DBMS Project

院系： 软件学院

专业： 软件工程

年级： 2010级

组员： 陈侨安（通软）

赵娜（计应）

吴尚君 （通软）

一. 项目题目

实现可扩展哈希。算法实现分为两大部分：

第一部分是建立索引。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。

第二部分是查询。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,查询结果按照 L_PARTKEY 进行排序。

只能使用 P 个页，每个页的大小为 8K bytes，一个哈希桶的大小和一个页的大小相同。比较 P=8，P=128 这两种情况下的哈希方法，包括桶的分裂方式、桶分裂时桶内数据的分配方式、I/O 的次数、目录的大小、桶的数量、查询的速度（每秒执行查询的数量）、I/O 用时占查询总用时的比例等

分别实现从低位和从高位进行扩展的哈希。

二．项目环境和工具

VS2010 + gedit(win 版本)

测试环境：WIN7(32)

CPU:AMD Athlon™ 64 X2 Dual Core Processor 3600+ 1900Mhz

RAM:2.00GB DDR2

硬盘:西数 1.00TB 7200 转/s

三．项目内容

1. 程序说明

读入lineitem.tbl，以L_ORDERKEY属性作为键值将记录放入合适的哈希桶内。

读入测试文件testinput.in内的数据，数据中包含多个需要查询的键值。

将通过键值查询得到的所有记录都输出到testoutput.out文件中。

hashindex.out为所建立的哈希目录，result.fil为已经完成哈希的数据。

本程序默认采取的方式为从低位到高位的哈希，可供选择页面只有8和128。

若想测试本程序的从高位到低位（不足位补0）的方式，需修改Manager::insertData() 函数中的insertLow(str) 为 insertUpOne(str), 并修改Manager::query() 中的 hashFromLow 为 hashFromUpOne, 并且需要修改Buffer::saveOutPage和Buffer::readOutPage的代码为注释部分代码。

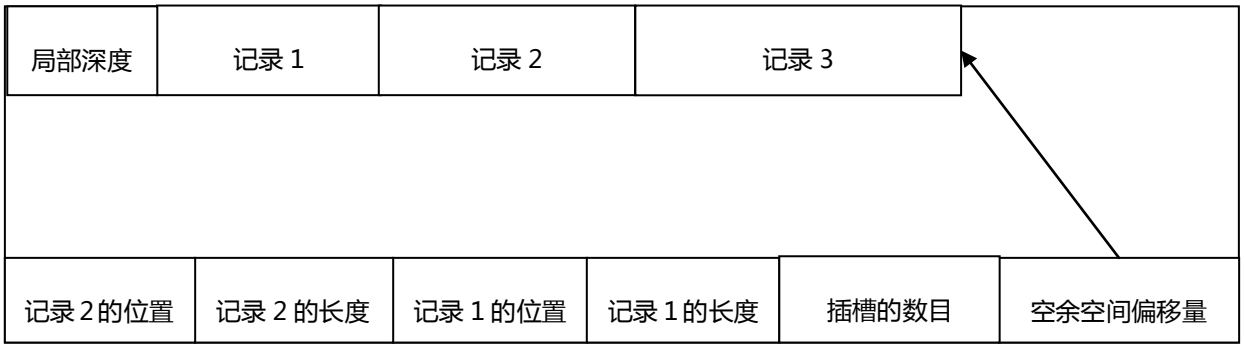
若想测试本程序从高位到低位（默认23位）的哈希方式，按照上一条修改即可，修改为inserUpTwo和hashFromUpTwo,Buffer中不需作上述修改。

2. 数据结构设计

1> Page

- 定义页大小为 8k，记录为变长的方式进行存储，即把记录当成变长字符串存储。
- 页中含有局部深度、记录、记录的长度、记录在页中的偏移量、空余空间便宜量、插槽的数量。以上数据均在页中以字符串的方式存储，需要时将其转换为整形。
- 考虑到对于 8K 的数据范围，可以用页内 4 个位（4 个 char）来储存整形变量，如局部深度，记录长度的数据等。

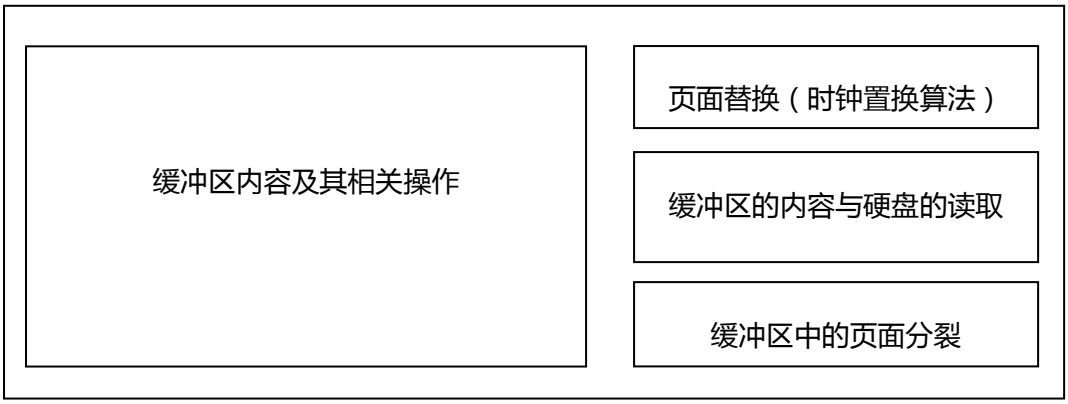
- 页的结构图如下：




```
class Page {
private:
    char content[ 8192 ];//页的内容
    int readInt ( int start, int end );                    //读取页中从start到end的整形数据
    void setInt ( int start, int end, int number );        //设置页中从start到end的位置的整形数据为number
    int getRecordOffset( int index );                      //返回第index条记录偏移量
    void setRecordOffset( int index, int number ); //把第index条记录便宜量设置成number
public:
    bool addRecord (char *rec );                           //往记录中添加数据，成功返回1，失败返回0
    char* getRecord ( int index );                         //返回页中第index个数据
    void adjustPage();                                     //调整页面的空余空间
    void clearPage( int localDepth );                 //清空页面的内容，并把页面局部深度设置为 localDepth
    void setLocalDepth( int number );                 //设置页面的局部深度
    int getLocalDepth();                               //返回页面的局部深度
    void setLocalDepth( int number );                 //设置页面的局部深度
    void setFreeOffset( int number );                 //将页面空余偏移量设置为number
    int freeOffset();                                 //返回页面空余偏移量
    void setSlotNumber( int numebr );                 //将页面插槽数目设置为number个
    int getSlotNumber();                              //返回页面的插槽数目
};
```

2> Buffer:

