关系运算算法

本小节主要介绍常用的关系操作算法，如选择、投影、连接，并给出各操作算法的执行代价估算。

选择运算算法

> [例7.2] 查询学号为2022001的学生。
>
> ```
> SELECT *
> FROM Student
> WHERE Sno = '2022001' ;
> ```

上述查询语句中，Student表中的数据以物理页的方式进行组织。表中的学生信息可以存放在多个物理页中，每个物理页可以存放多个学生的信息。多个物理页可以通过Innode结构或者B+树索引结构组织起来，如图7.4所示。图中的索引是基于主码Sno构建的，索引的层数为L。Student表占用了B个物理页。

选择运算只涉及一张表，一般采用全表扫描或者索引扫描的算法。

- 全表扫描（Table Scan）：首先基于Innode结构读取Student的第一个物理页到内存，然后检查内存中的每个元组，如果该元组满足选择条件则输出，最后重复上述操作直到处理完Student的B个物理页。全表扫描的执行代价为B，即cost=B。
- 索引扫描（Index Scan）：首先基于B+树结构找到“2022001”所在的叶子结点，然后通过叶子结点上的指针找到对应的物理页。整个执行过程中，需要读取B+树中从根结点到叶子结点L个物理块，以及读取查询结果所在的物理页数。因为Sno是主码属性，只会对应一个物理页，因此，该算法的执行代价为L+1，即cost=L+1。假设选择条件是非主属性的等值查询，那么索引扫描的执行代价为L+S，即cost=L+S，其中S是索引的选择基数（有S个元组满足条件），最坏的情况是S个元组存放在不同的物理页中。

基于上述描述可以得出：小表的选择运算通常采用全表扫描算法；对于大表的选择运算，如果有索引结构且索引的选择基数较低时，通常采用索引扫描算法；如果索引选择基数较高，或者查找的元组均匀地分布在表中（存储在B个物理页中），那么索引扫描算法性能则不如全表扫描算法。因为，此时的索引扫描不仅要读取表的物理页，还要对B+树进行扫描。

投影运算算法

> [例7.3] 查询学生表中学生所在的系。
>
> ```
> SELECT DISTINCT Dept
> FROM Student;
> ```

该查询语句中包含选择运算和投影运算，首先通过全表扫描算法依次读取Student表的元组，得到每个元组的Dept值；然后使用投影运算算法对得到的Dept值进行去重。

投影运算的关键是对属性列的值进行去重。为了便于去重操作，需先对属性值进行排序。目前，排序算法可以分为内排序和外排序。内排序算法用于所有数据都能存放于内存的情况，常用的算法有快速排序、插入排序、冒泡排序和堆排序等；外排序算法用于数据量太大而不能全部存放于内存的情况，常用的算法有两阶段多路归并排序算法。关系数据库管理系统的投影运算通常采用外排序算法。

外部排序算法的步骤如下：

- 第一步生成归并段：将所有数据分成多个内部已排序的归并段。假设生成k个归并段，那么首先从磁盘中读取第一个分段的数据，采用内排序算法进行排序，然后将排好序的分段写回磁盘，最后重复上述操作直到处理完k个归并段。整个过程涉及一次读数据和一次写数据，假设数据占用了B个物理页，则此阶段的执行代价为2B。
- 第二步多路归并排序：将所有归并段合并成一个排好序的大段。首先读取每个归并段的第一个数据，计算出最小的值作为输出；然后取最小值所在归并段的下一个数据，重复上述过程计算出最小的值；最后依次找出k个归并段中的所有数据。整个过程涉及一次读数据，因此执行代价为B。

投影运算的去重操作在外部排序的第二步中实现，即输出的最小值不重复。投影运算算法的执行代价为2B+B，即cost=3B。

连接运算算法

> [例7.4] 查询学生及其选修课程的信息。
>
> ```
> SELECT *
> FROM Student S, SC R
> WHERE S.Sno = R.Sno;
> ```

上述查询语句是Student表（S表）与SC表（R表）在主码Sno上的等值连接运算。常用的连接运算算法有：嵌套循环连接（Nested Loop Join）、排序合并连接（Merge Join）、散列连接（Hash Join）以及索引连接（Index Join）。下面将分别进行介绍。

（1） 嵌套循环连接

这是连接运算最简单直观的算法。算法的思想为：首先读取左表S的第一个元组，然后将该元组与右表R的所有元组进行等值连接条件检查，如果两个元组在连接属性Sno上相等，则串接后作为结果输出，之后依次读取S表余下的元组重复上述操作。这其实是一个双层嵌套循环的过程。

在关系数据库管理系统中，数据的存取是按照物理页读入内存的，而不是按元组读入的，因此，实际的嵌套循环连接算法步骤如下（假设内存可以容纳M个物理块）：

- 第一步， 首先读取左表S的M-1个物理块到内存中；然后读取右表R的1个物理块到内存，之后依次检查内存中S表的所有元组与R表的所有元组是否满足连接条件，如果满足则输出；最后读取右表R余下的物理块并重复上述操作；
- 第二步，读取左表S余下的物理块并重复第一步的操作。

嵌套循环连接算法的伪代码如下所示。

```
/*嵌套循环连接算法伪代码*/
Do load M-1 blocks MS from S into memory
    Do load One block MR from R into memory
        For each tuple s in MS Do
            For each tuple r in MR Do
                If(s.joinkey = r.joinkey)
                Then output(s,r);
            End
        End
    Repeat;
Repeat;
```

