

# 股票市场的主力资金流向计算 Report

2024 Fall 分布式存储与并行计算 Project 报告：第三组

姓名	学号	分工	贡献率
欧炜娟	12212252	问题解构, Driver、Reducer框架设计, 运行优化, 展示+ppt	50%
李怡萱	12212959	问题解构, Mapper框架设计, 结果可视化, 报告撰写	50%

## 股票市场的主力资金流向计算 Report

### 1. 问题描述

### 2. 问题理解与分析

#### 2.1 数据有效字段筛选

#### 2.2 计算流程安排

#### 2.3 难点分析

### 3. 整体技术方案

#### 3.1 数据读入与处理

##### 3.1.1 数据读取

##### 3.1.2 数据筛选

##### 3.1.3 主动买卖单判断

#### 3.2 时间相关处理

##### 3.2.1 时间窗口ID划分

##### 3.2.2 时间区间分配

#### 3.3 成交量、成交额、流通盘占比计算

#### 3.4 主力资金分类、统计与输出

##### 3.4.1 成交单分类标准

##### 3.4.2 主力资金流动计算公式

##### 3.4.3 最终输出

### 4. 代码模块化设计思路

#### 4.1 MapReduce计算模块化设计

##### 4.1.1 Driver 设计

##### 4.1.2 Mapper 设计

##### 4.1.3 Reducer 设计

#### 4.2 可视化代码模块设计

##### 4.2.1 `csvToHtml.java`

##### 4.2.2 `visualization.html`

##### 4.2.3 `style.css`

##### 4.2.4 可视化模块化优势

### 5. 结果分析

#### 5.1 结果准确性验证与可视化展示

##### 5.1.1 数据准确性

##### 5.1.2 可视化展示

#### 5.2 性能分析

##### 5.2.1 `SecurityID=000001, TIME_WINDOW=10` 情况举例分析

##### 1. 运行时间分析

##### 2. 占用内存分析

##### 5. 结论

## 1. 问题描述

在股市中，主力资金意指集中了大量资金的交易者（机构），具有主导股票涨跌的能力。识别主力资金流向对投资者决策至关重要。然而，交易所未提供实时资金流向数据，且无法直接获取交易者ID。因此，如何估算主力资金流向成为目前市场上的一个难题。

本项目使用深交所**Level-2数据**，包括某一天时间下的逐笔委托和逐笔成交数据，记录了所有买卖订单的详细信息。通过读取已有的数据，设计方法来区分主动买单和卖单，计算成交量和成交额，再根据成交单类型（大单、超大单）计算主力资金流向。

项目中，我们需要采取以下方式估算主力资金：

- 主力流入 = 超大买单金额 + 大买单金额
- 主力流出 = 超大卖单金额 + 大卖单金额
- 主力净流入 = 主力流入 - 主力流出

然后，输出包括19个指标，包含每个时间窗口内的主力资金流入、流出、净流入等指标，如超大买卖单成交量和成交额。另外，我们还需要实现前端网页的可视化展示，并进行准确性和速度测试。

## 2. 问题理解与分析

### 2.1 数据有效字段筛选

首先，数据筛选分析方面，我们需要提取trade表和order表的相关字段，来进行后续的计算。

先初步结合流程图，筛选逐笔成交、逐笔委托的数据表的有效字段如下：

- 逐笔成交数据表：`ChannelNo`, `ApplSeqNum`, `SecurityID`, `BidApplSeqNum`,  
`OfferApplSeqNum`, `Price`, `TradeQty`, `ExecType`, `tradetime`
- 逐笔委托数据表：`ChannelNo`, `ApplSeqNum`, `TransactTime`

其中，我们可以依靠`ChannelNo`, `ApplSeqNum`这两个主键字段，将两个表的数据进行结合和串联，并依据逐笔成交数据中的`BidApplSeqNum`和`OfferApplSeqNum`在委托索引表里查到委托时间`TransactTime`来判断主动单类型。

不过，上述的流程过于繁杂，而稍微再进行进一步分析，可以发现上述字段中的一些隐藏的信息：

- `ApplSeqNum`会随着时间而增加；
- Bid、Offer双方的`ApplSeqNum`在股票ID一定的情况下都是唯一的。

所以，我们可以仅利用逐笔成交数据表中的`BidApplSeqNum`和`OfferApplSeqNum`字段就可以判断主动单，无需使用逐笔委托数据表中的数据。

据此，我们优化精简了字段的筛选，可以将需要筛选的字段总结如下：

- 逐笔成交数据表：`SecurityID`, `BidApplSeqNum`, `OfferApplSeqNum`, `Price`, `TradeQty`, `ExecType`, `tradetime`

## 2.2 计算流程安排

在本问题中，我们需要分析股市中的主力资金流向数据，计算特定时间范围内的主力净流入、主力流入和主力流出等指标。我们将整体的计算流程安排为以下四个部分：

- 首先，根据我们更新后的数据预处理方案，我们可以在Mapper计算出满足筛选条件的每条交易数据的交易时间窗口、主动单类型，再将数据传输给Reducer；
- 接着，通过Reducer聚合相同时间窗口的数据，计算当前时间区间，再累加合并每个主动委托索引对应的成交量和成交额，记录买卖类型；
- 然后，根据判断超大额、大额、中额、小额买卖单的三个指标，判断成交单类别；
- 最后，根据主力数据的计算方式，计算主力流入、主力流出、主力净流入的数据，并进行标准化输出。

通过以上方式，我们可以通过分布式计算来准确地计算出给定股票的主力资金流向数据。

## 2.3 难点分析

在完成该项目的过程中，我们也遇到了一些难点，对此进行了总结并有针对性地进行分析：

- 读入数据处理**
  - 有效字段筛选：**在进行主力数据的计算之前，我们需要根据题目要求筛选出需要的参数。原本给定的数据集中，逐笔成交表（trade表）包括16个字段，逐笔委托表（order表）包括20个字段，我们需要根据题目的要求来确定计算方式，从而筛选出最有效的字段，以减少系统读写成本。
  - 主动单判断：**根据流程图，在给定股票ID `SecurityID`（同时已知对应频道 `ChannelNum`）的情况下，可以根据逐笔成交数据表的 `BidApplSeqNum` 和 `OfferApplSeqNum` 唯一确定逐笔委托数据表的两条买卖委托数据，通过比较其二的委托时间 `TransactTime` 的大小，来确定主动单。对于委托时间相同的数据，采用比较索引先后的方式来进行主动单判断。但是，此方式的整体流程需要涉及到两张超过1GB的表的处理，还需大量的查找与筛选工作，流程过于繁琐，耗时很长，需要优化提升。
- 任务性能优化**
  - 原始给定的数据集一共5.05GB，数据量庞大。即便根据最新的处理策略简化了运行流程，但是读入处理2-3GB的数据仍然要耗费大量时间，在进行Job Chain工作流程时，各个的Job的加载、分配也需要很长时间。所以，需要不断思考优化处理的策略，从而得以在处理大规模数据时提高MapReduce的运行效率。
- 复杂指标计算**

复杂指标计算是主力资金流向分析中的核心环节，其难点在于如何高效且准确地对大规模订单数据进行聚合、分类和统计。在同一时间窗口内，可能会出现多个主动委托索引相同的订单，为了确保数据的完整性和准确性，需要对这些订单的成交量和成交额进行合并。

  - 为此，我们设计了一种基于 `HashMap` 的聚合方法，将主动委托索引作为键，对应的值为包含累计成交量、成交额和买卖类型的数组。这样，通过一次遍历即可高效完成数据的聚合操作，避免了传统多次查找和重复计算的低效问题。同时也为后面其他指标的计算做铺垫。

- 聚合完成后，还需要对订单进行单类型的分类。这一过程基于成交量、成交额和流通盘占比等多维标准。分类的难点在于需要同时兼顾多种指标，并确保标准的合理性和适应性。我们采用了优先判断的方式，即优先满足超大单的条件，再依次判断是否为大单、中单或小单。这样的分层分类逻辑能够确保结果的准确性。
- 在数据统计阶段，不同类型的买单和卖单需要分别计算成交量和成交额。为提升计算效率，我们设计了数组结构来存储统计数据，每种单子类型对应数组的不同索引位置。分类完成后，直接根据分类结果更新对应的数组索引值，从而快速完成统计操作。这种设计避免了复杂数据结构带来的性能损耗，同时提高了系统的扩展性。

