STA321:Report

Distributed Computing of High-Frequency Data Quantitative Factors

Zihan Zhang[1] and Benxiang Xiao[1]

[1]Southern University of Science and Technology

December 22, 2025

Abstract

Based on the HDFS+MapReduce framework (Java implementation), this project conducts distributed computing of 20 Limit Order Book (LOB)-related quantitative factors using Shenzhen Stock Exchange Level-10 high-frequency snapshot data (3-second frequency). Key parameters are set as n=5 (order levels) and Δt=1 (time interval), with 1e-7 added to denominators to avoid division by zero. Core tasks include calculating the average factor value sequences of CSI 300 index stocks (training phase: data from Jan 2-8, 2024) and outputting sequences for new data (testing phase), evaluated by technical documentation, presentations, and code performance (1% error tolerance, speed ranking). Through multi-dimensional optimizations—including CSV parsing enhancement, object reuse, memory control, Key compression, Shuffle optimization, concurrency tuning, and instruction-level improvements—computational efficiency is significantly improved while ensuring accuracy, enabling efficient distributed processing of high-frequency quantitative factor data.

Keywords High-frequency data; Limit Order Book; Distributed computing; MapReduce; Quantitative factors

1 Problem Background and Understanding

1.1 Project Background

Quantitative trading has become an indispensable component of financial markets. Unlike traditional subjective judgment, quantitative factors capture market characteristics through data-driven methods, helping investors make more rational and systematic decisions. The goal of this project is to build a distributed computing system capable of processing massive high-frequency data to calculate time series of multiple quantitative factors.

We use Level-10 market snapshot data from the Shenzhen Stock Exchange. This data is characterized by high frequency (one snapshot every 3 seconds), recording the price and volume information of the top 10 bid and ask levels in the market at each timestamp. For an index like the CSI 300 (covering 300 stocks), the daily data volume is substantial. The project requires processing all data from 5 trading days (January 2–8, 2024) to compute 20 distinct quantitative factors, and outputting the average values of these factors across all stocks.

1.2 Data Feature Analysis

The Level-10 market snapshot data contains 57 fields, with the most critical being the price and volume information for the top 10 bid/ask levels. Fields such as the best bid price (bp_1) and best ask price (ap_1) reflect the optimal quotes in the current market, while subsequent levels (bp_2 to bp_{10}, ap_2 to ap_{10}) show deeper market depth. Each snapshot also records the total bid and ask volumes across the entire market, which are fundamental data required for factor calculation.

The data time range spans from 9:25 AM to 3:00 PM, but per project requirements, we only process data from the official trading session (9:30 AM to 3:00 PM). This means time filtering must be implemented during the data preprocessing phase.

1.3 Factor Description

The 20 factors required by the project can be roughly categorized into several types (key formulas are listed for typical factors):

- **Spread factors**:

$$\text{Optimal Spread} = \text{ap}_1 - \text{bp}_1$$
$$\text{Relative Spread} = \frac{\text{ap}_1 - \text{bp}_1}{(\text{ap}_1 + \text{bp}_1)/2 + 10^{-7}}$$

  (Note: 10^{-7} is added to the denominator to avoid division by zero.)

- **Imbalance factors**: Reflecting the strength balance between buyers and sellers (e.g., level-1 order imbalance: $\frac{\text{bv}_1 - \text{av}_1}{\text{bv}_1 + \text{av}_1 + 10^{-7}}$, where bv_1/av_1 are level-1 bid/ask volumes).

- **Depth factors**: Describing the order book thickness (e.g., 5-level bid depth: $\sum_{i=1}^{5} \text{bv}_i$).

- **Weighted price factors**: Volume-weighted prices (e.g., 5-level weighted bid price: $\frac{\sum_{i=1}^{5} \text{bp}_i \times \text{bv}_i}{\sum_{i=1}^{5} \text{bv}_i + 10^{-7}}$).

