

# 实验报告

—— FlyIsh 数据库设计 Stage 1



组号:

1

成员:

王迎旭 16340226 欧穗新 16340173

谢浩峰 16340253 彭伟林 16340181

吴聪 16340237

班级:

16 级软件工程教务三班

日期:

2018. 5. 14

# 目录

Part1: 需求分析	第 2 页
Part2: 设计思路	第 3 页
Part3: 测试程序	第 16 页
Part4: 实验分析	第 18 页
Part5: 表格填写	第 19 页
Part6: 项目探索	第 20 页
Part7: 项目分工	第 21 页

## Part1: 需求分析

### 1.1 建立数据库

目的:

1. 读取数据, 将数据按页存储在磁盘
2. 数据预处理, 将处理后的数据按页存储在磁盘。

### 1.2 处理查询

目标:

给定一个查询点, 表示为原始向量, 完成以下查询

1. 返回原始点集中距离查询点点最近的 K 个点 (KNN) 的 id

2. 返回处理后的点集中距离查询点点最近的 K 个点 (KNN) 的 id (注意要对查询点做预处理)

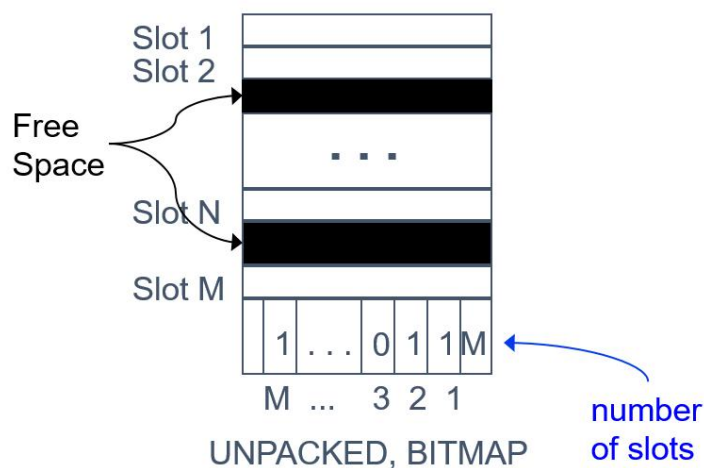
## Part2: 设计思路

### 2.1 页存储

考虑到项目最后部分要实现数据库的动态维护, 也即要能够很好地管理磁盘页, 跟踪空闲槽的使用等等, 所以按页存储时有必要设计好数据在页中如何存储。

#### 2.1.1 页存储设计要求

- ① 输入数据为  $n$  个  $d$  维向量, 每个向量都有一个 id 作为索引。
- ② 内存缓冲区中设置 50 个大小为 64KB (65536B) 的帧, 帧的大小与磁盘页大小 (64KB) 一致。帧用于读写和处理数据。
- ③ 文件 IO 以页为单位, 每次读取磁盘上的一页, 写时把一整页写到磁盘。
- ④ 磁盘页以定长记录存储。
- ⑤ 页格式使用右图所示的结构。



- ⑥ 文件以二进制方式读写和存储。
- ⑦ 此外我们还需要能够做到数据库的动态维护, 这涉及磁盘页的管理, 空闲槽的跟踪, 向量 id 的分配等问题。

#### 2.1.2 页实现

### 2.1.2.1 页头

① 如果我们把一页就简单地理解为磁盘上的一个文件，那么为了节省空间，我们就不必把页 id 放到一条记录中去(否则记录的格式为<pageId slotId vectorData>), 而是在页中的开头部分设计一个页头部分，其中就记录了该页的 id，表示为 pageId，大小为 32bit。

② 页头部分应该还需要指定每个槽的大小，表示为 slotSize，其大小为 32bit。这样我们在读取记录的时候，就可以知道该读取多少位（对于 mnist 原始数据，为 32 + 8 \* 784bit）。此外，这个数据也便于我们在动态维护的时候用于判断某个数据是否可以插入到该磁盘页中去。

③ 页头部分有必要记录该页有多少个槽。在确定了每个槽的大小之后，我们就可以确定每页大概可以有多少个槽，表示为 NumOfSize。因为我们已经用 32bit 来记录 pageId（页 id），用 32bit 来记录 slotSize（槽大小/每条记录的大小），用 32bit 来保存 numOfSlot，那么 numOfSlot 的计算不过是解一个方程，如下，其中 slotSize 已知：

$$\frac{(64 \times 1024 \times 8 - 32 \times 3 - numOfSlot \times 8)}{slotSize} = numOfSlot$$

$$numOfSlot = \frac{524192}{(slotSize + 8)}$$

以 mnist 数据集为例，其向量维度 d 为 784，则可求得 slotSize 为 32 + 8 \* 784 = 6304bit，表示一个槽大概会占 6304 个 bit，numOfSlot 的值为 83，表示在给定 64KB 的文件下，只能存储大概 83 条记录，也即只能有 83 个槽。

④ 页头部分还需要包括一个 bool 数组，因为我们使用 bool 数组来处理槽的使用情况（本想使用 byte 数组，但是 C 中没有，只能使用 bool，本质上没有区别，其大小为 1 个字节就可以了）。每个槽对应一个 Byte 位，值为 0 时表示槽为空，否则非空。利用上面算出的 numOfSlot，我们可以得知，我们的 bool 数组有多少位。