### ● 时间窗口计算

- 根据题目要求，我们需要根据给定的时间窗口参数，计算每一条成交数据的成交时间所对应的时间窗口。而由于原本时间数据以长数字形式输入，所以我们制定策略将其转化为时间进行比较，确定了对应的时间窗口后再创建函数来输出标准格式的时间区间数据。

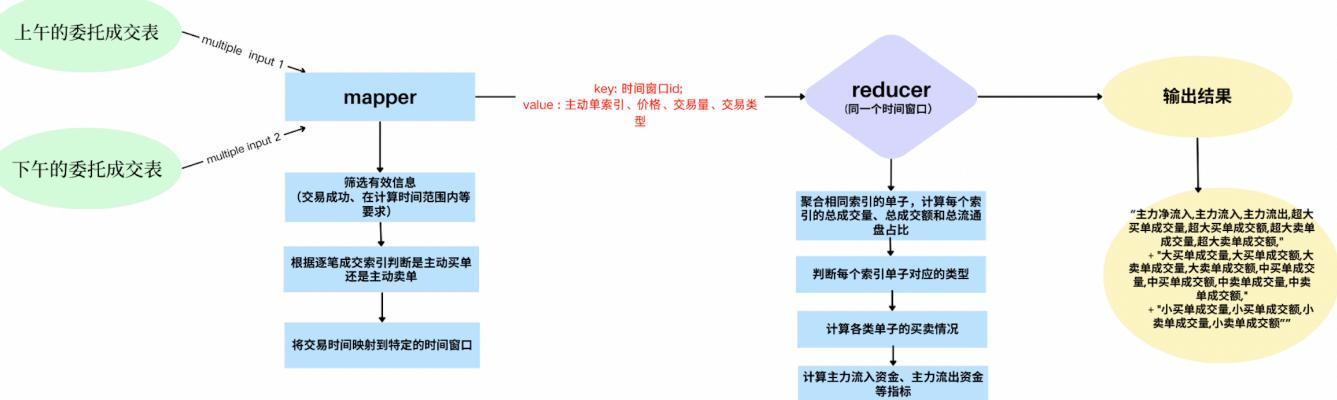
### ● 准确性保证

- 留意到在有些时间段，可能没有交易产生（如每日14:57-15:00的收盘集合竞价时间中，14:58-14:59就没有相关的交易数据），为了维护输出数据表的完整性，我们需要在聚合阶段判断出未产生交易的时间窗口，并补充相应的数据。

在下一个模块中，我们将更加详细地阐述我们的计算流程细节，描述我们针对以上难点来制定的解决方案。

## 3. 整体技术方案

接下来，我们绘制整体工作流程图，将设计的数据输入、预处理、指标计算、输出结果等步骤予以展示：



下面，我们将整体的工作流程按照需要执行的任务类别划分成数据读入与处理；时间相关处理；成交量、成交额、流通盘占比计算；主力资金分类、统计与输出这四个模块，一一详细阐释我们的整体技术方案。

### 3.1 数据读入与处理

#### 3.1.1 数据读取

首先，在数据读取方面，我们在 `FinalDriver` 类中，使用 `MultipleInputs` 来指定多个输入文件路径，也就是将早上和下午的逐笔成交数据表 `am_hq_trade_spot.txt` 和 `pm_hq_trade_spot.txt` 并行通过Mapper进行数据读入：

```
1 | MultipleInputs.addInputPath(job, new Path(TRADE_INPUT_PATH1), TextInputFormat.class,
2 | FinalMapper.class);
3 | MultipleInputs.addInputPath(job, new Path(TRADE_INPUT_PATH2), TextInputFormat.class,
4 | FinalMapper.class);
```

通过 `MultipleInputs` 方法，我们将两个逐笔成交数据表指定通过同一个 `FinalMapper` 类进行运算。在这种情况下，**Hadoop** 会自动并行处理两个输入文件，将每个输入文件分成多个逻辑分片（splits），然后分配给多个 Map 任务同时运行，从而实现并行处理，加速数据读取与处理的过程。

### 3.1.2 数据筛选

在 `FinalMapper` 类中的 `map()` 方法中，我们首先根据题目给定的条件和后续所需要的计算方式来筛选数据，仅处理符合特定条件的记录：

- `ExecType = F` 表示执行类型为“F”（有效交易记录）
- `SecurityID = 000001` 表示交易的证券代码为 `000001`（特定的股票代码）
- 交易时间需在有效的交易时段内（早上9:30到11:30，下午13:00到15:00）

筛选条件代码如下：

```
1 | // 处理符合条件的记录： 成交类型为F， 股票代码为000001。
2 | if ("F".equals(execType) && "000001".equals(securityID)) {
3 |     if (isWithinTradingTime(tradeTime)) {
4 |         // 进行后续的时间窗口计算、主动买卖单判断，并写入结果
5 |         ...
6 |     }
7 | }
```

### 3.1.3 主动买卖单判断

为了优化流程图中繁琐的主动单判断流程，我们观察到成交索引 `ApplSeqNum` 参数随着时间增长而逐渐递增，而且同一只股票的索引在同一天内不会出现重复情况，所以我们可以直接通过逐笔成交数据表，比较同一条交易数据的买方成交索引 `BidApplSeqNum` 和卖方委托索引 `OfferApplSeqNum` 的相对大小，来判断该条交易数据的主动单类型。所以，只需要处理逐笔成交数据表，就可以完成买卖单类型的判断，完成技术方案的大幅度优化。

该部分代码如下：

```
1 | long bidApplSeqNum = Long.parseLong(records[10]);
2 | long offerApplSeqNum = Long.parseLong(records[11]);
3 |
4 | // 确定交易类型并获取主动单索引
5 | String activeOrderIndex = (bidApplSeqNum > offerApplSeqNum) ?
6 |     String.valueOf(bidApplSeqNum) : String.valueOf(offerApplSeqNum);
6 | int tradeType = (bidApplSeqNum > offerApplSeqNum) ? 1 : 2;
```

## 3.2 时间相关处理

在本项目中，我们需要涉及到两处与时间有关的处理：

- 在 `FinalMapper` 中，我们将每个交易时间 (`tradeTime`) 映射到一个特定的时间窗口ID，并将其作为key传递给reducer；
- 在 `FinalReducer` 中，我们根据key的时间窗口ID来聚合位于相同时间窗口的数据，并按时间窗口的大小顺序，根据当前时间窗口ID将数据分配到正确的时间区间。

这两个步骤的具体实现逻辑如下：

### 3.2.1 时间窗口ID划分

在Mapper中，每个逐笔成交表的数据都会有成交时间 `tradetime` 的字段，而我们需要根据给定的 `TIME_WINDOW` 参数，将对应时间的数据划分进相应的时间窗口中。

- `getTimeInMinutes()` 方法：通过截取成交时间 `tradetime` 中的 `HHmm` 字段，将每个交易时间（格式为 `yyyyMMddHHmmssSSSS`）都被转换成从午夜开始的分钟数。具体代码实现为：

```
1 // 将时间字符串 (yyyyMMddHHmmssSSSS) 转化为分钟
2 public static int getTimeInMinutes(long tradetime) {
3     // 转换 tradetime
4     String timeStr = String.valueOf(tradetime).substring(8, 12); // 提取 "HHmm"
5     int hour = Integer.parseInt(timeStr.substring(0, 2));
6     int minute = Integer.parseInt(timeStr.substring(2, 4));
7
8     // 将时间转换为从午夜开始的分钟数
9     return hour * 60 + minute;
10 }
```

- `calculateTimeWindowID` 方法：将交易时间映射到早上和下午两个交易时段内的特定时间窗口。

- 具体计算公式为：

$$\begin{aligned} \text{早上: 时间窗口 ID} &= \frac{\text{目前分钟时间} - \text{早上开始分钟时间}}{\text{时间窗口参数}} + 1 \\ \text{下午: 时间窗口 ID} &= \frac{\text{目前分钟时间} - \text{下午开始分钟时间}}{\text{时间窗口参数}} + \text{早上的窗口数目} + 1 \end{aligned} \quad (1)$$

