

中山大学软件学院 12 级
数据库课程设计实验报告



中山大學
SUN YAT-SEN UNIVERSITY

目录

一、 小组成员.....	3
介绍了本小组实验成员的信息。	
二、 实验环境.....	3
介绍了本次实验的开发环境、开发语言等。	
三、 实验内容.....	3
简要回顾了本次实验的目标以及主要任务。	
四、 实验思路与设计.....	3
主要介绍本小组进行本次实验时的思路，给出了我们小组整个实验的设计过程，包括考虑的问题、采取的方案、以及所采用的优化。	
五、 代码结构与实现.....	3
包括了本次实验的类图设计，以及相应的实现方法。	
六、 测试结果与分析.....	3
主要展示了本次实验各任务的结果，以及对优化前后的结果的对比分析。	
七、 心得体会.....	3
包括了我们这次实验的收获和感想。	

一、 小组成员

1. 魏杰伟 12330318 weijieweijerry@163.com
2. 肖文惠 12330340 522171087@qq.com
3. 伍兴敏 12330337 530164193@qq.com

二、 实验环境

1. 开发系统：Ubuntu 12.04
2. 测试系统：Ubuntu 10.04/ Ubuntu 12.04/ Ubuntu 13.04/ Ubuntu 14.04
3. 测试计算机：Intel(R) Core(TM) i5-2450M CPU @ 2.50GHz 2.50GHz/
4.00GB RAM
4. 编译器：g++
5. 开发语言：C++

三、 实验内容

本次实验主要是设计一个基本的列存储的数据库 c-store 来处理 TPC-H 中生成的 orders.tbl 和 customer.tbl 这两张表, 实现导入数据、查询数据、压缩、拼接以及计数这五个功能。

四、 实验思路与设计

1. 实验思路

根据给出的需求文档, 我们可以明确地知道我们这次实验有四个环节

- 1) 不带压缩地导入 Orders 表, 并能实现通过输入 orderkey 查出该条记录, 回显 orderkey custkey totalprice shippriority;
- 2) 根据 custkey 列对 orderkey 和 custkey 进行排序, 并将 orderkey 按照行程长度编码压缩;

- 3) 在 2) 的基础上, 实现 Orders 表和 Customer 表的拼接;
- 4) 在 2) 的基础上, 统计 Orders 表记录的数目。

2. 实验设计

2.1 不带压缩地导入数据

对于第一个任务, 我们小组考虑了以下两个问题:

- 1) 对于题目给出的 128 个页是否要用全;
- 2) Orders 表和 Customer 表两张表的属性并不完全一样, 该怎样处理。

经过讨论, 我们小组决定设计两个函数 `loadData_o()` 和 `loadData_c()` 来导入这两张表。至于涉及的页的数目, 我们认为, 可以根据要存储的属性的数目来决定。如 Orders 表中, 我们需要存储 `orderkey`, `custkey`, `totalprice` 以及 `shippriority` 四个属性, 所以, 在导入 Orders 表的时候, 我们仅用到 4 个缓存页。

导入的时候, 通过遍历 `orders` 和 `customer` 表的每一行, 将须要的属性存放在相应的页面中, 当页面满了或者已经读到最后一页时, 将页面换出到磁盘中。

2.2 根据 `orderkey` 来查询记录

内存分页管理并且关系表属性顺序写入磁盘, 故来自同一条记录的不同属性的总偏移量(记录相对首元素的偏移量)相同, 所以查询 `orderkey` 所对应的属性, 实际上只需获得 `orderkey` 的总偏移量, 为了获得总偏移量必须解决以下两个问题:

- 1) 获得 `orderkey` 的页内偏移量(记录相对于页首元素的偏移量);
- 2) 获得 `ordekey` 所在页面的页首偏移量(页首元素相对于首元素的偏移量)。

经过一番思考, 我们小组采用的解决方案如下:

- 1) 考虑到 `orderkey` 有序, 故可以采用二分查找来确定其页内偏移量;
- 2) 为了获得 `orderkey` 的页首偏移量, 我们小组的实验过程中先后采用以下三种思路:

i. 【原始思路】

顺序读入 `orderkey` 的所有页面, 每读入一个页面就更新

页首偏移量并进行该页的二分查找，直至二分查找成功或者读到文件尾巴。

【分析】

通过顺序扫描 orderkey 来获得页首偏移量会导致许多不必要的磁盘 IO，浪费了许多时间。

二分查找

```
1 // Binary search for query.
2 int binarySearch(array[], int start, int end, int key)
3 {
4     // Omitted details
5 }
```

ii. 【优化思路】

通过维护一个查询表来迅速定位须要查找的 orderkey 所在的页面的页首偏移量，然后在页内进行二分查找。查询表的模式为<键值，页首偏移量>，其中以每页的首元素为键值。查询表在导入数据时生成并写入磁盘中。当进行查询时，重新读入查询表，通过遍历查询表确定 orderkey 所在的页面的页首偏移量。

【分析】

维护查询表来定位 orderkey 所在页面，使得查询只需要至多两次磁盘 IO，一个用于读入查询表，一个用于读入页面。缩短了查询所须要的时间。但是这种思路使得 c-store 必须付出一个代价来维护查询表，而且查询表占用了一定的磁盘空间。

iii. 【进一步优化】

考虑到 c-store 中页面是固定大小的，所以可以通过简化查询表来减少 c-store 维护查询表所付出的代价。修改查询表的模式为<键值>，而页首地址则在导入查询表的时候动态生成。在这里 tableFile 表示生成的查询表文件，pfile 表示 Orders 表文件。

生成查询表

```

1 // Omit details
2 int queryTable[MAX_SIZE_QUERY_TABLE];
3 int queryTableSize = 0;
4 int orderkeyData[MAX_ITEM];
5
6 while (fscanf(pfile) != EOF) {
7     if (count == MAX_ITEM) {
8         queryTable[queryTableSize++] = orderkeyData[0];
9         // Process orderkey, custkey etc.
10    }
11 }
12
13 if (count != 0) {
14     queryTable[queryTableSize++] = orderkeyData[0];
15     // Write other data into files
16 }
17
18 fwrite(queryTable, sizeof(int), queryTableSize, tableFile);

```

【分析】

此方法使得查询表的大小减少了一半，而由于 c-store 的页书并不会很多(规模 1 的 Orders 表共 1465 页 规模 10 的 Orders 表共 14649 页)，所以动态生成页首偏移量的代价可以忽略。

2.3 外排序

我们小组采用多路归并排序实现外排序，具体细分为以下步骤：

1) 读入数据：

分别将 orderkey 和 custkey 读入到内存中，每次分别读入 128 个 4K 页。将 orderkey 的 128 个 4K 页和 custkey 的 128 个 4K 页合并成 128 个二维元组<orderkey, custkey>的 8K 页。

2) 快速排序：

将这 128 个二维元组的 8K 页作为一个 128*8K 的大页面在内存中进行快速排序。将排序后的结果写入到临时文件中。重复上一步至所有数据都进行了快速排序。

快速排序

```

1 void quickSort(key[], otherField[], size)
2 {
3     // Omit details
4 }

```

3) 多路归并：

将临时文件导入到内存中（最多导入 127 页），归并所有临时文件页面，并将结果存放在另一个页面中，若该页面满了则写入磁盘中。

2.4 RLE 压缩

将 custkey 导入到页面中, 遍历页面遇到相同元素计数器自增, 不同则将上一步循环的 custkey 和计数器的值构成的元组写入磁盘并且重置计数器。

【优化处理】

一开始我们读一行数据, 进行一次归并。但考虑到这样反复从磁盘读写, 可能会造成大量 IO 消耗, 我们决定采取读一页再进行一次归并的策略。

优化前读入数据策略

```
1 item custkeyItem;
2
3 fread(&(custkeyItem.value), sizeof(int), 1, cfileAfterSort);
4 while (true) {
5     fread(&(nextCustkey), sizeof(int), 1, cfileAfterSort);
6     // Processing
7 }
```