- 作为内存中存放数据的缓冲区，对于放在哈希桶中的数据放在这里进行处理。
- 页面的存取，选择替换页面的时钟置换算法以及页面的分裂都将在这里实现。
- 相关功能逻辑概念图如下图所示
-




```
class Buffer {
private:
    Page** pool;                //缓冲区中的页
    int* pageId;                //缓冲区中页面所对应的页面号
    bool* ref_bit;              //引用位，1表示最近曾被使用过
    bool* pin_count;            //锁定位，当1时不能被替换
    FILE *outStream;            //输出文件“result.fil”,用于保存页的数据
    bool* dirty;                //记录修改位，检查记录是否被修改
    int size;                   //缓冲区的大小
    int currentPageId;          //最近一次所读取或存取的页面编号
    int current;                //作为时钟页面算法的下一个起始量
public:
    Buffer(int size);
    int choseByClock();                     //时钟页面算法，返回被替换的页面
    void saveOutPage ( int f_offset );      //保存缓冲区中目前第f_offset个页

    void readOutPage ( int f_offset, int pageId );
                                            //从硬盘读取编号为pageId的页到缓冲区第f_offset个页中

    Page* getPage( int f_offset );          //返回缓冲区中第f_offset个页

    void spiltPageLow ( int oldLocation, int newLocation, int hashKey, int localDepth );
    //从低位到高位的哈希的分裂，缓冲区中第oldLocation个页分裂到第newLocation个页中

    void spiltPageUpOne ( int oldLocation, int newLocationOne, int newLocationTwo, int hashKey,
int localDepth );
//从高位到低位（不足位补0）的哈希的分裂，将缓冲区中第oldLocation个页分裂到newLocationOne和
newLocationTwo中

    void spiltPageUpTwo( int oldLocation, int newLocation, int hashKey, int localDepth );
```

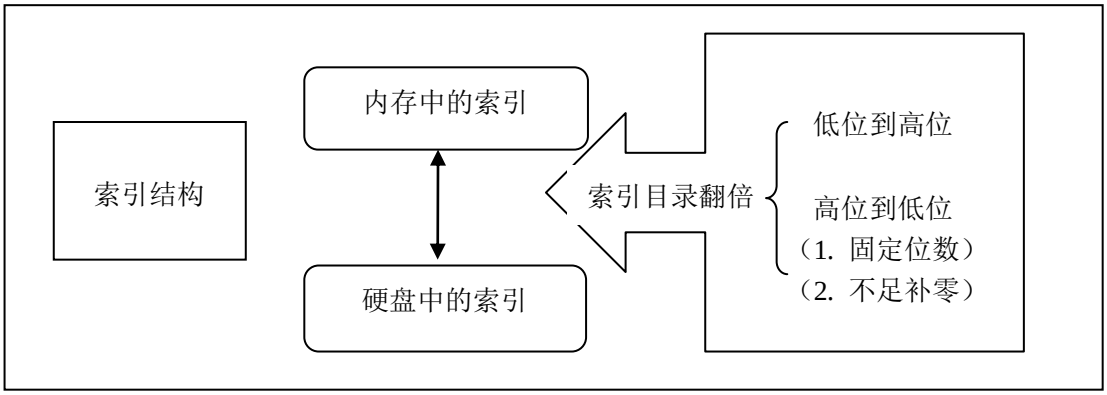
```
    //从高位到低位(固定位数)的哈希的分裂，将缓冲区中第oldLocation个页分裂到newLocation个页中

    bool insertData(char* str,int f_offset);          //将str数据插入到缓冲区中第f_offset个页中
    void setPageId (    int f_offset, int pId );      //设置缓冲区第f_offset个页的页面号为pId
    int getPageId ( int f_offset );                   //获取缓冲区中第f_offset个页
    void setPinCount ( int f_offset );                //锁定缓冲区中第f_offset个页，此时禁止被替换
    void clearPinCount ( int f_offset);               //释放缓冲区中第f_offset个页的锁

    void clearPage ( int f_offset, int localDepth );
    //将缓冲区第f_offset个页清空并设置局部深度为localDepth
};
```

3> Catalog

- 索引的建立与存储，并且其最大能放下的索引大小只能为 8K（2048 个 int）。需要取特定的 hashKey 的页面编号时可能需要在硬盘中载入数据才能读取。
- 不同的哈希方法的索引的翻倍方式不同，需要不同的函数实现。
- 修改 hashKey 所指向页面的 Id 编号的时候 需要在特定时间写数据进硬盘中进行保存。




```
class Catalog {
private:
    int size;                   //目录的总大小
    int globalDepth;            //全局深度
    FILE *indexStream;          //目录文件
    int pageId[ MAXSIZE ];      //目录索引，其大小被限制为8K
    int start;                  //目前目录中的起始位置
```

```
    int dirty;                          //目录修改位，表示目前目录是否被修改过，1为是
public:
    Catalog( int totalSize );
    void doubleCatalogLow();        //对于低位高高位的索引的目录翻倍
    void doubleCatalogUpOne();    //对于高位到低位（不足位补0）索引目录的翻倍
    void doubleCatalogUpTwo();    //对于高位到低位（固定位数）索引目录的翻倍
    void saveCatalog();                 //保存目前的索引目录
    int getPageId ( int hashKey );    //获取hashKey所指向的页面的编号
    void setPageId ( int hashKey, int number );
                                        //将键值为hashKey的所指向页的页面编号设置为number
    int getGlobalDepth ();               //返回目录的全局变量
};
```

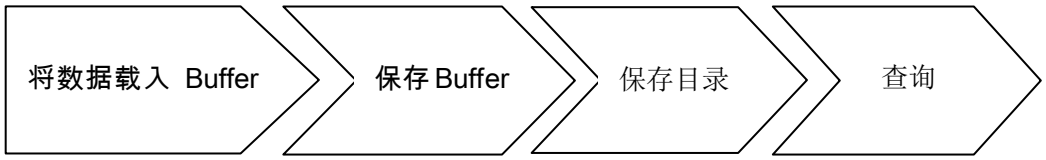
4> Func

- 分别实现从低位到高位和从高位到低位扩展的哈希及相关操作

```
int hashFromLow ( int number, int localDepth );        //从低位展开的hash
int hashFromUpOne ( int number, int localDepth );    //从高位展开的hash（不足补零）
int hashFromUpTwo ( int number , int bucketSize );    //从高位展开的hash（固定位数）
int getKey ( char*str );                                    //获取str的键值
int totalSize ( int globalDepth );                          //获取globalDepth深度下的数据规模
int shiftLeft ( int number, int i );                        //对number左移i位并返回结果
int getPartKey ( char* str );                               //获取str的L_PARTKEY
```