⑤ 记录在页头部分的页的其他信息以及方便跳转的一些位标志值，比如下一页的 pageId，最近插入的记录所在槽的 slotId，空的槽地址便于直接插入记录等等，不过暂时我们没有考虑，如果有必要添加也很方便，只是 numOfSlot 的求值部分那个方程需要做一点点的修改。

### 2.1.2.2 页的正文

使用<slot id>来标识一条记录。也即一条记录的格式为<slotId vectorData>, slot id 也即槽的 id, 大小为 32bit, 它对于一个完整的数据集是唯一的, 也即指定 slot id 可以唯一地获取一个完整数据集中的一条向量记录。

### 2.1.3 实际页的设计 (应用于对应文件)

#### 2.1.3.1 mnist 文件

```
#page header
4 bytes      (int pageId)
4 bytes      (int slotSize)
4 bytes      (int numOfSlot)
numOfSlot bytes (bool[] slotArray)
#page content
<4 bytes(int slotId) d bytes(unsigned char[] vector)>
<4 bytes(int slotId) d bytes(unsigned char[] vector)>
...
<4 bytes(int slotId) d bytes(unsigned char[] vector)>
```

#### 2.1.3.2 preprocess\_mnist 文件

```
#page header
4 bytes      (int pageId)
4 bytes      (int slotSize)
4 bytes      (int numOfSlot)
numOfSlot bytes (bool[] slotArray)
#page content
<4 bytes(int slotId) d*2 bytes(short int[] vector)>
<4 bytes(int slotId) d*2 bytes(short int[] vector)>
...
<4 bytes(int slotId) d*2 bytes(short int[] vector)>
```

#### 2.1.3.3 glove 文件

```
#page header
4 bytes      (int pageId)
4 bytes      (int slotSize)
4 bytes      (int numOfSlot)
numOfSlot bytes (bool[] slotArray)
#page content
<4 bytes(int slotId) d*4 bytes(float[] vector)>
<4 bytes(int slotId) d*4 bytes(float[] vector)>
...
<4 bytes(int slotId) d*4 bytes(float[] vector)>
```

### 2.1.3.4 process\_glove 文件

```
#page header
4 bytes      (int pageId)
4 bytes      (int slotSize)
4 bytes      (int numOfSlot)
numOfSlot bytes (bool[] slotArray)
#page content
<4 bytes(int slotId) d*4 bytes(float[] vector)>
<4 bytes(int slotId) d*4 bytes(float[] vector)>
...
<4 bytes(int slotId) d*4 bytes(float[] vector)>
```

### 2.1.4 页存储对后期编程上的帮助

总的来说，在我们按页存储以后，我们可以非常方便地调用页中的数据 and 修改其中的数据，同时保证其安全性。

当我们需要获取某个向量，只需要指定其 id（也即页格式中的 `slotId`），我们就能马上定位它。因为它所在的页（也即页格式中的 `pageId`），以及它在页中的槽的位置（在页的第几个槽，设为 `slotIndex`）都可以通过该向量求得。

求解公式为：

$$pageId = \frac{slotId}{numOfSlot}$$

$$slotIndex = slotId \% numOfSlot$$

当我们需要插入某个向量时，使用页头部分的 `slotSize` 可以判断出该向量是否和该页中其他向量格式相同（比如维度相同或其所属的域相同），这样就可以防止把不当的数据插入到页中。

## 2.2 建立数据库

### 2.2.1 读取原始数据

#### 2.2.1.1 二进制读取原始数据

① `void fileIO_origin_data_in_mnist(const char* infile, const int n, const int d)`

该函数用于读取原始数据 mnist，并将其按页存储到文件夹 origin\_data\_mnist 中，我们对于每个磁盘页创建一个文件，文件名正是该页的 pageId

```
48     memset(s, 0, sizeof(s));
49     sprintf(s, "%s%d", "origin_data_mnist/", fileNum);
50     fout = fopen(s, "wb");
```

以二进制的形式把页头信息写入，其中 numOfSlot 我们在前面页存储设计中已经提到如何计算求得。

```
51     // page id
52     fwrite(&fileNum, sizeof(int), 1, fout);
53     // slotsize = 32 + 8 * 784 = 6304bit = 788byte
54     int slotSize = 788;
55     fwrite(&slotSize, sizeof(int), 1, fout);
56     // number of slot
57     int numOfSlot = 83;
58     fwrite(&numOfSlot, sizeof(int), 1, fout);
59
60     // array to tag every slot
61     // in case the last file is not full
62     if (fileNum == (n / 83)) {
63         for (int k = 0; k < 83; k++) {
64             if (k < (n % 83)) {
65                 fwrite(&arrayToTagSlot, sizeof(bool), 1, fout);
66             }
67             else {
68                 bool tagSlotNotUse = false;
69                 fwrite(&tagSlotNotUse, sizeof(bool), 1, fout);
70             }
71         }
72     }
73     else {
74         for (int k = 0; k < 83; k++) {
75             fwrite(&arrayToTagSlot, sizeof(bool), 1, fout);
76         }
77     }
78 }
```