优化后读入数据策略

```
1 int ckeyAfterSort[MAX_ITEM];
2 item ckeyAfterCompress[MAX_ITEM];
3
4 // Compress custkey sort.
5 while(!feof(cfileAfterSort)) {
6     int num = fread(ckeyAfterSort, sizeof(int), MAX_ITEM, cfileAfterSort);
7     for (int i = 0; i < num; ++i) {
8         // Omit details
9     }
10 }
```

2.5 JOIN 操作

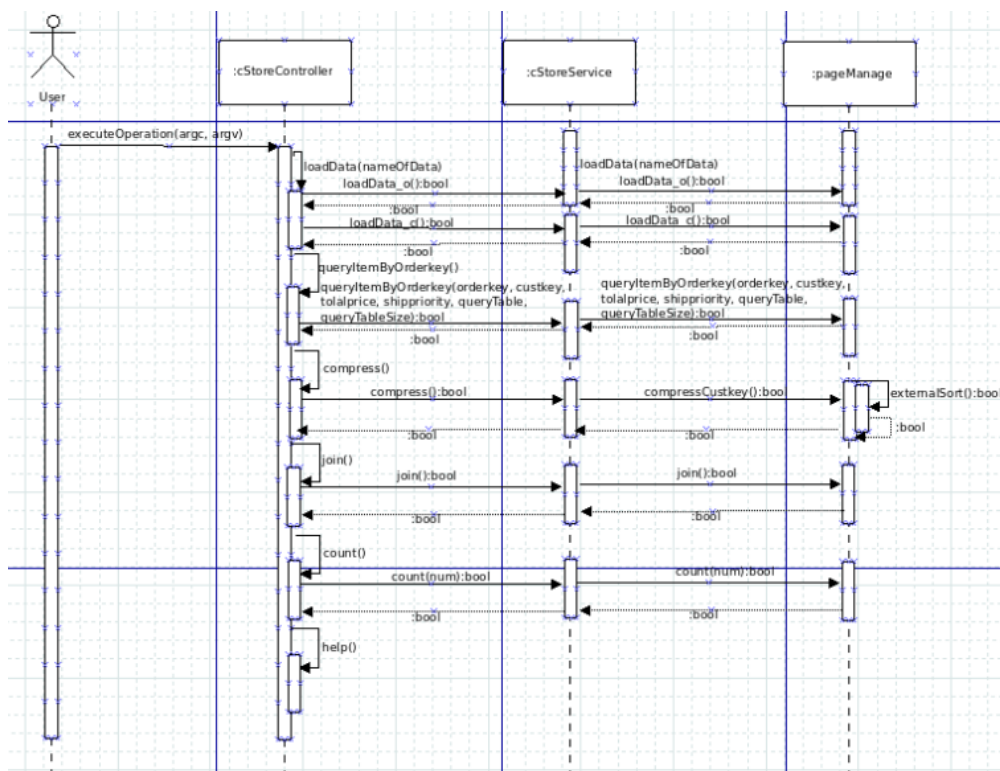
使用两个指针, 同时从 Customer 表压缩后的 custkey 列和 Orders 表的 custkey 列进行扫描, 遇到相同值时, 输出对应的 custkey 值和 orderkey 值。

2.6 COUNT 操作

通过累加 RLE 压缩后结果所有元组的第二项得到。

1) 时序图

从整个设计来看, 为了更好理解整个过程, 我们得出以下的 UML 时序图:



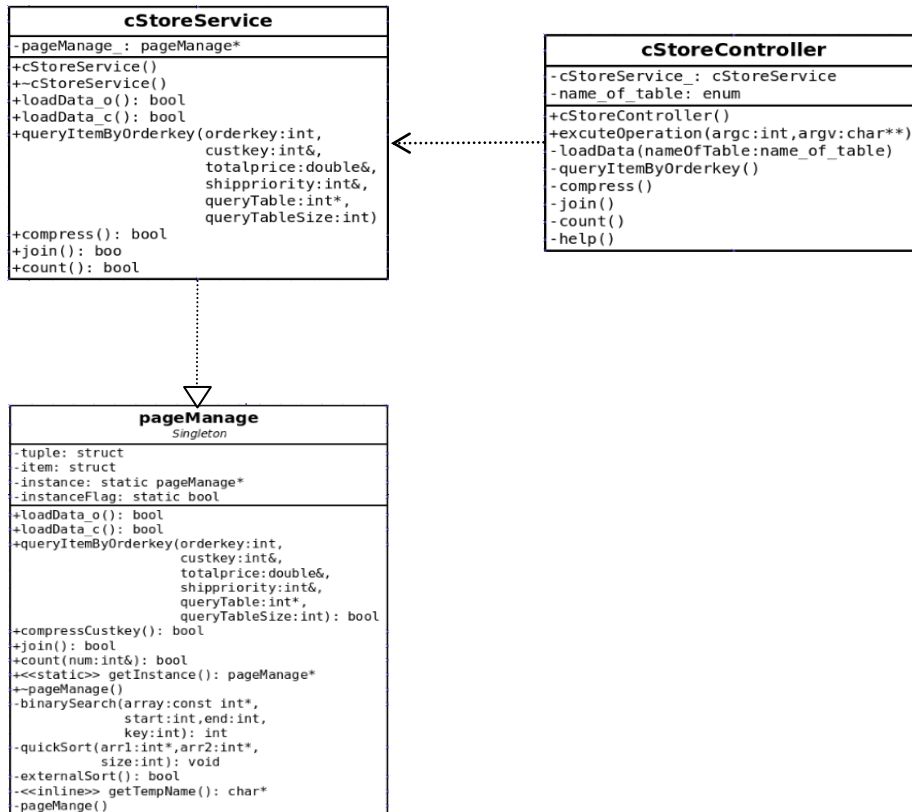
五、 代码结构与实现

1. 类图设计

经过以上讨论，我们小组最后决定设计三个类来实现这次实验：

- 1) pageManage 类：c-store 的后台(内核)，各种功能的具体的实现，使用单例模式。
- 2) cStoreService 类：实现 c-store 前端和后台的分离，实现数据对接，增强 c-store 的扩展性。
- 3) cStoreController 类：c-store 的前端，用于和用户交互。

得出的 UML 类图如下所示：



2. 伪代码及说明

2.1 导入表

实现导入 Orders 表的函数 loadData_o() 的伪代码如下：

```

1  orderkeyData, custkeyData, toatalData, shippriorityData is array
2  while tbl → okay, ckey, tprice, spriority
3      if count = 1024
4          queryTable[queryTableSize++] = orderkeyData[0]
5          orderkeyData, custkeyData, toatalData, shippriorityData → binary files
6      end if
7      okay → orderkeyData[count]
8      ckey → custkeyData[count]
9      tprice → toatalData[count]
10     spriority → shippriorityData[count]
11 end while
12 queryTable → file
    
```

实现导入 Customer 表的函数 loadData_c() 的伪代码与 loadData_o() 函数类似。

2.2 查询

实现查询的 queryItemByOrderkey(orderkey, custkey, totalprice, shippriority, queryTable[], queryTableSize) 函数的伪代码如下：

```

1  file → queryTable
2  position = 0
3  for pageId = 0 to queryTable Size
4      if orderkey = queryTable[pageId]
5          seek position
6          files → custkey, totalprice, shippriority
7      else if orderkey < queryTable[pageId]
8          break
9      end if
10     position += 1024

11 end for
12 position -= 1024
13 seek position
14 file → okeyPage(Array)
15 index = binarySearch okeyPage
16 seek position+index
17 files → custkey, totalprice, shippriority