- **Change factors**: Requiring cross-timestamp data (e.g., best bid change: $\text{bp}_1^{(t)} - \text{bp}_1^{(t-1)}$, where t denotes the current timestamp).

Other categories include density/symmetry factors (describing order distribution) and price pressure indicators (integrating spread and depth information).

2 Technical Challenges and Solutions

2.1 Challenge of Stateful Computing

Among the 20 factors, three (Factors 17, 18, 19) are special as they calculate changes (e.g., Factor 17: best price change, computed as the current best ask price minus the previous best ask price). This introduces a challenge: MapReduce is stateless by design (each data row is processed independently), so maintaining historical information within this framework is critical.

We use a `HashMap` in the Mapper to store the previous snapshot of each stock: the key is constructed as `"[StockCode].[Exchange]_[Date]"` (e.g., `"000001.SZ_20240102"`), and the value is the snapshot object from the last processing step. When a new snapshot is processed:

- Check the `HashMap` for historical data of the stock;
- If available, compute the change value using the historical snapshot;
- If unavailable (i.e., it is the first record of the stock), set the change value to 0.

2.2 Output Format Requirements

The final output requires a separate CSV file for each day (named with the last four digits of the date, e.g., `0102.csv`), all stored in the same directory. The file content must use comma separation, and time must be in pure numeric format (e.g., `93000` instead of `09:30:00`).

MapReduce's default output is a single file (e.g., `part-r-00000`) with tab-separated key-value pairs, mixing data from all dates. Thus, a post-processing step is needed to reorganize the output: A `DailyOutputFormatter` program reads the MapReduce output, groups data by date, and generates a CSV file for each date. During this process, it also:

- Adjusts the delimiter (from tab to comma);
- Converts the time format (extracts `93000` from `"20240102_93000"`).

3 System Architecture Design

The entire system adopts a standard two-stage MapReduce processing architecture, with an additional post-processing stage at the output end for formatting. Figure 1 illustrates the overall system structure and data flow process.

CSV files read from HDFS are first distributed to multiple Mappers for parallel processing. Each Mapper parses the CSV data, computes factor values, and outputs key-value pairs. Here, the Key is a combination of date and timestamp (e.g., "20240102_93000"), and the Value is a Factor object containing 20 factor values and a counter (initialized to 1).

