素称为 G^*的中间状态.

显然,G^*中从初始状态 g_0 到终止状态 g_m 的任意一条路径 $g_0=g_{i_1},g_{i_2},\dots,g_{i_k}=g_m(k\le m)$对应了对查询 Q 的一个处理过程,该过程依次处理的子查询是 $Q_{g_{i_j}g_{i_{j+1}}}$ $(1\le j<k)$.该过程对应的查询计划可以如下生成:对于路径上的第 1 条边 $g_{i_1}g_{i_2}$,根据 $Q_{g_{i_1}g_{i_2}}$ 的操作类型 $O\in\mathcal{F}=\{$HGJoin,T-HGJoin,IT-HGJoin,Bi-HGJoin$\}$构造一个操作 O_E(其中,$E=E_{g_{i_1}g_{i_2}}$).设 $g_0=g_{i_1},g_{i_2},\dots,g_{i_k}$ 的查询计划已经得到且将中间结果集构成的集族记为$\mathcal{B}_j$,考察路径中的边 $g_{i_j}g_{i_{j+1}}$,先根据 $Q_{g_{i_j}g_{i_{j+1}}}$ 的操作类型 $O\in\mathcal{F}$在查询计划中添加一个操作 O_E(其中,$E=E_{g_{i_1}g_{i_2}}$);然后考察 $V_{g_{i_1}g_{i_2}}$ 中的每个顶点 a,如果$\mathcal{B}_j$中存在与 a 相关的中间结果集,则在查询计划中添加操作 $Filter_a$ 并从相应的中间结果集引一条边指向操作 $Filter_a$,再从 $Filter_a$ 引一条边指向 O_E;再后,将 O_E 操作的中间结果集添加到$\mathcal{B}_j$中.此时,如果$\mathcal{B}_j$中存在可以合并的中间结果集,则在查询计划中添加操作 sort-merge,并从相应的中间结果集引有向边指向新添加的 sort-merge 操作,同时,将$\mathcal{B}_j$中合并前的中间结果集删除,插入合并后的中间结果集;重复上述过程直到$\mathcal{B}_j$中没有可以合并的中间结果集,令$\mathcal{B}_{j+1}=\mathcal{B}_j$.继续,生成其他边的查询计划,直到路径上所有边均被考虑.

在为路径 $g_0=g_{i_1},g_{i_2},\dots,g_{i_k}=g_m$ 生成查询计划的过程中,我们为 $g_{i_j}g_{i_{j+1}}$ 在查询计划中添加一组操作.考虑这组操作的时间代价之和 $w_{g_{i_j}g_{i_{j+1}}}$ 和空间代价的最大值 w.如果 w 小于等于可用内存空间大小,则令边 $g_{i_j}g_{i_{j+1}}$ 的权值等于 $w_{g_{i_j}g_{i_{j+1}}}$;否则,表明这组操作不可行,令边 $g_{i_j}g_{i_{j+1}}$ 的权值等于$+\infty$.通过该方式,G^*中任意边 gg'上的权值唯一确定,因为无论以什么样的路径到达状态 g,其中间结果集构成的集族是一样的,故为 gg'添加的操作相同.

© 中国科学院软件研究所 http://www.jos.org.cn

基于以上讨论,可以看到,加权查询图 $G^*=(V_{G^*},E_{G^*})$上的一条从 g_0 到 g_m 的路径对应了一个查询计划,查询计划的代价即为路径的加权长度;反之亦然.故,寻找最优的查询计划,即在 G^*上寻找从 g_0 到 g_m 的最短路径,可以用 Dijkstra 算法[15]求解.由 G^*的构造知道$|V_{G^*}|=2^m$,故 Dijkstra 算法求得最短路径的时间复杂度为 $O(2^{2m})$.

4.3 查询优化加速策略

由第 4.2 节得知,查询图 G^*的规模很大,导致 Dijkstra 算法求得优化查询计划的时间复杂度和空间复杂度很大.本节将给出一些启发式规则来加速 Dijkstra 算法的求解过程.