```

2.3 外排

【原始方法】

```

1  while file not end
2      file → tuple[128*1024]
3      quick sort tuple
4      tuple → tempFile
5  end while
6
7  for i = 0 to num_of_tempFiles
8      tempFiles[i] → head[i]
9  end for
10 while num_of_eof != num_of_tempFiles
11     for i = 0 to num_of_tempFiles
12         if tempFiles[i] is end
13             continue
14         end if
15         if head[i].custkey is min
16             minMax = head[i]
17             min_index = i
18         end if
19     end for
20     minMax → file
21     tempFiles[min_index] → head[min_index]
22     if tempFiles[min_index] is end
23         ++num_of_eof
24     end while
25 remove tempFiles

```

【优化后】

```

1  while file not end
2    file → tuple[128*1024]
3    quick sort tuple
4    tuple → tempFile
5  end while
6
7  for i = 0 to num_of_tempFiles
8    tempFiles[i] → head[i]
9    0 → cur[i]
10 end for
11
12 while num_of_eof != num_of_tempFiles
13   for i = 0 to num_of_tempFiles
14     if tempFiles[i] is end And len[i] == cur[i]
15       continue
16     end if
17     if head[i][cur[i]].custkey < minMax.custkey
18       head[i][cur[i]] → minMax
19       i → min_index
20     end if
21   end for
22   data_after_sort → files
23   files → head[min_index]
24   if tempFiles[min_index] is And cur[min_index] = len[min_index]
25     ++num_of_eof
26   end if
27 end while
28 remove tempFiles

```

2.4 RLE 压缩

【原始方法】

```

1  externail sort
2  while
3    file → nextCustkey
4    if file is end
5      custItem → file
6    end if
7    if nextCustkey = custkeyItem.value
8      ++custkeyItem.length
9    else
10     custkeyItem → file
11     custkeyItem.value = nextCustkey
12     custkeyItem.length = 1
13   end if
14 end while

```

【优化后】

```

1  externail sort
2  while cfileAfterSort is not end
3    (file → ckeyAfterSort) → num
4    for i = 0 to num
5      if (ckeyAfterSort = pre_cke)
6        ++length
7      else
8        if ckeyAfterCompress is full
9          ckeyAfterCompress → file
10        end if
11        (pre_cke, length) → ckeyAfterCompress[count++]
12        length = 1
13      end if
14      pre_cke = ckeyAfterSort[i]
15    end for
16  end while
17 ckeyAfterCompress → file

```

2.5 JOIN

实现拼接的 join() 函数的伪代码如下所示:

```

1 while o_keyfile is not end and c_keyfile is not end
2   if next_o_key
3     o_keyfile get next element
4   end if
5   if next_c_key
6     c_keyfile get next element
7   end if
8   if o_key.value = c_key
9     for i = 0 to o_key.length
10      file → o_okey
11      o_key.value o_okey → stdout
12    end for
13  else if o_key.value < c_key
14    next_o_key = true
15    next_c_key = false
16  else
17    next_o_key = false
18    next_c_key = true
19  end if
20 end while

```

2.6 COUNT

实现 COUNT 的 count(&num) 函数伪代码如下:

```

1 num = 0
2 while file is not end
3   file → item(Array)
4   for i = 0 to item size
5     num += item[i].length
6   end for
7 end while

```

六、 测试结果与优化

本次测试采用的表是规模 10 的 Orders 表 (1.7G) 和 Customer 表。

测试的时候, 参照 Readme 中的命令格式。可以通过输入 ./cStore help

来查看命令的具体操作。

```

----- cstore -----
Action :
load orders      - load the data from orders.tbl to cstore.
load customer    - load the data from customer.tbl to cstore.
retrieve orders  - query the item by orderkey.
compress orders 1 - compress the first column.
join             - join orders.tbl and customer.tbl
count           - count the items of orders.tbl.

