文章编号: 1001-2486 (2000) 01-0001-06

对象关系数据库中连接谓词索引技术的研究*

阳国贵, 吴泉源
(国防科技大学计算机学院, 湖南 长沙 410073)

摘 要: 讨论了一种适合于对象关系数据库的新型索引结构——连接谓词索引, 在介绍了谓词索引之后, 给出了基于该索引结构的连接算法, 并分析了连接算法的性能, 提出了根据性能计算来确定关系 R 和 S 中谁作为外关系, 从而降低算法代价的方法, 特别地, 本文还把连接谓词索引推广到一般形式, 可以用于多个关系表的连接。

关键词: 索引结构; 连接算法; 存取方法; 对象关系数据库
中图分类号: TP392 文献标识码: A

Researches on Join Index Technology in ORDB

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

Abstract: A new type of index structure called join predicate index (JPI) is described. The JPI can be used to improve the performance of joins in object relational data base systems. After the introduction of JPI, a join algorithm based on JPI is presented and the performance analysis formula is given. It is also proposed which should be the outer relation so as to decrease the expense of the join operation. Lastly, the basic JPI is extended to the join of n relations.

Key words: index structure; join algorithm; access methods; OODB

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、图像处理、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。根据 Stonebraker[1] 对 DBMS 的分类, 对象关系数据库技术属于分类矩阵中的第四块, 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。在复杂数据方面, 通常要求提供创建复杂类型的类型构造子, 如组合、集合和引用。对象关系数据库系统提供的查询语言将是 SQL 的一种自然发展, 如现在正在形成的 SQL3。对象关系数据库系统面临的严重挑战之一就是对这样的查询语言如何进行优化, 它所涉及到的优化问题将远比关系数据库复杂。如对 ADT 的处理, 继承层次的连接优化, 高价函数 (Expensive Function) 的处理等等[2]。

对象关系数据库中同样将面临关系的连接运算问题, 为了加速连接运算, 索引技术仍将发挥重要作用。针对对象关系数据库的特点, 需要研究连接谓词索引等新技术 (尽管它们在关系系统中同样可用)。如 select * from Emp where Redness (house) < 0.1, 该查询中涉及到用户定义的 Redness 函数, 它计算房子外围的红色程度 (从房子照片中计算), 如果为函数建立了索引 (索引项为函数值和元组标识), 则完成该查询的计算方法与关系具有属性索引时的计算方法一样, 从 Redness 函数索引中找出满足小于 0.1 的元组标识, 再进行后续处理。类索引是要求系统具有为用户定义的 ADT (抽象数据类型) 建立索引的能力。本文将主要研究一种新的连接谓词索引, 即对连接运算中的连接谓词建立索引, 与函数索引思想有类似之处, 但顾名思义, 它主要针对连接运算, 而函数索引针对选择运算。如 Select * from A1, A2, …, An Where P (A1.att1, A2. att2, …, An.attn), 即从 A1 到 An 表中找出那

* 收稿日期: 1999-10-22
基金项目: 国家部委项目资助 (98J15.2.5.KG0133)
作者简介: 阳国贵 (1964-), 男, 副教授, 硕士。

些满足谓词P 的元组，其中的P（A1. att1，A2. att2，…，An. attn）就为连接谓词的一般形式，而连接谓词索引的索引项可为（T1id，T2id，…，Tnid），即由各关系表中满足P 谓词的元组的元组标识符组成。

Valduriez[3] 提出了二元连接索引的思想，并给出了基于二元连接索引的连接算法及性能分析。但他给出的算法不够具体，并且认为连接索引中 Trid 的个数高于 Tsid 的个数时，就应把 R 当做外关系进行处理。然而，通过分析表明，这一观点是不正确的。需要对连接关系在担当不同角色（即作为连接关系的内关系还是外关系）时的连接运算代价进行分析，才能确定把哪个关系当做内关系或外关系。特别地，由于对象关系数据库中的连接谓词可以是用户自定义的，使得连接谓词更为一般，这样，需要研究连接谓词索引的一般形式。在连接算法的实现中，需要充分利用现代计算机系统提供的大容量主存来改进算法性能，如 Shaprio[4] 就大主存情况下的连接算法的性能进行了一些分析，指出利用大容量主存的 Hybrid Hash 算法在通常情况下优越于 Grace Hash Join 和 Simple Hash Join 算法。为此，本文在介绍基于连接谓词索引的连接算法时也将尽量考虑内存资源来优化算法的实现。