在下面的讨论中,用记号 w_g 表示 g_0 到 g 的当前最短路径的加权长度,其初始值为+∞,每次扩张到达 g 时,更新 w_g.由于 Dijkstra 算法采用 Best-first 的扩展策略(在当前可扩展结点中选取代价最小的状态进行扩展),故在选中状态 g 进行扩展时,从 g_0 到 g 的最短路径已经求得.利用这个性质,可以得到如下规则:

规则 1. 在 Dijkstra 算法运行过程中,如果当前选中的扩展状态 g 等于终止状态 g_m,或者 $w_g\geq w_{g_m}$,则输出 g_0 到 g_m 的当前最短路径并终止.

规则 2. 在 Dijkstra 算法运行过程中,当前选中的扩展状态为 g 且从 g 出发的任意一条有向边 gg'均有 $w_g+w_{gg'}>w_{g'}$成立($w_{gg'}$表示边 gg'的权值),则从 Dijkstra 算法的数据结构中删除 g.

性质 2. 规则 1 和规则 2 不影响查询优化的结果.

直观上,直接执行一个复杂操作的时间开销往往小于将该操作分解成一系列简单操作后的执行时间总和.因此,在 Dijsktra 算法中可以忽略那些后代状态已被扩展到的状态,以节省空间开销并加速查询优化过程.这即为如下规则:

规则 3. 对于 Dijsktra 算法中的当前扩展状态 g 和$\forall gg'\in E_{G^*}$,如果数据结构中已经存在 g'的后代状态,则放弃在 g'上的扩展;否则,根据算法完成在 g'上的扩展.

此外,Dijsktra 算法选中的当前扩展状态 g 可能有很多孩子状态,我们可以只扩展那些选择性较大的孩子状态,以进一步加速查询优化的过程.对于 g 的孩子状态 g',g'的选择性可以定义为边 gg'对应的那组操作的选择性.扩展 g 时,仅扩展其孩子状态中选择性较高的 C 个状态,其中,C 是算法运行时的一个参数.这样得到如下规则:

规则 4. 对于 Dijsktra 算法中的当前扩展状态 g,将其孩子状态选择性大小递减排序,结合强枝剪除规则依次处理每个孩子状态,直到完成 C 次扩展或所有孩子均被处理时,完成对 g 的扩展.

尽管规则 3 和规则 4 可能导致算法得不到最优的查询计划,但这两个规则可以有效地降低查询优化过程的时间复杂度.

性质 3. 利用规则 3 和规则 4,Dijsktra 算法的时间复杂度为 $O(C\cdot 2^m)$.

4.4 方法的扩展

本节对子图查询处理策略进行扩展,以处理带环的子图查询和带有邻接关系的子图查询.

带环的子图查询 Q 可以用如下简单的策略进行处理:删除环上的一条边 ab 并将 ab 添加到集合 D 中;重复上述过程,直到 Q 中不存在环为止.然后,调用前文的算法处理查询 Q 得到中间结果集;最后,对中间结果集中的每个结果,依次验证 D 中各条边的约束是否满足,输出满足 D 中所有边的约束的结果即为最终的查询结果.

下面讨论如何用前文所给算法来处理带有邻接关系的子图查询.由于第 2 节给出的可达编码不能判断邻接关系,为了处理带有邻接关系的子图查询,应该为每一个结点赋予相应的邻接编码.在邻接编码中,对每个后序编码为 i 的结点,区间$[i,i]$被赋予该结点每个父亲.对于标签 $t\in TAG$,令 $t.Plist$ 是集合$\{\langle i,n.id\rangle|n\in N_t,[i,i]$是 n 的连接编码中一个区间$\}$中的元素按照第 1 个分量递增排序后得到的序列.对标签 t,除了存储 $t.Alist$ 和 $t.Dlist$,还需要存储 $t.Plist$.采用这样的存储策略后,邻接关系的判定与判定可达关系完全相同,只是算法采用的输入数据集不同.事实上,对于查询 Q 中的每条边 ad,如果 $rel(ad)=AD$,则在前文的算法中令 $Alist=tag(a).Alist$ 或 $Alist_i=tag(a).Alist$,如果 $rel(ad)=PC$,则在前文的算法中令 $Alist=tag(a).Plist$ 或 $Alist_i=tag(a).Plist$,这样就可以使用前文的各个算法直接处理带有邻接关系的子图查询了.