```

1. 测试结果

1.1 任务一：导入表

导入 Orders 和 Customer 表的情况如下所示:

Orders 表

```

Loading data...
Success to load data. (18.00 sec)

```

Customer 表

```
Loading data...
Success to load data. (6.00 sec)
```

1.2 任务一：根据 ORDERKEY 来查询数据

对比文档给出的输出格式，我们的输入输出格式与其一致：

<p>以下是样例输入</p> <p>Sample Input:</p> <p>123</p> <p>4</p> <p>56</p> <p>19</p>	<pre>Remind : Please enter a orderkey. Enter <ctrl+d> to stop. query 7 7 39136 252004.18 0 (0.00 sec) 6000000 6000000 110063 37625.29 0 (0.00 sec) 8 Empty. (0.00 sec) 32 32 130057 208660.75 0 (0.00 sec) Finish.</pre> <p>对于每一个查询，输出该表已存的所有列，以空格隔开。例如，对于 ORDERS 表，则 primary key 对应 ORDERKEY，结果应为 4 列 (如果没有导入更多列)。</p>
---	---

在这里，我们将 orders.tbl 拆分成 100 个临时表，每个表的记录数为 150000。同时我们写了一个测试代码，对临时表里的每条记录的 orderkey 输入查询，对比查询得出的 custkey, totalprice, shippriority 的值是否和临时表里的一致。

	<pre>18 while(fscanf(qFile, "%d %d %lf %d\n", 19 &okey, &ckey, &tprice, &spriority) != EOF) { 20 double totalprice; 21 c.queryItemByOrderkey(okey, custkey, totalprice, shippriority, 22 queryTable, queryTableSize); 23 EXPECT_EQ(ckey, custkey); 24 EXPECT_EQ(tprice, totalprice); 25 EXPECT_EQ(spriority, shippriority); 26 count += 1; 27 } 28 printf("Test for temp99.bin.\n"); 29 printf("Total records: %d\n", count);</pre> <p style="text-align: right;">gtest 代码主体</p>
--	---

【检验查询结果准确性】

这里我们只展示部分测试截图。

Test for temp0.bin. Total records: 150000 [OK] cStoreService.query (3061 ms) [-----] 1 test from cStoreService (3061 ms total) [-----] Global test environment tear-down [=====] 1 test from 1 test case ran. (3061 ms total) [PASSED] 1 test.	Test for temp10.bin. Total records: 150000 [OK] cStoreService.query (3772 ms) [-----] 1 test from cStoreService (3772 ms total) [-----] Global test environment tear-down [=====] 1 test from 1 test case ran. (3772 ms total) [PASSED] 1 test.
Test for temp20.bin. Total records: 150000 [OK] cStoreService.query (4393 ms) [-----] 1 test from cStoreService (4393 ms total) [-----] Global test environment tear-down [=====] 1 test from 1 test case ran. (4393 ms total) [PASSED] 1 test.	Test for temp30.bin. Total records: 150000 [OK] cStoreService.query (5084 ms) [-----] 1 test from cStoreService (5084 ms total) [-----] Global test environment tear-down [=====] 1 test from 1 test case ran. (5084 ms total) [PASSED] 1 test.
Test for temp40.bin. Total records: 150000 [OK] cStoreService.query (5756 ms) [-----] 1 test from cStoreService (5757 ms total) [-----] Global test environment tear-down [=====] 1 test from 1 test case ran. (5757 ms total) [PASSED] 1 test.	Test for temp50.bin. Total records: 150000 [OK] cStoreService.query (6507 ms) [-----] 1 test from cStoreService (6507 ms total) [-----] Global test environment tear-down [=====] 1 test from 1 test case ran. (6507 ms total) [PASSED] 1 test.
Test for temp60.bin. Total records: 150000 [OK] cStoreService.query (7117 ms) [-----] 1 test from cStoreService (7117 ms total) [-----] Global test environment tear-down [=====] 1 test from 1 test case ran. (7117 ms total) [PASSED] 1 test.	Test for temp70.bin. Total records: 150000 [OK] cStoreService.query (7765 ms) [-----] 1 test from cStoreService (7766 ms total) [-----] Global test environment tear-down [=====] 1 test from 1 test case ran. (7766 ms total) [PASSED] 1 test.
Test for temp80.bin. Total records: 150000 [OK] cStoreService.query (8355 ms) [-----] 1 test from cStoreService (8355 ms total) [-----] Global test environment tear-down [=====] 1 test from 1 test case ran. (8356 ms total) [PASSED] 1 test.	Test for temp90.bin. Total records: 150000 [OK] cStoreService.query (8987 ms) [-----] 1 test from cStoreService (8987 ms total) [-----] Global test environment tear-down [=====] 1 test from 1 test case ran. (8988 ms total) [PASSED] 1 test.
Test for temp99.bin. Total records: 150000 [OK] cStoreService.query (9646 ms) [-----] 1 test from cStoreService (9646 ms total) [-----] Global test environment tear-down [=====] 1 test from 1 test case ran. (9646 ms total) [PASSED] 1 test.	