5> Manager

- Manager 对 buffer、catalog、func 进行调用以实现 hash 建立索引以及查询的功能
- Manager 工作图




```
class Manager{
private:
    FILE *inStream;          //输入文件路径
    FILE *testInStream;      //测试数据输入路径
    int currentPageId;       //上一次寻找到的页面Id
    Buffer *buffer;          //缓冲区
    Catalog *catalog;        //索引
    int pidCount;            //下一个新页的编号
```

```
public:
    Manager();
    void insertData();          //从testInStream中插入数据
    void saveBuffer();          //保存当前缓冲区到硬盘文件中
    void insertLow( char*str );         //对str进行低位到高位的哈希并插入到相应的桶中
    void insertUpOne ( char*str );      //对str进行高到低（不足位补0）的哈希并插入到相应桶中
    void insertUpTwo ( char*str );      //对str进行高到低（默认23位）的哈希并插入到相应桶中
    void saveIndex();                   //保存当前目录
    void query();                       //查询结果
};
```

3. 实验过程设计

- **从低位到高位进行扩展的哈希**

1．哈希函数

经过分析可知从低位到高位取localDepth的哈希与对一个数对其取2^localDepth的余结果是相同的。

```
int hashFromLow ( int number, int localDepth ){
    int value = 2^localDepth;
    返回 number 对 value的余数;
}
```

2．桶的分裂

当页面分裂的时候，分裂桶的哈希值肯定是原页的哈希桶加上2^本页深度，可以根据这个规则把数据分离开来。

```
循环{
    取桶的下一条记录
    if (下一条记录不存在) break;
    int hashKey = 对该记录进行从低到高的哈希
    if ( hashKey == 本页的哈希键 ) 该记录留在此桶中
    else{
        将该记录插入到分裂的桶中
        设置该条记录在原有桶的记录长度属性为-1
```

```
        }
    }
    循环{
        在本桶找出一条在该记录前有长度为-1的记录
        if(本记录不存在) break;
        在桶中将该记录往前移动，填补空位
        将该记录移动到前面记录长度为-1的槽中并更新相应的偏移值和记录长
    度值
    }
    更新本页的槽的数量
    int localDepth = 桶原本深度
    分裂桶深度 = 本桶深度 = localDepth + 1;
```

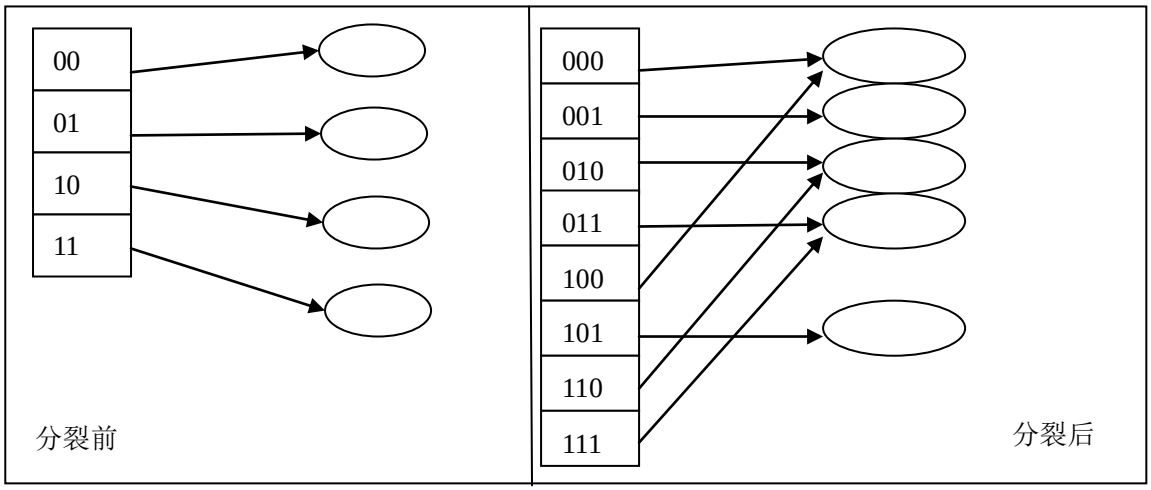
3．目录的维护

当页面分裂的时候，需要对目录进行相应的维护，此时有两种情况，当分裂页局部深度等于全局深度和分裂页局部深度少于全局深度。

(1) 分裂页的局部深度等于全局深度

因为当分裂时，原始页和分裂页的页面的哈希值只有最高位不同，所以可以通过原始页+(2^全局深度)找到新的分裂页。并且目录翻倍后多出的哈希键需要指向原有的哈希键（如下图）。

```
    int size = 目前目录大小
    全局深度++
    当前目录大小翻倍
    int pageId = 分裂页编号
    int location = 分裂页的键值
    for ( int i = 0; i < size; i ++ )
        哈希值为( i + size )的桶所指向的页面编号 = 哈希值为i的桶所指向的页
    面编号
    哈希值为location的桶所指向的页面编号 = pageId
```