注意,如果查询中某个祖先结点同时有可达出边和邻接出边,则复制该结点及该结点上的所有入边一次,该

© 中国科学院软件研究所 http://www.jos.org.cn

结点的一个拷贝通过可达出边连接到相应的孩子结点,另一个拷贝用邻接出边连接到相应的孩子结点.然后调用前文的方法进行处理.

5 实验结果

5.1 实验配置

我们用 Visual C++6.0 实现了本文给出的算法,所有 Alist 和 Dlist 被存储为文本文件.一个测试数据集采用 XMark[16],它具有图结构和比较复杂的模式,且带有环;生成数据集时选用的参数依次为 0.1~0.5.

另一个测试数据集是人工数据集.数据生成器有两个参数:第 1 个参数是具有同一标签的结点的数量 N,第 2 个参数是结点间存在边的概率 p(简称边概率).所有随机数据集中的图有 8 个标签{A,B,C,D,E,F,G,H}(标签是有顺序的).随机数据集合中的图可能是 DAG 也可能是一般的图(简称 GG).对于一般图,任意两个结点间存在边的概率是 p;对于 DAG,从标签较小结点到标签较大结点间有边的概率为 p,逆向边出现的概率为 0.

我们用运行时间(简称 RT)作为查询效率的测度.对于 XMark 上的查询,用两个查询来测试每个算法的效率,其中一个查询包含所有不在环中的结点,选择性较小;另一个查询不含环中的结点,选择性较大.为 HGJoin 算法选用的查询是 XMQS1:text→emph 和 XMQS2:person→bold;为 IT-HGJoin 算法和 T-HGJoin 算法选用的查询分别如图 5(a)~图 5(d)所示.为了与文献[2]中的算法进行比较,我们还设计了复杂的树结构查询 XMQW1 和 XMQW2,分别如图 5(g)和图 5(h)所示.为了深入研究查询优化的性能,我们设计了具有复杂结构的查询 XMQC1 和 XMQC2,分别在图 5(i)和图 5(j)中.图中的箭头表示子图查询中的 AD 关系.在随机数据集上,用一个查询来测试每个算法的效率.HGJoin 的查询是 RQS:A→E.IT-HGJoin 和 T-HGJoin 的查询在图 5(e)和图 5(f)中.用来测试查询优化策略的树结构查询 RQW 和复杂查询是 RQC 分别如图 5(k)和图 5(l)所示.

