中山大學

SUN YAT-SEN UNIVERSITY

数据库项目

technical report

院系： 软件学院

专业： 软件工程

年级： 2013 级

组员： 王文茏 13331256

项目介绍

主要涉及可扩展哈希在数据库中的应用。

读入由 tpc-h 生成的 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。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。

项目环境

系统： Windows 8.1 专业版 64 位
处理器：Intel® Core(TM) i3 CPU M 350 @ 2.27GHz 2.27 GHz
内存： 2 GB 金士顿 DDR3 1333MHZ
硬盘： 希捷 ST9320 320GB 7200 转/分

语言： C++
编辑器：Visual Studio 2013

项目架构

本项目共有 11 个文件：

ExtendibleHashingDB

|--main.cpp	主文件，程序的入口
|--Manager.h (.cpp)	主控程序，实现功能的主体
|--Buffer.h (.cpp)	缓存池，管理内存中各缓存页
|--Index.h (.cpp)	目录页，存储哈希值和桶号的对应关系
|--Page.h (.cpp)	页的基本类，实现一个页的基本功能
|--Function.h (.cpp)	项目中要用到的通用函数，如取哈希值，取键值等
|--option.h	程序的如最大使用页数，哈希方式等参数的定义文件

无论最大使用页数和哈希方式，项目运行时目录页占用 1 页，读写文件缓冲用占用一页，其他页都用来存储记录桶。

主要数据结构

基本页：

```
/*用来记录每条记录的长度和偏移值*/
typedef struct Rec_info {
  int16_t length;
  uint16_t offset;
} rec_info;

/* 每个桶用8188Byte存储记录，4Byte存储桶的记录数，
 * 局部深度和可用空间偏移
 */
typedef struct Tong {
  char contents[CONCAP];
  uint8_t num;
  uint8_t deepth;
  uint16_t offset;
} tongStruct;
```

每个 Page 页在内存中的分布如下：

<table>
<tr><td colspan="6">Info0</td><td></td></tr>
<tr><td colspan="7">Info1</td></tr>
<tr><td colspan="7">.....
.....
.....
.....</td></tr>
<tr><td>length1</td><td>offset1</td><td>length0</td><td>offset0</td><td>num</td><td>deepth</td><td>offset</td></tr>
</table>

其中蓝色背景的为 contents 的内容。记录的具体内容从 contents 的头向尾扩展，对应的记录信息如长度，偏移值等从 contents 的尾向头扩展。需要写入或写出时将 Tong 强制转换为 char*类型，按长度为 8192 进行存储或读取。

缓冲池：

```
class Buffer {
private:
  Page** pool;
  int* pageId;

  bool* ref_bit;
  bool* pin_count;
  bool* dirty;

  int size;
```

每个缓冲池有一个Page*的数组，用来存储各Page页的指针。pageId数组用来存储每个Page对应的pageId。ref_bit，pin_count数组分别存储各page的时钟标志和pin标记（用于时钟算法），dirty数组标志各page的修改状态，以确定换出时是否需要写回硬盘。