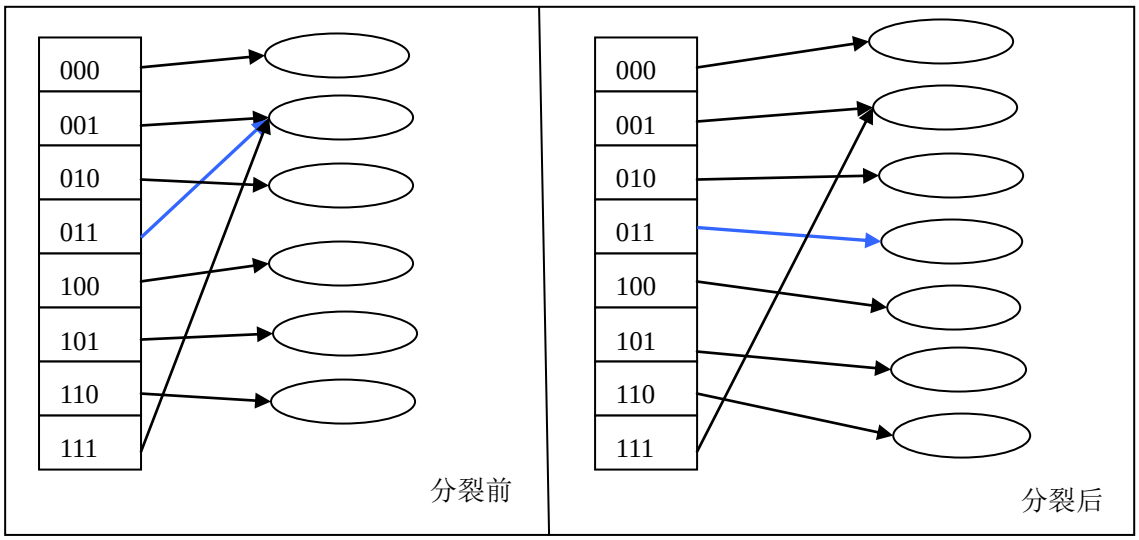
桶01分裂成001与101

(2)分裂页的局部深度少于深度

当分裂时，原始页和分裂页的页面的哈希值只有最高位不同，所以可以通过原始页+(2^全局深度)找到新的分裂页并更新，并且关联的索引项(当前页面键值的下一最高位为1的键值)也需更新，当前页面深度和分裂页面深度在更新完目录后通过页面分裂的函数进行更新。

```
 int pageIdNew =  分裂页编号
 int pageIdOld =  原本页编号
int hashKey = hashFromLow(  当前页面对应哈希键值 ,   当前页面深度  );
int depth =当前页面深度
int nowSize = 2 ^ depth
int totalSize = 2 ^  全局深度
哈希键位hashKey对应的页面编号  = pageIdOld
哈希键位hashKey + nowSize对应的页面编号  = pageIdNow
循环{
    int offset = 2 ^ ( depth + 1 )
    depth自增
    if ( hashKey + offset>=totalSize ) break;
    else  哈希键位hashKey + offset对应的页面编号  = pageIdNow
    if ( hashKey + offset+nowSize>=totalSize ) break;
    else  哈希键位hashKey + offset+nowSize对应的页面编号  = pageIdNow
}
```

全局深度++



桶001（实际值01）分裂成001与101

- **从高位到低位进行扩展的哈希（高位不足补0）**
- **哈希函数**

```
int hashFromUpOne ( int key, int localDepth ){
    if ( key的2进制位数  < localDepth ) return key;
    else  取高localDepth位并返回该值
}
```

- **桶的分裂**

当桶分裂时，按照分裂规制,哈希值为hashKey的桶必然分裂到hashKey * 2和hashKey * 2 – 1的桶中。并且由于数据的特殊性，哈希值为key的数据必然只能插入到哈希值为key所指向的桶中，并且在该桶分裂的时候留在本桶中。

```
int oldPageId =  本桶hashKey所指向页面的编号
int newPageIdOne =  分裂桶hashKey * 2所指向的页面编号
int newPageIdTwo =  分裂桶hashKey * 2  –  1所指向的页面的编号
循环{
    取本桶的下一条记录
    if (下一条记录不存在) break;
    int hashKey =  对该记录键值进行从高到低的哈希
    if ( hashKey ==  本桶的哈希值  )  该记录留在此桶中
```

```
        else if ( hashKey == 本桶哈希值 * 2 ){
            将该记录插入newPageIdOne中
            设置该条记录在原有桶的记录长度属性为-1
        }
        else ( hashKey == 本桶哈希值 * 2 + 1){
        将该记录插入newPageIdTwo中
        设置该条记录在原有桶的记录长度属性为-1
        }
    }
    循环{
            在本桶找出一条在该记录前有长度为-1的记录
            if(本记录不存在) break;
            在桶中将该记录往前移动，填补空位
            将该记录移动到前面记录长度为-1的槽中并更新相应的偏移值和记录长度值
    }
    更新本页的槽的数量
    int localDepth = 桶原本深度
    分裂桶newPageIdTwo = 分裂桶newPageIdOne深度 = 本桶深度 = localDepth + 1;
```

- **目录的维护**

当页面分裂的时候，需要对目录进行相应的维护，此时有两种情况，当分裂页局部深度等于全局深度和分裂页局部深度少于全局深度。

(1) 分裂页的局部深度等于全局深度

此时目录需要进行翻倍，由于在此哈希规制下，hashKey * 2 和hashKey* 2 + 1的桶必为hashKey的桶分裂出来。

```
    int size = 目前目录大小
    全局深度++
    当前目录大小翻倍
    int pageIdOne = 分裂桶1的页面编号, pageIdTwo = 分裂桶2的页面编号
    int pageIdOld = 原本桶的页面编号
```

int hashKeyOne = 分裂桶1的键值, hashKeyTwo = 分裂桶2的键值

int hashKeyOld = 原本桶的键值

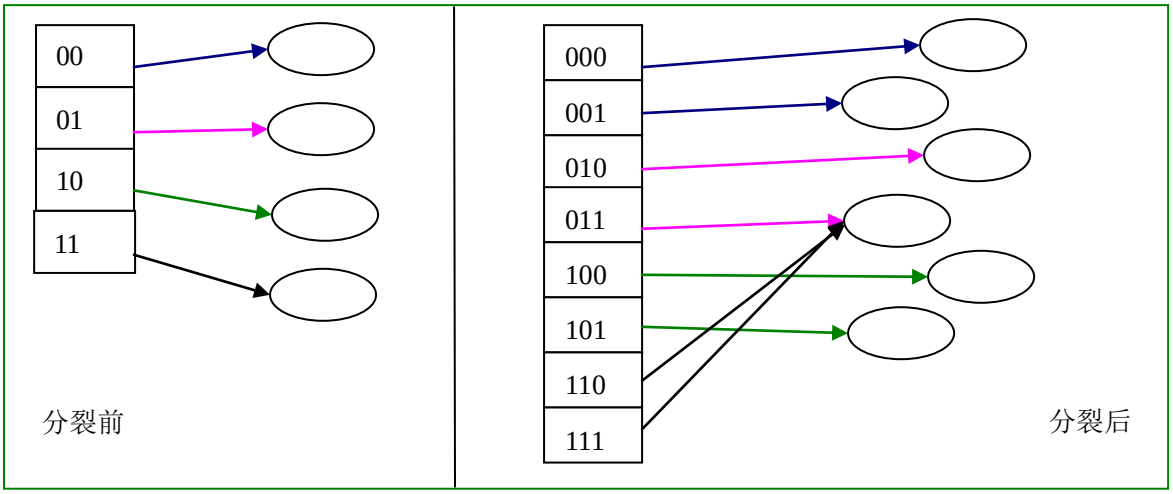
for (int i = 0; i < size; i ++)

