

数据库实验报告

题 目 数据库查询算法的实现

专 业 信息安全

学 号 1180300829

学 生 余 涛

指 导 教 师 史 建 焘

# 1实验目的

掌握多路归并排序算法，并用高级语言实现

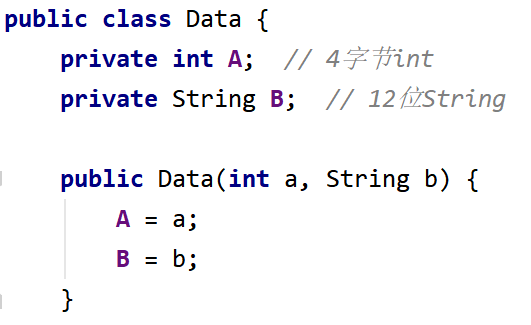
# 2实验环境

JAVA语言

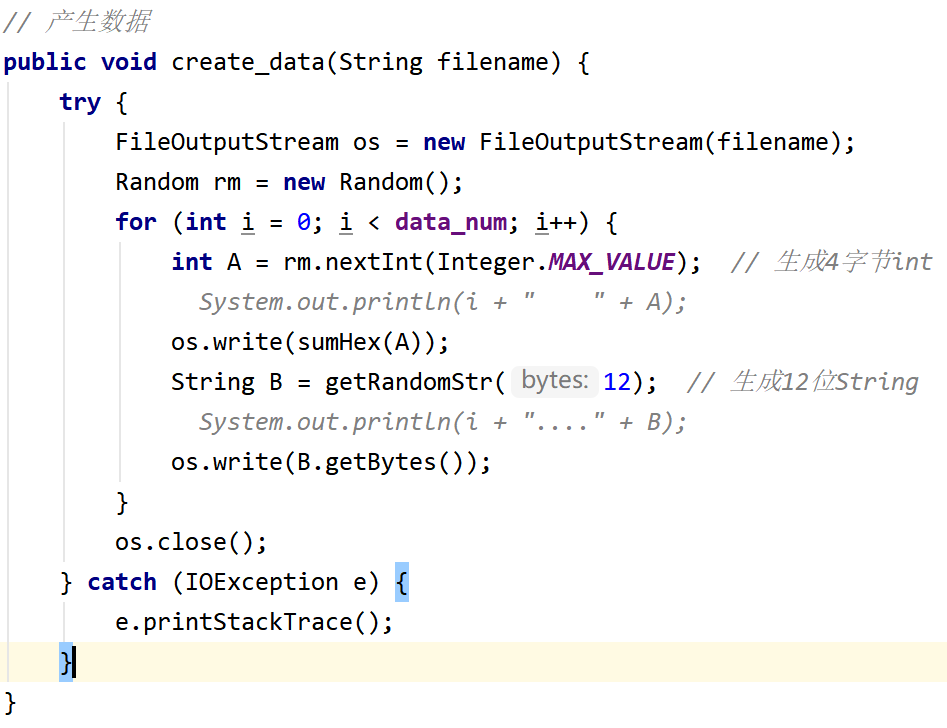
# 3 实验内容及要求

## 3.1随机生成具有1000000条记录的二进制文件，每条记录的长度为16字节

创建一个Data类储存数据：



创建数据代码如下:



## 3.2 分成小文件：

1MB内存 = 1024\*1024 B，而待排序文件大小=16B \* 1000000，若分为16个子集合，则每个子集合大小为1000\*1000B < 1MB。本机的内存页大小为4KB，1MB内存共有256个内存页，16<256，因此分为16个子集合满足两趟扫描算法的要求。

