Technical Report of Stock Analysis Project

Coauthors

Junjie Qiu, Qiang Hu

贡献比：50:50

1 问题描述

股票交易中涉及许多经典的数据处理问题，在交易中会产生大量的逐笔委托和逐笔成交数据。逐笔委托数据记录了投资者在股票市场上委托的**买入和卖出订单**的实时信息，它们会展示投资者正在以什么价格和数量来买入或卖出股票。逐笔成交数据记录了每一笔订单**成交或撤单**的实时信息，包括交易价格、交易数量、买卖方的账户信息以及交易 的时间等。在本问题中，我们关注在9:00至11:30、13:00至14:57的连续竞价时间所产生的数据。所谓连续竞价时间，也就是交易双方委托能进行实时匹配交易的时段。

在连续竞价时间段内，交易者可提交三种订单：限价单（OrderType=2）、市价单（OrderType=1）和本方最优订单（OrderType=U）。它们会按照一定的规则[1]进行撮合成交，成交和撤单的信息将会被记录到逐笔成交数据中。

市价单具有五档买卖的性质。五档买卖是指在交易所中，买卖双方各有五个不同的价格档位，分别称为买一、买二、买三、买四、买五和卖一、卖二、卖三、卖四、卖五。买一指的是买方愿意以最高的价格买入股票的数量，卖一指的是卖方愿意以最低的价格卖出股票的数量。买二、买三、买四、买五和卖二、卖三、卖四、卖五的价格逐渐递减或递增。在交易所中，买卖双方的订单会按照价格优先、时间优先的原则进行撮合成交。也就是说，如果买一的价格高于卖一的价格，那么就会成交。如果买一的价格低于卖一的价格，那么就不会成交。如果买一的价格等于卖一的价格，那么就会按照时间优先的原则进行撮合成交。

以下是五档买卖的流程图

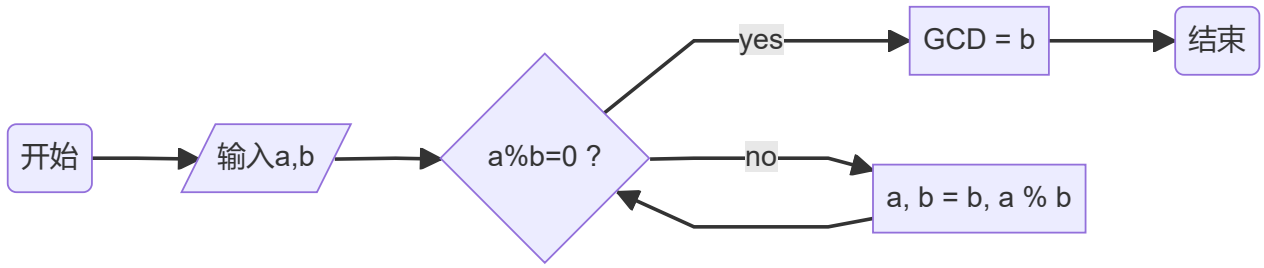

图 1 五档买卖流程图

但在实际记录中，市价单的OrderType均记录为1，缺乏更具细粒度的区分。我们希望确定每笔市价单的最低成交档位并记录，为了达成这个目的，我们需要调用逐笔委托和逐笔成交记录来对市价单成交挡位进行恢复工作。

2 任务理解和难点分析

2.1 任务理解

由于股票交易信息记录不包括市价单更细化的区分，但在实际交易过程中市价单的成交挡位对交易双方都非常重要，我们需要一个方法能够利用逐笔委托和逐笔成交信息来还原这个信息。而本任务考察的重点是利用MapReduce的特性，在Mapper中通过委托索引将逐笔委托和逐笔成交信息联系起来；在Reducer中结合委托和成交信息输出一个更细化的记录。

2.2 难点分析

本Project的主要难点有：

- 在Mapper中分别处理逐笔委托和逐笔成交数据，并在Reducer中将它们联系起来；
- 在Reducer中对不同种类的订单和委托单进行处理，输出更细化的记录；
- 在最后的排序中，对于同一时间戳的非撤单记录，需要按照委托索引进行排序；同时保证撤单记录能按撤单时间戳进行排序；
- 结合效率需求，对程序进行profile，找到效率瓶颈并进行优化。

