DBMS PROJECT

组号：个人成组

姓名：翁镇斌

学号：10389079

班级：10计应（2）班

1. 项目题目

可扩展哈希的实现。算法实现分为两大部分，第一部分是建立索引，第二部分是查询。建立索引是将输入的每一条记录根据指定的键值放入合适的哈希桶内，当哈希桶已满时，需要进行分裂。查询是根据输入的键值返回具有相同键值的记录，返回的记录可能有不止一条。

1. 实验环境

操作系统：Windows 7

IDE：Microsoft Visual Studio 2010

1. 实验思路
2. 假设有P个缓冲页，在本实验中，P等于8或者128。
3. P个缓冲页的分配方式如下，1个缓冲页用于读取 lineitem.tbl 中的数据，I个缓冲页用于存放索引，B个缓冲页用于存放哈希桶，P、I、B满足以下等式：

P=I+B+1。

当P=8时，令I=2,B=5;

当P=128时，令I=3,B=124。

1. 程序生成的索引文件放在hashindex.out中，另外有一个长度为4个BYRES（即1个long型的长度）的文件globalDepth.out，用于记录索引的全局深度，哈希桶全部放在LineItemRecord.out这个文件中。上述文件都以二进制的形式存储在硬盘中。
2. 哈希函数的设计（假设当前的全局深度为depth）。

从低位进行拓展的哈希：将orderkey对应的二进制数取后depth位，得到一个long型的整数，返回该值。

从高位进行拓展的哈希：保持orderkey对应的二进制数的最高位不变，即最左边的1不变，低位的数全部反转。然后取后depth位，得到一个long型的整数，返回该值。例如，当orderkey为11时，其对应的二进制数为1011，保持最高位不变，剩下的低3位全部反转，得到1110。当depth=1时，哈希函数返回0；当depth=2时，哈希函数返回2；当depth=3时，哈希函数返回6；当depth>=4时，哈希函数均返回14。

**下面将描述建立索引的设计思路。**

1. 对于lineitem.tbl，每次以二进制的格式读取8KB（即1个页面）到原始数据缓冲区。
2. 对于每一条记录，截取记录的orderkey，对它应用哈希函数，得到这条记录在索引文件中的偏移量，于是查找索引文件，得到该记录应该被放入的桶的编号。
3. 尝试将记录放入对应的哈希桶中。如果哈希桶没有满，则记录插入成功（记这种情况为Case 0#）。读取下一条记录，返回步骤F。
4. 如果哈希桶剩余的空间不够插入该记录，则应分裂。哈希桶的localDepth加1，并申请一个新桶，然后根据当前的localDepth，看哈希桶中的数据的orderkey从右往左数的第localDepth位，在一个bool型的数组中记录下这些区分位。如果区分位等于0，则该记录应该留在原来的桶，否则，这条记录应该被复制到新的桶中。
5. 如果分裂之后，原哈希桶为空，即所有的记录都移动到新的桶（记这种情况为Case 1#），那么只需要将原来桶的赋值给新的桶，同时修改新桶的桶号。如果分裂出来的两个桶都非空（记这种情况为Case 2#），那么扫描步骤F得到的用于区分的bool型数组，如果当前值为false，那么这条记录留在原来的桶中，否则，这条记录会被复制到新的桶，并在原哈希桶的这条记录的有关信息上做一个标记（在该条记录的长度上记为-1）。对于Case 2#，原来的桶中的数据已经出现了空的槽，所以应当调整原哈希桶中的记录以及重新设置桶和记录的相关信息。如果分裂之后，新的哈希桶为空，即所有的记录都留在了原来的哈希桶（记这种情况为Case 3#），保留这个空桶在哈希桶文件中。
6. 总之，对于插入一条记录，总是属于Case 0~3#中的一种，对于Case 0#，前面已经说过，是插入成功，读取下一条记录。对于Case 1~3#，即桶需要分裂的情况，我们得到新桶的ID（即桶号），以及分裂的桶的localDepth，如果localDepth大于globalDepth，那么目录应当加倍，同时，对于插入不成功的记录，对它的orderkey需要再次应用哈希函数计算，否则，目录无需加倍也不需要再次计算哈希值。不管目录是否加倍，得到了新桶的ID号之后，需要对目录进行调整，称之为重定位（relocation），并重新在目录中查找记录应当插入的桶号。返回步骤G。
7. 如果原始数据的缓冲页中记录已经读取完了，要从lineitem.tbl中读取下一页原始数据，即返回步骤E，知道lineitem.tbl读取完毕，所有的记录都已经建立了索引。