第一趟分成小文件如下：

1. // 划分小文件
2. **public** **void** split\_file(String filename) {
3. File myfile = **new** File(filename);
4. **if** (myfile.isFile() && myfile.exists()) {
5. **try** {
6. FileInputStream fi = **new** FileInputStream(myfile);
7. **for** (**int** i = 0; i < block\_number; i++) {  // 16个子文件
8. List<Data> templist = **new** ArrayList<>();
9. **for** (**int** j = 0; j < part\_file\_size / 16; j++) {  // 每个文件的记录总数
10. Data tempdata = get\_data(fi);
11. **if** (tempdata != **null**) {
12. templist.add(tempdata);
13. }
14. }
15. Collections.sort(templist, **new** Comparator<Data>() {  // 对子文件内按照A进行排序
16. @Override
17. **public** **int** compare(Data o1, Data o2) {
18. **return** o1.getA() - o2.getA();
19. }
20. });
21. FileOutputStream os = **new** FileOutputStream(prefix + i);
22. **for** (Data tempdata : templist) {  // 将templist中的所有记录读入一个小文件
23. os.write(sumHex(tempdata.getA()));
24. //                        System.out.println(tempdata.getA());
25. os.write(tempdata.getB().getBytes());
26. //                        System.out.println(tempdata.getB());
27. }
28. os.close();
29. //                    System.out.println("————————————————————————————————————————————————————————————");
30. }
31. fi.close();
32. } **catch** (IOException e) {
33. e.printStackTrace();
34. }
35. }
36. }