以二进制形式把页的正文部分写入，其中 ti 也即向量对应的 id。由于数据集中直观上看到向量的每个分量是浮点数，但我们又想只用 8bit 进行存储，所以我们先以二进制的形式读取向量中的每个分量，然后转换为 int 类型后再取低 8 位，以 unsigned char 类型保存。

```
83     // read vectorId and write it as ti
84     fscanf(fin, "%d", &ti);
85     fwrite(&ti, sizeof(int), 1, fout);
86
87
88     for (int j = 0; j < d; ++j)
89     {
90         fscanf(fin, "%f", &a);
91         temp = ((int)a) & 0x000000ff;
92         fwrite(&temp, sizeof(unsigned char), 1, fout);
93     }
94 }
```

② void fileIO\_origin\_data\_in\_glove(const char\* infile, const int n, const int d)

该函数用于读取原始数据 glove，并将其按页存储到文件夹 origin\_data\_glove 中，基本与①函数类似。不同的是槽的大小和槽的数量不一样了，且我们存储向量的每个分量是使用 float 来进行存储。

```
132     int slotSize = 1204;
133     fwrite(&slotSize, sizeof(int), 1, fout);
134     //number of slot
135     int numOfSlot = 54;
136     fwrite(&numOfSlot, sizeof(int), 1, fout);
```

```
165     for (int j = 0; j < d; ++j)
166     {
167         fscanf(fin, "%f", &td);
168         fwrite(&td, sizeof(float), 1, fout);
169     }
```

### 2.2.1.2 按页存储

对两个数据集进行按页存储，让每个数据集分成大小为 64k 的小文件，同时提供 api 对这些小文件进行读取，让内存中始终进行的 io 是以读取 50 个 64k 的文件为基础的。

如何保证我们页的大小是 64kb 呢？答案是，只要按照相应的页格式进行存储，就可以保证了。在前面的页格式设计中，对于给定的数据集，我们需要在页头中给出 numOfSlot，这个变量就指明了页中可以存多少个槽，也即可以存多少个记录，所以在对数据集进行存储时，每次存储了 numOfSlot 个向量后就开始往新页存储数据，这也就实现了按页存储。以 mnist 数据集为例：

对于 mnist 数据集，按照我们设计的页格式一页大概能有 83 个槽，所以下面的代码中有一个取余操作，用来判断是否是刚刚开始往一个页中写入二进制形式的页信息。



```

// read all vector
for (int i = 0; i < n; i++) {
    // Each page's header info
    if (i % 83 == 0) {
        if (fout != NULL)
            fclose(fout);
        memset(s, 0, sizeof(s));
        sprintf(s, "%s%d", "origin_data_mnist/", fileNum);
        fout = fopen(s, "wb");
        /** binary write page header
    * ...
    */
        fileNum++;
    }

    /** binary write page content
    * ...
    */
}

```

### 2.2.2 文件读取函数设计

**规定：在内存进行 50 个帧的操作**

这么做主要是为了模仿数据库，当然课件里面也有要求。大概就是模仿数据库，输入的数据先到内存再到磁盘（这个我们还没确定），要获取数据也是先从磁盘读到内存，然后要使用数据的时候就去内存（50 个帧）中找。同时也能把不同的功能分割开 也方便后续的维护和功能扩展。

#### 2.2.2.1 内存缓冲区设计：

由于一个帧的大小是固定下来的 64KB，磁盘文件大小也和内存缓冲区内的帧大小一致，在参考了课本中第九章对于文件管理方面的介绍，设置了 50 个 char 型数组。

**也即：char all\_pages[50][65536]**

每一个 page 是一个长度为 65536 字节的 char 型数组，使用 char 类型是因为在 C/C++ 中，char 是为数不多只占一个字节的基本数据类型，这样便于后面对于内存缓冲区内帧的操作。为了便于后续的操作，同时还设置了额外的变量来维护帧的信息，如：

① bool dirty[50];//[用来表示内存中的帧在从磁盘读取之后有无被改写（为后续的操作做好准备），以便知道哪些帧应该写回磁盘以保持数据一致性]

② int file\_index[50];//[用来记录内存中帧对应的磁盘文件的索引，以便写回]

③ `int head_index, end_index, pages_count;`//[三个变量用来表示内存缓冲区中的帧的情况，用了数组为基础的循环队列的思想，`head_index` 表示目前内存缓冲区中最早被加入的帧的索引，`end_index` 表示内存缓冲区中最后被加入的帧的索引，`page_count` 用来记录当前内存缓冲区有多少个帧。维护内存缓冲区使用了按时间计算的思想，当内存缓冲区满时，新加入内存缓冲区的会覆盖当前内存中最早写进的帧 (`head_index`) ]

④ `FILE* file_ptr;`//[用来将磁盘页读入到 50 帧的内存缓存区（也即 `all_pages`）的文件指针]

## 2.2.2.2 接口设计

所有的接口如下：

① `void init_pages();`

该接口用于初始化或重置接口所使用到的所有资源（也即前面提到的内存缓存区设计中的变量）。在使用所有其他的接口之前，需要先调用一次本接口。

```
35 void init_pages() {
36     for (int i = 0; i < 50; i++) {
37         memset(all_pages[i], 0, SIZEOFPAGE * sizeof(char));
38     }
39     memset(dirty, 0, 50 * sizeof(bool));
40     memset(file_index, -1, 50 * sizeof(int));
41     head_index = end_index = pageCount = 0;
42     file_ptr = NULL;
43 }
```