哈希值为(i + size)所指向的页面编号 = 哈希值为i所指向的页面编号

哈希值为hashKeyOne的桶所指向的页面编号 = pageIdOne

哈希值为hashKeyTwo的桶所指向的页面的编号 = pageIdTwo

哈希值为hashKeyOld的桶所指向的页面编号 = pageIdOld



桶10分裂成100与101

(2) 分裂页的局部深度少于深度

这里所指的桶的键值是指最早指向该桶的哈希键值，也就是hashFromUpOne(key,localDepth)，其中key为数据键值，localDepth为当前桶的全局深度。

int size = 目前目录大小

int pageIdOne = 分裂桶1的页面编号, pageIdTwo = 分裂桶2的页面编号

int pageIdOld = 原本桶的页面编号

int hashKeyOne = 分裂桶1的最早出现键值

int hashKeyTwo = 分裂桶2的最早出现键值

int hashKeyOld = 原本桶的最早出现键值键值

循环{

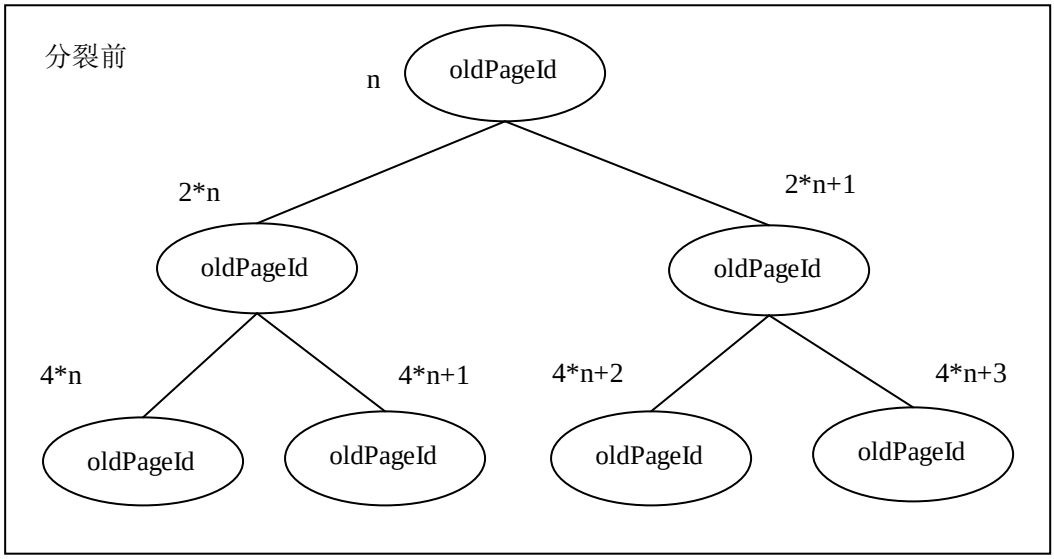
if (hashKeyOld仍有左子树节点没赋值){

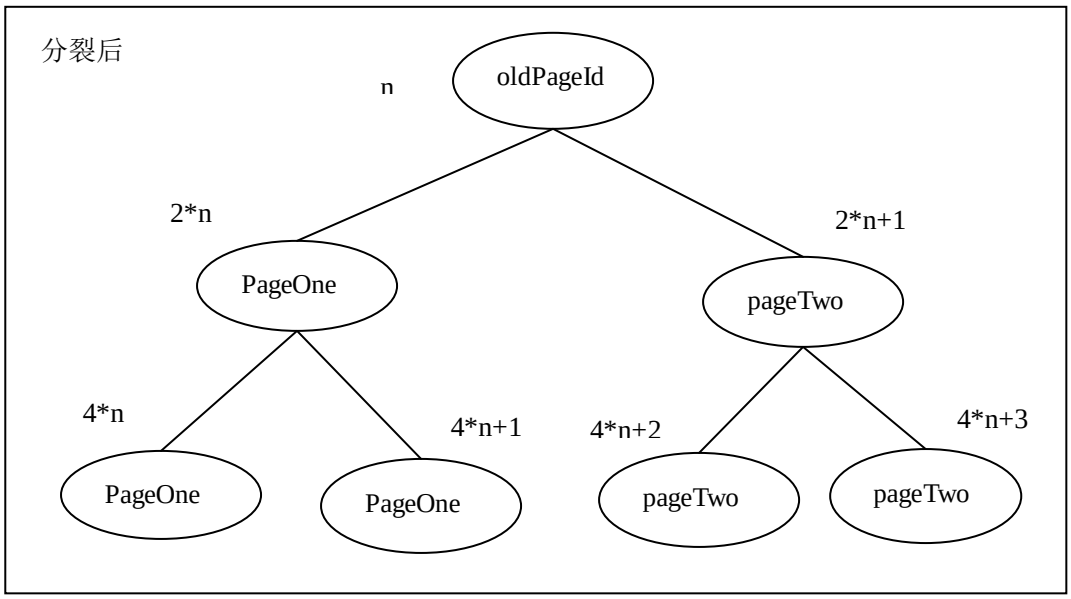
该节点指向的页面编号 = pageIdOne

}

```
                else break;
                if ( hashKeyOld仍有右子树节点没赋值 ){
                    该节点所指向的页面编号 = pageIdTwo
                }
                else break;
            }
```

具体分裂如下图。





- **从高位进行拓展的哈希（默认位数为23位）**

本哈希默认一个数有23位，在自行定义桶的分裂方式的规则下能进行对应的哈希。

- **哈希函数**

本哈希是默认一个数必定有23位，对其哈希的结果取最高的深度的位数(如13，当局部深度为10的时候，其哈希值为0)，在按照下列定义的桶的分裂和目录的维护下能进行相应的哈希。

```
int hashFromUpTwo ( int key, int localDepth ){
    int stack[ 23 ] = {0};          //默认为23位
    int temp = key;
    int stackCounter = 0;
    int result = 0;
    循环( temp 不 为 0 ){
        stack[ stackCounter ] = temp 余 2
        stackCounter ++;
    }
    循环( 取的位数不足localDepth位 ){
        result左移1位
        result = result + 要取的stack的下一位
    }
    return result;
}
```

- **桶的分裂**

对于一个哈希值为hashKey的桶，由于本哈希函数的特殊规则当其分裂时，桶内所有数据必然会分配到hashKey * 2和hashKey * 2 + 1所对应的桶中，因此可以把原有桶的部分数据移到hashKey*2的桶中成为分裂桶1（通过设置对应的页面编号实现），余下的数据移动hashKey*2+1的桶中成为分裂桶2。

```
int oldPageId = 本桶hashKey所指向页面的编号
int newPageId = 新的分裂桶所指向的页面编号
循环{
    取本桶的下一条记录
    if (下一条记录不存在) break;
    int hashKey = 对该记录键值进行从高到低的哈希（局部深度+1）
    if ( hashKey == 本桶的哈希值 * 2 ) 该记录留在此桶（分裂桶1）中
     else if ( hashKey == 本桶哈希值 * 2 + 1){
          将该记录插入到分裂桶2中
         设置该条记录在原有桶的记录长度属性为-1
    }
}
循环{
    在本桶找出一条在该记录前有长度为-1的记录
    if(本记录不存在) break;
    在桶中将该记录往前移动，填补空位
    将该记录移动到前面记录长度为-1的槽中并更新相应的偏移值和记录长度值
}
更新本页的槽的数量
int localDepth = 桶原本深度
分裂桶1所指向的页面编号 = oldPageId
分裂桶2所指向的页面编号 = newPageId
分裂桶1和分裂桶2的局部深度加1
```