【数据分析】

从图中查每个表的时间可知，随着查询的数据的值越大，数据处理所耗的时间也越多，我们认为，这可能是因为数据越大，计算机处理起来越困难。

【优化前后对比】

优化前	优化后
Test for temp0.bin. Total records: 150000 [OK] cStoreService.query (7955 ms) [-----] 1 test from cStoreService (7955 ms total) [-----] Global test environment tear-down [=====] 1 test from 1 test case ran. (7955 ms total) [PASSED] 1 test.	Test for temp0.bin. Total records: 150000 [OK] cStoreService.query (3061 ms) [-----] 1 test from cStoreService (3061 ms total) [-----] Global test environment tear-down [=====] 1 test from 1 test case ran. (3061 ms total) [PASSED] 1 test.

从上图可以看出，我们小组对查询的优化，让查询数据的速度提升了一倍甚至以上，可见我们的优化还是颇有成效的。

1.3 任务二：压缩数据

经 cStore 压缩后的结果如下：


Compressing data...
Success to compress. (25.00 sec)

压缩后	外排后 custkey 的情况	压缩后	外排后 custkey 的情况
<pre> Compress.cpp 1 1 15 2 2 3 3 4 19 4 5 17 generateTest.cpp </pre>	<pre> Compress.cpp 1 1 2 1 3 1 4 1 5 1 6 1 7 1 8 1 9 1 10 1 11 1 12 1 13 1 14 1 15 1 16 2 17 2 18 2 19 4 20 4 21 4 22 4 23 4 24 4 25 4 26 4 27 4 28 4 29 4 30 4 31 4 32 4 33 4 34 4 35 4 36 4 37 4 38 5 39 5 40 5 41 5 </pre>	<pre> 999980 1499996 9 999981 1499998 22 999982 1499999 18 999983 * compressData.tbl * generateTest.cpp </pre>	<pre> Compress.cpp 14999961 1499998 14999962 1499998 14999963 1499998 14999964 1499998 14999965 1499998 14999966 1499998 14999967 1499998 14999968 1499998 14999969 1499998 14999970 1499998 14999971 1499998 14999972 1499998 14999973 1499998 14999974 1499998 14999975 1499998 14999976 1499998 14999977 1499998 14999978 1499998 14999979 1499998 14999980 1499998 14999981 1499998 14999982 1499998 14999983 1499998 14999984 1499998 14999985 1499998 14999986 1499998 14999987 1499998 14999988 1499998 14999989 1499998 14999990 1499998 14999991 1499998 14999992 1499998 14999993 1499998 14999994 1499998 14999995 1499998 14999996 1499998 14999997 1499998 14999998 1499998 14999999 1499998 15000000 1499998 </pre>