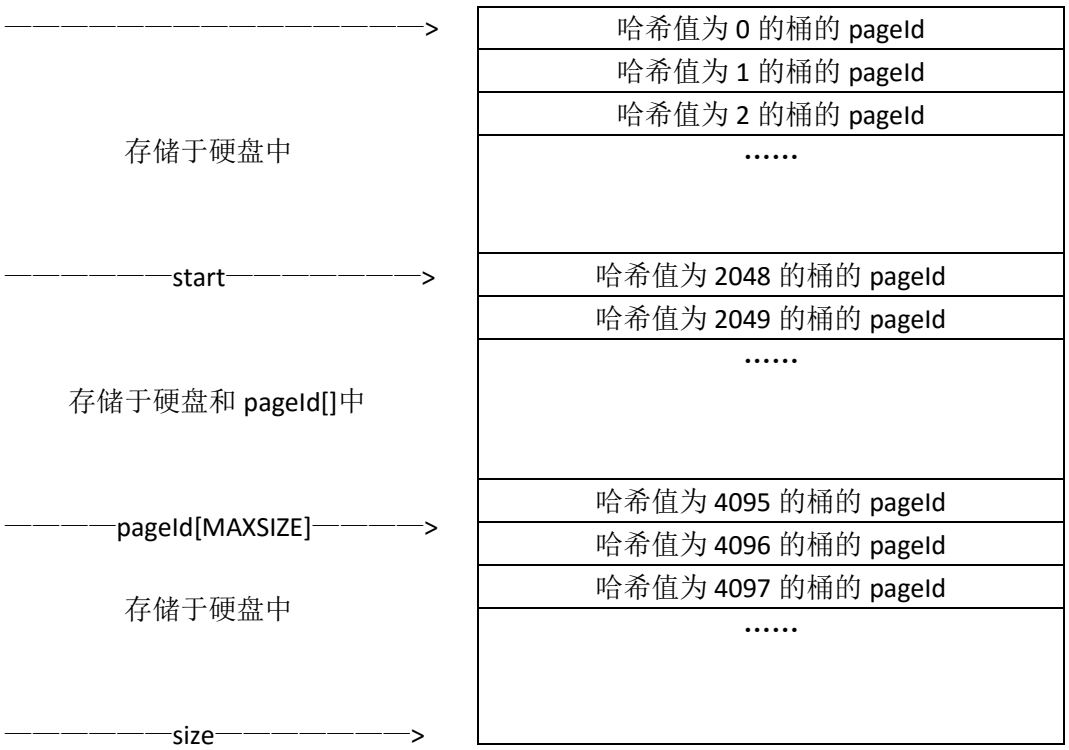
目录页：

```
class Index {

private:
  int size;
  int globalDepth;
  int start;
  int pageId[MAXSIZE];
  bool dirty;
```

size 表示目录页的总大小，包括在硬盘的部分。globalDepth 表示全局深度，start 表示目录当前在内存部分的头下标，pageId 表示以 start + (数组下标)为哈希值的桶的 pageId。

目录总的结构为：



主要算法

低位哈希算法：

哈希方法

直接将 key 值截取低 globalDepth 位。比如 13 二进制为 1101，则当全局深度为 1 时返回 1， 全局深度为 2 时返回 1，全局深度为 3 时返回 5， 全局深度大于等于 4 时返回值都为 13。

程序中使用的方法是 key 值对 1 左移 globalDepth 位后的值取余：

$$key \% (1 << depth)$$

桶分裂算法

假设原来桶中的记录哈希值为 hashKey，则分裂后桶中的记录的哈希值可能为 hashKey 或者 1 << localDepth + hashKey，即原来的 hashKey 最高位前补 0 或 1。所以桶分裂后桶中的记录会根据记录的 key 值的哈希判断是留在旧桶还是移动到哈希值为 1 << localDepth + hashKey 的桶中。另外由于采取的是变长记录，删除掉一条记录后需要对之后的所有记录进行移动，所以旧桶对移出的记录不立即删除，而是将其长度置为-1，等桶分裂完成后再一次性将旧桶中所有长度为-1 的记录删除。

void Buffer::spiltPage

当桶中存在下一条记录时循环：

计算该条记录的哈希值并与当前桶的哈希值进行比较

两者相同则进行下一次循环

两者不同则将该记录插入新桶中，并将该记录在当前桶的长度置为-1

对旧桶进行更新操作

void Page::adjustPage

当桶中存在下一条记录时取该条记录进行循环

得到该条记录的长度，判断其是否为-1

是，则继续循环

否，则取在其前面的距离最远一条长度为-1 的记录，覆盖为该条记录

更新桶的记录数和可用空间偏移

目录维护

桶分裂时需要对目录进行相应的维护，这里分两种情况：

一、 localDepth < globalDepth

这时需要将目录中新的哈希值所对应的桶号改为新桶桶号，同时如果需要的话，还要把另外一些关联的哈希值的对应桶号也改为新桶桶号。比如当前全局深度为 7，一个局部深度为 5，哈希值为 00110 的桶要进行分裂，这时不仅要将哈希值 01 00110 对应的桶更改为新桶桶号，还要将哈希值 11 00110 对应桶号做相同更改。

void Manager::insert

Value = 0

循环

Offset = value << (localDepth + 1)