- **目录的维护**

鉴于本哈希的特殊性，本目录的维护不同于正常的目录的维护，需要自行定义规则。

（1） 分裂桶的深度等于局部深度

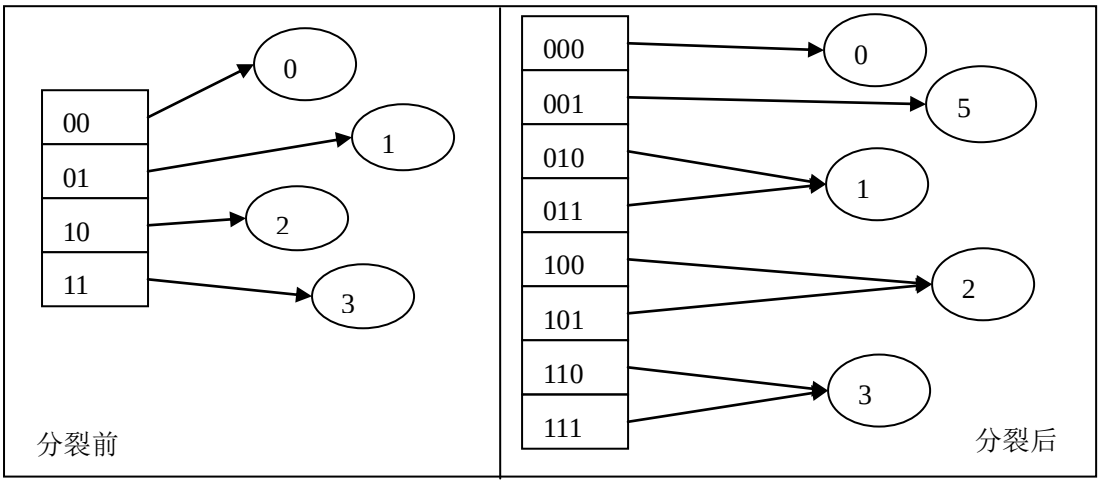
在目录翻倍时，由于哈希函数的特殊性，分裂桶的数据必然会放到hashKey*2和hashKey*2+1这两个桶当中，所以对于其他没翻倍的hashKey索引所指向的桶的编号必与hashKey／2所指向桶的索引编号相同。

```
int pageIdOne =  分裂桶1（原有桶）所对应页面编号
int pageIdTwo =  分裂桶2(新桶)所对应页面编号
int hashKeyOne =  分裂桶1原先对应的哈希键(hashKey)
int hashKeyTwo =  分裂桶2原先对应的哈希键(hashKey * 2 + 1)
 for ( int i =  翻倍前目录大小  * 2  –  1; i >= 0; i -- ) {
        索引i所指向的页面编号  =  索引i/2所指向的页面编号
 }
 索引hashKey * 2 + 1所指向页面编号  = pageIdTwo
 目录大小翻倍
 全局深度  ++ ;
```

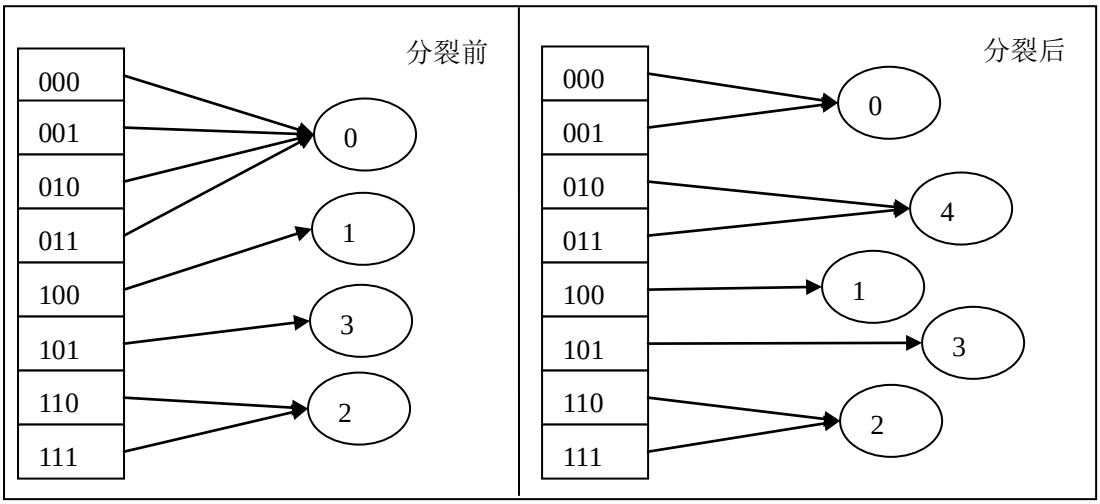
（2） 分裂桶的深度少于局部深度

下列所说的hashKey为指向该桶最小的hashKey,可通过key余2^localDepth余数获得。因为目录翻倍的特殊性，邻近的哈希值必定相同，此时可修改邻近的哈希值的索引。并且分裂后，指向新分裂桶的索引和指向原桶的索引的数目相同。注意此时局部深度没更新，更新在页面分裂过程中进行。

```
int pageIdOne =  分裂桶1（原有桶）所对应页面编号
int pageIdTwo =  分裂桶2(新桶)所对应页面编号
int start = hashKey左移（全局深度-当前局部深度）
                                    //start为分裂后最小指向分裂桶的键值
int size = 2^(全局深度-当前局部深度)
for ( int i = 0; i < size; i ++ ){
      索引start + i所指向的页面编号  = pageIdOne
      索引start + size + i所指向的页面编号  = pageIdTwo
}
```



全局深度等于局部深度时的分裂（00分裂成000与001）



全局深度少于局部深度时的分裂

- **时钟页面算法**

```
循环{
    若当前页的pin_count与ref_bit均为0 break;
    否则  如果当前位pin_count==0  将其ref_bit设为0;
    将当前位设为下一位（循环数组）
}
返回当前位并且把当前位设为下一位
```

- **记录的插入**

将一条记录插入到相应的桶中，插入操作在内存中实现。

```
if（ 记录插入的桶对应页面在缓冲区中 ）{
    把记录插入到缓冲区对应帧中
    修改对应的dirty位表示该帧被修改
}
else{
    int frameNum =  时钟算法选出一个被替换的页面
    if（ 找到该页面已被修改 ）{
        保存到硬盘对应位置中
        设该页dirty为0
    }
    从硬盘中读取对应页读到frameNum帧中
    将记录插入到frameNum帧中
```

```
    if ( 该页空间不足插入不成功 ) {
        该页分裂
        重新调用本函数重新插入此条数据
    }
    修改本帧dirty位为1
}
```