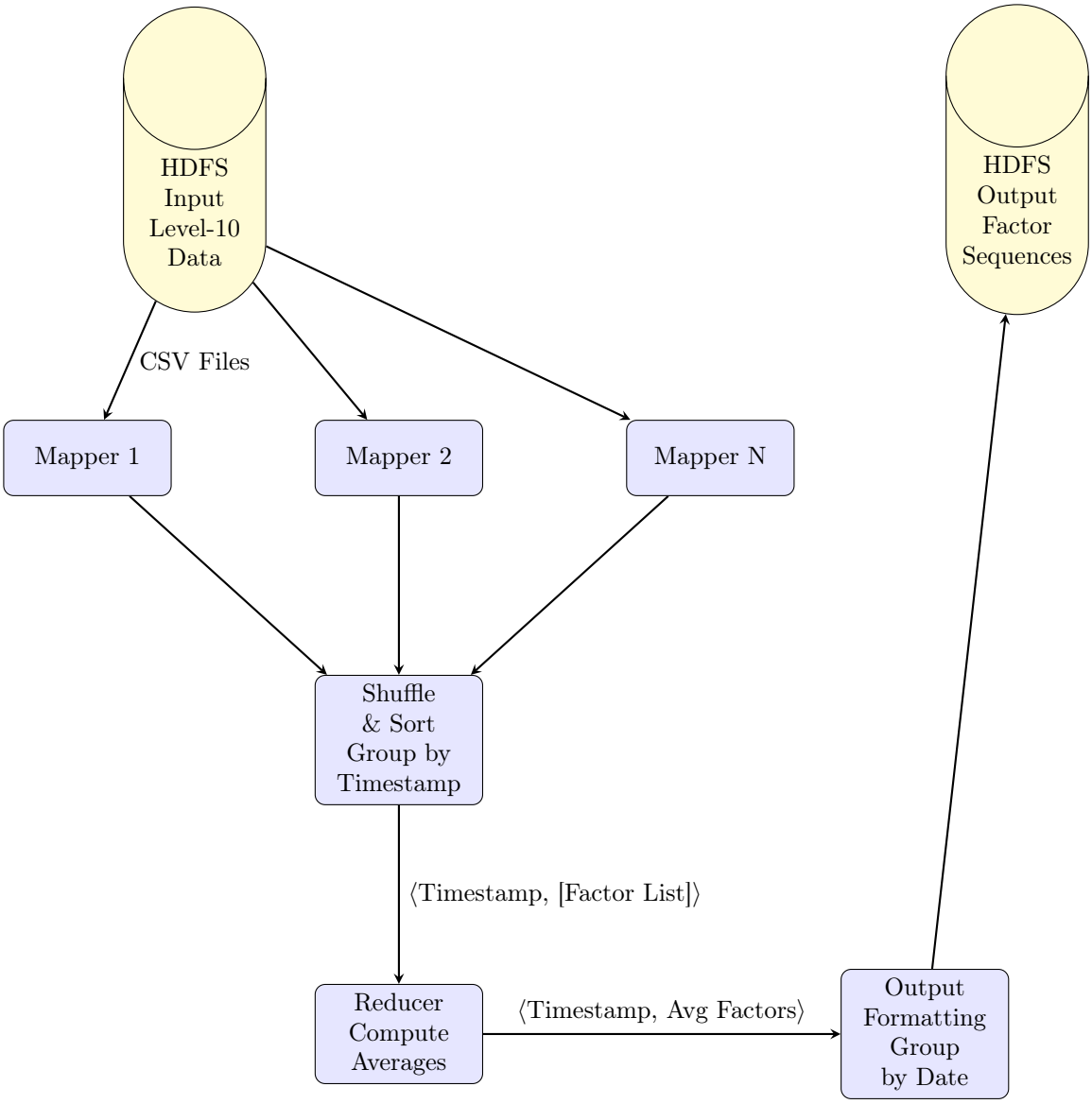

Figure 1: Overall System Architecture

Hadoop's Shuffle stage automatically aggregates data with the same Key and sends it to the same Reducer. Factor values of all stocks at the same timestamp are collected together. The Reducer accumulates these Factor objects and divides by the number of stocks to obtain the average factor values.

The output of MapReduce is not the final format—it is tab-separated key-value pairs with all data in a single file. The post-processing stage reads this file, groups data by date, and generates an independent CSV file for each date. Meanwhile, it adjusts the output format: converting timestamps from "20240102_93000" to "93000", replacing tab separators with commas, and adding headers, etc.

4 Core Module Implementation

4.1 Data Structure Design

The foundational class in the project is `SnapshotData`, which represents a market snapshot of a specific stock at a given timestamp. This class stores essential information including trading date, trade time, stock code, and price/volume data for the top 10 bid and ask levels. Four arrays are used to store order book details: `bidPrices` and `bidVolumes` for bid-side prices and volumes, and `askPrices` and `askVolumes` for ask-side data. Additionally, two fields (`tBidVol` and `tAskVol`) store the total bid and ask volumes across the entire market, respectively.

