路径表达式 面向对象 数据库系统 (16)

计算机科学1999Vol. 26№. 3

OODBMS

关于路径表达式的一些问题*)

66-69

Some Problems about Path Expression

王欣晖 王国仁 于 戈 郑怀远

(东北大学计算机科学与工程系 沈阳110006)

TP311.13

Abstract Research on path expression is now becoming a hot point in Object-Oriented Database System field. In this paper, first, we introduce the basic conception of path expression and some classic algorithms. Then, discuss search space and it's optimization. Finally, give cost model of two kinds of path expression algorithms.

Keywords Path expression, Cost model, Search space, OODBMS

1 引言

面向对象数据库系统(OODBMS)支持通过引用建立类之间的联系。类的属性可以是原子属性也可以是嵌套属性。由于存在使一个对象能引用另一个对象的嵌套属性,引出了对象导航的概念,OQL中对象导航的特性体现在路径表达式(Path Expression)上。围绕路径表达式各国的学者做了大量研究工作,法国的 Gardarin 等[1]给出了 DFF,BFF 和 RBFF 等三种路径表达式算法的代价模型和实验结果,美国威斯康星大学的 Lieuwen 等[2]给出了基于指针的并行连接算法的详尽的性能分析,德国的 Frohn 等[5]提出了一种基于规则的路径表达式的处理语言 PathLog,我国的学者王国仁等[6]提出了两种并行的处理路径表达式的算法,土耳其的 Ozkan 等[3]提出了一种自学习的路径表达式优化算法,意大利的 Bertino[4]提出了 OODBMS 中路径索引的组织结构。越来越多的事实表明,路径表达式的研究正成为当前 OQL 研究中的一个重点,但是其研究还远远没达到满意的程度。本文对路径表达式的概念、算法、搜索空间和代价模型作一简要论述。

2 路径表达式概述

2.1 路径表达式的概念

越来越多的数据库应用如 CAD,CAM,CASE 和地理信息系统(GIS)等要求 DBMS 能操作更复杂的数据类型表达更丰富的语义,而 OODBMS 则能轻松满足其要求。在 OODBMS 中通过引用(Reference)来表达复杂的对象结构,而且其嵌套长度可以是任意的。于是为了在面向对象数据库中定位某个对象,一种很自然的方式就是路径表达式。用户可以通过命名对象(Named Object)进入数据库,但更为一般的途径是得到一个对象后沿着该对象的导航链得到所要的对象。在 OQL[7]中用户可以用"."来描述复杂对象结构。参见图1,考虑如下的 OQL 查询:

p. lives-in. building. address. street

该查询从一个 Person 开始,遍历 Apartment 和 Building,找出他住处的街道名。这即是一个典型的路径表达式,依据文[7],路径表达式中可以包括复杂属性(非原子属性)和1-1的联系,但不能包括1-n或 n-n 的联系。例如,如果要得到 Person. P 的所有孩子的姓名,不能写成如 p. children. name 这样形式的路径表达式,因为其中的 p. children 是引用的聚集,聚集中的元素是 children 的 OID,路径表达式的处理程序不能识别 OID 类型的结果。对于这样的查询可以采用 SFW 语句处理:

SELECT c. name FROM p. children c

如果在一个路径表达式中包括1-n 或 n-m 的联系,应分成两个子路径表达式使每个表达式中都只包括1-1的联系。由于 OQL 的正交性,路径表达式可以如下所示地出现在 SFW 语句的任何地方:

*)本课题得到第六届霍英东青年基金和辽宁省自然科学基金资助。

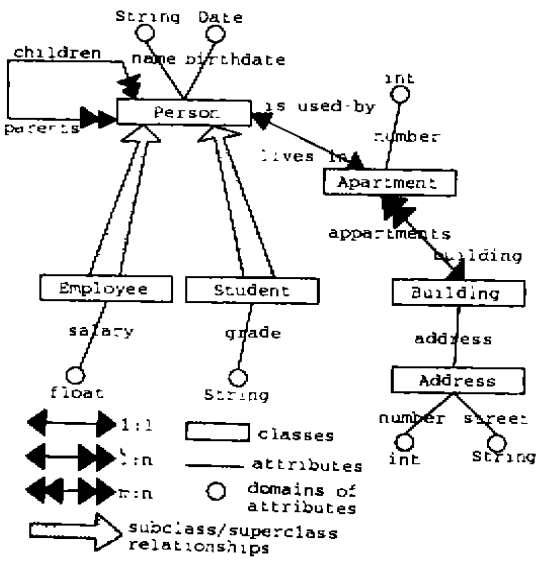

图1

```
SELECT c. lives-in. building. address
FROM p in Persons, c in p. children
WHERE p. lives-in. building. address. street=
        "Main Street";
```

在形式上，出现在 SFW 语句任一子句中的路径表达式是相同的，但实质上它们之间还是有区别的，考虑 p. children 这样简单的路径表达式，它出现在 select 子句和 where 子句中时其值为 p，而出现在 from 子句里时它的值应是 p 的孩子（为简单起见，此处认为 children 不是集合或列表属性）。在文[7]中规定的路径表达式在一定程度上是受限的，如它的谓词只能作用在表达式的最后一个类上，但事实上这样定义的可以被扩展成路径上所有的类上都可以作用谓词。一般的，任意路径表达式都可以形式化如下：

$$rv[p_1].NA_1[p_2].NA_2[p_3]\cdots NA_n[p_{n+1}] \quad (1)$$

其中 rv 是范围变量，$NA_i(1\leqslant i\leqslant n)$代表嵌套属性，类 C 的外延由 C_i 表示，$P_i(1\leqslant i\leqslant n+1)$是作用于 C_i 上的谓词。以该形式表示的路径表达式被称为限定路径表达式(Qualified Path Expression)。

2.2 一些典型的路径表达式算法

到目前为止学者们已经做了许多关于路径表达式算法及优化的研究工作。一般来说，这样的算法分两大类：基于指针追踪(Pointer Chasing)的算法和基于连接的算法。

1. 基于指针追踪的算法

(a)正向指针追踪。这是最自然最直接的处理路径表达式的算法，它依照对象的嵌套属性逐层向下计算，算法对第 k+1层聚集中没有被第 k 层聚集中对象引用到的对象不作处理，一次处理一个对象，操作粒度细且不产生中间结果，当系统的自由内存空间足够大，即将第 k—1层聚集中对象调入内存时，不会因为内存不足而将内存中驻留的第 k 层聚集中对象置换出去，算法的效率是最高的。文[1]的研究工作证明在有足够自由内存空间的情况下，该算法的效率是目前已有的所有处理路径表达式算法中最高的。但该算法的效率与自由内存空间大小有直接联系，同样的文[1]也表明了当自由内存空间逐渐减少时，性能开始降低，减少到某一阈值后，性能急剧下降。

(b)反向指针追踪。该算法仅当系统支持反向引用时有效，算法性能同正向指针追踪。

2. 基于连接的算法

(a)基于指针的连接算法：关于此类算法的研究也比较多，如 Shekita 和 Carey 在文[10]中分析了 nested-loops join，sortmerge join，hybrid-hash join 等三种算法。Lieuwen 在文[2]中评价了 hash-loops，probe-children，hybrid-hash/node-pointer，和 hybrid-hash/page-pointer 等四种算法，并给出了相应的并行算法。还有如正向指针连接等算法。不同于传统的基于连接属性匹配的连接算法，在基于指针的连接算法中匹配的是 OID。这些算法的优点是能容易被并行化，但它并没有反映出路径表达式级连(cascade)的特性，处理到第 i 层时该层的所有对象都要被调入内存，导致它效率不高，相对于基于显式连接的算法，它不能转化成半连接，这也使得它效率不高。

(b)基于显式连接的算法：在式(1)所表示的路径表达式出现在 where 子句中时，可以将其转化为式(2)所示的连接表达式来计算：

$$select_{NA_1}(C_1,p_1)\overset{NA_1}{\bowtie}select(C_2,p_2)\overset{NA_2}{\bowtie}\cdots\overset{NA_{n-1}}{\bowtie}select(C_n,p_n)\overset{NA_n}{\bowtie}select(C_{n+1},p_{n+1}) \quad (2)$$

