一个基于引用的连接算法 Sort-Loops

阳国贵, 吴泉源
(国防科技大学计算机学院, 长沙 410073)

摘 要: 引用既是对象关系数据模型中一种重要的建模设施, 同时它也有利于连接算法的设计和高效实现, 针对对象关系数据模型和查询语言的这个新特点, 文中提出了一个基于引用的连接算法 Sort- Loops. Sort-Loops 一次尽可能多地读入外关系, 并依据引用属性中的页面信息, 对读入的外关系元组进行排序, 这不仅避免了对同一内关系页面的多次重复存取, 同时, 也使对内关系页面的访问次序与物理地址顺序一致, 进一步提高了算法对内关系的存取性能. 文中还对该算法的性能进行了分析、比较, 证实了 Sort-Loops 是一个实用和高效的连接算法.

关键词: 连接算法; 对象关系数据库; 算法分析
中图分类号: TP392 文献标识码: A 文章编号: 0372-2112 (2001) 05-0615-04

Sort-Loops: A New Reference Based Join Algorithm

YANG Guo-gui, WU Quan-yuan
(*College of Computer, National Univ. of Defense Technology, Changsha* 410073, *China*)

Abstract: Reference is not only one of the important modeling mechanism, but also beneficial to the design and implementations of join algorithms. Based on the features of ORDM(Object-Relational Data Model) and the query language, a new reference based join algorithm Sort-Loops is represented. Sort-Loops attempts to read most of the outer table as possible in one pass, and then sorts the tuples according to their reference's PID. By the sorting techniques, there is no duplicate read of the same page in a pass, and the performance is further improved by reading the inner table in order of its physical address. Sort-Loops is a useful and high performance join algorithm. It is further validated by the analysis and comparison with other algorithms.

Key words: join algorithm; ORDB; algorithm analysis

1 引言

对象关系数据库技术是在继承传统的关系数据库技术的基础上, 增加面向对象特征, 使面向对象技术与关系数据库技术紧密结合, 可满足 CAD、CASE、图象处理、GIS 等新型数据库应用领域的需求而发展起来的一种新型数据库技术, 它既要提供管理复杂数据的能力, 又能提供强有力的查询功能. 对象关系数据模型提供创建复杂类型的类型构造子, 如组合、集合和引用(Reference) 等[1], 其查询语言将是 SQL 的一种自然发展, 如目前已经出现的 SQL: 1999. 模型和语言的新特点, 使得对象关系数据库系统的实现技术更为复杂, 在连接算法方面, 由于引用的引入, 需要研究新的基于引用的连接算法, 才能有效利用这一重要的建模设施.

在对象关系表中, 某列可以是引用类型, 用于指向某个类型的对象. 同样, 关系表中的某列也可以是集合类型, 而集合的成员是引用类型. 如在 Illustra 对象关系数据库中, 表中的每一行都有一个唯一的标识, 称为对象标识符(OID), 它是一个 64 位的唯一标识, 由系统赋予, 并且一经赋予就不再改变, OID 的组成包括两个部分: 其中的 24 位作为表标识, 另外的 40 位为表中的行标识. 实现上, 引用类型列中所存储的就是所指对象的 OID. 同样, ORACLEV8.0 对象关系数据库支持嵌套对象、可变数组、复杂结构类型, 关系表中的元组具有 ROWID, 该 ROWID 包含了元组所在的数据文件、块和块中的行序列等存储信息.

在面向对象数据库(如 Gemstone、ObjectStore、ORION、O++、O2 等) 中, 尽管允许集合属性和对对象的直接引用, 但由于早期的面向对象数据库系统并不提供类似于 SQL 的查询语言接口, 仅提供导航式对象访问和操作方式, 所以, 对象间的关联访问(或连接) 运算没有得到应有的重视. 在以后的发展中, 逐渐认识到了面向对象数据库系统也同样需要提供强有力的查询语言功能, 在 ODMG-93 中就定义了支持对象查询的 OQL 语言.

在关系数据库中, 连接运算十分费时, 为此, 对连接算法的研究也就十分重视, 典型的连接算法有嵌套循环、带索引的嵌套循环、排序归并、Hash Join、Hybrid Hash Join 等[2]. 当对象的 OID 带上存储位置信息(如所在的物理页面) 时, 就可根据对 OID 的引用直接找到该对象了. 为此, 利用该 OID 特点的一

收稿日期: 1999-08-13; 修回日期: 2000-12-20

些连接算法已被提出,如基于指针(pointer based)的 Hash Loops 连接算法和 Hybrid Hash 算法等[3]. 一般地,这些算法可以移植到对象关系数据库中. 对 Hash Loops 连接算法而言,它对内表的访问不是按物理地址的顺序进行的,这样,将导致数据读取时磁头来回频繁移动,使算法性能受到严重影响,据此,本文提出一种新的基于引用的连接算法 Sort Loops,该算法能克服上述不足,优化内存的使用,在一次循环中读入更多的元组,从而减少循环次数,改善算法性能.

