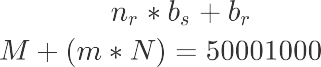
# 块嵌套循环连接

# 设计思想

连接运算需要设计到两个关系，其中嵌套循环算法是最简单的，适用于任何连接条件。但是其代价是非 常高的，由计算公式: 可得，假设Table R:M=1000,m=100000;Table S:N=500,n=40000. 最坏的代价: ，其时间复杂度 ，所以嵌套循环算法是需要进行优化的，因缓冲区太小而内存无法完全容纳任何一个关系时，以块的方式而不是以元组的方式处理关

系，可以减少不少块读写次数。这里使用了块嵌套循环算法。

**Block Nested-Loop Join**(块嵌套循环) 其优化思路是减少内层表的扫表次数，通过观察简单的嵌套循环查询，我们可以看到，左表的每一条记录都会对右表进行一次扫表，扫表的过程其实也就是从内存读取 数据的过程，那么这个过程其实是比较消耗性能的。所以缓存块嵌套循环连接算法意在通过一次性缓存 外层表的多条数据，以此来减少内层表的扫表次数，从而达到提升性能的目的。

在mysql 中使用了join buffer 的缓冲区去减少循环次数，这个缓冲区默认是256KB，可以通过命令

show variables like 'join\_%' 查看。

# 实现方案

1. Block Nested-Loop Join 通过一次缓存多条数据批量匹配的方式来减少外层表的IO次数,同时也减少了内层表的扫表次数
2. 伪代码算法实现如下

1

2

3

4

5

6

7

8

9

10

11

12

13

14

15

16

17

18

19

20

21

for each row in t1 matching range {

for each row in t2 matching reference key { store used columns from t1, t2 in join buffer if buffer is full {

for each row in t3 {

for each t1, t2 combination in join buffer {

if row satisfies join conditions, send to client

}

}

empty join buffer

}

}

}

if buffer is not empty { for each row in t3 {

for each t1, t2 combination in join buffer {

if row satisfies join conditions, send to client

}

}

}

1. 我们对基于元组的嵌套循环连接进行一下改进：每次讲尽可能多的第一个表中的元组放在内存，然 后对它们建立一个内存查找结构，即我们之前建立的AVL树;然后依次取出第二个表中的元组，根据这个元组连接属性上的值查询我们建立的第一个表的内存查找结构，若能找出能和这个元组连接的 第一个表中的元组，则把连接结果加入到结果表中。不断重复这样的步骤，直到我们取出了第一个 表中的元组为止。很明显，当第一个表能一次都放进内存时，这个算法就成了等值连接的一遍算 法。
2. 实现算法的C++代码如下:
   1. //迭代器
   2. struct iterator{ 3 //区间

4 int extent; //current which extent 5 //块

1. int blk; //in which block of the cur extent
2. //存储了当前取的块
3. char b[PAGE\_SIZE];
4. //记录位置
5. int t; //the tup of the cur page
6. int len; //tup len 12 };

13

1. struct iterator open\_iter(const char \*tbname);
2. int getnext\_iter(struct iterator \*iter,char \*rec);
3. //fetch next tuple,store in rec
4. int close\_iter(struct iterator\*iter); 18
5. //显示结果表里的所有记录
6. int print\_tuple(char \*rec,struct table\_def \*td); 21
7. //显示结果表的表头
8. int print\_tbhdr(struct table\_def\*td); 24
9. struct BNode{
10. int num;
11. int bf;
12. KeyType key;
13. InfoTyp data;
14. struct node \*lchild,\*rchild;
15. struct node \*same; 32 };

33

34 struct BNode \*InsertBT(struct BNode \*b,struct Record e,int \*taller); 35

1. //返回AVL树中具有相应key值的所有元组
2. struct Result \*SearchBt(struct BNode \*b,KeyType k); 38

39 int nest\_blk\_join(const char \*tb1,const char \*tb2) 40 {

1. BNode \*avl=NULL;
2. print\_tbhdr(td);
3. iterb1=open\_iter(tb1); 44
4. //建立内存查找结构
5. while(getnext\_iter(&itertb1,rec1)==0) 47 {

48 avl=InsertBT(avl,e,&taller); 49 }

50

1. itertb2=open\_iter(tb2);
2. while(getnext\_iter(&iterb2,rec2)==0) 53 {

54 //在内存查找结构中查找能进行连接的元组

55

56

57

58

59

60

61

62

63

64

65

res=SearchBT(avl,key); while(res!=NULL)

{

//进行连接并且输出print\_tuple(rec,td); res=res->netx;

}

}

return 0;

}

## 数据结构

基于块的嵌套循环连接中用到内存的查找结构：能在接近常量的时间内增加一个新元组，查找一个元组 是否存在，在这里使用了AVL树，平衡搜索树。在管理方面上AVL是比较有优势的。

BNode 是AVL树的结点的定义，每个结点里面保存了相应的key 值和相应的元组数据和其他的一些控制信息;

InsertBT() 函数实现了向AVL树中插入一条元组的功能;

SearchBT() 函数是返回AVL 树中具有相应key 值的所有元组;

## 文件结构

使用的是顺序文件.

文件中的记录一个接一个地顺序排列，记录可以是定长的或变长的，可以顺序存储或以链表形式存储，在访问时需要顺序搜索文件。顺序文件有以下两种结构：

第一种是串结构，记录之间的顺序与关键字无关。通常的办法是由时间决定，即按存入时间的先后排列，最先存入的记录作为第1个记录，其次存入的为第2个记录，依此类推。

第二种是顺序结构，指文件中的所有记录按关键字顺序排列。

## 实现细节

1. 特点：以元组（记录）为单位进行操作，按页操作。
2. 块嵌套循环连接：以块的方式循环，以减少块读写次数。
3. 一些改进：

连接属性是内层关系的码时，内层循环可中途跳出。

1. 外层关系不是每次读入一个磁盘块，而是每次读入内存最多能容纳的磁盘块数。即，如果内存有M 块，则一次读取外层关系r中的M – 1块，当读取到内层关系中的每一块时，把它与已读取到内存的外层关系中的M – 1块做连接。于是内层关系的扫描次数从次减少到



1. 内层循环轮流做正、反向扫描，重用缓冲区中的数据，以减少磁盘读取。
2. 内层循环利用索引。
3. 连接缓冲区在执行连接之前分配并在查询完成后释放。