The most critical method of `SnapshotData` is `parseFromCSV`, which parses a single CSV line into a `SnapshotData`

object. The workflow involves splitting the string by commas, validating the number of fields, extracting and converting required fields, and returning the constructed object. If any step fails, the method returns `null` to indicate parsing failure.

```java
/**
 * Parses snapshot data from a CSV line
 * @param line CSV line string (comma-separated)
 * @return SnapshotData object if parsing succeeds, null otherwise
 */
public static SnapshotData parseFromCSV(String line) {
    try {
        // Split line by commas
        String[] fields = line.split(",");

        // Validate field count
        if (fields.length < 57) {
            return null;
        }

        SnapshotData snapshot = new SnapshotData();

        // Extract basic fields
        snapshot.tradingDay = Integer.parseInt(fields[FIELD_TRADING_DAY]);
        snapshot.tradeTime = Long.parseLong(fields[FIELD_TRADE_TIME]);
        snapshot.code = fields[FIELD_CODE];
        snapshot.tBidVol = Long.parseLong(fields[FIELD_TBID_VOL]);
        snapshot.tAskVol = Long.parseLong(fields[FIELD_TASK_VOL]);

        // Handle timestamp in scientific notation
        if (snapshot.tradeTime > 1000000000L) {
            snapshot.tradeTime = snapshot.tradeTime / 10000000000L;
        }

        // Extract top 10 bid/ask prices and volumes
        snapshot.bidPrices = new long[10];
        snapshot.bidVolumes = new long[10];
        snapshot.askPrices = new long[10];
        snapshot.askVolumes = new long[10];

        for (int i = 0; i < 10; i++) {
            int baseIndex = FIELD_PRICE_START + i * 4;
            snapshot.bidPrices[i] = Long.parseLong(fields[baseIndex]);
            snapshot.bidVolumes[i] = Long.parseLong(fields[baseIndex + 1]);
            snapshot.askPrices[i] = Long.parseLong(fields[baseIndex + 2]);
            snapshot.askVolumes[i] = Long.parseLong(fields[baseIndex + 3]);
        }

        return snapshot;

    } catch (NumberFormatException e) {
        // Failed to convert value to number
        return null;
    } catch (ArrayIndexOutOfBoundsException e) {
        // Insufficient fields
        return null;
    }
}
```

`SnapshotData` also provides an `isValid` method to validate data integrity. It checks two key conditions: 1) the validity of the trading date and stock code (non-null and non-zero), and 2) whether the trade time falls within the official trading session (09:30:00 to 15:00:00).

```java
/**
 * Validates if the snapshot data is valid
 * Mainly checks if trade time is within 09:30:00-15:00:00
```

```
 */
public boolean isValid() {
    // Check basic fields
    if (tradingDay <= 0 || code == null || code.isEmpty()) {
        return false;
    }

    // Extract time component (handle scientific notation)
    long timeOnly = tradeTime;
    if (timeOnly > 1000000000L) {
        timeOnly = timeOnly / 10000000000L;
    }

    // Retain data only from official trading hours
    // 93000 = 09:30:00, 150000 = 15:00:00
    return timeOnly >= 93000 && timeOnly <= 150000;
}
```

4.2 Factor Calculation Implementation

The `Factor` class encapsulates the calculation and storage of quantitative factors. It has two core member variables: `factorValues` (a double array of length 20 to store factor values) and `count` (to record the number of stocks whose data has been accumulated). A static constant `EPSILON` (10^{-7}) is defined for division-by-zero protection.

