09020328-王亮-数据结构实践报告3

External Sort: Merge Sort

1.问题描述

利用归并排序思想，实现外排。

2.算法思想

先归并分治下去，当分治的尺寸小于或等于input cache的capacity时读取并排序，形成初始小串，然后对于小串进行归并。归并时两个input cache区先读取数据，然后循环判断选取较小的input cache的数据归并移动到output cache，output cache满时写回，继续此过程直至两个input cache绑定数据文件均被清空，同理往下一直进行到所有数据被处理。

3.功能模块设计

之前是用数组模拟文件，现在尝试实际用文件进行排序。

Cache类：

封装cache需要提供的功能，读取，写入，判空，判满。

```
class Cache {
public:
  int ioCount = 0;
  int data[CACHE_SIZE];
  int capacity = CACHE_SIZE;
  fstream fread;
  fstream fwrite;
  string filename;
  string writeFilename;
  Cache() {
    for (int i = 0; i < capacity; i++) {
      data[i] = INVALID;
    }
  }
  int &operator[](int i) {
    return data[i];
  }
  void clear() {
    for (int i = 0; i < capacity; i++) {
      data[i] = INVALID;
    }
  }
  bool isEmpty() {
    for (int i = 0; i < capacity; i++) {
      if (data[i] != INVALID) {
        return false;
      }
    }
    return true;
  }
  bool isFull() {
```

```
    for (int i = 0; i < capacity; i++) {
      if (data[i] == INVALID) {
        return false;
      }
    }
    return true;
  }
  bool cacheRead(string filename) {
    ioCount++;
    if (!fread.is_open()) {
      this->filename = filename;
      fread.open(filename);
    }
    else if (this->filename != filename) {
      fread.close();
      fread.clear();
      fread.open(filename);
      this->filename = filename;
    }

    if (fread.eof()) {
      return READ_OVER;
    }
    for (int i = 0; i < capacity; i++) {
      fread >> data[i];
      if (fread.eof()) {
        break;
        return READ_OVER;
      }
    }
    return READ_LEFT;
  }
  void cacheWrite(string file) {
    ioCount++;
    if (!fwrite.is_open()) {
      fwrite.open(file, ios::out | ios::app);
      writeFilename = file;
    }
    else if (writeFilename != file) {
      fwrite.close();
      fwrite.clear();
      fwrite.open(file, ios::out | ios::app);
    }
    for (int i = 0; i < capacity && data[i] != INVALID; i++) {
      fwrite << data[i] << " ";
    }
  }
};
```

基本数据产生：

```
void dataGenerator() {
  fstream fio;
  srand(time(0));
  fio.open("dataSet.txt", ios::out);
  for (int i = 1; i <= DATA_NUM; i++) {
    fio << rand() % DATA_MAX + 1 << " ";
    if (i % 10 == 0) {
      fio << endl;
    }
  }
  fio.close();
}
```

归并核心逻辑：

归并中分两种情况，单边cache写回，两边cache取最小。如果output已满直接写回，如果input为空继续向文件中读取数据，如果两个input cache绑定文件数据均处理完毕，则归并结束。

```
while (true) {
      // only one side left
      if (inputCache[0][cache0Pointer] == INVALID && inputCache[1]
[cache1Pointer] != INVALID) {
        outputCache[outputPointer++] = inputCache[1][cache1Pointer];
        inputCache[1][cache1Pointer++] = INVALID;
      }
      else if (inputCache[1][cache1Pointer] == INVALID && inputCache[0]
[cache0Pointer] != INVALID) {
        outputCache[outputPointer++] = inputCache[0][cache0Pointer];
        inputCache[0][cache0Pointer++] = INVALID;
      }
      // compare and choose which one to move to the output
      else if (inputCache[0][cache0Pointer] <= inputCache[1][cache1Pointer]) {
        outputCache[outputPointer++] = inputCache[0][cache0Pointer];
        inputCache[0][cache0Pointer++] = INVALID;
      }
      else {
        outputCache[outputPointer++] = inputCache[1][cache1Pointer];
        inputCache[1][cache1Pointer++] = INVALID;
      }
      // write immediately when output is full
      if (outputCache.isFull()) {
        outwrite(newfile, outputPointer);
      }
      // read when empty
      if (inputCache[0].isEmpty()) {
        inputCache[0].cacheRead(fileFirst);
        cache0Pointer = 0;
      }
      if (inputCache[1].isEmpty()) {
        inputCache[1].cacheRead(fileSecond);
        cache1Pointer = 0;
      }
      // quit when merge is done
      if (inputCache[0].isEmpty() && inputCache[1].isEmpty()) {
        outwrite(newfile, outputPointer);
```

```
        return file;
      }
    }
```