3 技术方案

权衡了多种方案后，我们最终决定采用以下技术方案：

- 在Mapper中，将逐笔委托和逐笔成交数据分别处理：对于逐笔委托，输出委托索引和委托信息分别作为key和value；对于逐笔成交，输出买卖方委托索引和成交信息分别作为key和value，其中买卖方委托索引是指买方委托索引或卖方委托索引。同时，在Mapper中，我们还需对Order和Trade进行筛选，以满足题目要求。
- 在Reducer中，对于同一委托索引的逐笔委托和逐笔成交信息，将它们通过Mapper输出的key联系起来。对于不同类型的订单和委托单，我们采用不同的处理方式：
  - 对于限价单，我们直接输出委托信息；
  - 对于本方最优订单，我们直接输出委托信息，且本方最优订单不用输出价格；
  - 对于市价单，我们需要在Reducer中对逐笔成交信息进行处理，找到最低成交档位并输出。
- 在最后的排序中，我们需要对同一时间戳的非撤单记录按照委托索引进行排序；同时保证撤单记录能按撤单时间戳进行排序。为了达成这个目的，在TradeMapper中，我们需要加上成交索引`ApplSeqNum`字段，它是一个递增的整数，可以用来对同一时间戳的非撤单记录按撤单时间进行排序。

3.1 整体技术框架

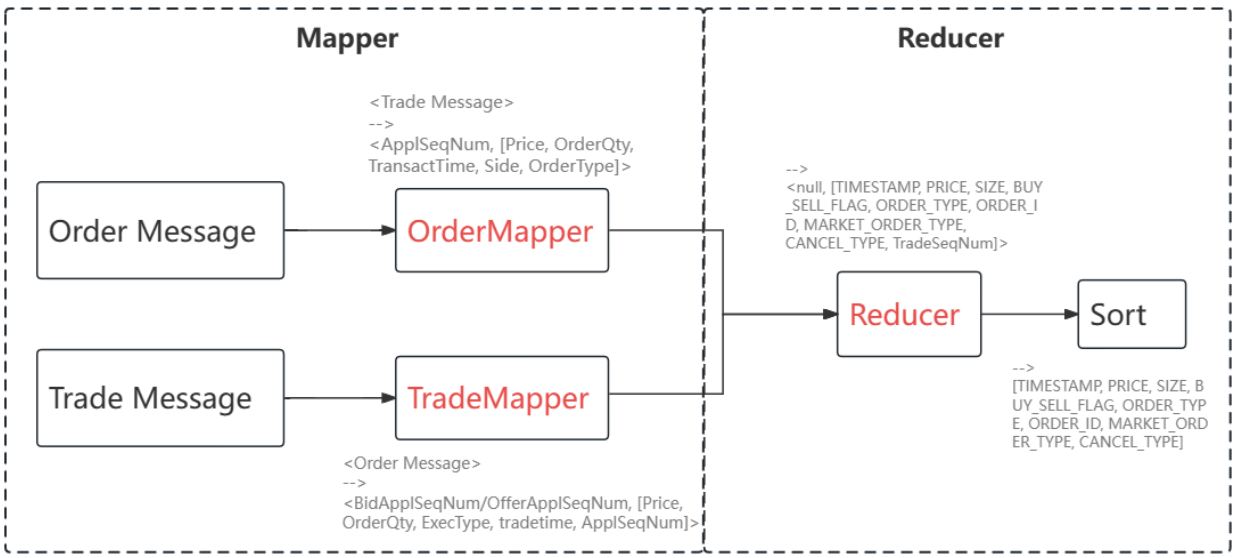

图 2 技术框架流程图

3.2 代码结构化设计思路

Project `src` 目录树结构：

```
├──main
│   ├──java
│   │   ├──driver
│   │   ├──mapper
│   │   └──reducer
│   └──resources
└──test
    └──java
```

我们对每一个用到的方法和重要的变量都撰写了文档，方便了开发和阅读，例如：

```
/**
 * @param t 转换前的时间
 * @return 转换后的时间
 */
protected String tConvert(String t)
```

```
/**
* 原时间格式
 */
protected SimpleDateFormat inputFormat = new SimpleDateFormat("yyyyMMddHHmmssSSS");
/**
 * 转换后的时间格式
 */
protected SimpleDateFormat outputFormat = new SimpleDateFormat("yyyy-MM-dd HH:mm:ss.SSS000");
```

3.3 代码细节

- Mapper中数据的筛选策略：

```
protected boolean validate(String securityID, String transactTime) {
    return securityID.equals("000001") && transactTime.compareTo("20190102093000000") >= 0 &&
 (transactTime.compareTo("20190102113100000") < 0 || transactTime.compareTo("20190102130000000") >= 0
) && transactTime.compareTo("20190102145700000") < 0;
}
```