② `int* get_vector_by_id(int id, const char* filePath);`

这个接口通过向量的 `id` 和文件的路径两个参数来获得一个向量的内容，返回值为得到的向量的内容。注意该接口返回的指针在使用后记得进行释放。

```
136 int *return_array = (int*)malloc(MNIST_VEC_DIM * sizeof(int));
137 int temp_page_index, temp_slot_index;
138 if (filePath == "origin_data_mnist/") {
139     temp_page_index = id / MNIST_SLOT_NUM;
140     temp_slot_index = id % MNIST_SLOT_NUM;
141 }
142 else if (filePath == "preprocess_data_mnist/") {
143     temp_page_index = id / PRE_PROCESS_MNIST_SLOT_NUM;
144     temp_slot_index = id % PRE_PROCESS_MNIST_SLOT_NUM;
145 }
146 }
```

这个接口先调用 `get_a_page` 获得页的索引

```
148 int real_array_index = get_a_page(temp_page_index, filePath);
```

然后再在相应的内存帧中读取向量的内容。

```

170     for (int i = 0; i < MNIST_VEC_DIM; i++) {
171
172         if (filePath == "origin_data_mnist/") {
173             return_array[i] = (int)all_pages[real_array_index][MNIST_SLOT_NUM + 12 + ((temp_slot_index) * (MNIST_VEC_DIM + 4)) + 4 + i] & 0x000000ff;
174         }
175         if (filePath == "preprocess_data_mnist/") {
176             return_array[i] =
177                 (((int)all_pages[real_array_index][PRE_PROCESS_MNIST_SLOT_NUM + 12 + ((temp_slot_index) * (2 * MNIST_VEC_DIM + 4)) + 4 + 2 * i] & 0x000000ff))
178                 | (((int)all_pages[real_array_index][PRE_PROCESS_MNIST_SLOT_NUM + 12 + ((temp_slot_index) * (2 * MNIST_VEC_DIM + 4)) + 4 + 2 * i + 1]
179                     & 0x000000ff) << 8);
180         }
181     }

```

当然该接口只能用来处理 mnist 数据集，为了处理 glove 数据集，我们还需要设计一个能够返回 float\* 类型的 get\_vector\_by\_id\_glove(int id, const char\* filePath)。这个就不重复写在这里了。

③ bool set\_page(int arr\_index, int page\_index, char \*page\_content);

这个接口暂时还没有用到，这个接口是设计给 fileIO 使用的，但是有些小问题就没有使用，这个接口将一个数组直接赋值到内存缓冲区的一个帧里。

④ bool set\_vector\_by\_id(int id, int \*new\_vector, const char\* filePath);

这个接口可以更新一个向量，更新后不写回文件，而是将 dirty 数组相应位置置 1。参数分别为向量索引，要更新的内容和文件路径。返回值代表是否成功操作。但是目前暂时还没有用到。

⑤ bool write\_page(int array\_index, const char\* filePath);

这个接口将内存缓冲区的一个帧写回磁盘中。参数分别是要写回去的内存缓冲区的索引和文件路径。返回值代表是否成功操作。

```

45     // 把指定帧的页覆盖地写回到磁盘中去
46     bool write_page(int array_index, const char* filePath) {
47
48         // 如果写回，那么磁盘中的数据的一致性就得到保证了，设为0
49         dirty[array_index] = 0;
50         char name_temp[40];
51         sprintf(name_temp, "%s%d", filePath, file_index[array_index]);
52         file_ptr = NULL;
53         file_ptr = fopen(name_temp, "w");
54         if (file_ptr == NULL) {
55             printf("Error when writing file %s\n", name_temp);
56             return 0;
57         }
58         fwrite(all_pages[array_index], 1, SIZEOFPAGE, file_ptr);
59         fclose(file_ptr);
60         file_ptr = NULL;
61         return 1;
62     }

```

⑥ int get\_a\_page(int page\_index, const char\* filePath);

这个接口不直接使用，通过其他接口调用该函数来读取一个帧，两个参数是要获得的帧的索引，后面的是文件路径（因为多个数据集都要操作这个函数）。该函数先是在内存帧中寻找要获得的页。

```

97      // 检查指定页是否已经在内存中，若在，返回对应内存中的帧
98      bool exist_in_memory_flag = 0;
99      for (int i = 0; i < 50; i++) {
100          if (file_index[i] == page_index) {
101              exist_in_memory_flag = 1;
102              return i;
103          }
104      }

```

若不存在于内存中则调用 `read_disk_file` 来读取文件。返回的数字代表了该页在内存中的索引。

⑦ `bool read_disk_file(int page_index, int array_index, const char* filePath);`

这个接口作用于，在给定的文件路径下(filePath)，把给定文件路径下的指定页(page\_index)读到内存缓存区中的指定帧(array\_index)。

首先我们需要判断指定帧中是否已经存在页

```

68      // 如果指定帧已经存在页，那么指定帧的页覆盖地写回到磁盘中去，如果写失败，直接返回。
69      // 如果写成功的话，继续将新的指定页写到这个指定帧以覆盖这个帧中原来的页
70      if (file_index[array_index] != -1) {
71          // 如果存在的指定帧的也没有修改，则不覆盖地写回磁盘
72          if (dirty[array_index] == 1 && !write_page(array_index, filePath)) {
73              return 0;
74          }
75      }

```

然后把磁盘中的指定页读到内存缓存区的指定帧中。

```

76      char name_temp[40];
77      sprintf(name_temp, "%s%d", filePath, page_index);
78
79      file_ptr = NULL;
80      file_ptr = fopen(name_temp, "r");
81      if (file_ptr == NULL) {
82          printf("Error when reading file :%s\n", name_temp);
83          return 0;
84      }
85      fread(all_pages[array_index], sizeof PAGE, 1, file_ptr);
86      file_index[array_index] = page_index;