但这样得到的式(2)与式(1)却并不等价，需要对式(2)的结果做 πC_1 操作，这样得到的结果才与式(1)等价。这种算法的优点在于它能将式(2)中的每个连接用半连接代替，于是得到式(3)：

$$(select(C_1,p_1)\propto(select(C_2,p_2)\propto(\cdots select(C_n,p_n)\propto select(C_{n+1},p_{n+1}))\cdots))) \quad (3)$$

基于半连接的算法能大大降低通讯开销和 CPU 开销，且无需投影操作，在执行半连接操作 $select(C_i,$

$P_i)\infty(select(C_{i+1},P_{i+1})$时，只有 $C_i.OID$ 和 $NA_i.OID$ 参与建立 $hash$ 表，使表所占的空间大大减少，一定程度上缓解了内存争夺的矛盾。

3 搜索空间

如第2.2节中描述的那样，路径表达式存在多种算法，那么在处理它时必然地存在选择最优算法的问题。参加构成路径表达式的多个聚集和多个算法形成了路径表达式求解的搜索空间，搜索空间的大小很容易随着聚集数和算法数的增长而呈几何级数形式增长。考虑如图2的对象关系，设有 n 个聚集，x 种连接算法，在仅有二元连接(不考虑多连接)的情况下，搜索空间 $SS(n)$ 由式(4)[1]给出。

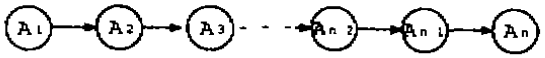

图2

$$SS(n)=\frac{(2n-2)!}{n(n-1)!}*x^{n-1} \quad (4)$$

式(4)中乘号左边的部分表示 n 个聚集最多可以够构成多少棵不同形态的树。若连接算法为三种，当聚集数为8时，搜索空间已经达到65462，也就是说存在65462种执行该路径表达式的途径，要想从这么多的途径中找出一条效率最高的来形成执行计划，是一件很困难的事。对此，目前通常的作法是采用启发式算法来解决。一种典型的方法是采用如第2.2节中介绍的指针追踪算法局部替代连接算法。图3中，圈代表连接操作，图中所示的由0-5六个聚集组成的路径表达式的搜索空间按式(4)计算应为 $SS(6)$，如果考虑将三角形围成的部分用第2.2节中介绍的指针追踪算法计算，则搜索空间降为如图3右半部分所示的 $SS(3)$，显然搜索空间的优化是惊人的。

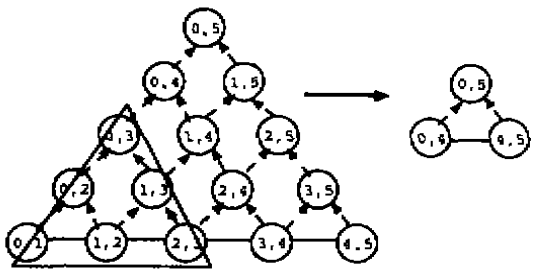

图3

4 代价模型

出于从理论上探讨路径表达式算法的优化搜索空间的需要，不同学者先后提出了不同的代价模型，本文分别介绍正向连接和正向指针追踪两种算法的代价模型。文中涉及的一些代价模型参数是：m：自由内存空间；p：页大小；$hash$：找到对象在内存中地址所用的时间，是纯 CPU 时间；$fan(C1,C2)$：从 $C1$ 到 $C2$ 平均对象引用个数。

首先不加证明地给出式(5)[1,2]，其含义为：从占了 m 页且含有 n 个对象的聚集中选择 k 个对象所需访问的页数。

$$Yao(n,m,k)=m*\left(1-\prod_{i=1}^{k}\frac{n-\frac{n}{m}-i-1}{n-i+1}\right) \quad (5)$$