**下面将描述查询数据的设计思路。**

1. 从testinput.in文件中，格式化读取需要查询的数据的总数。
2. 格式化读取orderkey，对它应用哈希函数，然后在目录中查找，得到哈希桶的ID，再在对应的哈希桶中查找。
3. 定义一个结构体HitRecord的数组（详见第四部分的数据结构设计）。在对应的哈希桶中（通过实验我们得知，每个orderkey对应的记录最多不超过10条，因此如果存在匹配的记录，这些记录可以在一个桶中全部查到），如果找到匹配的记录，记下它在该桶的slotId和记录的partkey。
4. 对步骤N得到的记下匹配记录的数组，根据它们的partkey调用C标准库中的函数qsort按照升序进行排序，然后，将这些匹配的记录都以二进制的格式输出到文件testoutput.out中。返回步骤M，直到读完所有需要查询的数据。
5. 数据结构设计
6. **索引页面**

const int page\_size = 8192;

const int index\_num = page\_size / sizeof(long) - 1;

struct indexPage

{

long indexPageId; // 索引页的ID

long index[index\_num]; // 索引，记录哈希桶的桶号

};

索引页面中，需要一个记录索引页面号的ID，剩下的空间都是用于记录哈希桶桶号的。于是，通过这个索引页，我们可以建立一个二级索引，即假设现在有一个目录文件的总的偏移量 dir\_num，只要将 dir\_num / index\_num 就可以得到索引页的ID，将dir\_num % index\_num 就可以得到在该页中的偏移量。于是，我们可以找到dir\_num指向的桶号。

在文件hashindex.out中，我们想找一个页面，假设它的页面ID为 indexPageId，那么只需要打开hashindex.out，使用fseek函数跳过 indexPageId \* page\_size 个BYTE即可定位到该页面，将其读入内存。

1. **桶页面**

/\* Through a simple test we know that each record takes average 120

Bytes approximately. Since 8192 / 120 = 68, we assume that the maximum

number of records in a page is 70. The remain memory space for record

data is 8192 - sizeof(long)\*4 - sizeof(slotDir) = 7616. \*/

const int record\_num = 70;

const int char\_num = 7616;

struct RecordInfo

{

long offset; //记录在桶中的偏移量

long length; //该记录的长度

};

struct bucketPage

{

char data[char\_num]; //存放记录的字符数组

RecordInfo slotDir[record\_num]; /\* 在桶中记录的索引，包括记录的偏移量和长度等信息 \*/

long slotNum; //桶中记录的数目

long toFreeSpace; //指向桶中存放记录的字符数组空闲位置的开始

long localDepth; //该桶的局部深度

long bucketId; //该桶的桶号

};

桶页面中，采用了变长记录的方式进行存储，具体的数据结构见类 bucketPage。通过一个小实验，我为桶中的数据做了一个安排，指定了桶中最多可以存储70条记录。在这个实验中，我们通过观察lineitem.tbl的数据，知道是不会产生溢出页的，因此，桶上我们不允许挂溢出页。

同时，我也设计了一个结构体 HitRecord，用于查询时可以记录匹配的记录在桶中的相关信息，具体如下。

struct HitRecord

{

long slotId; /\* 该匹配记录在桶中的槽号，记下这个信息可以通过桶的索引 slotDir 快速地定位到这条记录 \*/

long partkey; /\* 该记录的 partkey，记下这个信息我们可以根据 partkey 对匹配的记录进行排序 \*/

};

1. **管理索引页面的类**

const int indexBuf\_size

/\* 索引缓冲页面的个数 indexBuf\_size 会随着总缓冲页面的个数而变化，当P=8时，indexBuf\_size 的值为2，当P=128时，indexBuf\_size 的值为3 \*/

class IndexPage

{

// 类 IndexPage 的各种成员函数

private:

char cmd[100]; //索引文件的路径

char globalDepthFile[100]; //存放全局深度的文件的位置

long globalDepth; //全局深度

int current; //用于时钟页面算法，相当于时钟表面的指针

int pageNum; //索引页的总个数

int pin\_count[indexBuf\_size]; /\* 用于时钟页面算法，可以像别针一样固定住一个页面 \*/

bool ref\_bit[indexBuf\_size]; /\* 用于时钟页面算法，标识着该页面最近是否被使用过 \*/

bool dirty[indexBuf\_size]; //用于时钟页面算法，标识着该页面是否被修改过

indexPage indexBuf[indexBuf\_size]; /\* 索引页面的缓冲池。当P=128时，改用指针数组，即 indexPage \* indexBuf[indexBuf\_size]; \*/