2 连接算法

2.1 基本问题

假如在某公司的人事数据库中,有部门关系表 dept 和职员关系表 emp,其中部门表和职员表的元组类型分别为 dept_t 和 employee_t,其结构为:

```
create type employee_t (name varchar(30), startdate date, salary int,
                        address varchar(30), state char(2), zipcode int);
create table emp of type employee_t;
create type dept_t (dname varchar(30), manager varchar(30),
                    phone phone_t, manager_ref ref(employee_t),
                    workers setof(ref(employee_t)));
create table dept of type dept_t;
```

可以看到,dept_t 中的 manager_ref 为引用类型,指向 employee_t,workers 为集合类型,集合成员为引用类型. 为了对比,还包含了 manager 列. 当查找鞋帽部经理的起始工作日期时,可以采用如下方式:

```
select startdate from emp, dept
        where emp.name= dept.manager and dept.dname='shoe';
```

在对象关系数据库中,一种更为自然的方式可能是:

```
select deref(manager_ref) from dept where dname='shoe';        (1)
```

其中,deref 是用于引用类型的一个函数,得到引用所指的某种类型的元组. 处理类似于语句(1)的查询,可以采用基于指针的排序归并、基于指针的哈希连接,Hash Loop 以及本文将介绍的 Sort Loops 等算法,但本文具体讨论的 Sort Loops 形式将主要针对集合属性,而集合属性中的成员为引用类型. 事实上,语句(1)中的形式可看成集合属性的特例,即该集合属性仅含有一个引用成员. 关于集合属性的查询语句,试举例如下:

查找鞋帽部中工资高于其经理,且起始工作日期在 95 年 10 月 1 日后的员工. 这样一个查询,连接属性中的 Workers 即为一集合属性,套用 OQL 的写法,可以表述为:

```
select distinct struct(name: x.name, employees: (select y
                                                 from y in x.workers
                                                 where y.salary> x.manager_ref-> salary
                                                 and y.startdate> '1/10/95'))
from x in dept;                                                (2)
```

更一般地,可把上述查询问题归纳为:

$$\text{for } (x1 \text{ of Set1}; x2 \text{ of } x1->\text{set}) \text{ suchthat}(\text{Pred1}(x1, x2))\ S1. \tag{3}$$

其中,Pred1 为 $x1$, $x2$ 应当满足的条件谓词,而 $S1$ 是语句中的其他部分,如做属性投影运算等. 对于这类查询,一种直接的求解方式是:对每个 $x1$,利用引用值直接找到 $x2$,然后检查它们是否满足条件谓词 Pred1,在满足条件时,形成结果元组. 研究表明,这种直接的方式在性能上往往是低效的,为此需要寻找更好的连接算法. 在以下的讨论中,假定引用中包含了所指对象的物理位置信息,如块号 PID(page identifier). 下面将较详细地介绍 Sort Loops 算法,为了便于分析、比较,对 Hash Loops 算法、Probe Loops 算法也将做简要说明.

2.2 Sort Loops 算法

把式(3)中的 Set1 称为外连接关系,而 $x2$ 所在的关系称为内连接关系,则 Sort Loops 算法的基本思想是,当外连接关系 Set1 在内存中放不下时,分成若干遍进行,每遍尽可能多地读入 Set1 中的元组(当算法可用内存为 M 块时,为读内连接关系表 Set2 留出一块,即在每遍中可读入 Set1 中的 $M-1$ 块到内存.),对每遍中所读入的 Set1 中的元组,形成一组排序元素(Pid, Pointer),Pointer 为该元组在内存中的位置指针,而 Pid 即为式(3)中各 $X2$(即引用)中的 PID 部分(引用所指向的块号),并按 Pid 对这些排序元素进行排序,对同一遍中所有的排序元素排序之后(具有相同 PID 的 Pointer 可放在一起),将按排序次序依次读取 Pid 所标识的数据块,当该块被读入后,依据 Pointer 指针,找到相应的 Set1 中的元组,检查相应的元组对($X1$, $X2$)是否满足连接谓词 Pred1,若满足,则进行 $S1$ 语句的处理,获得一个结果元组,当完成该遍中所有元组的处理工作后,可开始下一遍的工作,直至把 Set1 中的所有元组处理完毕. 下面给出上述算法的伪代码描述:

```
Sort-Loops-algorithm ()
  {while(not eof(Set1)) do
    {phase1: Do get_next_page_set1(pbuffer);
                          /* get next page of Set1 Sequentially */
       while (not_end_of (pbuffer))
          {object= get_next_object();
           pointer= insert_into_mem_table(object);
           pids= extract_ref_pid(object);
           for each pid of pids
                do insert_into_sort_table(pid, pointer);}
       Until(available memory reached threshold);
     phase2: for(i = 0; i++ ; i< Max_Sort_Table_Length)
          {sort_table_entry= get_next_sort_table_entry(i);
          seq_read_page_of_set2 (sort_table_entry.pid, buffer);
          set1_objects= sort_table_entry.pointer;
          for each object1 of set1_objects
                        do join(buffer, object1);}
     }/* end of while */
  }/* end of the algorithm */
```