## 3.3 归并排序：

按照ppt的算法生成代码即可，代码内容如下：

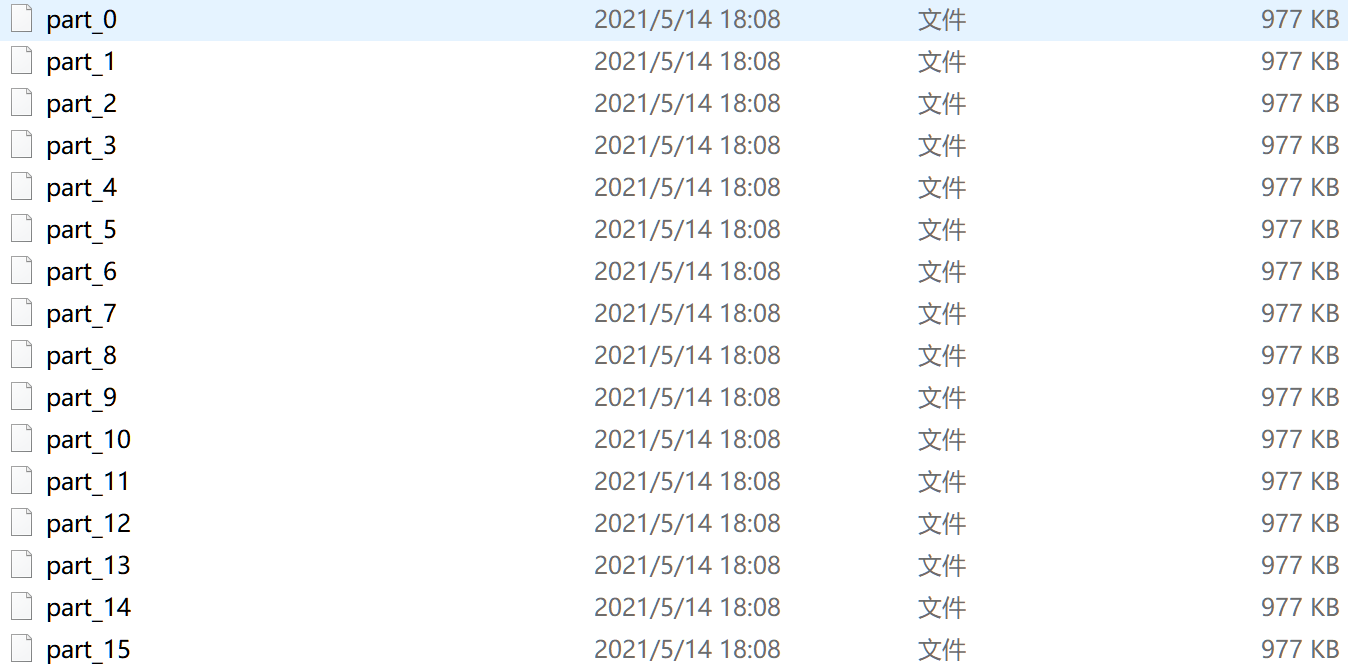
1. **public** **void** merge\_sort() {
2. **try** {
3. **int** num = 1;
4. //            int all = 1;
5. Data[][] M = **new** Data[block\_number][256];
6. Data[] M\_compare = **new** Data[block\_number];  // 比较缓冲区
7. Data[] M\_output = **new** Data[block\_number];  // 输出缓冲区
8. **int**[] Mi\_next = **new** **int**[block\_number];  // 记录Mi中元素移动的指针,即Mi中移到第几个元素了
9. **int**[] Si\_next = **new** **int**[block\_number];  // 记录Si中读到第几块了
10. **int**[] flag = **new** **int**[block\_number];  // 记录某一块是否读完
11. Data finished = **new** Data(-1, "finished");  // 记录已经读完的Data
12. FileOutputStream os = **new** FileOutputStream("src/merge\_sort\_result/result");  // 输出文件
14. **int** i;
15. **for** (i = 0; i < block\_number; i++) {
16. FileInputStream fi = **new** FileInputStream(prefix + i);
17. **for** (**int** j = 0; j < page\_size / 16; j++) {  // 读Si的第一块存到Mi中
18. M[i][j] = readData(fi);
19. }
20. M\_compare[i] = M[i][0];  // 第一个元素
21. fi.close();
22. }
23. **int** P\_output = 0;  // 输出指针起始位置
24. **while** (getmin(M\_compare) != -1) {  // 如果M\_compare有最小值
25. i = getmin(M\_compare);  // M\_compare最小值的位置
26. Data min\_i = M\_compare[i];  // M\_compare最小值
27. M\_output[P\_output] = min\_i;  // 将第i个元素存入M\_output的P\_output位置
28. P\_output++;  // P\_output指向下一个位置
29. Mi\_next[i] = Mi\_next[i] + 1;  // 记录Mi中的位置的指针指向下一个
30. **if** (min\_i.getA() == -1) {  // 为finished跳出循环
31. **break**;
32. }
33. **if** (P\_output == M\_output.length) {  // 如果P\_output指向结束位置，输出M\_output的内容
34. **for** (Data data : M\_output) {
35. System.out.println(num + ":" + data.getA());
36. //                        System.out.println(data.getA());
37. os.write(CreateData.intToBytes(data.getA()));
38. System.out.println(num + ":" + data.getB());
39. //                        System.out.println(data.getA());
40. os.write(data.getB().getBytes());
41. num++;
42. }
43. P\_output = 0;  // P\_output置于输出缓冲区起始位置
44. }
45. **if** (Mi\_next[i] < M[i].length) {  // 如果Mi有下一个元素，将Mi的下一个元素放入M\_compare的第i位置
46. M\_compare[i] = M[i][Mi\_next[i]];
47. } **else** {
48. FileInputStream fi = **new** FileInputStream(prefix + i);
49. fi.read(**new** **byte**[page\_size \* (Si\_next[i] + 1)]);  // 先读入已经读过的块，跳过
50. Si\_next[i] = Si\_next[i] + 1;
51. **if** (flag[i] == 0) {  // 如果还有下一块
52. **for** (**int** j = 0; j < page\_size / 16; j++) {  // 重新读入新的一块中所有内容到M
53. Data tempdata = readData(fi);
54. **if** (tempdata == **null**) {  // 没有下一块
55. flag[i] = 1;
56. M[i][j] = finished;
57. **break**;
58. }
59. M[i][j] = tempdata;
60. }
61. Mi\_next[i] = 0;
62. M\_compare[i] = M[i][0];
63. } **else** {
64. M\_compare[i] = finished;
65. }
66. fi.close();
67. }
68. }  // M\_compare中所有值均为Finished，没有最小值，算法结束
69. } **catch** (IOException e) {
70. e.printStackTrace();
71. }
72. }