另外，由于前方所有时间窗口区间都为左闭右开，最后还要单独将15:00的成交数据提取出来，加入当天最后一个时间窗口。

- 该部分的代码如下：

```
1 // 计算时间窗口ID
2 public static long calculateTimeWindowID(long tradetime) {
3     ...
4     if (currentTimeInMinutes >= morningStartInMinutes &&
5         currentTimeInMinutes <= morningEndInMinutes) {
6         // 早上 9:30 - 11:30 的时间段，计算属于哪个窗口
```

```

6         timeWindowID = (currentTimeInMinutes - morningStartInMinutes) /
7             TIME_WINDOW + 1;
8     } else if (currentTimeInMinutes >= afternoonStartInMinutes &&
9     currentTimeInMinutes < afternoonEndInMinutes) {
10        long interval = (morningEndInMinutes - morningStartInMinutes) /
11             TIME_WINDOW;
12        // 下午 13:00 - 15:00 的时间段，计算属于哪个窗口
13        timeWindowID = (currentTimeInMinutes - afternoonStartInMinutes) /
14             TIME_WINDOW + 1 + interval;
15    } else if (currentTimeInMinutes == afternoonEndInMinutes) {
16        long interval = (morningEndInMinutes - morningStartInMinutes) /
17             TIME_WINDOW;
18        timeWindowID = (currentTimeInMinutes - afternoonStartInMinutes) /
19             TIME_WINDOW + (morningEndInMinutes - morningStartInMinutes) / TIME_WINDOW;
20    }
21
22    return timeWindowID;
23}

```

### 3.2.2 时间区间分配

在根据时间窗口ID聚合相关的数据后，可以将数据根据key的时间窗口ID由小到大排序传入 `FinalReducer` 中，进行时间区间的计算分配。可以直接根据传入的时间窗口ID，调用 `calculateTimeInterval(timeWindowID)` 方法计算时间区间。其中涉及到的计算原理如下：

- `getTimeInMinutes()` 方法：与3.2.1 时间窗口计算部分的方法一致。主要用于将早上与下午的起始与终止时间（格式为 `yyyyMMddHHmmssSSS`）通过方法转换成从午夜开始的分钟数，来计算早上9:30-11:30的时间区间所对应的时间窗口ID数目，达成对传入时间窗口ID的分区。
- `addTime()` 方法：通过截断早上与下午起始时间的分钟部分（`HHmm` 部分），加上时间窗口分钟的数目，来实现对时间窗口所在时间区间的计算。可以用公式表示其中的计算过程：

$$\begin{aligned}
 \text{totalMinutes} &= \text{hour} \times 60 + \text{minute} + n \times \text{timeWindow} \\
 \text{newHour} &= \left( \frac{\text{totalMinutes}}{60} \right) \bmod 24 \\
 \text{newMinute} &= \text{totalMinutes} \bmod 60
 \end{aligned} \tag{2}$$

该部分的代码如下：

```

1 // 时间分钟的六进制运算
2 public static String addTime(String time, long n, long b) {
3     // 解析输入的时间，获取小时和分钟
4     long hour = Long.parseLong(time.substring(0,2));
5     long minute = Long.parseLong(time.substring(2,4));
6
7     // 计算总的分钟数
8     long totalMinutes = hour * 60 + minute + (n * b);
9

```

```

10     // 计算新的小时和分钟
11     long newHour = (totalMinutes / 60) % 24; // 处理24小时制
12     long newMinute = totalMinutes % 60;
13
14     // 格式化输出，保证输出为四位数
15     return String.format("%02d%02d", newHour, newMinute);
16 }

```

- **calculateTimeInterval()** 方法：调用 `getTimeInMinutes()` 将时间窗口ID分区到早上或下午，在调用 `addTime()` 将时间窗口ID映射到对应的起始时间 `timeWindowBegin` 与终止时间 `timeWindowEnd` 的区间中。对应的代码部分展示如下：

```

1 // 计算时间区间
2 public static String calculateTimeInterval(long timeWindowID) {
3     ...
4     // 获取早上和下午的timeWindowID间隔
5     long interval = (morningEndInMinutes - morningStartInMinutes) /
6     TIME_WINDOW;
7     ...
8     if (timeWindowID <= interval) {
9         // 早上 9:30 - 11:30 的时间段，计算属于哪个区间
10        timeWindowBegin = addTime(morningStart, timeWindowID-1, TIME_WINDOW);
11        timeWindowEnd = addTime(morningStart, timeWindowID, TIME_WINDOW);
12    } else if (timeWindowID > interval) {
13        // 下午 13:00 - 15:00 的时间段，计算属于哪个区间
14        timeWindowBegin = addTime(afternoonStart, timeWindowID-1-interval,
15        TIME_WINDOW);
16        timeWindowEnd = addTime(afternoonStart, timeWindowID-interval,
17        TIME_WINDOW);
17    }
18    return "20190102" + timeWindowBegin + "00000 to 20190102" + timeWindowEnd +
19    "00000";
20 }

```

- 缺失时间窗口数据的判断与补全：我们留意到在部分时间窗口内，可能没有产生交易，如14:57-15:00这段收盘集合竞价时间中，14:58-14:59分内没有相关的交易数据。所以，为了保证输出时间数据的完整性，我们在 `FinalReducer` 里加了一个检测指标 `index`，会随着时间窗口的处理和输出也相应自增；如果有时间窗口数据缺失导致跳过了该时间窗口的处理，就可以通过以下方法检测到，并补全输出该缺失时间窗口的数据：

```

1 if (index < timeWindowID){
2     long initial = index;
3     for (long i = initial; i < key.get(); i++){
4         String timeInterval = calculateTimeInterval(index);
5         context.write(new Text("0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0"), new
6         Text(timeInterval)); // 补足缺失的时间区间
7         index++;
8     }
9 }

```

### 3.3 成交量、成交额、流通盘占比计算

在 `FinalReducer` 类中，计算了每个时间窗口内的成交量、成交额，并根据成交量和流通股数据（如：`CIRCULATION_STOCK = 17170245800.0`）来计算流通盘占比。

在此之前，我们先需要根据主动单索引 `activeOrderIndex` 对数据进行聚合：

```
1 // 存储每个主动委托索引的累计成交量、成交额和买卖类型
2 Map<String, Object[]> activeOrderData = new HashMap<>();
3
4 // 遍历所有值，累加每个主动委托索引的成交量和成交额，并记录买卖类型
5 for (Text value : values) {
6     String[] fields = value.toString().split("\t");
7
8     try {
9         String activeOrderIndex = fields[0].trim();           // 主动委托索引
10        ...
11        activeOrderData.putIfAbsent(activeOrderIndex, new Object[]{0.0, 0.0,
12 tradeType}); // 根据相同的主动委托索引，累加该主动委托索引对应成交单数据
13        Object[] data = activeOrderData.get(activeOrderIndex);
14        ...
15    } catch (NumberFormatException e) {
16        System.err.println("Error parsing record: " + value.toString());
17    }
18 }
```

在聚合的过程中，根据成交量、成交额、流通盘占比的计算公式，来计算每个索引的成交单的相应指标：

1. 成交量：所有买卖订单的成交量的累计。

$$\text{成交量} = \sum_{i=1}^n (\text{买单成交量}_i + \text{卖单成交量}_i) \quad (3)$$

2. 成交额：通过成交价格 (`price`) 和成交量 (`tradeQty`)，根据单子总数累加计算得到的成交额。

$$\begin{aligned} \text{成交额}_i &= \text{价格}_i \times \text{成交量}_i \\ \text{总成交额} &= \sum_{i=1}^n (\text{价格}_i \times \text{成交量}_i) \end{aligned} \quad (4)$$