在上述描述中,外循环 while 的执行次数即为算法将外关系划分为多段执行时的遍数,在每遍的 phase1 阶段,依次读入外关系 Set1 的数据页面,对页面中的每个元组而言,先将该元组放到内存中的适当位置(由 insert_into_mem_table 完成),然后从该元组的连接集合属性中获取指向 Set2 的引用,并从中抽取出物理页面号 Pid,把排序元素(Pid, Pointer)插入到按 Pid 排序的排序表中(由 insert_into_sort_table 过程完成).

需要特别指出的是, insert_into_sort_table 过程的实现方式将直接影响到算法的性能, 在内存中实现排序的方法很多, 如基于优先队列(Priority Queue) . 但基于 Sort-Loops 算法中的 Pid 范围是已知的, 即关系 Set2 的页面范围, 据此, 可以预先分配一个排序表, 使得表项与 Pid 一一对应. 在下面的算法分析中, 对单个引用属性的情况, 采用基于优先队列的排序算法, 而对集合引用属性的情况, 采用第二种方式, 以少量空间换取时间.

下面用一个实例对上述算法思想进行说明, 假定在算法的 phase1 阶段, 读入内存的 Set1 中的元组(仅三个) 为:

Mat, …, {(# 10, 5)}; Com, …, {(# 11, 2), (# 20, 1)}; Phy, …, {(# 20, 2)}

上述各元组中的{}即为集合属性, 其中各成员为引用. 如(# 10, 5), 表示块号为 10, 块内号为 5 的一个引用值. 则完成 phase1 后在内存中形成的数据结构如图 1.

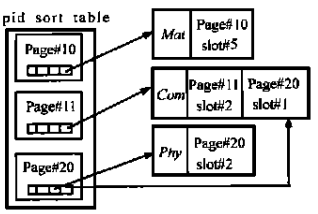

图 1

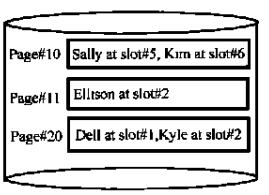

图 2

假定 Set2 中的部分数据如图 2 所示. 在 phase2 阶段, 由于有了 Pid 的排序表, 对 Set2 读取时, 可以按 Pid 顺序进行(由 seq_read_page_of_set2 过程完成) . 并且把该遍中所有对 Set2 中位于同一页面的所有元组的引用处理一次完成, 从而避免了对该页面的重读(见 phase2 中的 for 循环) . 对上例而言, 根据排序表, 将先读入 Set2 中的第 10 块, 找到相应的元组对 Mat/Sally, 检查该元组对是否满足连接条件, 如满足, 即形成相应的结果元组, 然后读取 Set2 中的第 11 块, 形成元组对 Com/Ellison, 并对该元组对进行相应的处理, 最后读取 Set2 中的第 20 块, 根据排序表中指向 Set1 中的元组的指针, 形成元组对 Com/Dell 和 Phy/Kyle, 同样需要对这两个元组对进行相应的处理, 值得指出的是, 这里的第 20 块只需读入一次即可. 要不采用排序而按直接方式进行时, 将先处理 Set1 中的第一个元组, 并读入 Set2 中的第 10 个页面, 按(# 10, 5) 找到相应的元组, 再处理第二个元组, 读入 Set2 中的第 11 个页面, 依(# 11, 2) 找到相应的元组, 再读入 Set2 中的 20 个页面, 依(# 20, 1) 找到相应的元组, 最后处理第三个元组, 将再次读入 Set2 中的 20 个页面, 由(# 20, 2) 找到相应的元组进行处理, 这样将两次读取 Set2 中的第 20 个页面.