};

1. **管理桶页面的类**

const int bucketBuf\_size

/\* 哈希桶缓冲页面的个数 bucketBuf\_size 会随着总缓冲页面的个数而变化，当P=8时，bucketBuf\_size 的值为5，当P=128时，bucketBuf\_size 的值为124 \*/

class Bucket

{

// 类 Bucket 的各种成员函数

private:

char cmd[100]; //桶文件的路径

int current; //用于时钟页面算法，相当于时钟表面的指针

int pageNum; //桶页面的总个数

int pin\_count[bucketBuf\_size]; /\* 用于时钟页面算法，可以像别针固定住一个页面 \*/

bool ref\_bit[bucketBuf\_size]; /\* 用于时钟页面算法，标识着该页面最近是否被使用过 \*/

bool dirty[bucketBuf\_size]; //用于时钟页面算法，标识着该页面是否被修改过

bucketPage bucketBuf[bucketBuf\_size]; /\* 桶页面的缓冲池 \*/

};

1. **页面管理综述**

在本实验中，我把缓冲页面分成三种，一种用于存放原始数据（即从lienitem.tbl中读取的数据），一种用于存放目录，剩下一种用于存放哈希桶。存放原始数据的页面只需要1个，所以不需要设计页面替换算法。而另外两种页面，存放目录和哈希桶的页面需要设计页面替换算法。

在管理页面的类中（包括管理索引页面和桶页面的），我设计了如下函数用于管理页面的函数。

int checkPage(const long pageId); /\* 检查页面号为 pageId 的页面是否在页面缓冲池中，如果存在，返回该页面在缓冲池的ID，即表明页面在第几个缓冲区域，否则，返回-1 \*/

int requestAPage(); /\* 向系统在缓冲池中申请一个缓冲区域，返回缓冲区域的ID \*/

int clockReplace(); /\* 使用时钟页面算法在缓冲池中选取一个可以被替换出去的缓冲页面，返回该区域的ID \*/

void writeBufPage(const int buf\_id); /\* 将缓冲池中第 buf\_id 个缓冲区域写出到桶文件中 \*/

void readBufPage(const int buf\_id, const long pageId); /\* 将页面号为 pageId 的页面读入到缓冲池第 buf\_id 个缓冲区域 \*/

int pinAPage(const int pageId); /\* 将页面号为 pageId 的页面固定在缓冲池中 \*/

页面调度算法如下。

1. 根据一个页面号 pageId 向系统申请一个页面，即调用 pinAPage 函数时，系统首先会调用 checkPage 函数检查缓冲池区域里面是否存在这个页面，如果存在，那么checkPage 函数返回缓冲区域的ID。假设该ID为 buf，pinAPage 函数将 pin\_count[buf] 加1，设置 ref\_bit[buf] 为 true。页面调度完成。如果 checkPage 函数返回-1，表明该页面不在缓冲池中，那么将执行步骤B。
2. 调用 requestAPage 函数向系统在缓冲池中申请一个缓冲区域。
3. requestAPage 函数将调用 clockReplace 函数在缓冲池中选择一个可以被替换出去的缓冲页面，如果该缓冲区域中的页面被修改过，即它对应的 dirty 标志位为 true，那么该页面需要被写回硬盘中，系统将调用 writeBufPage 函数将其写回，如果dirty 标志位为 false，那么该页面无需被写回，同时，由于该缓冲区域从逻辑上来说已经被清空，需要设置它的 pin\_count 为0，ref\_bit 和 dirty 标志位都为 false。requestAPage 函数最终将返回一个缓冲池中的缓冲区域的ID号，假设该ID值为 buf。
4. 此时，我们得到了缓冲池中空闲的缓冲区域，于是调用 readBufPage 将页面号为 pageId 的页面读入到缓冲池中的第 buf 个缓冲区域。同时，将 pin\_count[buf] 加1，设置 ref\_bit[buf] 为 true。页面调度完成。