【压缩率】

由下图可知，本次压缩的压缩率为 $7999856/60000000 * 100\% = 13.3\%$ 。

压缩前 custkey 文件的大小

	名称(N):	o_custkey.bin
	类型:	二进制 (application/octet-stream)
	大小:	60.0 MB (60,000,000 字节)

压缩后 custkey 文件的大小

	名称(N):	o_custkeyAfterCompress.bin
	类型:	二进制 (application/octet-stream)
	大小:	8.0 MB (7,999,856 字节)

【优化前后比较】

优化前	优化后
<pre>Compressing data... Success to compress. (25.00 sec)</pre>	<pre>Compressing data... Success to compress. (31.00 sec)</pre>

从上图看出，优化后反而压缩花费的时间更多。

1.4 任务三：JOIN 操作

Join 后的结果如下：

```
1499999 49998689
1499999 51693889
1499999 54747524
1499999 55439588
Success to join. (481.00 sec)
```

1.5 任务四：COUNT 操作

对 Orders 表中的 orderkey 进行计数结果如下：

```
xwh@xwh-HP-Pavilion-dv4-Notebook-PC:~/my4V3$ ./cStore count
counting data...
15000000 (0.00 sec)
```

七、心得体会

这次实验是要设计一个能处理两张表的列存储的数据库，工作量不小，但是我们都觉得收获不小。

【小组成员感受】

平时上课的时候，听着老师介绍外排、列存储的时候，有的地方还是比较不理解的。比如说外排的时候，对一列进行排序，其他列不动，这样感觉会影响表里的记录，即 key[0]可能对应的是 attr1[2]，而不是 attr1[0]。动手的时候才能更好地理解理论，当自己亲手写外排的时候，才更好地了解到外排的整个过程。在做完任务二之后，才知道外排后的结果会存在另一个文件里，而不影响原来的记录。实验对理论学习还是有很大地帮助的，可以巩固理论，加深对课本上理论的理解。感觉通过这次实验，我们对数据库的认识就不再局限于如何使用 SQL 语句来查询记录，还更深层次地明白我们对数据库进行的操作的设计原理。同时也加强了我们的动手能力。

除了有收获外，一个极大的感受是打代码需要细心和耐心。不细心，就容易在代码里出现小错误，比如多了一个}或者单词拼写错误，花上一个小时甚至以上，到最后发现可能只是因为自己粗心而造成的错误，就觉得有些伤感。没有耐心，程序调试就会把我们给弄倒，调试真的是一件很花时间、脑力、精力的事。机器和人之间的沟通，还是挺高深的！

【实验中遇到的问题】

这次实验我们的架构，其实是借鉴了实训时 Agenda 的架构设计，来进行修改的。但是，这一次的架构不是师兄给我们，而是让我们自己动手设计，自然也遇到了不少问题。

1. 一开始碰到的是，做好任务一、二后，代码在 Ubuntu 12.04 下能正常运行，但是在 Ubuntu 13.04 下导入数据就出现段错误。在检查之后，发现是主函数里 cStoreController_被定义成指针，导致 cStoreService 类的构造函数没有被调用。

2. 在写 gtest 测试代码的时候，出现多次某些函数未被使用的错误，就只能把一些与查询测试无关的函数注释掉才进行测试。

这些其实都是小细节来的，可能觉得两行代码不可能会出错，但是就是因为主函数的两行代码导致段错误。

如果要做好一件事，不管是不是打代码，都得更加严谨细心才行。一时着急，而没有注意一些细节问题，将来可能还要花更多地时间去修补自己着急造成的漏洞，就显得尤为得不偿失了。

【总的来说】

实验都是为了巩固理论，给学生更多动手的机会而设有的。这次实验，收获的不仅是一个自己设计的数据库，以及地理论的进一步的理解，还有更多为人做事上的心得。万事急不来，不管是做什么，都应细心细心。