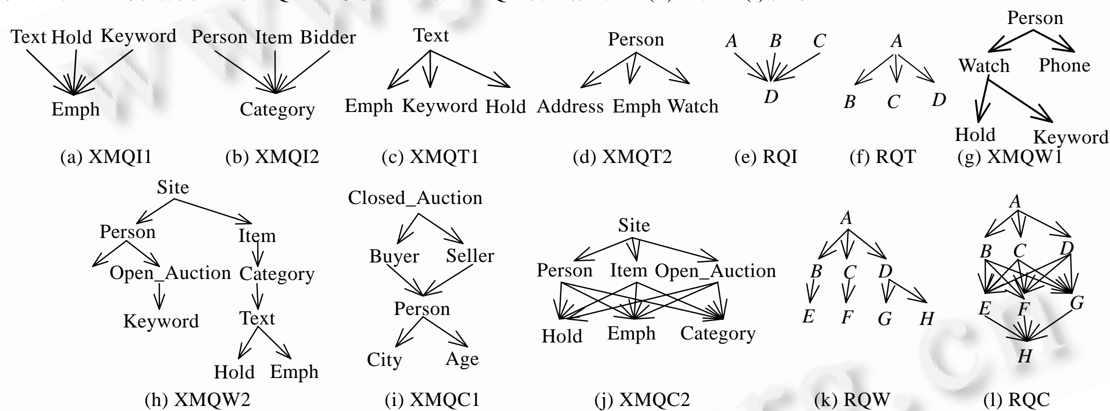


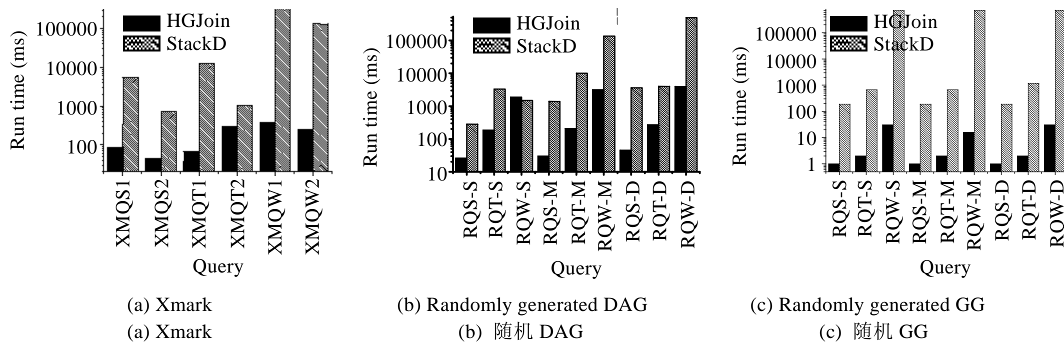
Fig.5 Query set

图 5 查询集合

5.2 与StackD算法的效率比较

据我们所知,StackD[2]算法是当前处理图结构 XML 数据上结构查询最高效的算法.因此,本节比较本文所给算法和 StackD 算法的效率.我们实现了文献[2]中的 StackD 算法.由于 StackD 算法是为树结构查询设计,这里仅比较不同算法执行可达查询、T-查询和子树查询的效率.在 XMark 测试集,N=4096,p 分别为 0.1,0.4,0.8 的 DAG 和一般图上进行实验,结果分别如图 6(a)~图 6(c)中所示.可以看到,本文所给算法的效率显著地超过了 StackD.特别值得注意的是,在边的密度较高的随机图上进行查询时,效率差别最大.出现这一现象的原因有两个:一是当图的边密度很大时,区间被很多结点共享.本文的方法可以将相同标签对应结点的区间统一处理,而 StackD 将其分开处理;二是当结点有很多区间的时候,StackD 需要维护一个很大的数据结构,在其上的操作代价较高.

© 中国科学院软件研究所 http://www.jos.org.cn



(a) Xmark (b) Randomly generated DAG (c) Randomly generated GG

(a) Xmark (b) 随机 DAG (c) 随机 GG

Fig.6 Experimental results for comparisions

图 6 比较实验的结果

5.3 查询优化策略的有效性

我们通过验证查询计划的质量和查询计划的效率来检验查询优化策略的有效性.在本节的实验中,我们把使用 1M 固定内存在 50M 的 XMark 文档上对查询执行查询优化策略.

为了验证查询计划的质量,我们比较 10 个随机查询计划和查询优化策略得到的查询计划的执行时间,结果见表 1.其中,时间单位是 ms;OPT 是利用规则 4 在 C=4 时得到的查询计划的执行时间;MAX,MIN 和 AVG 分别是 10 个随机查询计划执行时间的最大值、最小值和平均值.可以看到,本文提出的查询优化策略可以得到较优的查询计划.