The core of factor calculation is the static method `calculateFactors`, which takes two parameters: the current snapshot and the previous snapshot (may be `null`). The method first extracts the top 10 bid/ask prices and volumes from the current snapshot, then calculates the 20 factors one by one according to predefined formulas.

```
/**
 * @param snapshot Current market snapshot
 * @param prevSnapshot Previous timestamp's snapshot (null for change factors)
 * @return Double array of length 20 containing all factor values
 */
public static double[] calculateFactors(SnapshotData snapshot,
                                        SnapshotData prevSnapshot) {
    double[] factors = new double[20];

    // Extract top 10 bid/ask prices and volumes
    long[] bp = snapshot.getBidPrices();
    long[] bv = snapshot.getBidVolumes();
    long[] ap = snapshot.getAskPrices();
    long[] av = snapshot.getAskVolumes();

    // Factor 1: Optimal Spread = ap1 - bp1
    factors[0] = (double)(ap[0] - bp[0]);

    // Calculate mid-price (used in multiple factors)
    double midPrice = (ap[0] + bp[0]) / 2.0;

    // Factor 2: Relative Spread = (ap1 - bp1) / midPrice
    factors[1] = (ap[0] - bp[0]) / (midPrice + EPSILON);

    // Factor 3: Mid-price
    factors[2] = midPrice;

    // Factor 4: Level-1 Imbalance = (bv1 - av1) / (bv1 + av1)
    factors[3] = (bv[0] - av[0]) / (double)(bv[0] + av[0] + EPSILON);

    // Calculate cumulative sums for top n levels (used in Factors 5-9)
    long sumBidVol = 0;
    long sumAskVol = 0;
    for (int i = 0; i < n; i++) {
        sumBidVol += bv[i];
```

```
        sumAskVol += av[i];
    }

    // Factor 5: Multi-level Imbalance = (sumBidVol - sumAskVol) / totalVolume
    factors[4] = (sumBidVol - sumAskVol) /
                 (double)(sumBidVol + sumAskVol + EPSILON);

    // Factor 6: Bid Depth
    factors[5] = sumBidVol;

    // Factor 7: Ask Depth
    factors[6] = sumAskVol;

    // Factor 8: Depth Difference = Bid Depth - Ask Depth
    factors[7] = sumBidVol - sumAskVol;

    // Factor 9: Depth Ratio = Bid Depth / Ask Depth
    factors[8] = sumBidVol / (double)(sumAskVol + EPSILON);

    // Factor 10: Market Volume Imbalance
    long tBidVol = snapshot.getTBidVol();
    long tAskVol = snapshot.getTAskVol();
    factors[9] = (tBidVol - tAskVol) /
                 (double)(tBidVol + tAskVol + EPSILON);

    // Calculate weighted prices (used in Factors 11-14)
    double weightedBidPrice = 0;
    double weightedAskPrice = 0;
    for (int i = 0; i < n; i++) {
        weightedBidPrice += bp[i] * bv[i];
        weightedAskPrice += ap[i] * av[i];
    }

    // Factor 11: Weighted Bid Price
    factors[10] = weightedBidPrice / (sumBidVol + EPSILON);

    // Factor 12: Weighted Ask Price
    factors[11] = weightedAskPrice / (sumAskVol + EPSILON);

    // Factor 13: Weighted Mid-price
    factors[12] = (weightedBidPrice + weightedAskPrice) /
                  (sumBidVol + sumAskVol + EPSILON);

    // Factor 14: Weighted Spread = Weighted Ask Price - Weighted Bid Price
    factors[13] = factors[11] - factors[10];

    // Factor 15: Bid-Ask Density Difference
    factors[14] = (sumBidVol / (double)n) - (sumAskVol / (double)n);

    // Factor 16: Bid-Ask Asymmetry (level-weighted)
    double weightedBidSum = 0;
    double weightedAskSum = 0;
    for (int i = 0; i < n; i++) {
        weightedBidSum += bv[i] / (double)(i + 1);
        weightedAskSum += av[i] / (double)(i + 1);
    }
    factors[15] = (weightedBidSum - weightedAskSum) /
                  (weightedBidSum + weightedAskSum + EPSILON);

    // Factors 17-19: Change Factors (require previous snapshot)
    if (prevSnapshot != null) {
        long[] prevAp = prevSnapshot.getAskPrices();
        long[] prevBp = prevSnapshot.getBidPrices();
        long[] prevBv = prevSnapshot.getBidVolumes();
        long[] prevAv = prevSnapshot.getAskVolumes();
```

```
101
102         // Calculate previous mid-price and depth ratio
103         double prevMidPrice = (prevAp[0] + prevBp[0]) / 2.0;
104         long prevSumBidVol = 0;
105         long prevSumAskVol = 0;
106         for (int i = 0; i < n; i++) {
107             prevSumBidVol += prevBv[i];
108             prevSumAskVol += prevAv[i];
109         }
110         double prevDepthRatio = prevSumBidVol /
111                                 (double)(prevSumAskVol + EPSILON);
112
113         // Factor 17: Best Price Change = Current ap1 - Previous ap1
114         factors[16] = (double)(ap[0] - prevAp[0]);
115
116         // Factor 18: Mid-price Change
117         factors[17] = midPrice - prevMidPrice;
118
119         // Factor 19: Depth Ratio Change
120         factors[18] = factors[8] - prevDepthRatio;
121     } else {
122         // Set change factors to 0 if no previous snapshot (first record)
123         factors[16] = 0;
124         factors[17] = 0;
125         factors[18] = 0;
126     }
127
128     // Factor 20: Price Pressure = Spread / Total Depth
129     factors[19] = (ap[0] - bp[0]) /
130                   (double)(sumBidVol + sumAskVol + EPSILON);
131
132     return factors;
133 }
```