Hash = oldHash + (1 << localDepth) + offset

当得到的哈希值超出允许位数时退出循环

设置 hash 对应的桶号为新桶号

Value++

二、 localDepth == globalDepth

这时仅需要将目录翻倍后将哈希值为 oldHash + 1 << globalDepth 对应的桶号更改为新桶的桶号即可。

高位哈希算法：

哈希方法

从第 23 位开始从高到低将 key 值截取 globalDepth 位。比如 2816 二进制为 0000 0000 0000 1011 0000 0000，则当全局深度小于等于 12 时都返回 0， 全局深度为 13 时返回 1，全局深度为 14 时返回 4， 全局深度为 15 时返回值为 5 等等。

程序中使用的方法是 key 值右移 23 - globalDepth 位后与 $007fffff_{(16)}$取与的结果：

$$(\text{key} \gg (23 - \text{globalDepth})\ ) \ \&\ 007fffff$$

桶分裂算法

假设原来桶中的记录哈希值为 hashKey，则分裂后桶中的记录的哈希值可能为 hashKey << 1 或者(hashKey << 1) + 1，即原来的 hashKey 的两倍或者两倍加一。这里不会将旧桶中的记录分到两个新桶，而是将旧桶对应的哈希值翻倍，之后再像低位扩展一样，根据旧桶中的记录的哈希值进行判断，哈希值恰好是原哈希值两倍的记录留在旧桶，其他移往新桶，并在全部记录都移动好后再对旧桶进行整理。

目录维护

桶分裂时需要对目录进行相应的维护，这里也分两种情况：

一、 localDepth < globalDepth

这里跟低位扩展一样，不仅需要将新桶对应的哈希值更新，还需要对关联的哈希桶的哈希值进行更新。与低位扩展不同的是，高位扩展是在最低位后补 0 或 1。比如当前全局深度为 7，一个局部深度为 5，哈希值为 00110 的桶要进行分裂，这时不仅要将哈希值 00110 10 对应的桶更改为新桶桶号，还要将哈希值 00110 11 对应桶号做相同更改。

```
void Manager::insert
    start = localHash << (globalDepth - localDepth)
    size = 1 << (globalDepth - localDepth - 1)
    for (int j = size; j < (size << 1);   j++)
        设置 start + j 对应的桶号为新桶号
```