## 3.4 实验结果

（1）分成的16个小文件：

1000000个数据总大小为1000000 \* 16 B, 分成16个小文件，则每个小文件大小为1000000B，为977KB，小文件大小满足要求。





用winhex随便打开一个小文件查看：

按照A属性在小文件内部完成了排序：



（2）最后的输出文件：

源文件大小如下：

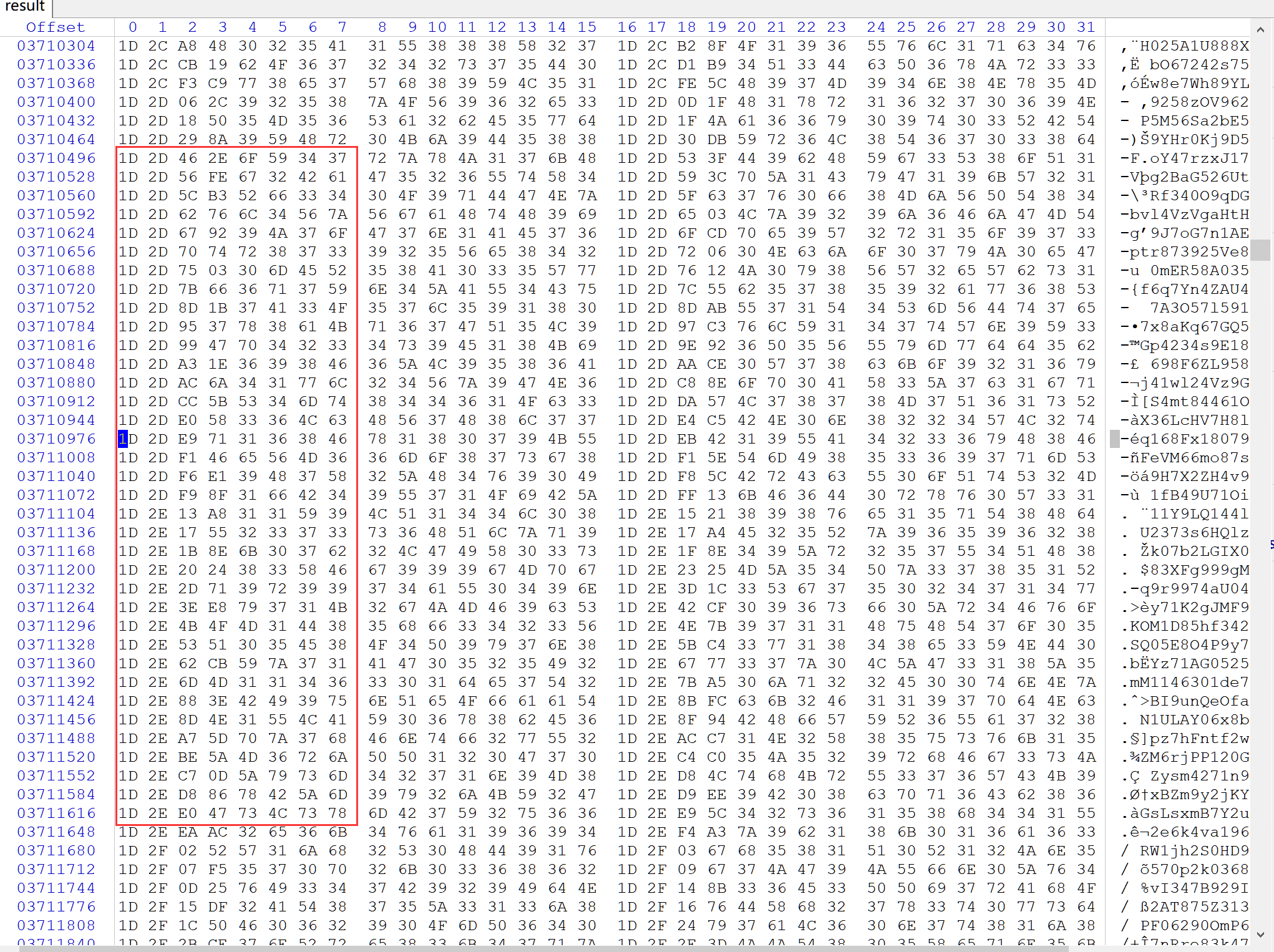


输出文件大小如下：



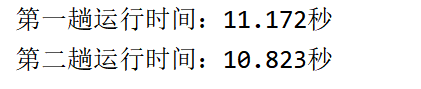


刚好1000000个数据，文件大小合适，然后查看内部A属性的顺序：



可见输出文件完成了排序。

（3）运行时间：



第一趟：使用java自带的comparator对子文件按照A属性排序，速度较快。

第二趟：由于每次读入文件的时候是一次性读入16个字节作为一个新的数据，相对于一次性读一个字节，效率较高。