需要注意的是，当缓冲池中的某个区域暂时不需要被用到时，应该将它对应的 pin\_count 应该减1，使得它有可能被时钟页面算法选中而被替换出去，否则，缓冲池中将很快就没有空闲的区域，页面调度将陷入死循环。

1. **处理建立索引与查询请求的类**

class HashIndex

{

// 类 HashIndex 的各种成员函数

private:

char cmd[100]; //源文件 lineitem.tbl 的路径

char queryInFile[100]; //查询输入文件的路径

char queryOutFile[100]; //查询结果输出文件的路径

char dataBuf[page\_size]; //用于存放原始数据（lineitem.tbl）的缓冲页面

IndexPage indexMgr; // 索引管理类的实例

Bucket bucketMgr; // 桶管理类的实例

clock\_t startTime, endTime; // 用于测试运行时间的变量

};

类 HashIndex 是与 main 函数直接联系的，当需要建立索引时，HashIndex 可以调用建立索引的函数，当需要查询时，HashIndex 中也有相应的接口。HashIndex 中还有用于从 lineitem.tbl 中读取原始数据的函数。通过类 HashIndex，我们建立起哈希索引数据库系统与外界通信的桥梁。

1. 实验过程设计
2. **时钟页面算法的伪代码**

当页面管理类（包括管理索引页面和桶页面的）创建的时候，current的值被初始化为0。

int clockReplace()

{

/\* flag = true when the clock replacement algorithm has been

implemented one time \*/

bool flag = false;

int tmp = current;

while (1)

{

如果 (current指向的缓冲池中的缓冲区域没有被固定住)

{

如果 (current指向的缓冲区域的ref\_bit为true，即该区域刚被使用过)

将这个值设置为false；

否则

返回current的值；

}

current加1，指向下一个缓冲区域，如果current的值等于缓冲池中缓冲区域的总个数，current将被设置为0；

如果 (tmp 等于 current)

{

如果 (flag为true，即时针页面算法的指针已经转过一圈了)

返回current的值；

将flag的值设置为true；

}

}

}

1. **索引目录加倍算法的伪代码**

index\_num 索引页面中每一页最多存放的目录条数

void doubling()

{

int src\_buf; //源页面在缓冲池中的缓冲区域ID

int dst\_buf; //目标页面在缓冲池中的缓冲区域ID

long src\_dir; //源索引条目在索引文件中的偏移量

long dst\_dir; //目标索引条目在索引文件中的偏移量

long src\_PageId; //源页面的ID

long dst\_PageId; //目标页面的ID

long total = 1 << globalDepth; //总共需要复制的索引条目的总数

src\_dir = 0;

src\_PageId = 0;

将0号索引页面固定住，将它在索引缓冲池中的区域ID号赋值给src\_buf；

dst\_dir = total;

dst\_PageId = dst\_dir / index\_num;

如果 (dst\_PageId 不等于 src\_PageId)

{

将索引页面ID为dst\_PageId的页面固定住，将它在索引缓冲池中的区域ID号赋值给dst\_buf；

}

否则

{

dst\_buf = src\_buf;

dst\_buf的pin\_count加1，ref\_bit设置为true;

}

dirty[dst\_buf] = true; // dst\_buf need to be written to disk

int i = 0, k = dst\_dir % index\_num;

while (1)

{

while (src\_dir < total && i < index\_num && k < index\_num)

{

indexBuf[dst\_buf].index[k] = indexBuf[src\_buf].index[i];

src\_dir,dst\_dir,i,k这四个变量都加1；

}

如果 (src\_dir 大于等于 total，即复制完成)

{

dst\_buf和src\_buf的pin\_count都减1，ref\_bit都设置为false;

break;

}

如果 (i 大于等于 index\_num，即源页面的指针已经指向页面的结尾)

{

src\_buf的pin\_count减1，ref\_bit设置为false; src\_PageId加1，指向下一页需要复制的索引页面；

if (src\_PageId 不等于 dst\_PageId)

{

将索引页面ID为src\_PageId的页面固定住，将它在索引缓冲池中的区域ID号赋值给src\_buf；

}

否则

{

src\_buf = dst\_buf;

pin\_count[src\_buf]++;

}

设置i等于0，指向新的源页面的开始；

}

if (k 大于等于 index\_num，即目标页面的指针已经指向页面的结尾)

