STA321 Project

> Dec.22nd, 2024 by Group 14 in Lab Session Tuesday 7-8, Peng Yang

Basic information

Member	Student ID	Lab Session	Contribution Rate
Tianyu Li	12212824	Tuesday 7-8 Yang	50%
Wentao Tian	12212813	Tuesday 7-8 Yang	50%

Contribution of work

Tianyu Li: Overall design of the structure, mapper coding, and data visualization.

Wentao Tian: Reducer coding, refinement of the computation process, and debugging.

1 Problem introduction

In the stock market, it is a crucial task to identify the main flow of capital with real-time order and trade data to help investors make decisions. However, the order and trade data are often of so large volume that the computation flow and program of main capital flow should be carefully designed in order to fulfill such demand.

In this project, we design a mapReduce parallel computing program that can run on Hadoop, thus not only the need to compute the main capital flow is met, but also volumes of different categories of orders are derived, and a visualization page is provided to demonstrate the results more intuitively.

2 Analysis

2.1 Task understanding & key points

- With the input `StockID`, filter out needed records with corresponding `SecurityID`.
- **Important:** check the `TransactTime`'s of the bid/offer orders or the `BidApplSeqNum` & `OfferSeqNum` themselves of a trade record to identify its proactive direction (buy/sell).

- Merge records with the same `BidApplSeqNum` / `OfferSeqNum` (based on its proactive direction).
- **Important:** calculate the quantity & amount (`Qty * price`) of each trade order, categorize it concerning its quantity & amount to **extra-large**, **large**, **medium**, and **small**.
- With the input length of `TimeWindow`, calculate the total quantity & amount of each category and main in/outflow (with data of extra-large and large orders) in each time window. The required fields of the output result are as below:

Fields	Fields
TIMEWINDOW	LARGE_SOLD_QUANTITY
MAIN_NET_INFLOW	LARGE_SOLD_AMOUNT
MAIN_INFLOW	MEDIUM_PURCHASED_QUANTITY
MAIN_OUTFLOW	MEDIUM_PURCHASED_AMOUNT
SUPER_LARGE_PURCHASED_QUANTITY	MEDIUM_SOLD_QUANTITY
SUPER_LARGE_PURCHASED_AMOUNT	MEDIUM_SOLD_AMOUNT
SUPER_LARGE_SOLD_QUANTITY	SMALL_PURCHASED_QUANTITY
SUPER_LARGE_SOLD_AMOUNT	SMALL_PURCHASED_AMOUNT
LARGE_PURCHASED_QUANTITY	SMALL_SOLD_QUANTITY
LARGE_PURCHASED_AMOUNT	SMALL_SOLD_AMOUNT

- Visualize the derived results.

3 Overall technical scheme

3.1 Project structure

```
./src
├── ./resources
└── ./java
    ├── ./driver
    |   └──./driver/StockDriver.java
    ├── ./mapper
    |   └──./mapper/Mapper2.java
    ├── ./reducer
    |   └──./reducer/StockReducer.java
    └── ./webApp
        ├── ./css
        |   └──./css/style.css // style of the webpage display
        ├── ./js
        |   └──./js/chart.js   // imported template package
        ├──./app.js            // read and import result data
        ├──./index.html        // setting the layout of the webpage
        └──./output.csv

```

- `./driver`

This package is responsible for managing the Hadoop job configurations.
The driver class initializes the MapReduce job, sets up input/output formats, and invokes the mapper and reducer classes.
It orchestrates the execution of the entire job.

- `./mapper`

  This package contains classes related to the Map phase of the MapReduce framework.
  Mapper classes implement the logic to process input data, transform it into key-value pairs, and emit intermediate results.
  Each mapper corresponds to a portion of the dataset being processed.

- `./reducer`

  This package includes classes responsible for the Reduce phase of the MapReduce process.
  Reducer classes aggregate and process intermediate key-value pairs emitted by the mappers.
  They execute the logic to summarize or aggregate data, producing the final output.

- `./webApp`

  This package is used to visualize our output.

3.2 Detailed scheme description

3.2.1 Driver

Job Configuration:

The class initializes a Hadoop Job instance named `"JoinExample"`.
It sets the framework name to "local" for local execution, which is useful for testing and debugging.

Parameters and Path Setup:

The class defines a specific stock identifier `SecurityIDQueried` and a time window `Mapper2.TimeWindow`.
It specifies input paths for trade data and an output path where results will be stored.

Multiple Input Paths:

The `MultipleInputs.addInputPath` method is used to specify multiple input sources, allowing the job to handle various data files—in this case, trade data processed by `Mapper2.class`

Reducer Configuration:

The `StockReducer.class` is set as the reducer for the job, which will process the key-value pairs outputted by the mappers.