- **目录索引的读取与保存**

本程序在内存的P（8、128）个页中选择1个页当作目录索引在内存中的缓冲区，当需要某一索引所指向的页面的编号时，先检查该索引是否在内存中能找到，若不能再查找硬盘并且读取。注意在程序结束时需要判断目录是否被修改过，如被修改过则需保存。

```
if (读取数据在内存目录索引中) 查找到相应值并返回
else{
    if (在内存中的目录已被修改) 把当前目录索引写到硬盘相应位置中
    在硬盘读取含有该索引的一个页到内存目录索引中
    找到相应的值并返回
}
```

- **缓冲区的页面的读取与保存**

缓冲区中有P-1个帧用于页面的在内存中的暂时存放，并且需要一个pageId来记录本页所对应的编号，每个页固定为8K。当读取或保存一个页时，在硬盘的第pageId个页读取或保存8K的数据即可。

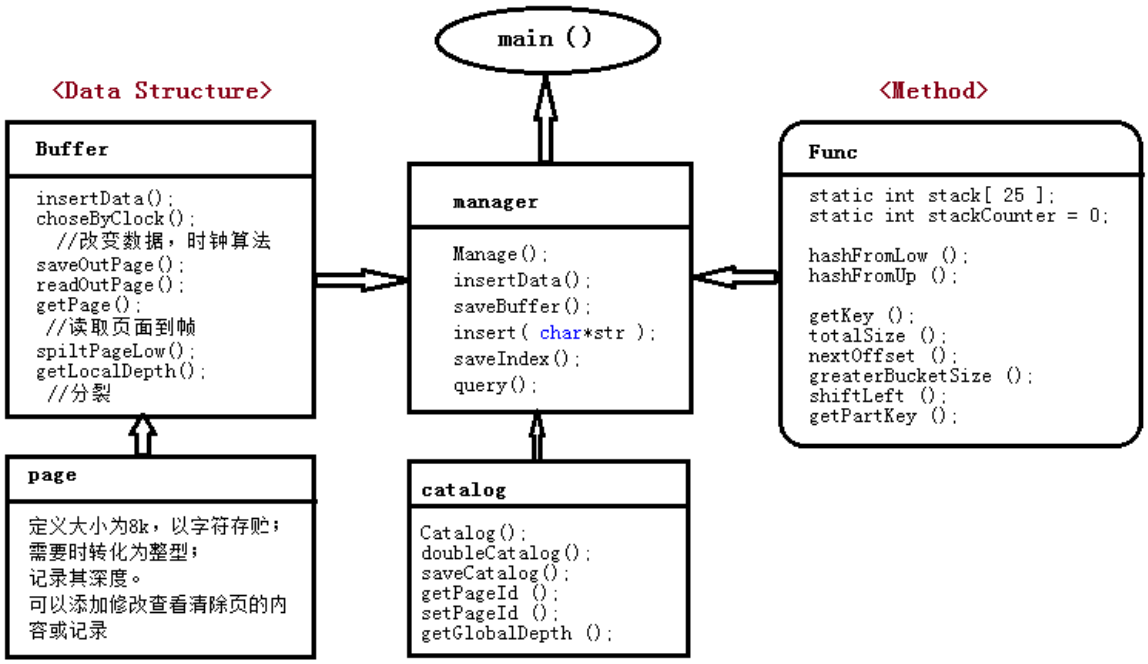
- **页面的分裂**

本程序将页面的分裂放在内存中运行。分裂的新桶在分裂完成后保存在缓冲区中，根据哈希函数的不同页面所需要缓冲区的页面数目也不同。在申请页面的时候，可以对缓冲区的某一个帧设置一个pin_count的锁，表示此页不能被替换来排除对页面的重复申请的情况，当完成分裂后释放锁并且设置相应帧dirty为真表示这些页面都被修改过。

从低位到高位哈希：2个帧

从高位到低位的哈希：（1）不足位补0:3个帧　　（2）默认23位：2个帧

4. 代码结构



5. 实验结果及分析

1> 结果

由于哈希桶的读取与写入次数远比索引的读取写入次数多，所以这里的I/O次数是读哈希桶的读取与写入数据的次数。高位到低位（不足位补0）的方法生成文件过大最后导致了程序的崩溃，因此没有记录，/O用时比例采用VS的profile得出,查询表为附录里的testinput.in（10000条随机生成的测试点）。

	I/O 的次数	目录的大小	桶的数量	程序运行时间	I/O 用时比例	哈希后的文件大小	查询速度(条/s)
P=8 从低位	3,143,026	4096KB	133,032	523.644s	82.62%	1,064,256KB	6857.933
P=128 从低位	3,129,746	4096KB	133,112	427.963s	81.34%	1,064,896KB	5087.116
P=8 从高位(1)	518,283	8192KB	368,496	588.480s	34.8%	2,947,968KB	561.814
P=128 从高位(1)	504,662	8192KB	368,496	657.949s	32.8%	2,947,968KB	622.442
P=8 从高位(2)	195,837	1024KB	153,274	116.241s	25.92%	1,226,192KB	1666.525
P=128 从高位(2)	156,392	1024KB	153,307	113.674s	22.39%	1,226,456KB	2077.2354

由于 lineitem.tbl 这个表导入生成数据过大，对高位（1）测试 orders.tbl 表

	I/O 的次数	目录的大小	桶的数量	程序运行时间	I/O 用时比例	哈希后的文件大小
P=8 从高位(1)	118，705	512KB	59，448	60.362s	36.32%	475,584KB
P=128 从高位(1)	118,573	512KB	59,448	42.889s	36.34%	475,576KB

2> 性能优化

1．本程序在不同机器上运行时间会有很大不同，这里程序运行时间只能作为参考。

2．若一个页面在缓冲区没被修改过，则不需写回硬盘中。

3．本程序的数据运算尽可能使用了位运算符。

4．对于输入的数据，采取 fread 读取一个页的空间的后进行数据处理，而不是对输入数据一个一个进行读取。

5．对于文件指针的定位采用当前位置到目的位置的偏移进行定位而不采用才文件头开始定位。

6．将局部深度放在桶中而不放在索引文件中，使一次能载入的索引数目更多。

7．当页处于分裂中时，对不属于该页的数据不采用把该数据重新清空赋值，而采用将其长度设置成-1，当页分裂完成后，再把页内空余的洞进行压缩。

8．页目录的更新才取尽可能让处于附近的目录在尽可能少的情况下更新。

3> 结果分析

对于 P=8 与 P=128 时不同哈希方法，其速度提升不明显。原因是限制速度的瓶颈在于 I/O 的速度。观察两者的 I/O 次数，发现两者都十分相近，由此可以推理出影响本程序速度在于 I/O 的次数。