为了验证查询优化策略的效率,我们比较采用不同策略时优化查询 XMQC1 和 XMQC2 的时间与转移数,结果见表 2.其中,$time_i$ 和 $state_i$ 表达了应用策略 i 的查询优化时间和转移数;EXE-Time 为执行规则 4(k=4)得到的查询计划的时间,时间的单位是 ms.可以看到,启发式规则可以有效地减小转移的数量,得到较优的查询计划;并且相对于查询执行时间,查询优化的时间很小.

Table 1 Quality of query plans

表 1 查询计划的质量

Query	OPT	MAX	MIN	AVG
XMQC1	17 328	169 797	55 641	101 715
XMQC2	62 720	234 640	66 953	1 542 434

Table 2 Efficiency of query optimization

表 2 查询优化的效率

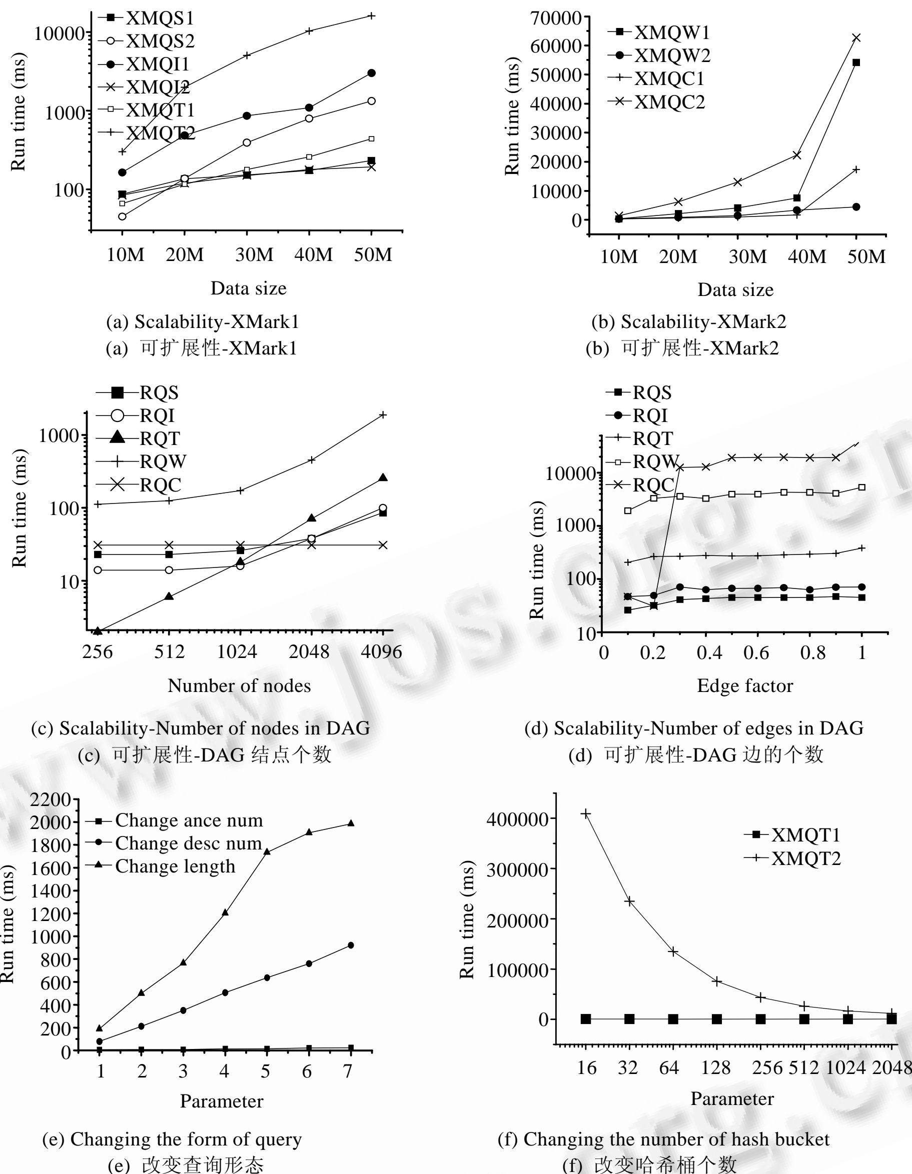
Query	$Time_1$	$Trans_1$	$Time_2$	$Trans_2$	$Time_3$	$Trans_3$	$Time_4$	$Trans_4$	EXE-Time
XMQC1	47	2 860	16	1 249	2	16	1	16	17 328
XMQC2	104 968	4 324 639	16 109	95 941	35	5	34	5	62 720