二、 localDepth == globalDepth

这时仅需要将目录翻倍后将哈希值为 (oldHash << 1) + 1 对应的桶号更改为新桶的桶号即可。

时钟算法：

每个桶都有一个 pin 标志位和 ref 标志位。Pin 表示该页不可被换出，ref 表示当前页的时钟标志。

int Buffer::choseByClock

从 **currentPage** 开始循环取 **buffer** 中的下一页（首尾循环）

若当前页的 **pin** 与 **ref** 均为 **false** 则退出循环

否则如果当前页 **pin** 为 **false**，将其 **ref** 设为 **false**

返回当前位，并把 **currentPage** 设为下一页

插入记录算法：

void Manager::insert

对将要插入的记录的键值进行哈希得到哈希值

通过得到的哈希值查找 **index** 目录，找到其应放入的页的 **pageID**

如果该页不在内存中

用时钟算法得到一可用来替换的页

如果该页的 **dirty** 位为 **true**，表示该页已修改

将该页写回硬盘

从硬盘读入要插入的页，放于替换出去的页的内存中

将该记录插入内存中对应的页

如果插入不成功，则需要进行分裂

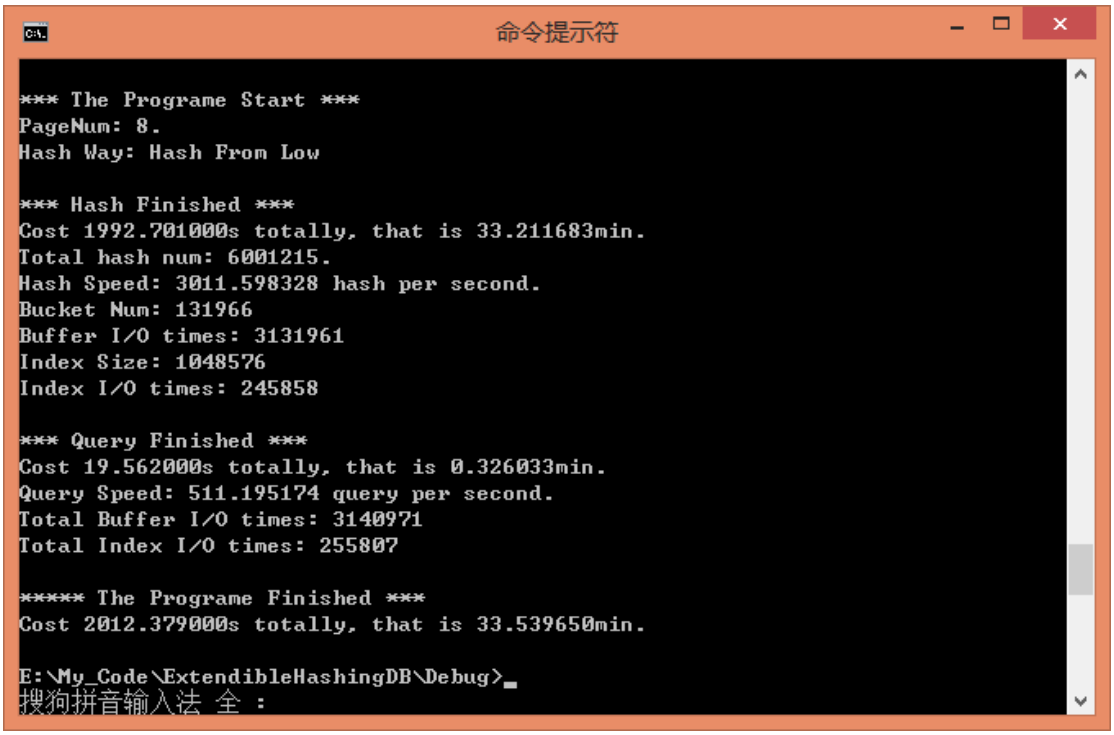
页分裂

递归调用自身再次进行插入

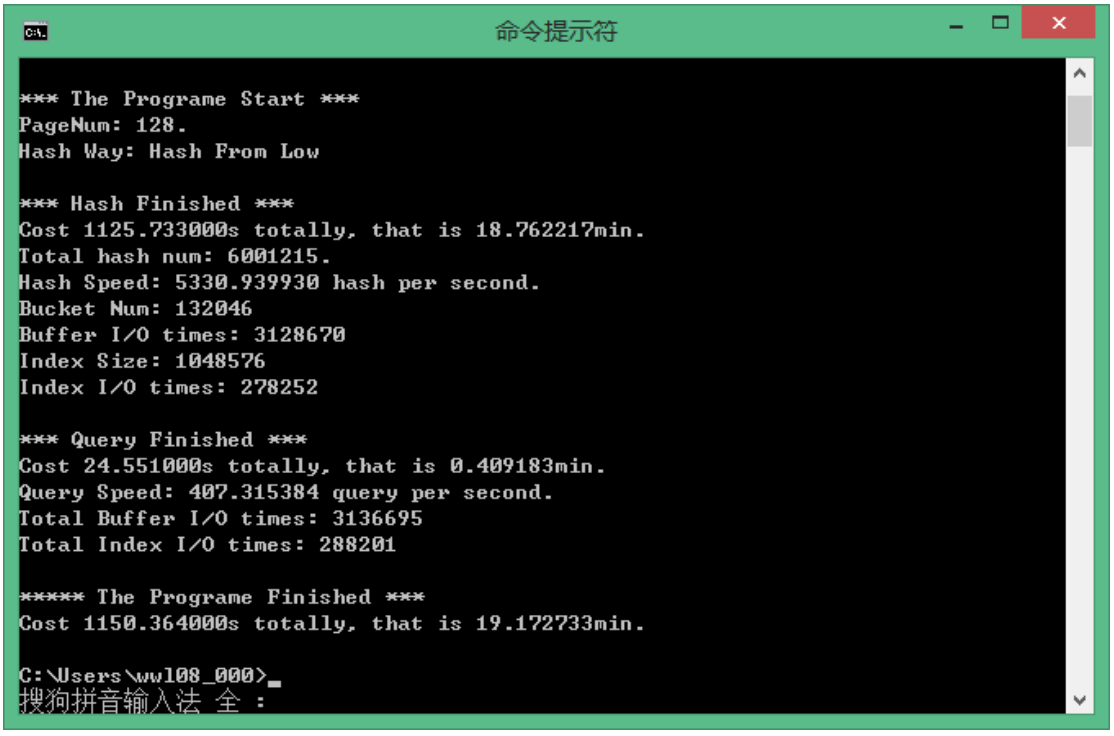
修改该页 **dirty** 位为 **true**

性能测试

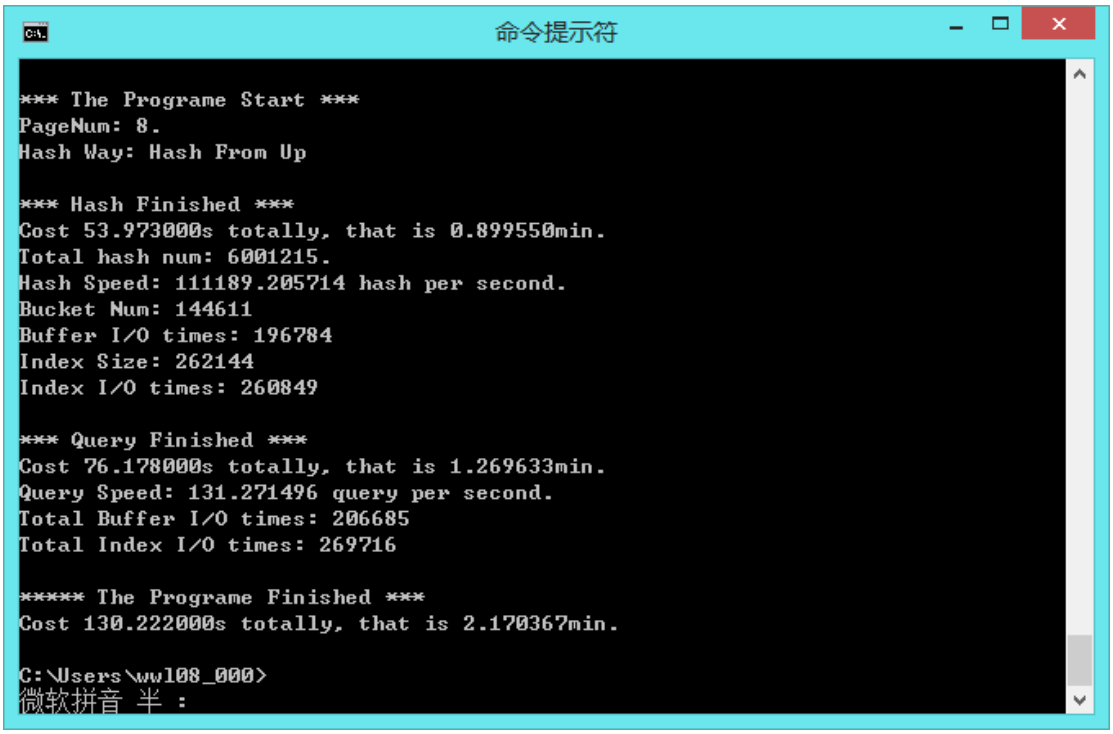
以下为软件在本人电脑中运行的截图：



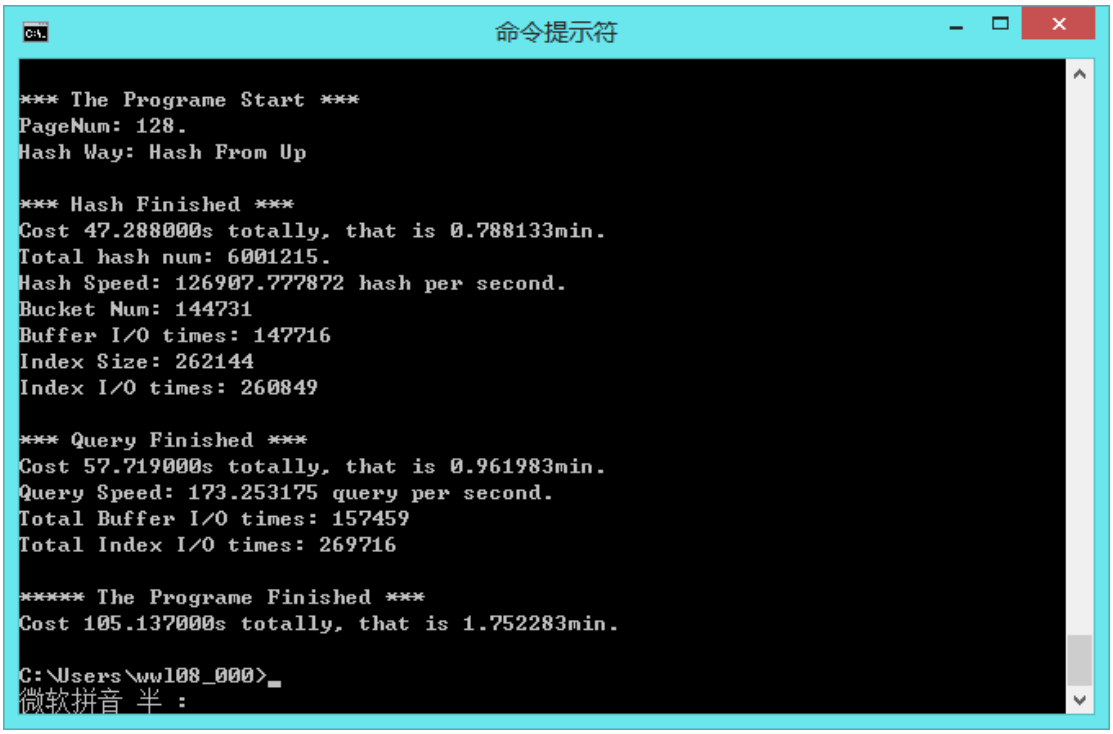
图一：低位扩展哈希，8 页



图二：低位扩展哈希，128 页



图三：高位扩展哈希，8 页



图四：高位扩展哈希，128 页

可得到表格：

表一：哈希方式和缓存页数与程序效率的联系

		建立索引		记录查找	
低位扩展	8 页	1992.701	3011.598/s	19.562	511.195
	128 页	1125.733s	5330.940 /s	24.551s	407.315/s
高位扩展	8 页	53.973s	111189.206 /s	76.178s	131.271/s
	128 页	47.288s	126907.778 /s	57.719s	173.253/s

通过对比低位哈希和高位哈希时的建立索引耗时和查找记录耗时，可以看出建立索引时高位扩展比低位扩展速度要快很多，相差了两个数量级。这很大程度上是因为 listitem.tbl 文件本身是有序的，这就导致高位扩展建立索引时数据会按照顺序写入桶中，并且前期因为数据的哈希值都为 0，导致桶和目录的不断分裂，而由于数据的有序性，后来插入的数据的桶命中率会很高，这就大大降低了 I/O 次数，最终导致了高位哈希的建立索引的速度较快。