本文的后续部分将依次讨论二元连接谓词索引及连接算法，连接算法的性能分析，一般连接谓词索引及连接算法，最后是本文结语。

1 二元连接谓词索引及连接算法

1.1 二元连接谓词索引

对于以下查询 Select * from R，S where R. A= S. B，它求出满足 R. A= S. B 的元组对，然后在选择属性列上做投影。假定每个元组都有由系统产生的唯一的元组标识，如 surrogate（用 rid 表示元组标识，而 rid 标识的元组为 Trid），则 R. A= S. B 这个连接谓词为真的（rid，sid）偶对就称为连接索引项，这些索引项的集合称为连接谓词索引，以下有时也简称连接索引，所以连接索引可以看成连接谓词的外延（或具体化，materialization），更一般地，R 和 S 上的连接谓词索引为：

JI= {（rid，sid）| f（Trid. A，Tsid. B）is TRUE}

其中 f 是定义连接谓词的布尔函数。为此，二元连接谓词索引可以看成为一个二元关系，该关系中的每个元组仅由两个元组标识组成，所以这样的关系在体积上会很少，便于使用，连接索引可以聚集文件的形式保存到磁盘上。对于上面的 JI 而言，可以 rid 为聚集码或以 sid 为聚集码进行聚集，考虑到实际查询中可能包含对 R 或 S 关系的选择操作，有时可以先做选择，再进行连接。在这种情况下要根据 R 或 S 的 id 到 JI 索引中去找到相应的满足连接谓词的 S 或 R 中的元组，这样对于从 R 出发到 JI 中找 sid 的情况而言，建立以 rid 为聚集码的索引文件是合适的，反之，应建立以 sid 为聚集码的 JI 聚集文件。当然，考虑到上述两种情况都会发生，也可采用一种对称式办法，即分别以 rid 和 sid 建立两个聚集文件（用 B+ 树方式实现），以便于上述情况的使用。