```

(注：在实际中直接操作到的接口只有①②⑦，通过调用接口，将文件操作与操作代码分离开，将所有的文件操作和内存缓存区中的帧的操作封装并进行抽象，一方面这是模拟实际商业数据库在文件存储方面的操作，二是方便后续的对这个功能的扩展和修改)

## 2.2.3 数据的预处理

## 2.3 处理查询



### 2.3.1 查询函数接口：

```
struct Pair* KNN(const int pointId, int k, int dim, const char* filePath)
```

#### 2.3.1.1 接口功能

输入要查询的点的 id，想要查的最近邻个数 k，以及数据集中元组的维数 dim，数据集所在文件路径；

```
202 // KNN求算磁盘中所有页中向量集合里距离给定点最近的K个点
203 struct Pair* KNN(const int pointId, int k, int dim, const char* filePath) {
```

然后将返回距离查询点最近的 K 个近邻组成的数组（struct Pair[]），每个元素都是一个 struct Pair，结构中包含 id 和到目标点的距离；

```
205     struct Pair* topK = (struct Pair*)calloc(k + 1, sizeof(struct Pair));
```

通过调用这个函数完成查询，遍历返回的数组就可以看到得到的结果

```
218     pageIndex = 0;
219
220     for (int i = 0; i < k + 1; ++i)
221     {
222         topK[i].id = -1;
223         topK[i].distance = MAX_NUM;
224     }
225
226     // 根据id寻找给定要查找的点
227     int* point = get_vector_by_id(pointId, filePath);
228     if (point == NULL) {
229         printf("point can not find: in KNN function\n");
230         return &topK[1];
231     }
232
233     // 求解KNN
234     for (int i = 0; i < totalFile / 50 + 1; ++i)
235         // for (int i = 0; i < 1; ++i)
236     {
237         KNN_ForPagesInMemory(point, topK, k, dim, filePath);
238     }
239
240     free(point);
241     return &topK[1];
```

#### 2.3.1.2 主接口实现

① 使用 get\_vector\_by\_id 接口得到目标点的信息，初始化 topK（距离最近的前 k 个点的数组）数组，将它们的 id 置为-1，距离置为 MAXNUM（100000000）；

② 从磁盘中读取 50 页内容到内存中，这 50 页内容使用内存中的 50 个帧存储（每个帧都是 64k byte）—— char [50][65536]，也即模拟一个 50 \* 64K byte 的内存缓冲区，将遍历内存缓存区中的所有向量并与目标点求算几何距离，将距离和 id 都存入 struct Pair ListForKNN[] 数组（存放了 50 页中所有点 id、到目标点的距离）；

③ 将这 ListForKNN 数组与 topK 数组中内容合并、排序

④ 将排序结果中的前 k 个重新写到 topK 中，如果已完成对所有磁盘页的遍历，则结束算法，返回 topK；否则继续从步骤 2 开始，读入还未读过的新的磁盘页；

[

★注：

上述步骤 3 中

① 合并：申请数组的时候为 ListForKNN 额外申请 k 个空间，然后将 topK 写入 ListForKNN 数组最前面（下标 0 到 k-1）；

② 排序：使用归并排序，将 ListForKNN、ListForKNN 的始末下标传入 mergeSort，mergeSort 将根据 ListForKNN[i].distance 排序，将 distance 小的排在前面，最后排序内容将在全局变量 ListForKNN 中；

]

### 2.3.1.3 功能接口

① void KNN\_ForPagesInMemory(const int \* point, struct Pair\* topK, int k, int dim, const char\* filePath)

```
244 void KNN_ForPagesInMemory_glove(const float* point, struct Pair* topK, int k, int dim, const char* filePath) {
245     // 初始化listForKNN
246     int totalSlot = 0;
247     totalSlot = GLOVE_SLOT_NUM;
248
249     for (int j = 0; j < k + 1; ++j)
250     {
251         listForKNN[j].id = topK[j].id;
252         listForKNN[j].distance = topK[j].distance;
253     }
254     for (int j = k + 1; j < 50 * totalSlot + k + 1; ++j)
255     {
256         listForKNN[j].id = -1;
257         listForKNN[j].distance = MAX_NUM;
258     }
259 }
```

这个函数即前述算法中循环的单步——2-4 步，它会把传入的 topK 和全局变量 ListForKNN 进行排序，将排序结果的前 K 个元素继续存入 topK 数组以进入下一个循环；

② int\* get\_vector\_by\_id(int id, const char\* filePath)

根据所需要的元素的 id 获取对应的向量元素

```

135 int* get_vector_by_id(int id, const char* filePath) {
136     int *return_array = (int*)malloc(MNIST_VEC_DIM * sizeof(int));
137     int temp_page_index, temp_slot_index;
138     if (filePath == "origin_data_mnist/") {
139         temp_page_index = id / MNIST_SLOT_NUM;
140         temp_slot_index = id % MNIST_SLOT_NUM;
141     }
142     else if (filePath == "preprocess_data_mnist/") {
143         temp_page_index = id / PRE_PROCESS_MNIST_SLOT_NUM;
144         temp_slot_index = id % PRE_PROCESS_MNIST_SLOT_NUM;
145     }