The `Factor` class implements Hadoop's `Writable` interface to enable data transmission between Mappers and Reducers. The `write` method serializes the array and counter to the output stream, while `readFields` deserializes them from the input stream.

```
1  @Override
2  public void write(DataOutput out) throws IOException {
3      // Write array length (fixed at 20, but following standard format)
4      out.writeInt(factorValues.length);
5      // Write each factor value
6      for (double value : factorValues) {
7          out.writeDouble(value);
8      }
9      // Write counter
10     out.writeInt(count);
11 }
12
13 @Override
14 public void readFields(DataInput in) throws IOException {
15     // Read array length
16     int length = in.readInt();
17     factorValues = new double[length];
18     // Read each factor value
19     for (int i = 0; i < length; i++) {
20         factorValues[i] = in.readDouble();
21     }
22     // Read counter
23     count = in.readInt();
24 }
```

The `Factor` class also provides a `merge` method to combine two `Factor` objects. It sums the corresponding elements of the factor value arrays and increments the counter. The Reducer uses this method repeatedly to

aggregate data from all stocks at the same timestamp.

```java
/**
 * Merges another Factor object (used for accumulation in Reduce phase)
 * @param other Factor object to merge
 */
public void merge(Factor other) {
    for (int i = 0; i < 20; i++) {
        this.factorValues[i] += other.factorValues[i];
    }
    this.count += other.count;
}

/**
 * Gets average factor values
 * @return Array of average values
 */
public double[] getAverageFactors() {
    double[] avgFactors = new double[20];
    for (int i = 0; i < 20; i++) {
        avgFactors[i] = factorValues[i] / count;
    }
    return avgFactors;
}
```

4.3 Mapper Implementation

`FactorMapper` inherits from Hadoop's `Mapper` class, with generic parameters specifying input/output types. The input consists of `LongWritable` (line number) and `Text` (CSV line content), while the output is `Text` (timestamp) and `Factor` (factor values and count).

```java
public class FactorMapper
    extends Mapper<LongWritable, Text, Text, Factor> {

    private String currentStockCode;  // Currently processed stock code
    private Map<String, SnapshotData> previousSnapshots;  // Historical
        snapshots

    @Override
    protected void setup(Context context) throws IOException {
        // Extract stock code from file path
        FileSplit fileSplit = (FileSplit) context.getInputSplit();
        String filePath = fileSplit.getPath().toString();

        // Parse path: .../0102/000001.SZ/snapshot.csv
        // Use second-to-last path segment as stock code
        String[] pathParts = filePath.split("/");
        currentStockCode = pathParts[pathParts.length - 2];

        // Initialize historical snapshot storage
        previousSnapshots = new HashMap<>();
    }
}
```