- Mapper中时间戳比较大小的效率分析：

经过我们的测试，对于时间戳，直接使String比较比转换成Long后再比较有显著的效率提升，因此我们采用了直接比较String的方式。

以下为Profiler的测试结果示意图：

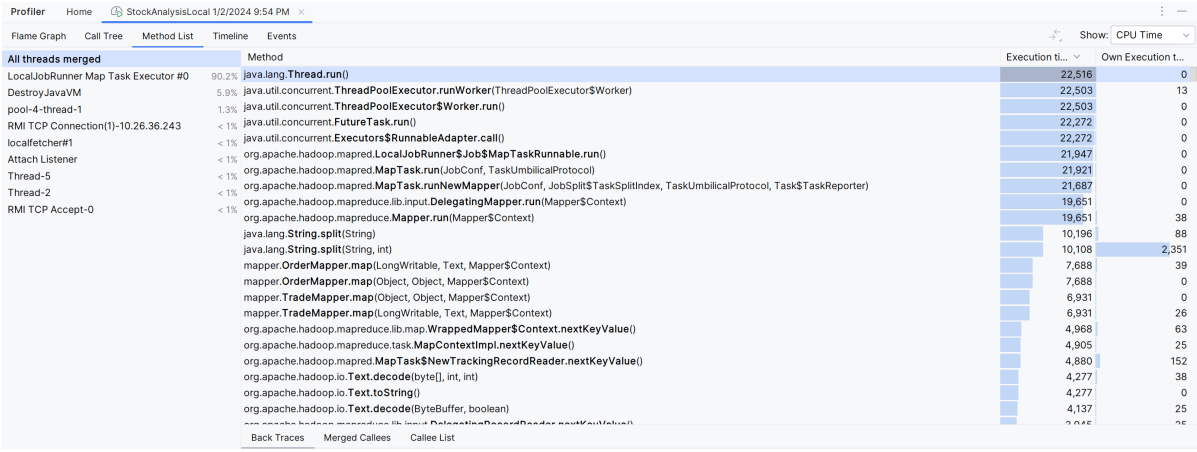

图 3 Profiler 示意图

通过对Profiler进行分析，我们可以发现，直接比较字符串程序本地电脑总共耗时41s；而转换为Long再比较程序本地电脑耗时45s。因此 ，我们直接比较字符串：

- 字符串连接效率分析：

经过我们的测试，发现在Mapper中，直接使用字符串连接的效率和使用StringBuilder的效率相差不大，因此我们采用了字符串连接的方式。

以下为两种实现方式：

1. 直接连接字符串，方法总共耗时121ms

```
v = tConvert(order[2]) + "," + price + "," + order[1] + "," + order[3] + "," + order[4] + "," + key + ",," + 2;
context.write(null, new Text(v));
```

2. 使用StringBuilder连接字符串，方法总共耗时120ms

```
sb.append(order[2]).append(",").appned(price).append(",").append(order[1]).append(",").append(order[3]).appned(",").append(order[4]).append(",").append(key).append(",,").append(2);
context.write(null, sb.toString());
```

- 对市价单价位的恢复：

```
// 判断传入value是order还是trade
for (Text value : values) {
    String[] split = value.toString().split(",");
    if (!split[4].equals("b") && !split[4].equals("s")) {
        // v: Price, OrderQty, TransactTime, Side, OrderType
        order = split;
    } else {
        // v: Price, TradeQty, ExecType, tradeTime, b or s
        if (split[2].equals("F")) {
            priceSet.add(split[0]);
        } else {
            cancelList.add(split);
        }
    }
}
```

为了方便记录价位，我们使用了一个`HashSet`来记录所有成交价位。同时，为了减少因为对象创建所带来的损耗，我们将`priceSet`和`cancelList`均设置为成员变量，方便重复调用。

```
protected ArrayList<String[]> cancelList = new ArrayList<>();
protected HashSet<String> priceSet = new HashSet<>();
```

- 排序逻辑：

```
// 重写排序规则
lines.sort((o1, o2) -> {
   // 字段分割
   String[] o1s = o1.split(",");
   String[] o2s = o2.split(",");
   if (o1[6].equls("2") && o2[6].equls("2")) {
       // 如果两个都非撤单
       if (!o1s[0].equals(o2s[0])) {
           // 按照时间排序
           return o1s[0].compareTo(o2s[0]);
       } else {
           // 按照订单号排序
           return Integer.parseInt(o1s[5]) - Integer.parseInt(o2s[5]);
       }
   } else if (o1[6].equls("2") && o2[6].equls("1")) {
       // 先处理非撤单
       return 1;
   } else if (o1[6].equls("1") && o2[6].equls("2")) {
       // 先处理非撤单
       return -1;
   } else {
       // 如果两个都是撤单
       if (!o1s[0].equals(o2s[0])) {
           // 按时间排序
           return o1s[0].compareTo(o2s[0]);
       } else {
           // 按撤单顺序排序
           return Integer.parseInt(o1s[7]) - Integer.parseInt(o2s[7]);
   }

});
```