对于第一种从高位到低位的哈希的方式，其生成索引数据十分大，而且其 I/O 比从低位到高位方式少很多。究其原因，是因为 lineitem 数据的特殊性，因为 lineitem 是有序的表，按顺序插入导致的结果是一个桶必然不能存放少于其对应哈希键值的数据，因为本组数据中，相同的数据最多只有 7 条，那么就导致了大量的桶只有 1-3 个空页导致生成数据较大。

观察可以发现，第二种从高位到低位的哈希方式是最快的，在相同条件下其拥有最小的 I/O 次数，其 I/O 占用时间比例最少。分析其与数据的插入顺序的特殊性有关，在刚开始时，数据由于哈希出哈希键相同导致密集堆积从而导致桶的不断分裂，当全

局深度升到一定的时候，由于页面已基本完成了所有分裂，所以数据可以不断插入而不导致分裂，并且由于哈希函数的特殊性导致数据在后面的桶的大量堆积，从而出现了目前这种结果。

可以发现 P=8 与 P=128 对于从低位到高位的哈希在速度上有明显的差别，一方面是由于数据规模与 P 相比比较大，P 的作用比较明显。从而得出在一定的哈希函数下，P 越大会使时间有效缩短。但是对于第二和第三种方法是会导致桶的不断分裂，并且由于是有序插入的，数据的有序性导致数据很大程度上会插入到后面的新桶中，而刚好新桶又容易在缓冲区中，从而使限制 I/O 次数的主要原因是页面需要不断分裂，从而导致 P 不同而有明显的差别。

对于两种不同的高位哈希方式，发现在本组测试数据的情况下第二种方式更优。

查询的中，第二种查询速度最慢，其原因是过多的桶导致数据的不密集堆积从而导致对于均匀分布的数据，对每一次查询都需载入一个桶，导致查询速度变慢。并且通过对比我们可以得出，当桶数越少的情况下，查询速度越快，因为数据的大量密集导致数据容易查找。

通过上面的比较我们可以知道，从低位到高位的哈希所生成的结果文件大小会比较小但是速度比较慢。而从高位到低位的哈希（不足位补 0）适合大量键值相同的数据插入，此时能有效利用空间，但是对于正常的数据比较浪费空间。而对于从高位到低位的哈希（默认 23 位），虽然空间的利用率比较低，但是在本组数据的情况下哈希的速度十分快，能有效提高程序速度。

4> 思考与分析

观察数据可以知道，相同键值的记录最多不超过 7 条，并且在实现时候，本程序没有

采用溢出页的处理方式，因为考虑的页的不断分裂与添加溢出页的操作相比，页的不断分裂相对简单。但是添加溢出页对不同的哈希方式有不同的影响，从而导致不同的结果。

鉴于数据的特殊性，可以使用 MRU(最近最多使用)的替换策略，因为本数据是有序的，所以得出的哈希键值也是有序的，通过使用 MRU 能有效地优化 I/O 时间，此为可优化方案之一。

对于页面内的局部变量，记录长度，记录偏移量等，由于本程序为了实现简单并且考虑到一条记录一般长度为 80，与其对比差 1 个数量级，所以采取 4 位 char 字节存取‘0’-‘9’的数据，需要时再转换。如为了进一步压缩空间，可只取 2 个 char 位，采取 128 进制，此时能进一步压缩页面空间。

由于为了代码的美观性，采取 C++的类的实现方式，如需进一步优化性能，可以把其改成 C 的结构体的版本。

四．心得体会

1） 实验过程中碰到的问题及其解决方法

1． 问题：对哈希索引的文件采取了 r+的读取方式，文件在读入的时候不能读取足够的位数。

解决方案：将其改为 wb+的方式，因为 r+的方式不是为二进制文件操作的形式，由于在保存的时候采取了二进制保存，导致读取结果不正确。

2． 问题：页面在分裂的时候虽然有数据分离出来到新的分裂页中，但是在原页中仍有该条记录存在，虽然该条记录长度为-1。

解决方案： 检查后发现，对从 start 到 end 位置的文件清空采取了 memset, 此时对 end – start 位清空，但是实际是 end-start+1 位。

3． 问题：页面分裂后目录维护出错。

解决方案：通过 cout 对其每条记录添加相应输出 debug.

4． 问题：对于超过 2G 的文件无法通过 fseek 寻址，出现错误。

解决方案：通过__fseeki64 寻址，偏移量采取__int64 的位数。（在 linux 下为 longlong）.

5．问题：从高位到低位（不足位补 0）出现爆栈的错误。

解决方案：原因是因为对于插入的时候采用了递归调用，对于不能插入的数据通过重复调用插入函数。由于此哈希的特殊性，导致一条记录不断插入，桶不断分裂导致爆栈的错误。通过将其修改为循环后即可。后来修改程序后发现是程序写错了而不是递归错了。

6．问题：从高位到低位（不足位补 0）在 2G 内的文件的结果正确，对于超过 2G 后结果文件生成迅速疯长到 6G 后崩溃。

解决方案：经过 3 小时的调试后发现，是由于对两个 int*int 的结果是不能直接赋值到 int64 中的，必需将其中一个 int 强制转换为 int64 位。

2） 实验心得

其实本 project 相当有趣，在写本程序的过程中不断出现错误，通过不断重复的调试，通过采取 cout 输出数据和断点调试结合的方式改正了错误，并且在调试过程中提高了自己写代码的严谨程度。很多地方出现错误的地方或许是因为写代码写得太快导致的小错误，也有思考得不够周全的小错误。这次的实验能进一步提高自己的代码能力。

其实对于本程度的速度非常不满意，其优化方案如上面分析所示。而且本程序

的框架写得比较不美观，很多地方都没有使用异常处理，导致调试困难。这也说明了我们下一步所需要的学习就是程序的异常处理，如何写出健壮和美观的程序对我们以后的发展至关重要。

刚开始还以为这个 project 比较简单，实现起来很简单，实现后却发现有1000+的代码量。并且实现这个 project 用了 2 个星期的时间，从侧面证明了我们的代码能力有待提高。

在 debug 过程中，我们发现了很多 silly mistake,如__int64 c = a * b（a、b 为 int）的结果是不正确的。尽管每一个小错误都可能需要调试 2-3 小时，但是感觉我们收获了很多，这些正是这些我们平时比较少用的东西。在相当大的数据规模下，一些我们平时经常犯的错误马上显现出来。

我们小组虽然有组员逃课通宵熬夜来完成这个 project,但是收获的东西是很多的，而这些正是我们平时上课所不能获得的东西。在这次 project 中，我们小组学会了团体合作、学会了如何合并代码、学会了想让其他人看代码更整洁、美观。

虽然在做这次 project 中有很多不愉快的地方，但是总体上来说还是得多于失。如果有时间和精力的话，我们或许考虑再跟冯老师做 project。谢谢冯老师给了我们第一个晚上通宵写代码的机会，谢谢冯老师在外部施加了点推力迫使我们向前进。