4.测试结果与分析

由后两组数据可以看出，io数应可由总数据量及cache大小按一定关系表示出。

```
constexpr int DATA_NUM = 1000;
constexpr int CACHE_SIZE = 100;

cache_0: 54
cache_1: 65
output: 51
sum: 170
```

```
constexpr int DATA_NUM = 1000;
constexpr int CACHE_SIZE = 200;

cache_0: 26
cache_1: 25
output: 20
sum: 71
```

```
constexpr int DATA_NUM = 1500;
constexpr int CACHE_SIZE = 100;

cache_0: 59
cache_1: 63
output: 64
sum: 186
```

```
constexpr int DATA_NUM = 3000;
constexpr int CACHE_SIZE = 200;

cache_0: 53
cache_1: 69
output: 64
sum: 186
```

5.实验总结

在一定程度上感受到归并排序相对于其他排序在外排情形下适用性更佳，比如可以很方便地在保持总体思路不变的情况下，准备更多的cache用于多路归并，在多线程支持下进行并行处理，加快处理速度。亦或是对于多路数据来源的排序，完全是出于直觉的考量。

6.源代码

```
#include <bits/stdc++.h>
using namespace std;
constexpr int DATA_NUM = 1000;
constexpr int CACHE_SIZE = 100;
constexpr int CACHE_NUM = 2;
constexpr int DATA_MAX = 10000;
constexpr int INVALID = -1;
constexpr int READ_OVER = 1;
constexpr int READ_LEFT = 0;
class Cache {
public:
  int ioCount = 0;
  int data[CACHE_SIZE];
  int capacity = CACHE_SIZE;
  fstream fread;
  fstream fwrite;
  string filename;
  string writeFilename;
  Cache() {
    for (int i = 0; i < capacity; i++) {
      data[i] = INVALID;
    }
  }
  int &operator[](int i) {
    return data[i];
  }
  void clear() {
    for (int i = 0; i < capacity; i++) {
      data[i] = INVALID;
    }
  }
  bool isEmpty() {
    for (int i = 0; i < capacity; i++) {
      if (data[i] != INVALID) {
        return false;
      }
    }
    return true;
  }
  bool isFull() {
    for (int i = 0; i < capacity; i++) {
      if (data[i] == INVALID) {
        return false;
      }
    }
    return true;
  }
  bool cacheRead(string filename) {
    ioCount++;
    if (!fread.is_open()) {
      this->filename = filename;
      fread.open(filename);
    }
```

```
    else if (this->filename != filename) {
      fread.close();
      fread.clear();
      fread.open(filename);
      this->filename = filename;
    }

    if (fread.eof()) {
      return READ_OVER;
    }
    for (int i = 0; i < capacity; i++) {
      fread >> data[i];
      if (fread.eof()) {
        break;
        return READ_OVER;
      }
    }
    return READ_LEFT;
  }
  void cacheWrite(string file) {
    ioCount++;
    if (!fwrite.is_open()) {
      fwrite.open(file, ios::out | ios::app);
      writeFilename = file;
    }
    else if (writeFilename != file) {
      fwrite.close();
      fwrite.clear();
      fwrite.open(file, ios::out | ios::app);
    }
    for (int i = 0; i < capacity && data[i] != INVALID; i++) {
      fwrite << data[i] << " ";
    }
  }
};
void dataGenerator() {
  fstream fio;
  srand(time(0));
  fio.open("dataSet.txt", ios::out);
  for (int i = 1; i <= DATA_NUM; i++) {
    fio << rand() % DATA_MAX + 1 << " ";
    if (i % 10 == 0) {
      fio << endl;
    }
  }
  fio.close();
}
class CacheSet {
public:
  Cache inputCache[CACHE_NUM];
  Cache outputCache;
  fstream fio;
  int segmentNo = 0;
  CacheSet(string filename) {
    fio.open(filename, ios::out | ios::in);
```

```
  }
  void outwrite(string newfile, int &outputPointer) {
    outputCache.cacheWrite(newfile);
    outputCache.clear();
    outputPointer = 0;