5.4 参数对算法效率的影响

可扩展性实验测试的是相同查询在模式相同而大小不同的文档上的运行时间.这组实验中,对于 XMark,我们把参数从 0.1 变化到 0.5,结果如图 7(a)和图 7(b)所示.图 7(a)的纵轴是时间的 log 值.可以看到,XMQS1,XMQS2,XMQI1,XMQI2,XMQT1,XMQT2 和 XMQW1 的查询时间和文档大小大致呈线性关系.当文档很大时,XMQW1,XMQW2 和 XMWC1 的处理时间增长很快.这是因为查询的主体部分与 XML 数据中的环和完全二分图相关,这部分的结果是相关结点的笛卡尔积.当结点增多时,最终结果和中间结果的数量增长大于线性.因此,查询时间的增长快于线性.

www.jos.org.cn

© 中国科学院软件研究所 http://www.jos.org.cn



(a) Scalability-XMark1
(a) 可扩展性-XMark1

(b) Scalability-XMark2
(b) 可扩展性-XMark2

(c) Scalability-Number of nodes in DAG
(c) 可扩展性-DAG 结点个数

(d) Scalability-Number of edges in DAG
(d) 可扩展性-DAG 边的个数

(e) Changing the form of query
(e) 改变查询形态

(f) Changing the number of hash bucket
(f) 改变哈希桶个数

Fig.7 Results of changing parameters
图 7 修改参数的实验结果

对于随机的数据集(DAG),固定边概率为 0.1,结点数量从 256 变化到 4 096,结果在图 7(c)和图 7(d)中,运行时间和结点数量的值均是 log 值.与上一段讨论的原因相同,RQS,RQI 和 RQW 的查询处理时间随结点数量的增长快于线形.RQC 处理时间受结点数量的影响不显著,这是因为查询优化为 RQC 选择了自底向上的处理策略,而子查询({E,F,G},H)的选择性非常小.再固定结点数 4 096,,将边概率从 0.1 变化到 1.0,结果在图 7(d)中.可以看到,RQS,RQI,RQT 和 RQW 的运行时间随数据中边的数量影响不大.这是由于当边较密集时,所有结点的区间趋同,这些区间得以在存储安排时被合并.RQC 是个例外,当边概率从 0.2 变化为 0.3 时,运行时间变化非常明显.这是因为 RQC 比较复杂,仅当边的密度超过一个阈值的时候,查询结果的大小变化很大.

对于随机的数据集(GG),一组实验固定边概率为 0.1,结点的数量从 512 变化到 4 096;另一组实验固定图的

www.jos.org.cn

© 中国科学院软件研究所 http://www.jos.org.cn

结点数为 4 096,边概率从 0.1 变化到 1.0.这两组实验的运行时间均很小(0ms~5ms),而且变化不明显.这是因为在这些情况下,图中的所有结点几乎均在同一个强连通分支中.根据存储策略,同一个强连通分支中结点的区间完全相同并且仅包含一个区间.为了节省空间,我们不给出这部分结果.