从式(5)可以得到，在路径表达式中从聚集 C_{i-1} 到 C_i 所访问的页数的计算公式，见式(6)，其中 $\|C_i\|$ 代表 C_i 聚集中的对象个数，$|C_i|$ 代表 C_i 聚集所占的页数：

$$M=\sum_{i=i+1}^{x}yao(\|C_i\|,|C_i|,X_i) \quad (6)$$

4.1 正向指针追踪的代价模型

该算法的代价分析包括分析 CPU 开销和分析 I/O 开销两部分，以下的公式均出自文[1]。

1. $Cost_{CPU}$ 包括遍历对象图，将 OID 转换为内存地址和计算谓词的开销。

$$Cost_{CPU}=\text{hash}*\|C_1\|*(1+\sum_{i=1}^{n-1}\prod_{j=1}^{i}(\text{fan}(C_j,C_{j+1})*\text{Sel}_j)) \quad (7)$$

其中 $hash*\|C_1\|$ 表示聚集 C_1 中对象的 OID 转换开销，Sel_j 表示聚集 j 的选择率。

2. $Cost_{I/O}$ 取决于自由内存空间的大小，内存足够大时，由式(8)给出。

$$Cost_{I/O}=|C_1|+\sum_{i=2}^{n}yao(\|C_i\|,|C_i|,X_i) \quad (8)$$

X_i 是在类似 $rv[p_1].NA_1[p_2].NA_2[p_3]\cdots NA_{i-1}[p_i]$ 这样的子路径表达式中，第 i 个聚集中的对象个数。对 X_i 可由式(9)给出精确计算。

$$X_i=(1-(1-\frac{1}{\|C_i\|})^{X_{i-1}\cdot Sel_{i-1}\cdot fan_{(i-1,i)}})*\|C_i\| \quad (9)$$

其中 $(1-\frac{1}{\|C_i\|})^{X_{i-1}\cdot Sel_{i-1}\cdot fan_{(i-1,i)}}$ 表示第 i 聚集中对象不包括在路径表达式计算中的概率，式(9)可简化计算如式(10)。

$$X_i\begin{cases}\approx X_{i-1}*Sel_{i-1}*fan_{(i-1,i)} & \|C_i\|\gg X_{i-1}*Sel_{i-1}*fan_{i-1,i}\\ =\|C_i\| & \|C_i\|<X_{i-1}*Sel_{i-1}*fan_{i-1,i}\end{cases} \quad (10)$$

当自由内存空间较少时，在处理路径表达式的过程中会发生频繁的 I/O 操作，此时的 $Cost_{I/O}$ 是自由内

存空间的函数，由于情况太复杂很难给出公式。在最坏的情况下，$Cost_{I/O}=\|C_1\|*(1+\sum_{i=1}^{n-1}\prod_{j=1}^{i}(fan(C_j,C_{j+1})*Sel_j))$，此时自由内存空间小到一次只能读入一个对象，这样每发生一次读入下一层对象的操作时都会因为内存空间的不足而将当前层的对象置换出去。

4.2 正向连接的代价模型

同样的，该算法的代价分析包括分析CPU开销和分析I/O开销两部分。

1. $Cost_{CPU}$包括找到C_2中对象内存地址的CPU开销加上投影连接结果的CPU开销：

$$Cost_{CPU}=\|C_1\|*fan(C_1,C_2)*hash+Cost_{output}(\|C_1\|*Sel_1*fan(C_1,C_2)*Sel_2*proj) \quad (11)$$

其中$\|C_1\|*fan(C_1,C_2)*hash$是$C_2$对象的OID转换开销，$Cost_{output}(\|C_1\|*Sel_1*fan(C_1,C_2)*Sel_2*proj)$是$C_1$和$C_2$两个聚集作连接和投影结果的开销，由于这样两个聚集之间的操作可以并行执行，所以从时间上来看，$Cost_{CPU}$可以用C_1，C_2两个聚集之间操作的$Cost_{CPU}$来粗略表示（实际上应该用最大的$Cost_{CPU}$来表示）。