3. 流通盘占比：成交量与流通盘的比值。

$$\text{流通盘占比}_i = \frac{\text{成交量}_i}{\text{流通盘}} \quad (5)$$

上述三个指标的具体计算代码如下：

```
1 // 遍历所有值，累加每个主动委托索引的成交量和成交额，并记录买卖类型
2 for (Text value : values) {
```

```

3   String[] fields = value.toString().split("\t");
4
5   try {
6       String activeOrderIndex = fields[0].trim();           // 主动委托索引
7       double price = Double.parseDouble(fields[1]);        // 成交价格
8       double tradeQty = Double.parseDouble(fields[2]);     // 成交量
9       double amount = price * tradeQty;                   // 成交金额
10      ...
11      Object[] data = activeOrderData.get(activeOrderIndex); // 遍历该索引对应的单
12      data[0] = (double) data[0] + tradeQty; // 累加成交量
13      data[1] = (double) data[1] + amount; // 累加成交额
14      data[2] = tradeType;                // 更新买卖类型（确保一致）
15  } catch (NumberFormatException e) {
16      System.err.println("Error parsing record: " + value.toString());
17  }
18 }
19
20 // 在后续根据主动单索引，判断单子类型并分类统计
21 for (Map.Entry<String, Object[]> entry : activeOrderData.entrySet()) {
22     ...
23     double totalTradeQty = (double) data[0]; // 累计成交量
24     double totalAmount = (double) data[1]; // 累计成交额
25     int tradeType = (int) data[2];
26     double circulationRatio = totalTradeQty / CIRCULATION_STOCK; // 流通盘占比
27     ...
28 }

```

## 3.4 主力资金分类、统计与输出

在 `FinalReducer` 中，在计算完相应的成交单指标后，我们需要根据成交单类型（超大单、大单、中单、小单）对买单和卖单进行分类，并统计主力流入和流出。

### 3.4.1 成交单分类标准

判断方式	超大单	大单	中单	小单
成交量	$\geq 200,000$	$\geq 60,000$	$\geq 10,000$	其他
成交额	$\geq 1,000,000$	$\geq 300,000$	$\geq 50,000$	其他
流通盘占比	$\geq 0.003$	$\geq 0.001$	$\geq 0.00017$	其他

文档已说明，成交量、成交额或流通盘占比落入的区间的较高标准即是该单的类型。由此，该部分的代码实现为：

```

1 // 确定单子类型索引
2 int orderTypeIndex;
3 if (totalTradeQty >= 200000 || totalAmount >= 1000000 || circulationRatio >= 0.003)
{
    orderTypeIndex = 0; // 超大单
} else if (totalTradeQty >= 60000 || totalAmount >= 300000 || circulationRatio >=
0.001) {
    orderTypeIndex = 1; // 大单
} else if (totalTradeQty >= 10000 || totalAmount >= 50000 || circulationRatio >=
0.00017) {
    orderTypeIndex = 2; // 中单
} else {
    orderTypeIndex = 3; // 小单
}

```

### 3.4.2 主力资金流动计算公式

在计算主力资金流动的数据前，我们需要先根据上述的成交单判断结果以及买卖类型，来累加合并相同单类型的数据；之后，根据题意中的主力资金计算公式来计算相应的主力数据：

1. 主力流入：

$$\text{主力流入} = \text{超大买单金额} + \text{大买单金额} \quad (6)$$

2. 主力流出：

$$\text{主力流出} = \text{超大卖单金额} + \text{大卖单金额} \quad (7)$$

3. 主力净流入：

$$\text{主力净流入} = \text{主力流入} - \text{主力流出} \quad (8)$$

我们在根据买卖类型聚合数据的过程中，就可以同时实现主力数据的计算。该部分代码实现为：

```

1 // 根据买卖类型累加数据
2 if (tradeType == 1) { // 买单
3     buyQty[orderTypeIndex] += totalTradeQty;
4     buyAmount[orderTypeIndex] += totalAmount;
5     if (orderTypeIndex == 0 || orderTypeIndex == 1) {
6         mainFlowIn += totalAmount; // 计算主力流入
7     }
8 } else if (tradeType == 2) { // 卖单
9     sellQty[orderTypeIndex] += totalTradeQty;
10    sellAmount[orderTypeIndex] += totalAmount;
11    if (orderTypeIndex == 0 || orderTypeIndex == 1) {
12        mainFlowOut += totalAmount; // 计算主力流出
13    }
14 }

```

```
15 // 计算主力净流入  
16 netMainFlow = mainFlowIn - mainFlowOut;  
17
```

### 3.4.3 最终输出

在 `FinalReducer` 中，在计算出所有所需数据后，我们使用 `StringBuilder` 类型的变量来构造最终的输出格式，包括主力净流入、主力流入、主力流出、不同单子类型的买卖成交量和成交额，以及其所处的时间窗口，并将结果写入输出文件。

```
1 StringBuilder resultBuilder = new StringBuilder();  
2 resultBuilder.append(netMainFlow).append(",")  
3     .append(mainFlowIn).append(",")  
4     .append(mainFlowOut);  
5  
6 // 写入买卖单类型统计数据  
7 for (int i = 0; i < 4; i++) {  
8     resultBuilder.append(",").append(buyQty[i])  
9         .append(",").append(buyAmount[i])  
10        .append(",").append(sellQty[i])  
11        .append(",").append(sellAmount[i]);  
12 }  
13  
14 Text outputValue = new Text(resultBuilder.toString());  
15 context.write(outputValue, new Text(timeInterval));
```

## 4. 代码模块化设计思路

在 Hadoop MapReduce 中，代码模块的设计需要考虑如何通过解耦的方式，使得各个部分得以分块化，能够独立进行优化、扩展和复用。在我们设计项目的代码结构时，主要将模块分为 MapReduce 计算模块与可视化模块。

我们的项目文件结构如下：

```
1 STA321-24F-Project  
2   └── .idea  
3   └── data  
4     ├── Final_10.csv  
5     └── Final_20.csv  
6   └── STA321-Project [mapreduce-lab8-ans]  
7     └── src  
8       └── main  
9         └── java  
10        └── driver  
11          └── FinalDriver  
12        └── mapper  
13          └── FinalMapper  
14        └── reducer
```

```
15 |           |           |   └── FinalReducer  
16 |           |           |   └── CsvToHtml  
17 |           |       └── resources  
18 |           |           └── META-INF  
19 |           |               └── log4j.properties  
20 |       └── target  
21 └── pom.xml  
22 └── visualization  
23     ├── script.js  
24     ├── style.css  
25     └── visualization.html  
26
```

接下来，我们将一一阐述MapReduce计算模块与可视化模块的代码分块化设计思路及其优势。

## 4.1 MapReduce计算模块化设计

在我们的技术方案中，我们只设计了一个Mapper和一个Reducer，并通过一个Driver实现任务的调度。下面将分别详细阐述 `FinalDriver`、`FinalMapper` 和 `FinalReducer` 的设计思路，描述各个模块的逻辑结构和内部封装的函数，分析其模块化的优势。

### 4.1.1 Driver 设计

`FinalDriver` 是整个 MapReduce 作业的入口点，主要负责的任务有：

- 作业的配置与初始化；
- 输入路径与 Mapper的关联；
- Mapper和Reducer执行的指定；
- 输出类型和路径的设定；
- 计算作业的提交与执行。

`FinalDriver` 主要负责作业的流程控制，通过集中配置任务来控制作业的执行，使得我们可以方便地扩展作业类型（例如，修改输入路径、输出路径或作业参数），具有高度可扩展性；而具体的 Mapper 和 Reducer 逻辑被独立封装，从而便于单独调试和优化。

### 4.1.2 Mapper 设计

`FinalMapper` 负责对输入数据进行过滤、初步处理并输出中间结果。具体任务包括：

- 从每行数据中提取相关字段；
- 对数据进行筛选，确保仅处理特定的 `ExecType` 和 `SecurityID`；
- 筛选交易时间处在合法区间内的数据；
- 计算时间窗口 ID，并计算主动单。

在 `FinalMapper` 中，许多判断和计算的逻辑分为多个独立的函数，比如 `isWithinTradingTime`、`calculateTimeWindowID`，这些时间计算的逻辑独立封装，使代码中其他部分无需关心具体的时间加减实现，增强了代码的灵活性。另外，每个函数承担单一功能，便于后期修改或复用，也可以减少代码编写错误、调用出错的可能性。此外，`FinalMapper` 的 `map()` 方法只关注数据的过滤与处理，其他的计算与数据格式的转换都被封