2.3 Hash-Loops 算法

当外连接关系 Set1 在内存中放不下时, Hash-Loops 算法同样分成若干遍进行, 每遍尽可能多地读入 Set1 中的元组, 对每遍中所读入的 Set1 中的元组, 形成一组元素(Pid, Pointer), Pointer 为该元组在内存中的位置指针, 而 Pid 即为式(3) 中各 $X2$(即引用) 中的 PID 部分(引用所指向的块号), 并根据某个哈希函数对 Pid 进行计算, 由哈希值确定这些元素所在的位置, 完成同一遍中所有元素的处理并形成哈希表之后(具有相同 PID 的 Pointer 可放在一起), 将依次按照哈希表中的 PID 读取相应的数据页面, 依据 Pointer 指针, 找到相应的 Set1 中的元组, 检查相应的元组对($X1$, $X2$) 是否满足连接谓词 Pred1, 若满足, 则进行 $S1$ 语句的处理, 获得一个结果元组, 当完成该遍中所有元组的处理工作后, 可开始下一遍的工作, 直至把 Set1 中的所有元组处理完毕.

2.4 Probe-Loops 算法

Probe-Loops 算法将把 M 块内存中的 $M-1$ 块用作保存 Set2 中的数据页面的缓冲区, 算法的基本思路是: 先从 Set1 中读入一个数据页面, 对该页面中的每个元组, 根据其连接集合属性, 找到各引用值, 对每个引用值, 先根据其 PID 值在内存中寻找相应的数据页面, 当该页面在内存时, 即可完成相应的处理工作. 在内存找不到时, 将把 Set2 中的相应数据页面读入内存, 当内存有自由空间时, 将相应页面读入即可, 否则, 可按某种淘汰算法, 先选择将被覆盖的页面, 然后将相应页面读入.

3 性能分析与对比

假设 Set1 中的每个元组对 Set2 中元组的引用是均匀分布的, 且每个 Set1 元组中包含的引用个数的平均数为 m 个, 而一个 Set2 元组平均被 Set1 中的 n 个元组所引用. 机器速度为 Mips, 执行比较、交换操作的时间分别为 compare 和 swap, 而排序元素的插入时间为 move, 完成一次磁盘读或写操作的时间为 IO, 内存页面数为 M、磁盘页面的大小为 P, |Set1|、||Set1|| 分别表示 Set1 的页面数和元组数, |Set2|、||Set2|| 分别表示 Set2 的页面数和元组数, 表示块号 Pid 所需的字节数为 K, 而在内存中指向一个 Set1 元组的指针所需的字节数为 L, Set1 和 Set2 中元组的大小为 Oset1 和 Oset2 个字节, 哈希表的膨胀因子 F 等[2].

引理 1　文件的记录数为 n, 块数为 m, 记录在各块中均匀分布, 为此, 每块中有 n/m 个记录. 当从该文件中任选 k 个记录时, 所需存取的文件块数为[4]:

$$Y(k, m, n) = m * \left[1 - \prod_{i=1}^{k} \frac{n - (n/m) - i + 1}{n - i + 1}\right]$$

对 Sort-Loops 算法而言, 从 Set1 中读入一个数据块, 完成 phase1 中的内 while 循环所需的额外空间(字节数) extra 平均为:

$$\text{extra} = [\,\|\text{set1}\|/|\text{set1}| \times m/n \times (K + n \times L)\,],$$

内存中一次可读入 Set1 的块数、指向 Set2 的 OID 的数目和算法执行外循环的次数分别为:

$$\text{pages} = \frac{M-1}{1 + \text{extra}/P},\ \text{references} = \text{pages} \times \|\text{set1}\|/|\text{set1}| \times m/n,$$

$$\text{cycles} = |\text{set1}|/\text{pages}.$$

而一次循环中所需存取的 set2 的数据块数为: blocks = Y(references, |set1|, ||set2||). 由于针对集合属性采用预留排序表方式, 所以算法的执行时间 T 为:

$$T = |\text{Set1}| \times \text{IO} + \|\text{Set1}\| \times m \times \text{move} + \text{cycles} \times Y(\text{references}, |\text{set2}|, \|\text{set2}\|) \times \text{IO}.$$

否则, 执行时间为: $T = |\text{Set1}| \times \text{IO} + \|\text{Set1}\| \times m \times \log$

(‖Set1‖ × m/cycles) × (compare + swap) + cycles × Y(references, |set2|, ‖set2‖) × IO.