{

dst\_buf的pin\_count减1，ref\_bit设置为false;

dst\_PageId加1，指向下一页需要复制的索引页面；

将索引页面ID为dst\_PageId的页面固定住，将它在索引缓冲池中的区域ID号赋值给dst\_buf；

dirty[dst\_buf] = true;

设置k等于0，指向新的目标页面的开始；

}

}

索引的全局深度加1

}

1. **桶分裂之后索引目录重新定位的伪代码**

initDir 为orderkey在本次插入操作之前在索引文件的偏移量

bucketId 为新分裂出来的桶号

initDepth 为initDir指向的桶分裂之前的局部深度

void relocation(const long initDir, const long bucketId, const int initDepth)

{

int buf;

long total, indexPageId, i, dir\_num, step = 1 << initDepth, jump = 1 << (globalDepth - 1);

total = jump / step; // 计算出总共有多少个索引条目需要重定位

dir\_num = initDir % step + step; //计算出应该从哪个索引条目开始重定位

step <<= 1; /\* step为两个相邻的需要重定位的索引条目之间的间隔 \*/

for (i = 0; i < total; i++, dir\_num += step)

{

获得需要重定位的索引条目所在的索引页面的ID，设其为indexPageId;

通过页面调度函数，将该页面载入缓冲池中，将其缓冲区域的ID赋值给buf;

将 bucketId 赋值给这一个需要重新定位的索引条目;

设置缓冲区域 buf 的 dirty 位为 true;

}

}

1. **桶分裂算法的伪代码**

buf\_id 缓冲池中第 buf\_id 个缓存区域中的桶需要分裂

bucketId 用于记录分裂出来的新桶的桶号

split 函数将返回分裂的情况标识

int split(const int buf\_id, long & bucketId)

{

int buf, i, k, cas;

bool splitBit[record\_num];

申请一个新的页面用于放置新分裂的桶，将这个缓冲区域的ID赋值给 buf;

将这个桶固定住，同时初始化该桶的相关数据；

for (i = 0; i < bucketBuf[buf\_id].slotNum; i++)

{

截取桶中记录的orderkey，并得到其分裂的标志位，记录在splitBit数组中；

如果 (该分裂位为1)

新桶的记录总数加1；

}

如果 (新桶的记录总数等于原来桶的记录总数)

{

将 buf\_id 指向的缓冲区域的值赋给buf指向的缓冲区域，同时更新buf指向的缓冲区域的桶号；

将 buf\_id 指向的缓冲区域的 slotNum 和 toFreeSpace 设置为0；

cas = 1;

}

否则，如果 (buf指向的缓冲区域中的桶的slotNum不等于0)

{

bucketBuf[buf].slotNum = 0;

for (i = 0; i < bucketBuf[buf\_id].slotNum; i++)

{

如果 (splitBit[i]为true)

{

将 buf\_id 指向的区域的第i个记录复制到新的桶中，同时更新新桶相关的信息；

将 buf\_id 指向的区域的第i个记录的长度设置为-1（做一个标记）；

}

}

cas = 2;

}

否则

{

cas = 3;

}

将 buf 指向的缓冲区域解除固定；

返回 cas 的值；

}

1. **当桶中出现空槽时调整桶中记录数据的算法的伪代码**

buf\_id 缓冲池中第 buf\_id 个缓存区域中的桶需要调整

void modifyInfo(const int buf\_id)

{

int i;

long toFreeSpace, length, slotNum = 0;

bool flag = false;

将 buf\_id 指向的缓冲区域固定在缓冲池中；

for(i = 0; i < bucketBuf[buf\_id].slotNum; i++)

{

如果 (flag为false 并且 buf\_id 指向的桶的第i个槽的长度为-1)

{

flag = true;

将 buf\_id 指向的桶的第i个槽的偏移量赋值给 toFreeSapce;

跳过下面的步骤马上进入下一轮循环；

}

否则，如果 (flag为真)

{

将 buf\_id 指向的桶的第i个槽的长度赋值给 length;

如果 (length 等于 -1)

跳过下面的步骤马上进入下一轮循环；

将 buf\_id 指向的桶的第i个槽的记录移动到 toFreeSpace 指向的该桶的字符数组的起始位置；

更新这个桶中第 slotNum 个记录的偏移量和长度；

toFreeSpace += length;

}

slotNum++;