装成独立方法，在 `map()` 方法里进行调用，提高了代码的灵活性，这使得 Mapper 的扩展变得更加容易。

### 4.1.3 Reducer 设计

`FinalReducer` 负责对 Mapper 输出的中间结果进行聚合与统计，输出最终的交易数据分析结果。任务包括：

- 根据时间窗口 ID 对数据进行分组，并映射到响应时间窗口；
- 计算每个时间窗口内的主力净流入、主力流入、主力流出等指标；
- 生成输出，并处理表头和缺失时间窗口的输出。

在 `FinalReducer` 中，聚合逻辑已经与统计过程分离，可以便于将来对不同交易数据进行更多元化、更定制化的处理方式进行扩展。另外，Reducer 主要负责数据的聚合与分析，计算逻辑和数据输入被封装为独立函数，函数的封装也提高了代码的可维护性、复用性与扩展性。

## 4.2 可视化代码模块设计

可视化代码分成三个模块：`csvToHtml.java`, `visualization.html`, `style.css`。

### 4.2.1 `csvToHtml.java`

这是一个读取CSV数据，并根据数据获取的函数生成 `script.js` 文件来可视化数据的Java类。

- `readCsv` 方法

该方法可以从`\data\`路径加载CSV文件，读取数据并返回一个列表，列表的每个元素是一个字符串数组。具体来看，它可以实现用 `BufferedReader` 和 `FileReader` 逐行读取CSV文件，分割每行内容并存储到一个 `List<String[]>` 数据结构中。

从模块化设计的角度来看，该方法将文件读取逻辑封装在一起，功能单一，遵循了单一职责原则。并且，这样设计可以将这个方法单独提取出来，可以根据目标文件路径修改读取路径，而不影响后续的方法，提高代码的可复用性。

- `generateJs` 方法

该方法根据提供的数据生成 JavaScript 文件，供前端使用（包括图表的数据和页面交互的逻辑）。具体来看，它使用 `FileWriter` 来写入 JavaScript 文件内容。并生成折线图、饼状图和柱状图的数据，动态构建图表数据和更新的函数。并且，该方法包含图表初始化、更新图表（例如，销毁旧图表并生成新图表）的逻辑，以及页面元素的交互操作（如，点击“更新数据”按钮，可以更新图表数据）。

分析该模块的设计优势，该方法将数据写入JS文件的功能集中在一起，对于图表展示相关的功能，如果需求类似（如更新不同数据、不同类型的图表），该方法可作为模板进行修改和复用，更为方便快捷。

- 3. 辅助方法 (`getLineLabels`, `getLineData`, `dataToJson`)

- 方法功能：

- `getLineLabels`：生成折线图的标签数据。
    - `getLineData`：生成特定列的数据，用于折线图。
    - `dataToJson`：将整个CSV数据转换为JSON格式，供前端使用。

在这个部分中，每个方法承担一个单一任务，功能明确，展现了代码块中清晰的逻辑。

## 4.2.2 visualization.html

`visualization.html` 是前端界面的代码，分为**head**、**body**、**加载js文件（Chart.js）**的脚本部分。它可以直观地生成对应的网页，展示股票资金流向的数据，包括不同的图表和数据表格。用户直接查看全天的股票主力净流入、主力流入、主力流出折线图，也可以选择时间窗口来查看相应时间窗口的数据的饼状图、柱状图分析，所有的数据展示部分均通过图表来直观显示，且页面采用了模块化布局，便于用户交互和数据更新。

## 4.2.3 style.css

`style.css` 文件设置了html网页的字体、排版、布局等来调整页面的视觉效果和布局结构。在这个文件中，我们定义了**h1 样式**、**body 样式**、**.chart-container 样式**、**.chart 样式**、**.data-table 样式**、**.data-display 样式**等样式，它们使得页面中的标题居中、表格整齐、图表之间的间距适中，同时通过Flexbox布局保证图表的对齐和分布，确保整个页面看起来清晰、简洁且有条理。

## 4.2.4 可视化模块化优势

本项目将读取CSV、生成 `script.js`、`visualization.html`、`style.css` 文件模块化设计，有一些明显的优勢：

### 1. 可维护性强

每个模块都聚焦于单一的功能，使得代码更易于理解和维护。`csvToHtml.java` 只负责处理数据的读取和生成，而前端部分则由HTML、CSS和JavaScript文件组成，各自承担不同的任务。这样能够降低系统复杂度，也便于在需要时修改某一部分而不影响其他模块。

### 2. 可复用性高

在 `csvToHtml.java` 中，`generateJs` 方法可以针对不同的需求（如不同的数据或图表类型）进行调整，而不需要从头编写每个图表的JS逻辑。在未来的项目或需求变更中，可以快速进行功能调整或扩展，减少重复劳动，模块化设计使得代码的可复用性大大增强。

### 3. 方便进行单元测试和调试

在可视化的部分里，各个文件模块负责的功能单一，方便进行单元测试。我们可以独立测试CSV文件读取功能，确保数据正确解析；也可以单独测试生成的 `script.js`，确保图表正确渲染。这使得项目的调试变得更为高效，提高了绘图改图的效率，也减少了错误的发生。

# 5. 结果分析

## 5.1 结果准确性验证与可视化展示

### 5.1.1 数据准确性

根据老师给出的样例，我们在 `SecurityID=000001`，`TIME_WINDOW=10或20` 的情况下分别计算了主力数据结果。

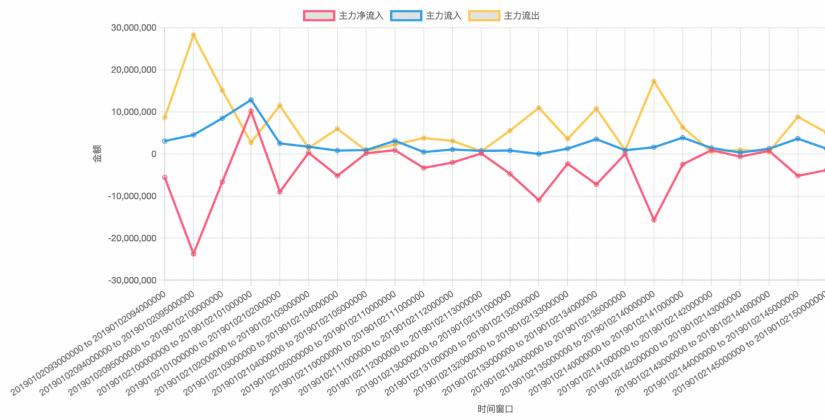
通过与老师给出的样例数据 `000001-时间间隔10min-results.csv`、`000001-时间间隔20min-results.csv` 进行对比，可以在老师给定的评分程序下，验证特定的股票在给定时间窗口下每个时间窗口的主力净流入数据，可以取得满分的效果。

## 5.1.2 可视化展示

通过可视化展示模块的代码，我们可以将计算结果的csv文件的数据进行分析，可以通过前端网页界面来展示数据的分布情况，更加清晰直观地展示我们计算出的数据。以 `SecurityID=000001` , `TIME_WINDOW=10` 的数据为例，进行展示如下：

股票资金流向可视化

全天股票资金流向走势图



选择时间窗口

选择时间窗口: [时间窗口16: 201901021330000000 to 201901021340000000] 更新数据

选择时间窗口

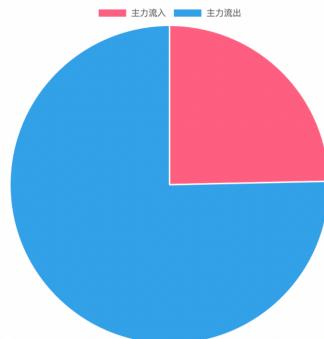
选择时间窗口: [时间窗口16: 201901021330000000 to 201901021340000000] 更新数据