1.2 基于连接谓词索引的连接算法

当存在 R 和 S 间的连接索引时，且 R，S 均按 surrogate 建立聚集索引，JI 按 rid 聚集，连接算法依次读入 JI 块。根据该块中的 rid，读入包括 Trid 的数据块，从该数据块中找到 Trid，并且将之放入到 Rk 中，直到使用完内存为止，然后对读入的 JI 块按 sid 排序（便于对 S 文件的处理），最后按 JI 块中的 sid 找到相应的包含 Tsid 的数据块，在该数据块中找到 Tsid，形成结果元组，直至把 JI 中的使用索引项处理完毕。具体算法为：

```
JRSji_Algorithm (R, S, JI, M)
{
bool Pre_Pass_Complete= TRUE;
m= M- 3; /* 分别为关系R、S 和结果元组保留一块作为缓冲区,
                                m 表示当前可用的内存块数, M 为该算法执行的内存块* /
while (not eof (JI) ||not Pre_Pass_Complete)
    {m= M- 3;
```

```
        m= m- 1;  /* 上遍中 JIk 中还有一块没有处理完，算法需要对内存进行有效管理，要保留上遍
                     中没有处理完的那一块，并且使位置指针不变。当然，算法也可采用预测的办
                     法，使得当有足够的空间时才读入下一 JI 块。* /
        if (Pre_Pass_Complete) {
                     m= m+ 1;       read_ JI (JI_ Buffer);       /* 读入 JI 的下一块* /
                     m= m- 1;       variable_ initial ();}
R_Temp_Buffer_Current_Positon= 0;
while (m> 0 and not eof (JI))
        {  /* 根据内存的大小，尽可能多地读入 JI 块* /
          while (Current_JI_Page_Position < = Current_JI_Page_End_Position)
             {  /* 根据 JIi 块完成 R 与 JIi 的半连接  * /
              T_JI= get_current_JI (Current_JI_Page_Position);
              rid= T_JI. rid;
        next_r_read: if (R_Temp_Buffer_Current_Positon> R_Temp_Buffer_End_Position)
              {/*   R_Temp_Buffer 中的元组搜索完后，再读入下一块 * /
                    Get_Page_R (R_Temp_Buffer, rid);
                    R_Temp_Buffer_Current_Position= R_Temp_Buffer_Begin_Position;
                    R_Temp_Buffer_End_Positon= end_of_R_Temp_Buffer;
                                  }
              Trid_Pointer= Find_Match (R_Temp_Buffer_Current_Position, rid);
                R_Temp_Buffer_Current_Position+ + ;
                If (Trid= = NULL ) goto next_r_read;
                if (Rk_Current_Page_Position< = Rk_Current_Page_Position_End_Position)
                   {
        Rk_Pointer= Insert (Trid_Pointer, Rk [Rk_Current_Page] [Rk_Current_Page_Position]);
                   Rk_Current_Page_Position+ + ;}
                   else if (m< 0)
                          {  m= m- 1;
                             Rk_Current_Page= assign_a_new_page_for_Rk ( ) ;
                             Rk_Current_Page_Position= Rk_Current_Page_Begin;
Rk_Pointer= Insert (Trid_Pointer, Rk [Rk_Current_Page] [Rk_Current_Page_Position]);
                          Rk_Current_Page_Position+ + ;
                   }
                          else {Pre_Pass_Complete= False;
                            goto phase2;  /*  把目前为止的 Rk 与 S 做连接* /   }
             JI_Buffer [ i] [Current_JI_Page_Position] . rid= Rk_Pointer;
             /*  使得 JI 中的 rid 改为指向 R 元组的指针* /
             Current_JI_Page_Position+ + ;
     } /*  end of while (Current_JI_Page_Position < = Current_JI_Page_End_Position)  * /
          if (m> 0) {get_Next_Of_JI ();
                     Current_JI_Page_Position= 0;}
} /*  end of while (m> 0 and not eof (JI))  * /
phase2:  sort JIK on sid;
```

```
for all ji In JIK {   /* 从排序的sid 中依次根据 sid 读取相应的 S 关系中的数据块 */
  next_s_read:   if (S_Temp_Buffer_Current_Position> S_Temp_Buffer_End_Position)
                 {get_page_s (S_Temp_Buffer, sid);
                 S_Temp_Buffer_Current_Position= S_Temp_Buffer_Begin_Position;}
                    Tsid_Pointer= Find_Match_S (S_Temp_Buffer, ji. sid);
                 /* 注意, 不要在找完后让 S_Temp_Buffer_Current_Position 下移, 因为可能有重复
                    的 sid 要查找* /
                       if (Tsid_Pointer= = NULL) goto next_s_read;
                       Tresult= Form_Result_Tuple (ji. rid, Tsid);
                          }
}   /* end of while (not eof (JI) || not Pre_Pass_Complete) */
} /* end of the algorithm */
```

本算法充分利用了已经读入内存的数据块, 尽量减少了对同一数据块的重读。也不必为 JI 排序留出空间, 可直接在 JIk 中进行。另外, 当找到与 rid 匹配的 Trid 时, 不必在 JIk 中另留空间来指向 Trid, 可直接使 rid 域指向 Trid 即可。

2 连接算法的性能分析

可以看到, 上述算法根据内存的大小, 分成多遍执行, 每遍中包括四个主要步骤, 即先读入 JI 块, 执行 R 与该块的半连接 (该步骤将尽量使用现有内存); 然后对读入的 JI 块进行排序, 以减少对 S 表中相同数据块的多次重复读入; 根据内存中的 JIk 块, 依次找到 S 表中的元组, 与相应的 R 中的元组形成最终的结果元组。为了能了解上述连接算法的性能, 需要对算法进行性能分析, 特别地, 性能公式的给出有助于对连接运算的不同计算过程进行分析, 即对 R 和 S 的连接, 到底是用 R 作为外连接关系, 还是用 S 作为外连接关系给出根据, 从而更好地对连接运算进行优化。

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:

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

算法分析中用到的部分符号为: |X| 表示关系 X 所占的数据块数, ||X|| 为关系 X 中的元组数, SR 表示关系 R 与 JI 连接的选择率, SS 表示关系 S 与 JI 连接的选择率, IO 为读入一个数据块所花费的时间, comp 为进行一次比较操作的时间, move 移动一个记录所花费的时间, FO 为索引块中所包含的索引项数。