我们测试了查询形式对算法效率的影响.所有的实验在结点数为 4 096、边概率为 0.1 的随机数据集上运行.我们分别测试在查询祖先数量和后代数量从 1 变化到 7 时 T-HGJoin 和 IT-HGJoin 算法.对应 IT-HGJoin 算法的查询以 H 为后代,并且$\{A\},\{A,B\},\ldots,\{A,\ldots,G\}$分别为祖先集合.对应 T-HGJoin 的查询以 A 为祖先,$\{B\},\{B,C\},\ldots,\{B,\ldots,H\}$分别是后代集合.我们还测试了路径查询的长度对效率的影响.我们将路径查询的长度从 2 变换到 8.查询是 $A\rightarrow B,A\rightarrow B\rightarrow C,\ldots,A\rightarrow\ldots\rightarrow H$.这 3 组实验的结果在图 7(e)中.可以看到,本文算法的查询时间和祖先与后代的数量大致呈线性关系.这是因为利用哈希表,所有祖先和后代可以直接输出而不需要和其他结点进行比较.另外一个结果是,路径查询的运行时间随路径查询的长度慢于线性.这是因为查询优化策略可以从选择性较小的步骤开始,而不是按照路径顺序进行.

哈希表中桶的个数是 T-HGJoin 的一个重要参数,下面的实验研究它对算法效率的影响.我们把桶数从 16 变换到 2 048.在 50M XMark 上运行 XMQT1 和 XMQT2,结果在图 7(f)中.哈希桶的数量对 XMQT1 影响较小.这是因为对应查询中 4 个结点的元素在 XML 图的树结构中,每一个结点的编码仅有一个区间,查询执行时仅需同时处理 3 个区间.而在 XMQT2 中,运行时间和哈希桶个数近似为对数关系.这是因为在处理 XMQT2 的过程中,有很多中间结果存储在哈希表中.更多的桶不仅会减小磁盘 I/O 数量,而且会减小排序和连接的代价.

6 结 论

图结构 XML 文档上的子图查询面临诸多挑战性的研究问题.本文研究了图结构 XML 数据上子图查询高效处理的问题.基于可达编码,本文提出了基于哈希的结构连接算法(HGJoin)来处理图结构 XML 数据上的可达查询;然后对 HGJoin 进行了拓展,分别给出了处理 IT-查询、T-查询和 CBi 查询的算法.以这些算法为基本操作,以相应算法的开销为代价构造代价模型,本文随后提出了一般 DAG 子图查询的处理策略和查询优化策略.本文的方法经简单扩展,还可以直接用于处理带环的子图查询或带邻接关系的子图查询,现有方法不具备这样的特征.实验结果表明,所给查询优化策略可以高效地生成有效的查询计划,本文提出的查询处理策略在时间效率上优于现有算法.本文的未来工作包括设计有效的索引结构来加速查询、设计更加高效的全局算法等.

References:

[1] Rdf/xml syntax specification. 2004. http://www.w3.org/TR/rdf-syntax-grammar/

[2] Chen L, Gupta A, Kurul ME. Stack-Based algorithms for pattern matching on dags. In: Böhm K, Jensen CS, Haas LM, Kersten ML, Larson P, Ooi BC, eds. Proc. of the 31st Int'l Conf. on Very Large Data Bases. Trondheim: ACM, 2005. 493−504.

[3] Bruno N, Koudas N, Srivastava D. Holistic twig joins: Optimal XML pattern matching. In: Franklin MJ, Moon B, Ailamaki A, eds. Proc. of the 2002 ACM SIGMOD Int'l Conf. on Management of Data. Madison: ACM, 2002. 310−321.

[4] Vagena VJTZ, Moro MM. Twig query processing over graph-structured XML data. In: Amer YS, Gravano L, eds. Proc. of the 7th Int'l Workshop on the Web and Databases. 2004. 43−48. http://webdb2004.cs.columbia.edu/papers/proceedings.pdf

[5] Wang H,Wang W, Lin X, Li JZ. Labeling scheme and structural joins for graph-structured XML data. In: Zhang YC, Tanaka K, Yu JX, Wang S, Li ML, eds. Proc. of the 7th Asia-Pacific Web Conf. Shanghai: Springer-Verlag, 2005. 277−289.