上面给出了 Sort-Loops 算法的代价公式, 限于篇幅, 其余算法的有关代价分析公式这里从略, 可参见[5]. 当参数设置为: Mips = 100000000; IO = 0.002; P = 4096; ‖Set1‖ = 100000; ‖Set2‖ = 100000; n = 1; m = 1; Oset1 = 400, Oset2 = 100; F = 1.2; 不考虑磁盘顺序读取和随机读取的差异, 且采用基于优先队列的排序方法时的性能对比为图 3 所示. 而图 4 中的 IO = 0.02; Oset2 = 400, 其余参数和假设不变.

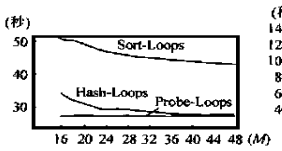

图 3 对比分析之一

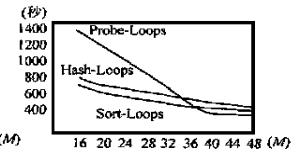

图 4 对比分析之二

从上可以看出, 当关系 Set2 较小, 可以全部放入内存时, Probe-Loops 算法的性能将好于 Sort-Loops 与 Hash-Loops, 但当 Set2 较大, 在内存放不下时, 性能显著下降, 见图 4. 由于采用基于优先队列的排序方法, 故排序时间不容忽视, 导致图 3 中 Sort-Loops 的性能不如 Hash-Loops. 但当降低 I/O 速度时(相当内存排序时间降为次要因素时), Sort-Loops 的性能好于 Hash-Loops, 如图 4. 而当采用预留排序表时, Sort-Loops 所花的排序时间进一步降低, 将表现出更好的性能.

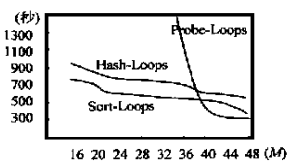

图 5 对比分析之三

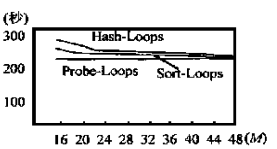

图 6 对比分析之四

下面的分析主要针对集合引用属性, 参数设置为: IO = 0.02; n = 50; m = 50; Oset1 = 400, Oset2 = 100; 其余不变, 但采用预留排序表方式(图 5). 当 | |Set2| | = 100000; M = 10; n = 100, 其余参数和假设不变时, 性能分析结果见图 6.

从上可以看出, Sort-Loops 算法采用预留排序表空间方式实现集合引用属性的连接处理时, 具有很好的性能.

4 总结

针对对象关系数据模型的特点, 本文提出了一个基于引用的连接算法 Sort-Loops, 该算法能根据内连接关系表(Set2)页面地址范围已知的特点, 预留出排序表空间, 加快获得排序表的速度, 完成排序以后, 将根据排序表, 对内连接关系进行顺序读取, 进一步提高算法性能. 目前, 计算机系统主频的速度提高比 I/O 速度的增长快, 这样使得连接操作的速度受限于 I/O, 通过分析, 对单引用连接属性而言, 即使采用一般的排序方法, Sort-Loops 的性能也可好于 Hash-Loops 算法. 而对基于集合引用属性的连接操作而言, 在采用预留排序表的情况下, Sort-Loops 算法将好于 Hash-Loops 算法. 当内连接关系表(Set2)可以放入内存时, Probe-Loops 将比 Sort-Loops 和 Hash-Loops 好, 否则, 性能很差.

参考文献:

[1] 阳国贵等. 对象关系数据库系统与技术 [J]. 计算机科学, 1998, 25(6).

[2] Leonard D. Shapiro. Join processing in database systems with large main memories [J]. ACM Transactions on Database Systems, Sep. 1986, 11(3): 239-264.

[3] Eugene Shekita and Michael J. Carey. A performance evaluation of pointer-based joins [J]. Proc. 1990 ACM SIGMOD, June 1990.

[4] Yao S B. Approximating block accesses in database organizations [J]. Commun. of ACM 1977, 20(4): 260-261.

[5] 阳国贵等. 对象关系数据库中若干关键问题的研究 [R]. 内部研究报告 9908, 1999.

[6] Michael Stonebraker, et al. Objects-Relational DBMSs The Next Great Wave [M]. Morgan Kaufmann Publishers, Inc. 1996.

作者简介:

阳国贵 1985 年毕业于中南矿冶学院计算机专业, 获学士学位, 1988 年毕业于中国人民大学信息系, 获硕士学位. 后到国防科技大学工作, 从事数据库技术的研究与教学, 先后发表学术论文三十余篇, 主编或参与出版数据库方面的著作三部. 主要研究兴趣包括面向对象数据库、并行数据库、对象关系数据库等.