数据展示

主力净流入	主力流入	主力流出	超大买单成交量	超大买单成交额	超大卖单成交量	超大卖单成交额	大买单成交量	大买单成交额	大卖单成交量	大卖单成交额	中买单成交量	中买单成交额	中卖单成交量	中卖单成交额	小买单成交量	小买单成交额	小卖单成交量	小卖单成交额
-7241504.859999999	3522491.18	1.076399604E7	150258.0	1383876.180000002	702000.0	6459157.42	231900.0	2138615.0	467422.0	4304838.62	235700.0	2171151.0	392177.0	3612019.4	253856.0	2338795.76	208622.0	1921274.6

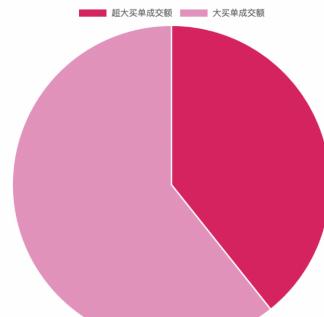
主力流入流出占比

主力净流入:  
**-7241504.859999999**



主力流入单类占比

超大买单成交额  
大买单成交额

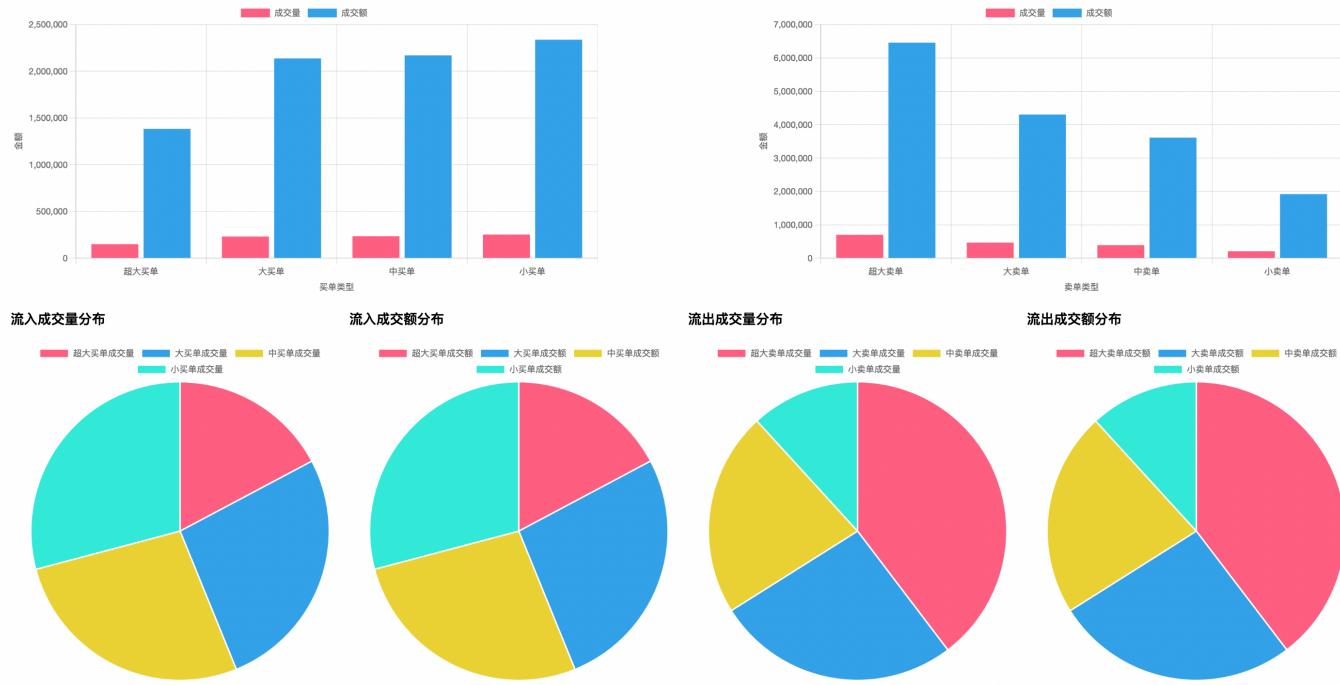


主力流出单类占比

超大卖单成交额  
大卖单成交额



## 资金出入流向分布



## 5.2 性能分析

### 5.2.1 SecurityID=000001 , TIME\_WINDOW=10 情况举例分析

#### 1. 运行时间分析

运行该部分的时间记录截图如下：

```
root@e624817ac95d:~# hadoop jar /opt/project/Final_10.jar
Output path exists. Deleting: /data/project/output
2024-12-22 14:37:16,840 INFO client.DefaultNoHARMFailoverProxyProvider: Connecting to ResourceManager at /0.0.0.0:8032
2024-12-22 14:37:17,176 WARN mapreduce.JobResourceUploader: Hadoop command-line option parsing not performed. Implement the Tool interface and execute your application with ToolRunner to remedy this.
2024-12-22 14:37:17,185 INFO mapreduce.JobResourceUploader: Disabling Erasure Coding for path: /tmp/hadoop-yarn/staging/root/.staging/job_1734351359284_0076
2024-12-22 14:37:17,483 INFO input.FileInputFormat: Total input files to process : 2
2024-12-22 14:37:17,533 INFO mapreduce.JobSubmitter: number of splits:18
2024-12-22 14:37:17,620 INFO mapreduce.JobSubmitter: Submitting tokens for job: job_1734351359284_0076
2024-12-22 14:37:17,620 INFO mapreduce.JobSubmitter: Executing with tokens: []
2024-12-22 14:37:17,738 INFO conf.Configuration: resource-types.xml not found
2024-12-22 14:37:17,739 INFO resource.ResourceUtils: Unable to find 'resource-types.xml'.
2024-12-22 14:37:17,786 INFO impl.YarnClientImpl: Submitted application application_1734351359284_0076
2024-12-22 14:37:17,820 INFO mapreduce.Job: The url to track the job: http://e624817ac95d:8088/proxy/application_1734351359284_0076/
2024-12-22 14:37:17,820 INFO mapreduce.Job: Running job: job_1734351359284_0076
2024-12-22 14:37:23,892 INFO mapreduce.Job: Job job_1734351359284_0076 running in uber mode : false
2024-12-22 14:37:23,893 INFO mapreduce.Job: map 0% reduce 0%
2024-12-22 14:37:41,218 INFO mapreduce.Job: map 22% reduce 0%
2024-12-22 14:37:42,231 INFO mapreduce.Job: map 33% reduce 0%
2024-12-22 14:37:57,445 INFO mapreduce.Job: map 44% reduce 0%
2024-12-22 14:37:58,456 INFO mapreduce.Job: map 67% reduce 0%
2024-12-22 14:38:12,634 INFO mapreduce.Job: map 83% reduce 0%
2024-12-22 14:38:13,643 INFO mapreduce.Job: map 94% reduce 0%
2024-12-22 14:38:15,658 INFO mapreduce.Job: map 100% reduce 0%
2024-12-22 14:38:16,666 INFO mapreduce.Job: map 100% reduce 100%
2024-12-22 14:38:17,687 INFO mapreduce.Job: Job job_1734351359284_0076 completed successfully
```

#### ● Map任务时间：

- 总共启动了18个Map任务 (Launched map tasks=18)。

- 所有Map任务在占用的插槽中总共消耗了 **256,519毫秒**。
  - Map任务的vcore时间为 **256,519 vcore-milliseconds**, 表明每个Map任务占用了一个vcore的时间。
- Reduce任务时间:**
    - 总共启动了1个Reduce任务 (`Launched reduce tasks=1`)。
    - 所有Reduce任务在占用的插槽中总共消耗了 **18,615毫秒** (即约 **18.6秒**)。
    - Reduce任务的vcore时间为 **18,615 vcore-milliseconds**, 这表示Reduce任务在CPU核心上的占用时间。
  - 整体执行时间:**
    - Map任务完成后, Reduce任务开始执行, 最终所有任务的总执行时间 (Map + Reduce) 从 `14:37:17` 到 `14:38:17`, 共 约**60秒**, 任务顺利完成。