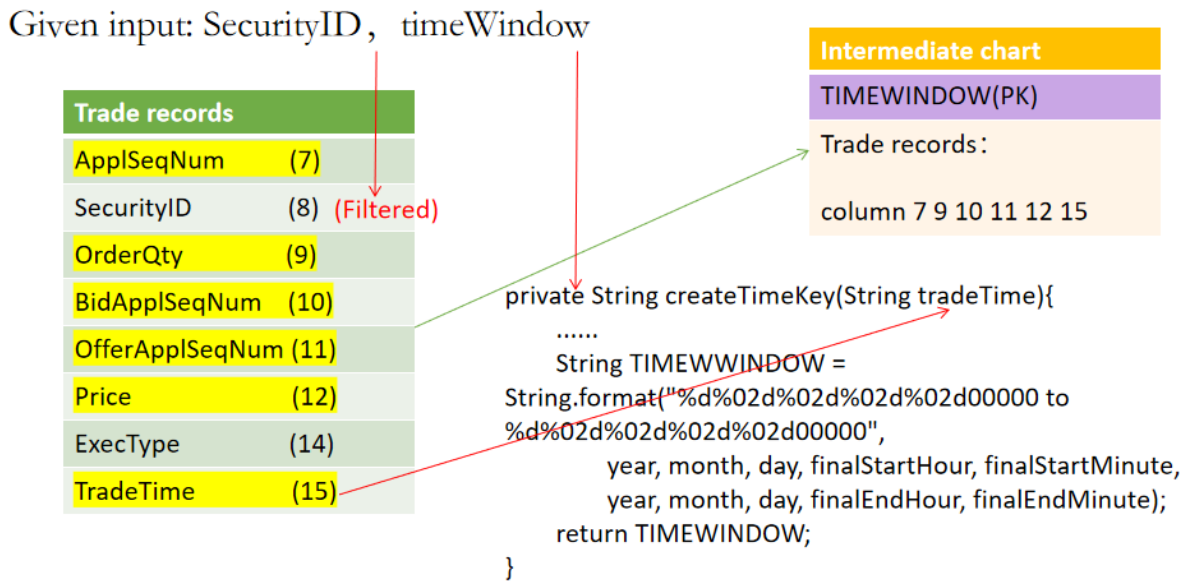
Output Format Configuration:

The output key-value types are set to Text, allowing for structured textual outputs.

Job Submission:

The job is submitted for execution, with the program waiting for its completion. In the event of an error, it prints the stack trace and exits with a non-zero status.

3.2.2 Mapper



Static Parameters:

The class defines two static variables: `TimeWindow` and `filter`, which are used to filter trade data based on specified criteria.
These parameters are set externally, typically by the driver class (`StockDriver.class`).

Map Method:

The map method processes each record of trade data:

- It splits the input text into fields based on whitespace or tab characters to extract relevant information.
- It retrieves the `SecurityID` (from the trade) and the `ExecType` (execution type), filtering for completed trades (`ExecType == "F"`) for the specified security ID.
- It extracts additional fields such as trade ID, purchase order ID, sell order ID, price, trade quantity, and trade time.
- A time window key is generated using the `createTimeKey` method, which converts the trade time into a formatted string reflecting the appropriate time window.

Context Output:

If the generated `timeWindowKey` is valid, the method writes the key-value pair to the context, associating the time window with trade details (trade ID, order IDs, price, and quantity).

Time Key Generation

Key Method: `createTimeKey(String tradeTime)`

```
1 private String createTimeKey(String tradeTime) {
2 // Method implementation
3 }
```

Description

This method converts the input trade time into a time window format, which allows for time-based aggregation of trade data. The implementation includes:

- Parsing the trade time to extract year, month, day, hour, and minute.
- Setting a base time of 9:30 AM, ensuring that inputs before this time return `null`.

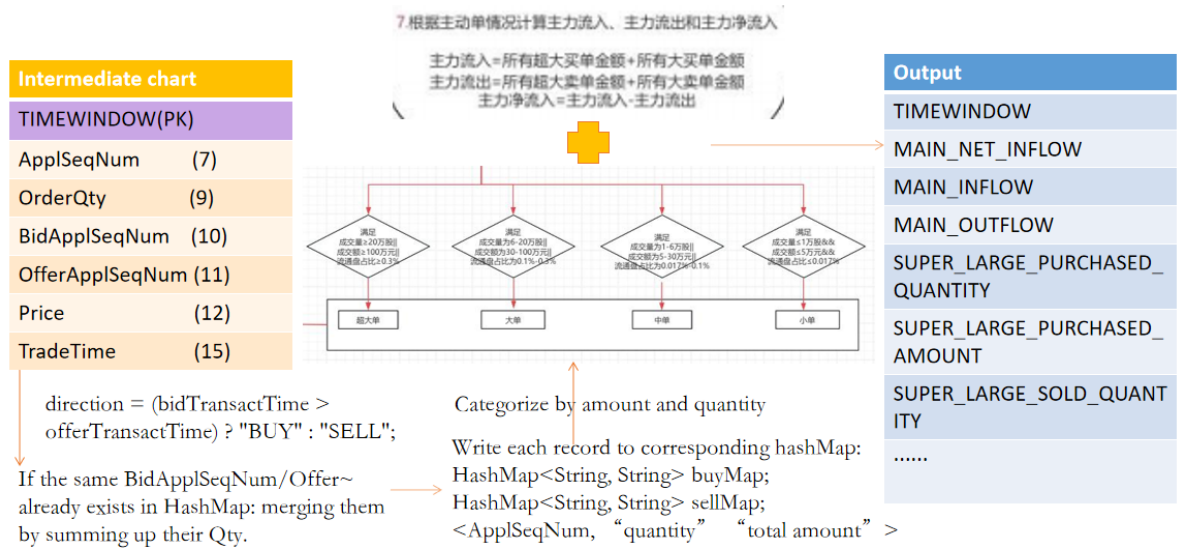
- Calculating the appropriate time slot index based on the `TimeWindow`.
- Generating formatted strings representing the start and end of the time window.