排序规则：
（1）不同时间，时间小的排前面
（2）相同时间，先排订单，再排撤单
（2）订单按订单号进行排序，撤单按交易先后顺序排序

在以上排序规则的逻辑下，我们可以保证同一时间戳的非撤单记录按照`委托索引`进行排序；同时保证撤单记录能按撤单时间先后进行排序。同时由于MapReduce输出的结果数据量不大，使用新的MapReduce任务进行排序的效率较低，我们使用快速排序的方式进行排序。

- 排序阶段的文件读入设计

```java
// 获取HDFS的文件系统对象
FileSystem fs = FileSystem.get(conf);

// 读取文件
Path pt = new Path(args[2], "part-r-00000");
// BufferedReader
BufferedReader br = new BufferedReader(new InputStreamReader(fs.open(pt)));
ArrayList<String> lines = new ArrayList<>();
// 读取，直到下一行不存在为止
do {
    lines.add(br.readLine());
} while (br.ready());
```

由于我们需要在Docker环境中运行，文件都会由HDFS管理，故我们需要获得**HDFS的文件系统对象**，通过文件系统对象获得文件的实际路径。

- 排序阶段的文件输出设计

```java
// 输出表主体到HDFS
Path pt2 = new Path(args[2], "sorted.csv");
// BufferedWriter
BufferedWriter bw = new BufferedWriter(new OutputStreamWriter(fs.create(pt2, true)));
// 写出每行
for (String line: lines) {
bw.write(line);
bw.newLine();
}
// 刷出缓存
bw.flush();
```

由于BufferedWriter可能出现缓存中的数据尚未输出程序就结束的问题，我们需要在最后使用`flush()`方法来强制写出缓存。

3.4 版本控制和代码开源

在完成本Project的过程中，我们使用了小粒度高数量的版本管理模式，为我们稳健的程序构建提供了很好的基础。

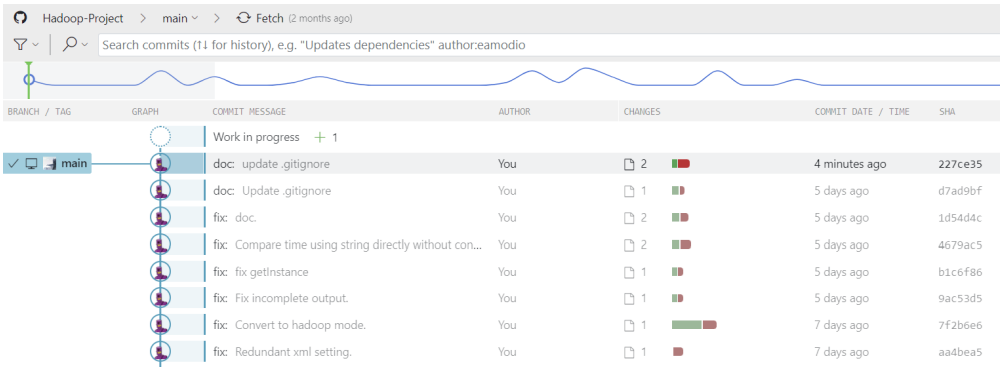

图 4 GitLens 插件截图

以下是GitHub自动生成的总结：

> Excluding merges, **1 author** has pushed **9 commits** to main and **9 commits** to all branches. On main, **11 files** have changed and there have been **105 additions** and **115 deletions**.

整个项目可以通过开源仓库GitHub/Hadoop-Project进行复现。

4 程序运行和测试结果

4.1 运行指令

- 进入jar包所在目录

```
cd ~/project
```

- 删除可能存在的运行结果

```
hdfs dfs -ls /
hdfs dfs -rm -r /proj_output
```

- 运行程序，结果输出至`Sorted.csv`

```
hadoop jar Hadoop-Project-1.0-SNAPSHOT.jar driver.StockAnalysis /data/order /data/trade
/proj_output
```

- 检验结果

```
hdfs dfs -head /proj_output/sorted.csv
```

4.2 测试结果

Docker环境测试结果：34s

csv-comparison-tool测试得分：100

1. 详见Project题目.pdf文件 ↩