```

③ void MergeSort(struct Pair\* list, int begin, int end)

```

68 // 归并排序中的分治步骤
69 void MergeSort(struct Pair* list, int begin, int end) {
70     if (begin < end) {
71         int middle = (begin + end) / 2;
72         MergeSort(list, begin, middle);
73         MergeSort(list, middle + 1, end);
74         Merge(list, begin, middle, end);
75     }
76 }

```

这里我们采用归并排序，先将队伍不断切断细分（分治），然后使细分的那些队伍有序（72-73 行）之后将较短的有序的队伍进行归并（74 行）最终将得到完整的、有序的队伍

④ void Merge(struct Pair\* list, int begin, int middle, int end)

```

39 void Merge(struct Pair* list, int begin, int middle, int end) {
40     int i = 0;
41     int j = 0;
42     int k = 0;
43
44     for (i = begin, j = middle + 1, k = 0; k <= end - begin; k++)
45     {
46         if (i == middle + 1) {
47             tempPointList[k] = list[j++];
48             continue;
49         }
50         if (j == end + 1) {
51             tempPointList[k] = list[i++];
52             continue;
53         }
54         if (list[i].distance < list[j].distance) {
55             tempPointList[k] = list[i++];
56         }
57         else {
58             tempPointList[k] = list[j++];
59         }
60     }
61
62     for (int i = 0, j = begin; i <= end - begin; ++i, ++j)
63     {
64         list[j].id = tempPointList[i].id;
65         list[j].distance = tempPointList[i].distance;
66     }
67 }

```

这个函数是归并算法中的归并步骤，将两段分别有序的队伍合并成一段有序的队伍

具体做法是：如果两个队伍都没到队尾，那么比较当前元素大小（这个大小通过该点与查找点的几何距离衡量），将较小的元素放到最终队列，直到有一个队伍检查的队尾，此时将另一个队伍中的元素按顺序放入到最终队列。最后我们可以得到合并的有序的队伍

⑤ `double dist(const int* pointA, const int* pointB, int dim)`

```
18 // 求出两个向量之间的距离
19 double dist(const int* pointA, const int* pointB, int dim) {
20     double result = 0.0;
21     for (int i = 0; i < dim; ++i)
22     {
23         result += pow(pointA[i] - pointB[i], 2);
24         // printf("%lf %d\n", result, i);
25     }
26     return pow(result, 0.5);
27 }
```

这个函数就是求距离的函数（计算目标点和其他点之间的距离），即计算 `pointA` 和 `pointB` 的几何距离——求和  $(\text{pointA}[i] - \text{pointB}[i])^2$ ，然后开根号即得两点之间几何距离

### 2.3.1.3 查询实验

#### 2.3.1.3.1 实验要求

- ① 重复实验 5-10 次，每次查询 1000 个点，`k` 取 200
- ② 循环调用 `experiment_for_1000_times` 函数 5-10 次

#### 2.3.1.3.2 实验过程

接口函数将在 `for` 循环中生成 1000 次随机 `id`（每次都设置 `srand(time(NULL))`），以产生真正的随机数；然后调用 `KNN` 函数，传入之前随机生成的 `id`（维度为 784，`k` 为 200），查询前 `k` 个最近邻，并将每一次查询结果存入指定的 `file` 中（`resultx.txt`），其中 `x` 是第 `x` 实验；

最后我们可以在 `result.txt` 中看到第 `x` 组实验

通过修改传入 `experiment_for_1000_times` 的参数 `mode` 来决定是查询原始数据还是处理后的数据，`mode==0`，查询原始数据，`mode==1` 查询处理后数据