而查找数据时低位哈希却比高位哈希快了 3 倍多，这应该是因为测试时 testinput.in 文件中的数据是随机生成的，高位扩展发挥不了其优势，由于数据的随机性目录页的命中率和缓存页的命中率都下降的缘故，而低位扩展数据分布得较为均匀，所以命中率会较高，最终导致低位扩展速度较快。

而对比 8 页和 128 页还可以发现无论是低位还是高位扩展，更多的缓存页都会一定程度上提高索引和搜索时的速度，但提高程度并不是特别明显。

表二：哈希方式和缓存页数与存储空间和 I/O 的联系

		哈希桶的数量	目录的大小	I/O 的次数	I/O 用时占比
低位扩展	8 页	131966	4096KB	3396778	85.48
	128 页	132046	4096KB	3424896	86.95
高位扩展	8 页	144611	1024KB	476401	55.64
	128 页	144731	1024KB	427175	48.64

**I/O 用时比例是参考 VS2013 的性能资源测试报告中 msvcr120d.ddl 的独占样本比例*

首先可以明显看出高位哈希的 I/O 次数和 I/O 用时占比明显低于低位扩展，其中的原因在上一步已经分析过，是因为高位扩展本身所需的 I/O 次数比较少，占比也就会比较少。

目录大小方面，高位索引的目录也比低位索引的小，但哈希桶的数量却比低位扩展的少，这应该是由于低位扩展的数据分布不均匀才导致桶的数目少但目录却更大的结果。

性能优化

1、尽量使用位运算

在一些频繁使用的函数如哈希函数中尽量使用位运算而避免使用乘除法，一些简单的如乘二除二的运算也用左右移位来替代；

2、将调用次数较多的函数改为宏函数或去掉函数调用直接计算；

优化程序时利用 VS 的性能报告发现一些需要频繁使用的函数用时占比较多，将其改为宏调用或直接计算后可加快 10% ~ 15%的程序用时。

3、使用 Win API 代替 C 库函数 (暂未实现)

直接调用 Win API：CreateFile，ReadFile，WriteFile，SetFilePointer。使用 Win API 会降低程序的可移植性，但是对于数据库这种对性能要求较高的系统，必然会针对平台进行优化，所以使用 Win API 是很正确的做法。

预计效率要比使用 fread 等高不少。但由于 CreateFile 和 ReadFile 和 WriteFile 会自动把原本单字符的\n 读入或写出为\n\r 的双字符，修改成本高，经过一天的尝试均告失败后，由于时间关系，最终决定暂时不做这部分的优化。

4、让 BUFF 在空间上连续

申请 BUFF 时候，让所有页位于一个连续的空间内，这样能够显著提高 Cache 的命中率，最终能够提升速率。

心得体会

通过这次实验发现低位哈希和高位哈希各有特点，低位哈希查找速率快，高位哈希建立索引的速度较快，但这次的数据输入是有序的，而且在实际应用中，用户对得到数据的速度往往会比较高，用户会希望搜索数据时程序反应快，但在录入大量数据时，用户会更能等待，并且低位扩展哈希实现起来也更加遍历，这也许就是实际应用中大多使用低位扩展的原因。另外就是发现了相较于算法的优化，可用内存的提高对于程序的效率，最起码相对于这个项目来说，是微不足道的，这也是我一开始所没想到的。

这次实验我是个人组队，实验中所有的代码都要自己写，这虽然是个比较大的挑战，但整个项目下来，最终将运行速度从将近一个小时优化到 100 秒左右，这其中的成就感还是很不错的，所谓无限风光在险峰，挑战越大，成功后的优越感就会越大，这也是我为什么选择个人组队的原因，这不仅锻炼了我的代码能力，项目完成后的成就感更是无可比拟的礼物。

最后，就是感谢老师可以提供这个有趣的项目给我们练习，在写项目的时候我有种乐在其中的感觉。这次提交的代码由于时间关系还没有做到最好的优化，我会在考试结束后，利用暑假的时间对该项目继续进行进一步的优化。