## 2. 占用内存分析

执行该任务的内存记录截图如下:

```
2024-12-22 14:38:17,687 INFO mapreduce.Job: Job job_1734351359284_0076 completed successfully
2024-12-22 14:38:17,788 INFO mapreduce.Job: Counters: 54
  File System Counters
    FILE: Number of bytes read=835624
    FILE: Number of bytes written=6867936
    FILE: Number of read operations=0
    FILE: Number of large read operations=0
    FILE: Number of write operations=0
    HDFS: Number of bytes read=2329491803
    HDFS: Number of bytes written=5589
    HDFS: Number of read operations=59
    HDFS: Number of large read operations=0
    HDFS: Number of write operations=2
    HDFS: Number of bytes read erasure-coded=0
  Job Counters
    Launched map tasks=18
    Launched reduce tasks=1
    Data-local map tasks=18
    Total time spent by all maps in occupied slots (ms)=256519
    Total time spent by all reduces in occupied slots (ms)=18615
    Total time spent by all map tasks (ms)=256519
    Total time spent by all reduce tasks (ms)=18615
    Total vcore-milliseconds taken by all map tasks=256519
    Total vcore-milliseconds taken by all reduce tasks=18615
    Total megabyte-milliseconds taken by all map tasks=62675456
    Total megabyte-milliseconds taken by all reduce tasks=19061760
  Map-Reduce Framework
    Map input records=14772443
    Map output records=24983
    Map output bytes=785652
    Map output materialized bytes=835726
    Input split bytes=4536
    Combine input records=0
    Combine output records=0
    Reduce input groups=24
    Reduce shuffle bytes=835726
    Reduce input records=24983
    Reduce output records=25
    Spilled Records=49966
    Total vcore-milliseconds taken by all reduce tasks=18615
    Total megabyte-milliseconds taken by all map tasks=262675456
    Total megabyte-milliseconds taken by all reduce tasks=19061760
  Map-Reduce Framework
    Map input records=14772443
    Map output records=24983
    Map output bytes=785652
    Map output materialized bytes=835726
    Input split bytes=4536
    Combine input records=0
    Combine output records=0
    Reduce input groups=24
    Reduce shuffle bytes=835726
    Reduce input records=24983
    Reduce output records=25
    Spilled Records=49966
    Shuffled Maps =18
    Failed Shuffles=0
    Merged Map outputs=18
    GC time elapsed (ms)=10445
    CPU time spent (ms)=48490
    Physical memory (bytes) snapshot=9809813504
    Virtual memory (bytes) snapshot=49489564224
    Total committed heap usage (bytes)=113444543744
    Peak Map Physical memory (bytes)=602095616
    Peak Map Virtual memory (bytes)=2695297664
    Peak Reduce Physical memory (bytes)=301637632
    Peak Reduce Virtual memory (bytes)=2611580928
  Shuffle Errors
    BAD_ID=0
    CONNECTION=0
    IO_ERROR=0
    WRONG_LENGTH=0
    WRONG_MAP=0
    WRONG_REDUCE=0
  File Input Format Counters
    Bytes Read=0
  File Output Format Counters
    Bytes Written=5589
```

分析运行截图可得:

- Map任务的内存使用量为 **262,675,456 MB-ms**, 可见Map任务的内存使用相对较高, 但仍然处于可接受范围, 峰值内存使用为 **602 MB**。

- Reduce任务的内存使用量为 **19,061,760 MB-ms**, 可见Reduce任务的内存占用相对较低, 峰值内存为 **301 MB**。
- 总的物理内存和虚拟内存快照显示, 任务总体使用了 **约9.81 GB** 的物理内存和 **49.49 GB** 的虚拟内存。

## 5. 结论

目前, 该任务在Map阶段所耗时间居多, Reduce阶段所耗时间较少。由于任务在Map阶段读取了大量数据, 且Map任务与Reduce任务的内存和CPU使用量差异较大, 因此该任务的性能瓶颈可能出现在Map阶段的数据处理和Shuffle阶段的数据传输。

如果要进一步改善, 可能要通过进一步优化内存管理、数据的分区和Shuffle策略, 才可以在未来的任务中减少内存的峰值使用并提高任务的性能。

### 5.2.2 稳定性分析

我们在当前给定数据集下, 验证程序在不同参数设置下的表现。

- 改变 `TimeWindow` 的设置: 我们的任务执行时间大体不会发生变化。

以下是以 `SecurityID=000001`, `TIME_WINDOW=10` 和 `TIME_WINDOW=20` 的情况下的运行时间截图对比:

```
root@e624817ac95d:~# hadoop jar /opt/project/Final_10.jar
Output path exists. Deleting: /data/project/output
2024-12-22 14:37:16,849 INFO client.DefaultNoHARMFailoverProxyProvider: Connecting to ResourceManager at /0.0.0.0:8032
2024-12-22 14:37:17,176 WARN mapreduce.JobResourceUploader: Hadoop command-line option parsing not performed. Implement the Tool interface and execute your application with ToolRunner to really this.
2024-12-22 14:37:17,185 INFO mapreduce.JobResourceUploader: Disabling Erasure Coding for path: /tmp/hadoop-yarn/staging/root/.staging/job_1734351359284_0076
2024-12-22 14:37:17,483 INFO input.FileInputFormat: Total input files to process : 2
2024-12-22 14:37:17,533 INFO mapreduce.JobSubmitter: number of splits:18
2024-12-22 14:37:17,620 INFO mapreduce.JobSubmitter: Submitting tokens for job: job_17343513592
84_0076
2024-12-22 14:37:17,620 INFO mapreduce.JobSubmitter: Executing with tokens: []
2024-12-22 14:37:17,738 INFO conf.Configuration: resource-types.xml not found
2024-12-22 14:37:17,739 INFO resource.ResourceUtils: Unable to find 'resource-types.xml'.
2024-12-22 14:37:17,786 INFO impl.YarnClientImpl: Submitted application application_1734351359284_0076
2024-12-22 14:37:17,828 INFO mapreduce.Job: The url to track the job: http://e624817ac95d:8088/proxy/application_1734351359284_0076/
2024-12-22 14:37:17,828 INFO mapreduce.Job: Running job: job_1734351359284_0076
2024-12-22 14:38:13,892 INFO mapreduce.Job: Job job_1734351359284_0076 running in uber mode : false
2024-12-22 14:37:23,893 INFO mapreduce.Job: map 0% reduce 0%
2024-12-22 14:37:41,218 INFO mapreduce.Job: map 22% reduce 0%
2024-12-22 14:37:42,231 INFO mapreduce.Job: map 33% reduce 0%
2024-12-22 14:37:57,445 INFO mapreduce.Job: map 44% reduce 0%
2024-12-22 14:37:58,456 INFO mapreduce.Job: map 67% reduce 0%
2024-12-22 14:38:02,634 INFO mapreduce.Job: map 83% reduce 0%
2024-12-22 14:38:11,643 INFO mapreduce.Job: map 94% reduce 0%
2024-12-22 14:38:15,650 INFO mapreduce.Job: map 100% reduce 0%
2024-12-22 14:38:16,666 INFO mapreduce.Job: map 100% reduce 100%
2024-12-22 14:38:17,687 INFO mapreduce.Job: Job job_1734351359284_0076 completed successfully
root@e624817ac95d:~# hadoop jar /opt/project/Final_20.jar
Output path exists. Deleting: /data/project/output
2024-12-22 15:00:49,066 INFO client.DefaultNoHARMFailoverProxyProvider: Connecting to ResourceManager at /0.0.0.0:8032
2024-12-22 15:00:49,446 WARN mapreduce.JobResourceUploader: Hadoop command-line option parsing not performed. Implement the Tool interface and execute your application with ToolRunner to really this.
2024-12-22 15:00:49,455 INFO mapreduce.JobResourceUploader: Disabling Erasure Coding for path: /tmp/hadoop-yarn/staging/root/.staging/job_1734351359284_0077
2024-12-22 15:00:49,764 INFO input.FileInputFormat: Total input files to process : 2
2024-12-22 15:00:49,812 INFO mapreduce.JobSubmitter: number of splits:18
2024-12-22 15:00:49,901 INFO mapreduce.JobSubmitter: Submitting tokens for job: job_17343513592
84_0077
2024-12-22 15:00:49,901 INFO mapreduce.JobSubmitter: Executing with tokens: []
2024-12-22 15:00:58,043 INFO conf.Configuration: resource-types.xml not found
2024-12-22 15:00:58,043 INFO resource.ResourceUtils: Unable to find 'resource-types.xml'.
2024-12-22 15:00:58,090 INFO impl.YarnClientImpl: Submitted application application_1734351359284_0077
2024-12-22 15:00:58,121 INFO mapreduce.Job: The url to track the job: http://e624817ac95d:8088/proxy/application_1734351359284_0077/
2024-12-22 15:00:58,122 INFO mapreduce.Job: Running job: job_1734351359284_0077
2024-12-22 15:00:56,196 INFO mapreduce.Job: Job job_1734351359284_0077 running in uber mode : false
2024-12-22 15:00:56,198 INFO mapreduce.Job: map 0% reduce 0%
2024-12-22 15:01:11,603 INFO mapreduce.Job: map 6% reduce 0%
2024-12-22 15:01:13,613 INFO mapreduce.Job: map 28% reduce 0%
2024-12-22 15:01:14,620 INFO mapreduce.Job: map 33% reduce 0%
2024-12-22 15:01:28,854 INFO mapreduce.Job: map 44% reduce 0%
2024-12-22 15:01:29,865 INFO mapreduce.Job: map 61% reduce 0%
2024-12-22 15:01:30,882 INFO mapreduce.Job: map 67% reduce 0%
2024-12-22 15:01:44,020 INFO mapreduce.Job: map 78% reduce 0%
2024-12-22 15:01:45,033 INFO mapreduce.Job: map 94% reduce 0%
2024-12-22 15:01:47,052 INFO mapreduce.Job: map 100% reduce 31%
2024-12-22 15:01:48,050 INFO mapreduce.Job: map 100% reduce 100%
2024-12-22 15:01:49,070 INFO mapreduce.Job: Job job_1734351359284_0077 completed successfully
```