Output Format

The method returns the time window as a string in the format:

`yyyyMMddHHmmSSSss` to `yyyyMMddHHmmSSSss`

3.2.3 Reducer



`Reduce` Method:

The main method in this class, reduce, processes each unique key (representing a time window) and its associated values (trade records).
Data Structures:
Two HashMaps, `buyMap` and `sellMap`, are used to store proactive purchase and sale order records respectively.

Processing Trade Records:

The method iterates through each trade record, splitting the string data into relevant fields—such as **order ID**, **price**, **trade quantity**, and **transaction times**.
It determines the direction of the trade (`BUY` or `SELL`) based on transaction times and merges order records with the same IDs to aggregate data.

Calculating Totals:

The method calculates total inflows and outflows by iterating through both `buyMap` and `sellMap`.
It categorizes trade orders into different classes (**extra-large**, **large**, **medium**, and **small**) based on predefined criteria using the `classifyOrder` method.

Formatting Output:

The final calculation results include the **main net inflow**, **main inflow**, **main outflow**, and categorized trade **quantities** and **amounts**.
The results are formatted using the `formatValue` method, which ensures a consistent numerical output format.

Supporting Methods

1. `formatValue(double value)`

```
    public String formatValue(double value) {
        DecimalFormat df = new DecimalFormat("#.00");
        if (value == Math.floor(value) && !Double.isInfinite(value)) {
            return String.valueOf((int)value); // return as integer if applicable
        }
        return df.format(value); // return formatted string for float
    }
```

This method formats numerical values to two decimal places for float representations and returns integers as whole numbers when appropriate.

2. `classifyOrder(int tradeQty, float price)`

```
    private String classifyOrder(int tradeQty, float price) {
    // Order classification logic
    }
```

The method categorizes orders based on their quantity and price, returning labels such as "**extra-large**," "**large**," "**medium**," "**small**," or "**unknown**."

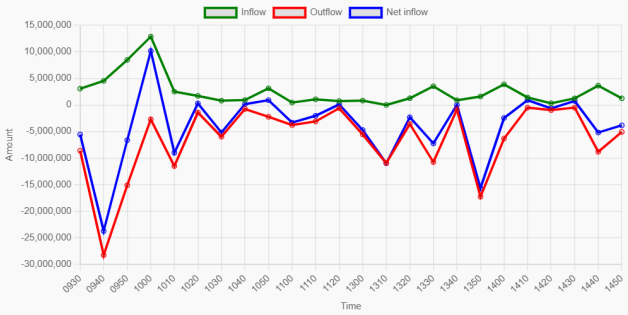
3.2.4 Visualization

We design a web page utilizing the `chart.js` package and the main inflow/outflow/net inflow data are plotted in a single chart. Furthermore, the quantity & amount/inflow & outflow for each category of orders are demonstrated in 2 charts respectively.

Our web page can be easily visited by `index.html`, here is a screen shot of it (take the data of stock 000001 and a time window of 10min as example):

Stock Market Data Visualization

Main flow



Extra-large order

Quantity

Amount

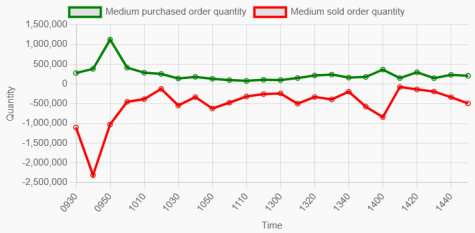
Large order

Quantity

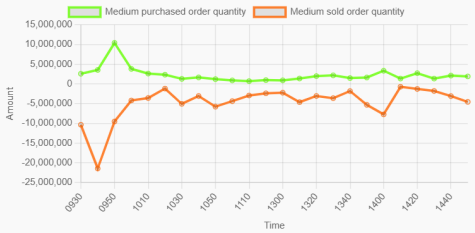
Amount

Medium order

Quantity



Amount



Small order

Quantity

Amount