实验最终结果见相关文件（`resultx.txt` 文件）

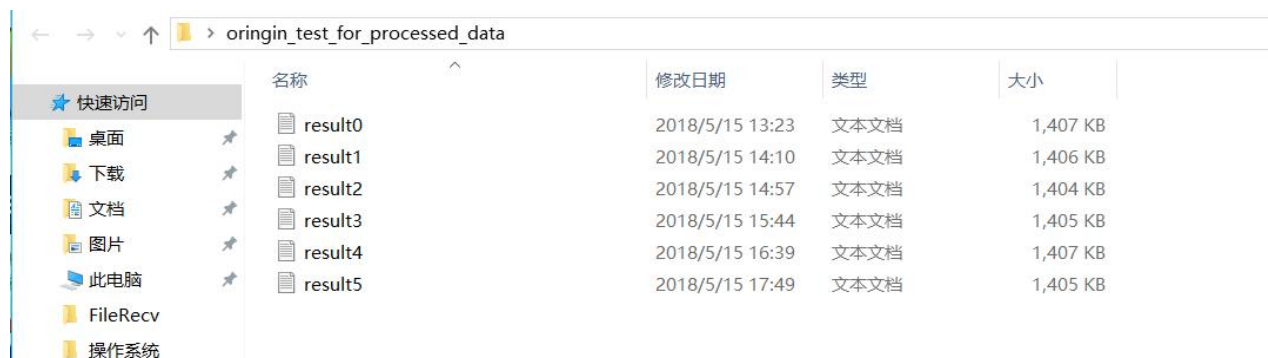


## Part3: 测试程序

### 3.1 编译运行

```
PS E:\桌面\数据库\大作业> .\a
the 0th point for search: 40045
Error when reading file :preprocess_data_mnist/1464
Error when reading file :preprocess_data_mnist/1465
Error when reading file :preprocess_data_mnist/1466
Error when reading file :preprocess_data_mnist/1467
Error when reading file :preprocess_data_mnist/1468
Error when reading file :preprocess_data_mnist/1469
Error when reading file :preprocess_data_mnist/1470
Error when reading file :preprocess_data_mnist/1471
Error when reading file :preprocess_data_mnist/1472
Error when reading file :preprocess_data_mnist/1473
Error when reading file :preprocess_data_mnist/1474
Error when reading file :preprocess_data_mnist/1475
Error when reading file :preprocess_data_mnist/1476
Error when reading file :preprocess_data_mnist/1477
Error when reading file :preprocess_data_mnist/1478
Error when reading file :preprocess_data_mnist/1479
Error when reading file :preprocess_data_mnist/1480
Error when reading file :preprocess_data_mnist/1481
Error when reading file :preprocess_data_mnist/1482
Error when reading file :preprocess_data_mnist/1483
Error when reading file :preprocess_data_mnist/1484
Error when reading file :preprocess_data_mnist/1485
Error when reading file :preprocess_data_mnist/1486
Error when reading file :preprocess_data_mnist/1487
Error when reading file :preprocess_data_mnist/1488
Error when reading file :preprocess_data_mnist/1489
Error when reading file :preprocess_data_mnist/1490
Error when reading file :preprocess_data_mnist/1491
Error when reading file :preprocess_data_mnist/1492
Error when reading file :preprocess_data_mnist/1493
Error when reading file :preprocess_data_mnist/1494
Error when reading file :preprocess_data_mnist/1495
Error when reading file :preprocess_data_mnist/1496
Error when reading file :preprocess_data_mnist/1497
Error when reading file :preprocess_data_mnist/1498
Error when reading file :preprocess_data_mnist/1499
the time for this search is : 2.571
the 1th point for search: 40058
```

### 3.2 运行结果



名称	修改日期	类型	大小
result0	2018/5/15 13:23	文本文档	1,407 KB
result1	2018/5/15 14:10	文本文档	1,406 KB
result2	2018/5/15 14:57	文本文档	1,404 KB
result3	2018/5/15 15:44	文本文档	1,405 KB
result4	2018/5/15 16:39	文本文档	1,407 KB
result5	2018/5/15 17:49	文本文档	1,405 KB

处理前:

```

the 997th point for search: 46952
30706->39758->9452->34582->33578->39788->9914->40180->41432->6730->22362->39800->15060->19074->42100->12652->64
->6146->38722->36712->8158->42402->25408->88->41620->32036->4106->36362->57494->57494->58408->41384->34246->473
the time for this search is : 2.407

the 998th point for search: 46967
649->58295->7759->41129->33209->829->35219->2913->2479->33207->53009->10917->47963->47045->56195->56195->58301->
->6567->20541->35575->44217->15591->302->16207->46323->59959->43281->39337->33975->20949->28249->10913->41351->1
the time for this search is : 2.367

the 999th point for search: 46979
25689->4583->6561->58021->58021->9413->2587->12775->44681->35723->44671->38735->39125->12881->12783->39061->276
->16587->35528->38657->38827->35891->32691->50337->22663->32923->4022->35727->50->50726->6750->16626->23439->559
the time for this search is : 2.36

the average time of search is : 2.49041

```

处理后：

```

the 997th point for search: 33272
26124->57198->17510->45338->51507->17008->42481->58392->58392->8140->30387->35791->52412->19316->52494
->51485->34106->6159->22944->1538->49802->48864->9016->15483->41666->40616->22121->35718->17332->40798
the time for this search is : 2.901

the 998th point for search: 33293
57241->10083->16903->23589->22445->35431->10071->743->19127->32273->59901->36351->781->577->33381->286
->2597->51303->18909->31297->859->10599->12573->22669->33345->58350->58350->5475->19811->43170->11729-
the time for this search is : 2.997

the 999th point for search: 33313
18374->36249->55159->28677->55207->13425->12515->36779->4611->39227->28753->30110->19641->54266->19315
539->20307->35084->24468->40487->1574->28761->14443->18755->41895->5669->12171->54517->58974->58974->3
the time for this search is : 2.884

the average time of search is : 2.8877

```

## Part4: 实验分析

### 4.1 性能评测：

A. 对原始数据的查询：每 1000 组查询平均用时 0.1212 秒

B. 对处理后的数据的查询：每 1000 组查询平均用时 0.134 秒

对比这两种数据查询时间，我们会发现，后一个时间要比前一个长一些，原因粗略分析如下：

① 因为数据经过预处理之后都变大了，所以数据计算稍微变大了

② 数据经过预处理之后磁盘页数变多了，排序次数也就变多了，所以用时变长了

通过和其他一些同学的交流，发现一个情况就是，我们小组的查询格外的慢（最开始我们的结果是每个查询大概 2-3s 左右，别人只要 0.5s 左右）

于是我们进行如下探究：

首先我们猜想可能是我们对内存缓冲区进行了模拟——开辟了大小 50\*64k byte 的大小，并且每次排序都要从磁盘页读入 50 页到内存缓冲区，再从内存缓冲区获取相应数