}

更新 buf\_id 指向的桶的 slotNum 和 toFreeSpace；

将 buf\_id 指向的缓冲区域解除固定；

}

1. 代码结构
2. **建立索引的代码结构**

do{

读取一个页面的数据到原始数据缓冲区中；

截取每条记录的orderkey；

对该orderkey应用哈希函数找到它在索引中的偏移量；

在索引文件中查找对应的哈希桶；

while(1)

{

尝试将记录插入哈希桶中；

如果记录插入成功，跳出循环；

如果插入不成功，判断得到的桶的局部深度是否大于索引的全局深度，如果大于，索引目录加倍，并重新计算orderkey的哈希值；

重新定位索引目录；

}

} while(读取的原始数据等于8KB);

1. **查询数据的代码结构**

读取需要查询的数据的总数 queryNum;

for (i = 0; i < queryNum; i++)

{

读取需要查询的orderkey;

对该orderkey应用哈希函数找到它在索引中的偏移量；

在索引文件中查找对应的哈希桶；

调用查询函数；

将查询结果输出到 testoutput.out 中；

}

1. 性能优化
2. 不使用标准化读入原始数据，而是用二进制的形式一次性读入8KB的数据，提高I/O速度，减少I/O的次数。
3. 改用pow(2.0, x)函数为移位操作 1 << x，减少函数调用。
4. 将频繁使用的类成员函数改为内联(inline)函数，例如数字字符转化为整型的函数，哈希函数等。
5. 使用Windows的API进行优化。将需要频繁调用的文件读写模块的C库函数改为CreateFile，ReadFile，WriteFile，SetFilePointer 等函数。优化之后，原来创建索引的时候I/O占总用时约90%已经降低到现在的约65%，运行时间也有不同程度的减少。需要注意的是，编译的时候VS2010的项目属性应该调整字符集(Character Set)从原来的Use Unicode Character Set改为Use Multi-Byte Character Set。
6. 实验结果
7. **测试平台**