  }

  void runGenerator(string sortname, int length) {
    ofstream fout(sortname);
    int temp[length];
    for (int i = 0; i < length; i++) {
      fio >> temp[i];
    }
    sort(temp, temp + length);
    for (int i = 0; i < length; i++) {
      fout << temp[i] << " ";
    }
    fout.close();
    fout.clear();
  }
  int mergeSort(int left = 0, int right = DATA_NUM - 1) {
    if (right - left + 1 <= CACHE_SIZE) {
      runGenerator("segment" + to_string(segmentNo) + ".txt", right - left + 1);
      return segmentNo++;
    }
    int mid = (left + right) >> 1;
    return merge(left, right, mergeSort(left, mid), mergeSort(mid + 1, right));
  }
  int merge(int left, int right, int first, int second) {
    string fileFirst = "segment" + to_string(first) + ".txt";
    string fileSecond = "segment" + to_string(second) + ".txt";
    int cache0Pointer = 0;
    int cache1Pointer = 0;
    int outputPointer = 0;
    int file = segmentNo;
    ++segmentNo;
    string newfile = "segment" + to_string(file) + ".txt";

    inputCache[0].cacheRead(fileFirst);
    inputCache[1].cacheRead(fileSecond);

    while (true) {
      // only one side left
      if (inputCache[0][cache0Pointer] == INVALID && inputCache[1]
[cache1Pointer] != INVALID) {
        outputCache[outputPointer++] = inputCache[1][cache1Pointer];
        inputCache[1][cache1Pointer++] = INVALID;
      }
      else if (inputCache[1][cache1Pointer] == INVALID && inputCache[0]
[cache0Pointer] != INVALID) {
        outputCache[outputPointer++] = inputCache[0][cache0Pointer];
        inputCache[0][cache0Pointer++] = INVALID;
      }
      // compare and choose which one to move to the output
      else if (inputCache[0][cache0Pointer] <= inputCache[1][cache1Pointer]) {
```

```
        outputCache[outputPointer++] = inputCache[0][cache0Pointer];
        inputCache[0][cache0Pointer++] = INVALID;
      }
      else {
        outputCache[outputPointer++] = inputCache[1][cache1Pointer];
        inputCache[1][cache1Pointer++] = INVALID;
      }
      // write immediately when output is full
      if (outputCache.isFull()) {
        outwrite(newfile, outputPointer);
      }
      // read when empty
      if (inputCache[0].isEmpty()) {
        inputCache[0].cacheRead(fileFirst);
        cache0Pointer = 0;
      }
      if (inputCache[1].isEmpty()) {
        inputCache[1].cacheRead(fileSecond);
        cache1Pointer = 0;
      }
      // quit when merge is done
      if (inputCache[0].isEmpty() && inputCache[1].isEmpty()) {
        outwrite(newfile, outputPointer);
        return file;
      }
    }
    return file;
  }
  void getIOCount() {
    int count = 0;
    // count for each
    for (int i = 0; i < CACHE_NUM; i++) {
      count += inputCache[i].ioCount;
    }
    count += outputCache.ioCount;
    cout << "cache_0: " << inputCache[0].ioCount << endl;
    cout << "cache_1: " << inputCache[1].ioCount << endl;
    cout << "output: " << outputCache.ioCount << endl;
    cout << "sum: " << count << endl;
  }
};
int main() {
  dataGenerator();
  CacheSet alpha("dataSet.txt");
  alpha.mergeSort();
  alpha.getIOCount();
}
```