据（本来可以直接从磁盘页中读取数据而无需写到内存缓冲区，再从内存缓冲区读取数据），这样带来了额外的开销，所以导致了我们的实验要花上 4-5 倍的时间。

同时，我们在对读取数据进行单独的时间计算时候发现，单单对数据的读取就花费了大概 2 秒左右的时间，这也就说明了，我们的查询大部分时间花费在了读取上面（也即对缓冲区的模拟上），而非我们的排序算法出了问题。

但偶然的的机会我们把代码移植到 Linux 系统跑了一下，发现我们查询的时间变的格外快，只需零点几秒，这又使得我们召集所有组员一起讨论，讨论发现，是系统编译器的优化问题，并且我们发现，**添加-O3 优化编译后查询时间有一个数量级的提升**，马上，我们有开始对数据进行重新查询，最后我们确定是编译器的优化问题导致我们在 Windows 下查询时间的增加。

#### 4.2 不同算法的运行时间对比

为了解决时间过于冗余的问题，我们小组尝试使用在进行 knn 计算时候的时候不真正的进行排序而选择使用插入的方法，举例，就以  $k=200$  为例来说，用一个 200 的有序数组存前 200 个点和他们的距离，后面的点先算距离，如果距离大于数组最后一个点的距离，直接丢掉这个点，小于的话只需在 200 个数组做插入出来；事实证明，这样做确实是可以减少查询的时间，但是有个问题就是**这样的查询是不是会受到 k 值的影响**；为了进一步验证我们的想法，我们使用了  $k=2000$ ，这个数字，再次对 knn 进行计算，完成查询之后，结果显示这时候我们小组的原设计的程序进行查询花费的时间仍旧是稳定在 2.5s 左右，而与我们进行对比的这种插入查询算法时间已经差不多到达了 3s，差不多是原来的 3 倍，数据的对比也就客观体现了我们小组所设计的程序的稳定性。

#### 4.3 不同环境下的运行时间对比

以上的运行时间全是在 Windows 环境下得出的结果。我们后来在 Linux（虚拟机）下运行了实验。发现查询时间只需要 0.8s 左右。而在 Windows 环境下需要 2.1 秒。

```
can not find message in the slot:
can not find message in the slot:
can not find message in the slot:
can not find message in the slot:
can not find message in the slot:
can not find message in the slot:
can not find message in the slot:
0.817219
```

```
page index to: 750; the 15th sort:
totalTime1: 0.27
totalTime2: 1.818
查询所用时间为 2.177 秒
0.089
```

在 Linux 并且做出优化后的运行时间仅仅需要 0.1 秒



```

the 707th point for search: 13079
Error when reading file :origin_data_mnist/723
Error when reading file :origin_data_mnist/724
Error when reading file :origin_data_mnist/725
Error when reading file :origin_data_mnist/726
Error when reading file :origin_data_mnist/727
Error when reading file :origin_data_mnist/728
Error when reading file :origin_data_mnist/729
Error when reading file :origin_data_mnist/730
Error when reading file :origin_data_mnist/731
Error when reading file :origin_data_mnist/732
Error when reading file :origin_data_mnist/733
Error when reading file :origin_data_mnist/734
Error when reading file :origin_data_mnist/735
Error when reading file :origin_data_mnist/736
Error when reading file :origin_data_mnist/737
Error when reading file :origin_data_mnist/738
Error when reading file :origin_data_mnist/739
Error when reading file :origin_data_mnist/740
Error when reading file :origin_data_mnist/741
Error when reading file :origin_data_mnist/742
Error when reading file :origin_data_mnist/743
Error when reading file :origin_data_mnist/744
Error when reading file :origin_data_mnist/745
Error when reading file :origin_data_mnist/746
Error when reading file :origin_data_mnist/747
Error when reading file :origin_data_mnist/748
Error when reading file :origin_data_mnist/749
the time for this search is : 0.132003

```

## Part5: 表格填写

	查询所需时间(s)	mAP (以原始数据的KNN作为真正KNN)	mAP (以使用处理后数据的KNN作为真正KNN)
真正KNN (使用原始数据)	0.1212		
真正KNN (使用处理后数据)	0.134		

## Part6: 探索

在这个阶段，我们在完成基本实验任务的同时，还进行额外的探索，在建好数据集以后，我们分别在 Windows 和 Linux 下完成查询，发现两者的查询时间存在一个数量级的差别，这引发我们的思考，经过层层实验，先是 windows 下查询，发现 windows 的优化很差，查询用时较长，然后是简单在 windows 下的 bash 环境（也就是模拟 Linux 环境）下完成查询，时间在平均每个点零点几秒接近 1s，然后又在 Linux 查询（没有加 -O3 优化编译），查询时间为平均每个点 0.3s 左右，最后是在 Linux 下查询（加 -O3 优化编译），查询时间为平均每个点 0.1s 左右。可见，编译器优化对于我们实验的影响是有多么的大，在 Windows 下我们得出数据的时候我们甚至以为自己实验出问题了，偶

然机会我们在 Linux 跑了一下，发现这个优化问题并一直探索下去，感觉真的是实验的发现可能就是在那么一刹那的东西

## Part7: 项目分工

姓名	谢浩峰	王迎旭	彭伟林	吴聪	欧穗新
分工	设计了在内存中的 50 个帧的操作方式，设计并实现了几个操作的接口	实验报告的撰写，不同操作环境的程序测试与 debug	文件读写以，分页模式的设计，缓冲区的优化访问	页存储的设计，数据预处理，以接口函数的拓展	Try 文件的实现，KNN 的设计，以及排序算法的设计与实现