CPU：Pentium(R) Dual-Core T4300 2.10GHz

内存：2G

分区格式：NTFS

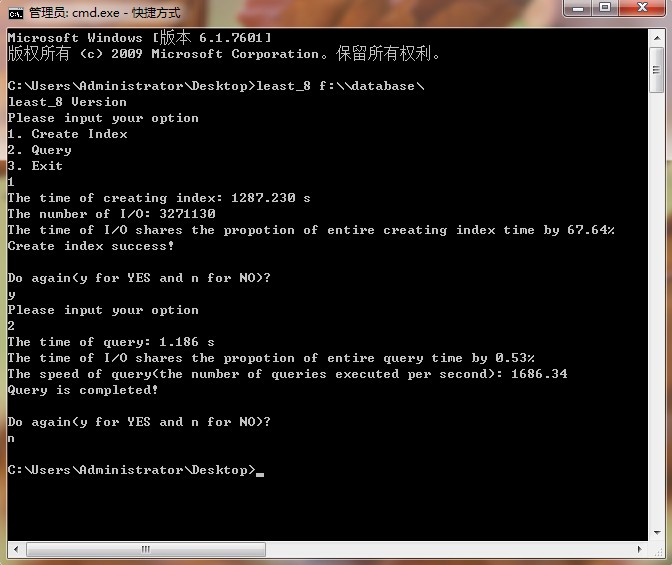
分区可用空间：约20G

操作系统：Windows 7

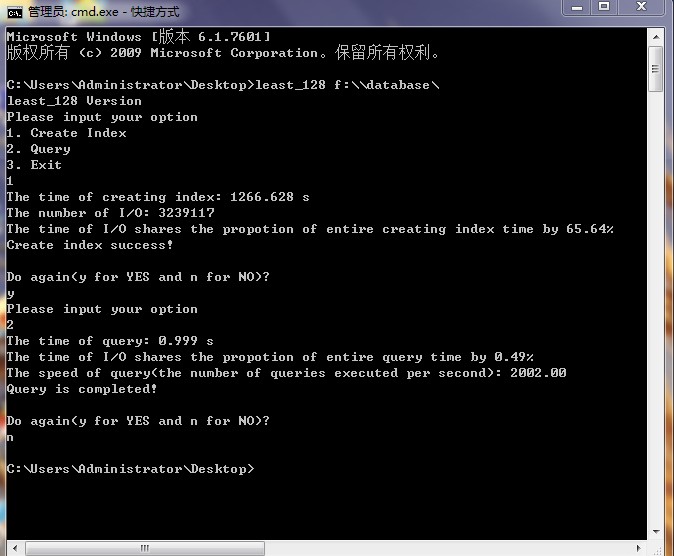
测试数据：730M

1. **测试结果**

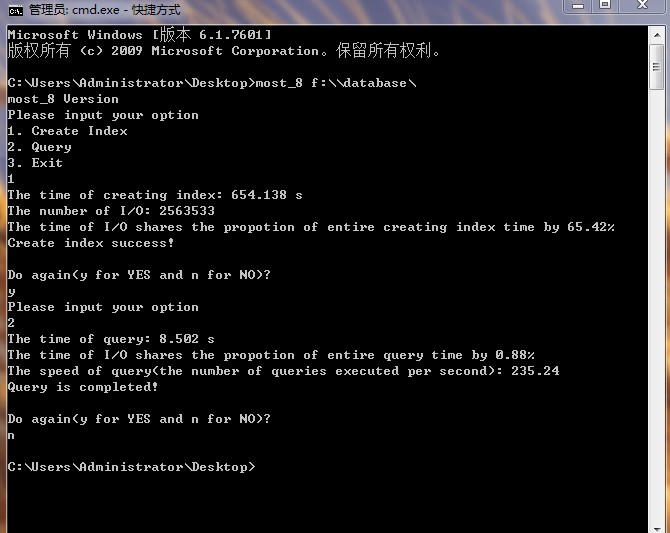
**least\_8版本测试结果截图**

****

**least\_128版本测试结果截图**

****

**most\_8版本测试结果截图**

****

**most\_128版本测试结果截图**

****

为了比较创建索引时不同版本程序的性能，根据实验结果可画出如下表格（见表一）。

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 版本 | 创建索引的时间 | I/O次数 | I/O占总用时的比例 | 索引文件的大小 | 桶文件的大小 |
| least\_8 | 1287 s | 3271130 | 67.64% | 4104KB | 1,076,064KB |
| least\_128 | 1267 s | 3239117 | 65.64% | 4104KB | 1,076,064KB |
| most\_8 | 654 s | 2563533 | 65.42% | 4104KB | 1,547,936KB |
| most\_128 | 626 s | 2488849 | 64.27% | 4104KB | 1,547,936KB |

表一

观察表一，我们可以发现同一种哈希方法，使用128个页面版本的程序都比使用8个页面版本的程序要快。这是因为页面缓冲多了，I/O次数就减少了，观察表中I/O次数的统计数字我们可以发现实验结果符合我们的假设。

从表中也可以看出，I/O是运行这个程序占用时间最多的部分。

而对比使用相同的页面缓冲页，高位拓展的哈希比低位拓展的哈希运行更快，效率上有大幅的提升。单看I/O次数就可以知道原因了，高位拓展的哈希的I/O次数明显比低位拓展的少。这是因为，在lineitem.tbl中，记录是按照orderkey是升序存放的，相邻之间记录的orderkey也相差不大。因此，相邻记录的orderkey的高位几乎都是一样的，从高位进行拓展（虽然实验中我采用了特殊的方式定住最高位，然后将剩余的低位全部反转，再进行拓展。）大大提高了申请页面时的命中率，因为该页面很可能刚被使用过，刚好存在缓冲池中，不需要请求I/O。反观从低位拓展的哈希，由于相邻的数据地位极大的概率存在不同，因此申请页面是的命中率比较低，这增加了I/O的次数，从而降低了程序运行的效率。虽然高位拓展哈希得到的桶文件比较大，但是这种牺牲硬盘空间换取运行效率的做法在创建索引的时候是值得的。

测试查询的时候，我写了一个产生1~6,000,000之间的整数的程序，生成了2000个随机数用于查询。为了比较查询时不同版本程序的性能，根据实验结果可画出如下表格（见表二）。

|  |  |  |  |
| --- | --- | --- | --- |
| 版本 | 查询的总时间 | 每秒执行查询的数量 | I/O占总用时的比例 |
| least\_8 | 1.186 s | 1686 | 0.53% |
| least\_128 | 0.999 s | 2002.00 | 0.49% |
| most\_8 | 8.502 s | 235 | 0.88% |
| most\_128 | 3.416 s | 585 | 0.75% |