假设S表占据B(S)个物理页，R表占据B(R)个物理页，那么嵌套循环连接算法的执行代价为：cost=B(S)+B(S)B(R)/(M-1)。该算法的执行代价比较高，尤其是对两张大表的等值连接。通常，采用嵌套循环连接算法时会选择较小的表（占用页数较少的表）作为左表S，这样可以减少算法的I/O代价。如果B(S)小于M-1，那么执行代价仅为B(S)+B(R)。

（2） 排序合并连接

排序合并连接算法是等值连接常用的算法。该算法的步骤如下：

- 第一步，使用外排序算法对Student表（S表）和SC表（R表）按连接属性Sno进行排序；
- 第二步，首先读取S表和R表的多个物理页到内存中；然后依次读取内存中两个表的元组进行连接条件检查。如果两个元组的Sno相等，则将它们串接起来进行输入，再读取S表的下一个元组继续进行检查；如果两个元组的Sno不相等，则读取Sno较小元组所在表的下一个元组，再进行连接条件检查。如果某张表在内存中的元组被处理完，那么从磁盘中读取该表的物理页到内存，重复之前的操作直到某个表的所有元组被扫描结束。

该算法的伪代码如下：

```
/*排序连接算法伪代码*/
Sort S and R by Join key
Do load blocks MS from S and blocks MR from R into memory
    Read the first tuple s from MS
    Read the first tuple r from MR
    While NOT EOF of MS or MR
        If (s.joinkey = r.joinkey)
            output(s,r) and Read next tuple s from MS
        ElseIf (s.joinkey < r.joinkey)
            Read next tuple s from MS
        ElseIf (s.joinkey > r.joinkey)
            Read next tuple r from MR
    End
Repeat;
```

该算法的执行代价为：cost=5(B(S)+B(R))，其中，第一步排好序的S表和R表需写入磁盘，其执行代价为4(B(S)+B(R))；第二步中S表和R表只扫描一次，其执行代价为B(S)+B(R)。排序合并连接算法适用于两个表已经按照连接属性排序的场景，此时它的执行代价仅为B(S)+B(R)。

（3） 散列连接

散列连接算法的思想是：首先利用基于连接属性Sno的散列函数将S表和R表分别划分成多个小表（S1,S2,S3...Si..）和（R1,R2,R3...Ri...）,S和R每个对应小表中的Sno取值范围一致，即Si表中元组的Sno与Ri表中元组的Sno取值一致；然后分别对S和R的对应小表做连接操作，直到处理完所有对应的S小表和R小表并输出连接结果。

散列连接算法的步骤如下（假设内存能够容纳M个物理页）：

- 第一步， 连接属性Sno作为散列关键字，利用相同的Hash函数分别将S表和R表划分成M-1个小表。对于S表的预处理，每次读取一个物理页到内存，利用Hash函数将物理页中的元组写入对应的Hash桶中（总共有M-1个Hash桶），最后再将Hash桶中的元组写入磁盘物理页中。利用同样的方法对R表进行预处理；

- 第二步， 读取S表和R表的第一个小表S1，R1的所有物理页到内存，然后依次检查内存中S1表的所有元组与R1表的所有元组是否满足连接条件，如果满足则输出串接结果，之后重复上述操作直到处理完S表和R表的所有小表。

该算法的伪代码如下：

```
/*散列连接算法伪代码*/
//对S表利用散列函数Hash进行划分
Do load block MS from S into memory
    For each tuple s in MS Do
        i = Hash(joinkey)
        Si = Si union s
    End
Repeat;
//对R表利用散列函数Hash进行划分
Do load block MR from R into memory
    For each tuple r in MR Do
        i = Hash(joinkey)
        Ri = Ri union r
    End
Repeat;
//对S表和R表的每一个小表进行连接
For i=0:N
    Do load blocks from Si and Ri into memory
     For each tuple s in Si Do
        For each tuple r in Ri Do
            If(s.joinkey = r.joinkey)
            Then output(s,r);
        End
    End
End
```

该算法的执行代价为：cost=3(B(S)+B(R))，其中，第一步需要分别对S表和R表进行一次读和一次写，其代价为2(B(S)+B(R))，第二步需要对S表和R表读一次，其代价为B(S)+B(R)。上面介绍的散列连接算法假设两个表的对应小表在第二步中可以完全放入内存中。如果不需要这个前提条件的散列连接算法以及改进算法可以参见其他参考资料。

（4）索引连接

索引连接算法是利用连接属性上的索引来实现连接操作。例7.4中，Student表（S表）在Sno上构建了B+树索引，SC表（R表）在Sno和Cno上构建了B+树索引结构。SC表上的索引称为组合索引，该索引的层数为L，索引在Sno上的选择基数为V（平均有V个元组具有相同的Sno值）。

算法的执行步骤如下：

- 第一步，读取S表的第一个物理页到内存，获得内存中S表每个元组的Sno，然后根据R表的索引结构找到R表中具有相同Sno值的元组，之后将S表的元组与R表的元组进行串接作为输出结果；
- 第二步，读取S表剩余的物理页，并重复第一步操作。

该算法的伪代码如下：

```
/*索引连接算法伪代码*/

Do load blocks MS from S into memory
For each tuple s in MS Do
    List r[] = Search R-index(s.joinkey)
    Do load sizeof(r[]) blocks MR from R into memory
    For each tuple r in r[]
        output(s,r)
    End
End
```

该算法的执行代价为：cost=B(S)+|S|*L*V，其中|S|表示S表中的元组个数。整个执行过程中，S表只需要读取一次，其代价为B(S)，而对于S表中的每一个元组都需要读取一次R表的索引结构找到满足连接条件的叶子结点，然后根据叶子结点的指针读取R表的物理页，其代价为|S|*L*V。