有了上述准备后, 下面来分析连接算法的 I/O 和计算工作量。从 JRSji_Algorithm 算法可知, 需要为关系 R、S 和结果关系各留一块作为缓冲区, 这就是 m= M- 3 的原因, 可令 M′ = M- 3; 内 while 循环要先把部分 JI 和 R 块读入内存, 而 JI 和 R 关系的体积之和为 |JI| + |R| * SR, 那么, 需要执行的遍数, 即外 while 循环调用内循环执行的次数 N 为

$$N = ceil\ ((|JI| + |R| * SR) / |M'|)$$

对每遍而言, 内 while 读入 JI 的元组为 |JI|/N 块, R 中的元组为 |R| * SR/N 块, 由于 JI 为聚集文件, 仅需 |JI|/N 次 I/O 即可。为读取 R 中的 ||R|| * SR/N 个元组 (体积为 |R| * SR/N), 需访问的 I/O 次数将包括两部分: 对数据文件进行操作的 I/O 和为索引文件进行操作的 I/O, 前者需要进行的 I/O 次数相当于从含有 ||R||/N 个元组, 大小为 |R|/N 个页面的文件中选择 ||R|| * SR/N 个元组时导致的 I/O 次数, 由引理 1 可知, 其次数为

$$IOrf = Y\left(\frac{\|R\| * SR}{N}, \frac{|R|}{N}, \frac{\|R\|}{N}\right)$$

对于索引结构部分而言, 假定由于每个索引块中的索引项较多, 聚集索引树高为 2 层, 并且第一

层（根节点）常驻内存。由于对数据文件要访问 IOrf 块，所以，对索引文件中的第二层索引项读取的个数为 IOrf 个，而该层上涉及到的索引项查找数为|R|/N，每个索引块中有 FO 个索引项，则索引块数目为|R|/（N* FO），为此，从|R|/N 个索引项中读取 IOrf 个索引项将导致的 I/O 次数为

$$IOrci = Y(IOrf, |R|/(N*FO), |R|/N)$$

为此，IO 时间为 IOr=（IOrf+ IOrci）* IO。

对于读入的每一块，调用 Find_Match 过程，要查找与 rid 相匹配的元组 Trid。块中元组数为‖R‖/|R|，为此，进行的比较运算时间为（‖R‖/|R|）* comp，共有 IOrf 块，所以比较时间为（‖R‖/|R|）* IOrf* comp；对于 Insert 操作而言，要把关系 R 缓冲区中的元组移入到 Rk 中，每个移动操作的时间为 move，则所需时间为（‖R‖* SR/|N|）* move；所以 CPU 时间为：CPU r=（‖R‖/|R|）* IOrfcomp+（‖R‖* SR/|N|）* move。

在 JIk 和 Rk 读入内存这个过程完成后，将对 JIk 进行排序，排序的计算开销为

$$CPUst = \frac{\|JI\|}{N}\left(\log 2 \frac{\|JI\|}{N}\right) * comp + \left(\frac{\|JI\|}{N}\right) * move$$

作为最后一步，完成与关系 S 的连接，其 I/O 和 CPU 的分析与前相似，只是要注意的是，由于 JI 按 rid 聚集，而不是按 sid 聚集的，所以 sid 的范围就将分散在整个 S 表中，为此，查找（‖S‖* SS）/N 个 S 中的元组要读取的 S 文件的块数为

$$IOsf = Y\left(\frac{\|S\| * SS}{N}, |S|, \|S\|\right)$$

读取索引文件的块数为

$$IOsci = Y(IOsf, |S|/FO, |S|)$$

为此，相应的 IO 时间为 IOs=（IOsf+ IOsci）* IO。

CPU 时间为：CPUs=（‖S‖/|S|）* IOsf* comp+（‖S‖* SS/|N|）* move。

所以，每遍的处理时间为 Ti= IOr+ IOs+ CPUr+ CPUs+ CPUst+ |JI|/N，算法所用的总时间为

$$T = \sum_{i=1}^{N} Ti = N * Ti$$

此据，对算法的执行时间进行了模拟分析，M，FO，|JI|，|R|，‖R‖，|S|，‖S‖，SR，SS 的取值范围分别为

M	FO	JI	|R|	‖R‖
500~ 50 000	300~ 500	10~ 10 000	900~ 25 000	90 000~ 250 000

|S|	‖S‖	SR	SS
900~ 25 000	90 000~ 250 000	0.001~ 1	0.001~ 1

当 M，FO，|JI|，|R|，‖R‖，|S|，‖S‖，SR，SS 分别为 1024，300，100，3 000，50 000，2 000，35 000，1，1，关系 R 为外关系时，连接运算的 IO 次数估计为 9018，反之，S 为外关系时，仅为 8024，而连接索引中不同的 r 的数目明显多于 s，有人认为此时关系 R 应作为外关系[3]，这与上述是不一致的，为此，把谁作为外关系应当依据性能分析来确定，在一般情况下，仍可使尺寸小的为外关系。