可见二者用时均在1分钟左右, 因时间窗口设置导致的变化对输出结果无明显影响, 可见程序运行的稳定性。

- 改变 `SecurityID` 的筛选条件: 我们的任务执行时间大体不会发生变化。

以下是 `TIME_WINDOW=10` 的条件下, `SecurityID=000001`, 和 `SecurityID=000166` 的情况下的运行时间截图对比:

```

root@e624817ac95d:~# hadoop jar /opt/project/Final_10.jar
Output path exists. Deleting: /data/project/output
2024-12-22 14:37:16,849 INFO client.DefaultNoHARMFailoverProxyProvider: Connecting to ResourceManager at /0.0.0.0:8032
2024-12-22 14:37:17,176 WARN mapreduce.JobResourceUploader: Hadoop command-line option parsing not performed. Implement the Tool interface and execute your application with ToolRunner to remedy this.
2024-12-22 14:37:17,185 INFO mapreduce.JobResourceUploader: Disabling Erasure Coding for path: /tmp/hadoop-yarn/staging/root/.staging/job_1734351359284_0076
2024-12-22 14:37:17,483 INFO input.FileInputFormat: Total input files to process : 2
2024-12-22 14:37:17,533 INFO mapreduce.JobSubmitter: number of splits:18
2024-12-22 14:37:17,620 INFO mapreduce.JobSubmitter: Submitting tokens for job: job_1734351359284_0076
2024-12-22 14:37:17,620 INFO mapreduce.Job: The url to track the job: http://e624817ac95d:8088/proxy/application_1734351359284_0076/
2024-12-22 14:37:17,620 INFO mapreduce.Job: Running job: job_1734351359284_0076
2024-12-22 14:37:23,892 INFO mapreduce.Job: Job job_1734351359284_0076 running in uber mode : false
2024-12-22 14:37:23,893 INFO mapreduce.Job: map 0% reduce 0%
2024-12-22 14:37:41,218 INFO mapreduce.Job: map 22% reduce 0%
2024-12-22 14:37:42,231 INFO mapreduce.Job: map 33% reduce 0%
2024-12-22 14:37:57,445 INFO mapreduce.Job: map 44% reduce 0%
2024-12-22 14:37:58,456 INFO mapreduce.Job: map 67% reduce 0%
2024-12-22 14:38:12,634 INFO mapreduce.Job: map 83% reduce 0%
2024-12-22 14:38:13,643 INFO mapreduce.Job: map 94% reduce 0%
2024-12-22 14:38:15,658 INFO mapreduce.Job: map 100% reduce 0%
2024-12-22 14:38:16,666 INFO mapreduce.Job: map 100% reduce 100%
2024-12-22 14:38:17,687 INFO mapreduce.Job: Job job_1734351359284_0076 completed successfully
root@e624817ac95d:~# hadoop jar /opt/project/Final_15.jar
Output path exists. Deleting: /data/project/output
2024-12-22 15:06:28,084 INFO client.DefaultNoHARMFailoverProxyProvider: Connecting to ResourceManager at /0.0.0.0:8032
2024-12-22 15:06:28,438 WARN mapreduce.JobResourceUploader: Hadoop command-line option parsing not performed. Implement the Tool interface and execute your application with ToolRunner to remedy this.
2024-12-22 15:06:28,451 INFO mapreduce.JobResourceUploader: Disabling Erasure Coding for path: /tmp/hadoop-yarn/staging/root/.staging/job_1734351359284_0078
2024-12-22 15:06:28,773 INFO input.FileInputFormat: Total input files to process : 2
2024-12-22 15:06:28,813 INFO mapreduce.JobSubmitter: number of splits:18
2024-12-22 15:06:28,903 INFO mapreduce.JobSubmitter: Submitting tokens for job: job_1734351359284_0078
2024-12-22 15:06:28,903 INFO mapreduce.JobSubmitter: Executing with tokens: []
2024-12-22 15:06:29,026 INFO conf.Configuration: resource-types.xml not found
2024-12-22 15:06:29,026 INFO resource.ResourceUtils: Unable to find 'resource-types.xml'.
2024-12-22 15:06:29,062 INFO impl.YarnClientImpl: Submitted application application_1734351359284_0078
2024-12-22 15:06:29,085 INFO mapreduce.Job: The url to track the job: http://e624817ac95d:8088/proxy/application_1734351359284_0078/
2024-12-22 15:06:29,086 INFO mapreduce.Job: Running job: job_1734351359284_0078
2024-12-22 15:06:35,176 INFO mapreduce.Job: Job job_1734351359284_0078 running in uber mode : false
2024-12-22 15:06:35,180 INFO mapreduce.Job: map 0% reduce 0%
2024-12-22 15:06:52,536 INFO mapreduce.Job: map 17% reduce 0%
2024-12-22 15:06:53,544 INFO mapreduce.Job: map 33% reduce 0%
2024-12-22 15:07:08,755 INFO mapreduce.Job: map 44% reduce 0%
2024-12-22 15:07:09,755 INFO mapreduce.Job: map 67% reduce 0%
2024-12-22 15:07:10,756 INFO mapreduce.Job: map 83% reduce 0%
2024-12-22 15:07:12,956 INFO mapreduce.Job: map 94% reduce 0%
2024-12-22 15:07:23,956 INFO mapreduce.Job: map 100% reduce 100%
2024-12-22 15:07:26,968 INFO mapreduce.Job: map 100% reduce 100%
2024-12-22 15:07:27,977 INFO mapreduce.Job: map 100% reduce 100%
2024-12-22 15:07:27,990 INFO mapreduce.Job: Job job_1734351359284_0078 completed successfully

```

可见二者用时同样均在1分钟左右，不因筛选出的数据量的差异而产生明显变化，可见程序运行的稳定性。

## 5.3 项目启示

根据整体计算流程的设计、优化，以及对输出结果的分析，我们得到了以下的启示：

- 在实际生产中，可以利用MapReduce的分布式计算，在短时间内处理大规模的数据，并具有较高的准确率。
- 在本项目中，数据量规模仍然适中，在具有更大数据规模的情形下，分布式运算会更具有需求市场。
- 需要以全局视角的来设计问题的处理流程，在优化时注重实际的业务需求，并与技术相结合，来理解大数据处理计算的实际业务逻辑。