2. $Cost_{I/O}$的情况较复杂。考虑C_1，C_2两个聚集作连接的情况，显然将聚集C_1中的对象调入内存是必需的$Cost_{I/O}$，在聚集C_1被完全调入内存后，如果此时的自由内存空间满足$m\geqslant yao(\|C_2\|,|C_2|,X_2)$，则聚集$C_2$可以被一次调入内存；再考虑连接结果的回写，易知在内存足够大时，C_1，C_2两个聚集正向连接的$Cost_{I/O}=Cost_{readC_1}+Cost_{readC_2}+Cost_{write\ Result}$。复杂的情况在于自由内存空间不满足$m\leqslant yao(\|C_2\|,|C_2|,X_2)$时，对于聚集$C_1$中的每个对象，只要它与聚集$C_2$中的所有对象做完连接后就没用了，一旦一页中所有的聚集C_1的对象完成了连接，这一页就可以从内存中被置换出去。可见最有效率的调度方法是在内存中保留一页聚集C_1的对象，其余的$m-1$页用于聚集C_2的对象。对于聚集C_2来说，其任意一个对象处于内存中的概率是$(m-1)/|C_2|$，于是由式(12)给出因自由内存空间不足所引起的调度C_2聚集中对象的额外I/O开销。

$$\left(1-\frac{m-1}{|C_2|}\right)*\|C_1\|*Sel_1*Fan(C_1,C_2) \quad (12)$$

合并上述分析得式(13)，给出了在两个聚集间作正向连接的$Cost_{I/O}$。同样由于两个聚集之间的操作可以并行执行，$Cost_{I/O}$可以用C_1，C_2两个聚集之间操作的$Cost_{I/O}$来粗略表示。

$$Cost_{I/O}=Cost_{output}(\|C_1\|*Sel_1*Sel_2*fan(C_1,C_2),proj)+|C_1|+yao(|C_2|,|C_2|,X_2)+\left\{\left(1-\frac{m-1}{C_2}\right)*\|C_1\|*Sel_1*Fan(C_1,C_2)\right.$$
$$\left.\text{if } m<yao(\|C_2\|,|C_2|,X_2)\right\} \quad (13)$$

本节给出两种算法的代价模型，目的仅在于说明代价模型的分析范围，方法和手段并不是说这就是标准，事实上代价模型相当多，但本质上有许多相同之处。

参考文献

1 Gardarin G, et al. Cost-Based Selection of Path Expression Processing Algorithms in Object-Oriented Databases. In: Proc. of 22nd Intl. Conf. on VLDB. India, 1996

2 Lieuwen D F, et al. Parallel Pointer-based Join Techniques for Object-Oriented Databases. [Technique report CCSTR-92-1099]. Computer Science Department, University of Wisconsin, Madison, 1992

3 Ozkan C, et al. A heuristic approach for optimization of path expressions in object-oriented query languages. In: Proc. of the 6th Intl. Conf on Database and Expert Systems Applications. London, September 1995

4 Bertino D, Foscoli P. Index Organizations for Object-Oriented Database System. IEEE Transactions on Knowledge and Data Engineering, 1995, 7(2)

5 Frohn J, et al. Access to Objects by Path Expressions and Rules. In: Proc. of 20th Intl. Conf. on VLDB. Chile, 1994

6 Wang G R, et al. Parallel Algorithms for Path Expressions in Object-Oriented Database Systems. Information System and Technologies for Network Society, Japan, September 1997

7 Cattell R. The object database standard: ODMG-2 0, Morgan Kaufmann Pubishers, Inc. 1997

8 Carey J N, et al. The OO7 Benchmark. In: Proc. of SIGMOD. 1993. 12～21

9 Carey J M, et al.: [The OO7 Benchmark CS Tech Report]. Univ. of Wisconsin-Madison, April 1993

10 Shekita E, Carey M. A performance evaluation of pointer-based joins. In: Proc of the 1990 ACM SIGMOD Intl. Conf. on Management of Data. Atlantic City, NJ, USA, p300～311

11 Yao S B. Approximating the number of accesses in database organizations. CACM, 1997, 20(4): 260