3 三元及多元连接谓词索引

上面介绍了二元连接谓词索引及连接算法的性能分析，事实上，上述算法还可推广到三元或多元，当连接谓词为 P（X.A，Y.B，Z.C）时，其三元连接索引谓词表示表 X、Y、Z 中的元组满足 P 的元组标识三元偶对（xid，yid，zid）的集合。同样，可以利用该索引来进行连接运算。为篇幅所限，这里并不给出基于连接谓词索引的三元连接算法，而只给出相对于上述二元连接的不同点。三元连接 Join（X，Y，Z），可以依二元连接算法来进行，即首先读入连接索引 JI，找到相应的 X 关系中的有关元组，与二元连接一样，尽量可能多地读入 JI 和 X 中的元组来减少外层循环的次数（遍数），但这时

将遇到与二元连接中不一样的问题，即对内存中的 JI 按 yid 排序后，无法再对 zid 排序了，这是因为没有内存空间来存放 Y 关系中的元组了，只能在按排序的 yid 读出所需的 Tyid 后，再根据 zid 去找 Z 表中的相应元组，并在形成结果元组后输出。由于无法对 zid 排序了，这样对 Z 表的查找将是不利的，这种不利表现在两个方面：首先是对同一 Tzid 的多次读取将更为严重，而在排序时，相同的会排在一起，只要在读入缓冲区中多次查找则可，而不必引起 I/O，对没排序的 zid 而言，关系 Z 的同一块的多次读入将难于避免了（在排序情况下，在不同遍中，可能引起对同一块的多次读入）。其次，在没有排序的情况下，访问的邻接性更差（假定 Z 关系按 zid 聚集），这样也将导致 I/O 的增大，在最坏情况下，每个 Tzid 的查找将导致读入 Z 关系中的一个数据块。

为此，可对上述算法过程做一个改进，即为关系 Y 留出足够的空间，使得算法过程为：首先读入适量的 JIk 和 Rk，然后对 JIk 按 yid 排序，把相应的 Tyid 读入内存，再次对 JIk 按 zid 排序，根据排序的 zid 来获取 Tzid，并最终形成结果元组。为了能给 Yk 留出合适的空间，可以先分析 JI 中不同的 xid 和 yid 个数，关系 X 的元组尺寸为 Txsiz，Y 的元组尺寸为 Tysiz，则当为 Xk 分配一块时，随即为 Yk 预留 M″块，M″= ‖yid‖* Tysize/（‖xid‖* Txsize），‖yid‖和‖xid‖分别为 JI 中 yid 和 xid 的个数，M″用于 Yk 的读入过程，这样，算法将分成 N= （|JI| + |X| * SX+ |Y| * SY） /M 遍完成（即外 while 循环的循环次数）。

多元连接谓词索引和多元连接算法可按上述三元算法的思想进行，使之从三元推广到多元。同样，可对三元或多元算法给出类似于二元时的性能分析。

4 结语

针对对象关系数据库中用户可定义一般连接谓词的情况，给出了基于连接谓词索引的连接算法和性能分析，并把二元连接索引推广到一般形式。值得指出的是，由于索引的存在，要影响到其他操作过程，如对关系表的插入和删除操作，给这些操作带来新的额外开销，有关这方面的分析可参考属性索引时的处理办法加以解决。另外，通过研究表明，把连接属性上不同值的个数较多的那个关系作为外关系进行连接，将使连接运算代价较小的说法是不正确的，为此，需要在查询处理优化过程中分别计算把 R 或 S 作为外关系时的代价，从中选出代价小的作为连接过程。显然，带连接索引的连接运算不像两关系在没带索引时只需把小关系作为外关系那样简单，因为为了分析的准确，引入了较复杂的分析公式，但通常情况下，把小关系作为外关系仍是可取的。

参考文献:

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

[2] 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).

[3] Valduriez P. Join Index. ACM TODS [J]. 1987, 12 (2): 219-246.

[4] Shapiro L D. Join Processing in Database Systems With Large Main Memories [J]. ACM TODS 1986, 11 (3): 239-264.

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