表二

从表二中可以看出，采用哈希函数相同时，使用不同缓冲页面的程序版本的查询效率相差不大。对比使用不用的哈希函数，低位拓展哈希的查询效率要明显地高于高位拓展哈希的。观察表二，可知I/O占各个程序版本查询总用时的比例都不到1%，我猜想主要的原因不是I/O，而是哈希函数的复杂度。低位拓展的哈希函数十分简单，只需要一两步操作就可以获得，相比之下高位拓展的哈希函数就复杂得多了，因此，在查询的时候，由于低位拓展哈希函数的简单性，低位拓展哈希方法占有明显优势。

1. 心得体会

终于到了可以写心得体会的时候了，要知道，我等这一天都等了两个星期了。

在本次实验中，我学到了不少东西，简单列举如下。

1. 实验中我想定义一个可以在整个项目中使用的变量，刚开始经常遇到编译器提示链接的时候该变量已经在其他OBJ文件中定义，最终我采用了extern声明的方法解决了问题。
2. 本实验中对于I/O效率的要求是很高的，因此如何提高I/O速度成了一个重要的问题。一开始我使用了fread，fwrite函数，后来改成了Windows的API，其实第一次改的时候并没有发现效率上有什么提高，因为那时候同时运行了较多的程序了，因此差别不大。后来我再想优化I/O效率时，上网查了一下发现fread，fwrite在Windows平台上其实也是调用API的，因此又测试了一下，发现效率确实提高了。
3. 实验过程中最麻烦的莫过于debug了，建立索引的部分又得全部代码几乎写好才能真正debug，因此，面对这么多的代码，一开始debug起来效率不高，往往一个小问题要查很久才能找出来。后来debug多了，也积累了经验，调试起来也就更加得心应手了。
4. 在查询的时候，我本来想把文件testinput.in以一页8KB的方式读入，后来发现这会出现问题。因为在Windows下，一般的编译器都会把回车换行符认为是一个字符‘\n’，但是，使用fread读入的时候，程序会认为回车换行符是两个字符‘\r’和‘\n’。因此，即使我后来改了fseek的跳转，下次读入依然没有办法读入原始数据的8KB，而是程序认为的8KB，这当然比原始数据的8KB少。后来，我只能改为fscanf的标准化读入了。

在本次实验中，也是有一些遗憾的。

其一是，优化之后，我最快的运行时间是10.5分钟，最慢的是21.5分钟，这相比一些同学的程序就非常慢了，虽然运行的操作系统不同，但差别的确有点大。

其二是，在低位拓展哈希写完后，我花了3天的时间尝试写高位拓展哈希。期间我尝试了几种方法的，但都是无果而终。我个人认为，高位拓展哈希有一个显著的特点，就是当深度超过orderkey的二进制位数时，桶分裂的时候，原来的数据需要重新拿出来哈希，而此时不像低位拓展哈希，低位拓展哈希的话，记录不管怎么分配，总是落在原来的桶和分裂出来的新桶中，而高位拓展哈希则有可能落在除了这两个桶之后其他的桶中，并且有可能造成其他的桶分裂。这是一个区别于低位哈希的显著特点。

我后来采用了双临时文件的方式处理这种连续桶分裂的问题，但是在索引目录定位的问题上卡住了，总是找不到合适的方法进行正确的定位。最后，我只能采用了特殊的高位拓展哈希方式，定住最高位，将剩余的低位全部反转，接下来的操作就跟低位拓展十分相似了。

虽然采用了“伪高位拓展哈希”的处理方式，实验的结果一开始还是出乎我的意料的。因为这种高位拓展哈希建立索引的速度比低位拓展哈希的快了很多，原因是原始数据lineitem.tbl中相邻记录的orderkey之间相差不大，它们的二进制高位几乎都是相同的，因此这种方法处理的时候页面的命中率会高很多，I/O自然就少了。我想，这个结果也反映了高位拓展哈希的特点之一吧。

这一次的项目我是一个人做的，一个人做有好处也有坏处，好处就是自己可以对整个实验中写出来的全部代码十分了解，坏处就是没有人帮忙写实验报告。不过，实验最终还是完成了，在这个过程中，我想感谢很多跟我一起讨论过的同学，给予我帮助的同学，尤其是许俊伟、连荣忠和区展明（排名不分先后，呵呵）。