[6] Cohen E, Halperin E, Kaplan H, Zwick U. Reachability and distance queries via 2-hop labels. Dey KM, Giesen J, Goswami S, Zhao W, eds. Proc. of the 13th Annual ACM-SIAM Symp. on Discrete Algorithms. San Francisco: ACM/SIAM, 2002. 937−946.

[7] Schenkel GWR, Theobald A. Hopi: An efficient connection index for complex xml document collections. In: Bertino E, Christodoulakis S, *et al*., eds. Proc. of the 9th Int'l Conf. on Extending Database Technology. Heraklio: Springer-Verlag, 2004. 237−255.

[8] Cheng J, Yu JX, Lin X, Wang H, Yu PS. Fast computation of reachability labeling for large graphs. In: Ioannidis YE, Scholl MH, Schmidt JW, *et al*., eds. Proc. of the10th Int'l Conf. on Extending Database Technology. Munich: Springer-Verlag, 2006. 961−979.

© 中国科学院软件研究所 http://www.jos.org.cn

[9] He H, Wang H, Yang J, Yu PS. Compact reachability labeling for graph-structured data. In: Herzog O, Schek HJ, Fuhr N, *et al*., eds. Proc. of the 2005 ACM CIKM Int'l Conf. on Information and Knowledge Management. Bremen: ACM, 2005. 594−601.
[10] Wang H, He H, Yang J, Yu PS, Yu JX. Dual labeling: Answering graph reachability queries in constant time. In: Liu L, Reuter A, Whang KY, Zhang JJ, eds. Proc. of the 22nd Int'l Conf. on Data Engineering. Atlanta: IEEE Computer Society, 2006. 75.
[11] Jagadish HV, Agrawal R, Borgida A. Efficient management of transitive relationships in large data and knowledge bases. In: Clifford J, Lindsay BF, Maier D, eds. Proc. of the 1989 SIGMOD Int'l Conf. on Management of Data. Oregon: ACM, 1989. 253−262.
[12] Bray JP, Sperberg-McQueen CM, Yergeau F. Extensible markup language (XML) 1.0. 3rd ed. In: Proc. of the W3C Recommendation 2004. 2004. http://www.w3.org/TR/REC-xml/
[13] Polyzotis N, Garofalakis MN. Structure and value synopses for XML data graphs. In: Bressan S, Chaudhri AB, Lee ML, Yu JX, Lacroix Z, eds. Proc. of the 28th Int'l Conf. on Very Large Data Bases. Hong Kong: Morgan Kaufmann Publishers, 2002. 466−477.
[14] Polyzotis N, Garofalakis MN, Ioannidis YE. Selectivity estimation for XML twigs. In: Proc. of the 22nd IEEE Int'l Conf. on Data Engineering. New York: IEEE Computer Society Press, 2004. 264−275.
[15] Cormen T, Leiserson C, Rivest R, Stein C. Introduction to Algorithms. Cambridge: MIT Press, 1990.
[16] Schmidt A, Waas F, Kersten ML, *et al*. XMark: A benchmark for XML data management. In: Bressan S, Chaudhri AB, Lee ML, *et al*., eds. Proc. of the 28th Int'l Conf. on Very Large Data Bases. Hong Kong: Morgan Kaufmann Publishers, 2002. 974−985.

王宏志(1978－),男,辽宁丹东人,博士,讲师,主要研究领域为 XML 数据管理,信息集成.

李建中(1950－),男,博士,教授,博士生导师,CCF 高级会员,主要研究领域为数据库系统实现技术,数据仓库,半结构化数据,传感器网络,压缩数据库技术,Web 数据集成,数据挖掘,计算生物学.

骆吉洲(1975－),男,博士,讲师,主要研究领域为压缩数据库.

www.jos.org.cn
www.jos.org.cn