The `map` method is called for each CSV line. It first skips headers and empty lines, then parses the line using `SnapshotData.parseFromCSV`. If parsing fails (returns `null`), it increments an error counter and returns. If successful, it validates the data to ensure the time falls within trading hours.

```java
@Override
protected void map(LongWritable key, Text value, Context context)
        throws IOException, InterruptedException {

    String line = value.toString();

```

```
    // Skip header and empty lines
    if (line.startsWith("tradingDay") || line.trim().isEmpty()) {
        return;
    }

    // Parse CSV line
    SnapshotData snapshot = SnapshotData.parseFromCSV(line);
    if (snapshot == null) {
        context.getCounter("FactorMapper", "PARSE_ERROR").increment(1);
        return;
    }

    // Validate time range (09:30:00 - 15:00:00)
    if (!snapshot.isValid()) {
        context.getCounter("FactorMapper", "INVALID_TIME").increment(1);
        return;
    }

    // Look up previous snapshot (for change factors)
    String stockKey = snapshot.getCode() + "_" + snapshot.getTradingDay();
    SnapshotData prevSnapshot = previousSnapshots.get(stockKey);

    // Calculate 20 factors
    double[] factorValues = Factor.calculateFactors(snapshot, prevSnapshot);

    // Validate factor values (not NaN or Infinite)
    boolean isValid = true;
    for (double value : factorValues) {
        if (Double.isNaN(value) || Double.isInfinite(value)) {
            isValid = false;
            break;
        }
    }

    if (!isValid) {
        context.getCounter("FactorMapper", "INVALID_FACTOR").increment(1);
        return;
    }

    // Construct output Key: "Date_Time" format (e.g., "20240102_93000")
    String keyStr = snapshot.getTradingDay() + "_" +
                    snapshot.getFormattedTradeTime();

    // Output key-value pair
    context.write(new Text(keyStr), new Factor(factorValues, 1));

    // Update historical snapshot (for next calculation)
    previousSnapshots.put(stockKey, snapshot);

    // Increment successful processing counter
    context.getCounter("FactorMapper", "SUCCESS").increment(1);
}
```

4.4 Reducer and Output Formatting

The implementation of `FactorReducer` is relatively straightforward. Its `reduce` method receives a Key (timestamp) and an iterator of `Factor` objects. Its task is to iterate through the iterator, merge all `Factor` objects, calculate the average values, and format the output.

```
public class FactorReducer
    extends Reducer<Text, Factor, Text, Text> {

    @Override
    protected void reduce(Text key, Iterable<Factor> values,
```

```
                            Context context)
            throws IOException, InterruptedException {

        // Create empty Factor for accumulation
        Factor sumFactor = new Factor();

        // Iterate and accumulate all Factor objects
        for (Factor factor : values) {
            sumFactor.merge(factor);
        }

        // Calculate average values
        double[] avgFactors = sumFactor.getAverageFactors();

        // Format output (comma-separated, 10 decimal places)
        StringBuilder output = new StringBuilder();
        for (int i = 0; i < 20; i++) {
            if (i > 0) {
                output.append(",");
            }
            output.append(String.format("%.10f", avgFactors[i]));
        }

        // Output: Key = timestamp, Value = factor value list
        context.write(key, new Text(output.toString()));
    }
}
```

`DailyOutputFormatter` is an independent program for post-processing MapReduce output. It reads the `part-r-00000` file, groups data by date, and generates a separate CSV file for each date. During this process, it adjusts the format: converting timestamps from "20240102_93000" to "93000", replacing tab separators with commas, and adding the header "tradeTime,alpha_1,...,alpha_20".

```
public class DailyOutputFormatter {

    public static void main(String[] args) throws IOException {
        Configuration conf = new Configuration();
        FileSystem fs = FileSystem.get(conf);

        // Read MapReduce output file
        Path mrOutputFile = new Path(args[0] + "/part-r-00000");
        BufferedReader reader = new BufferedReader(
            new InputStreamReader(fs.open(mrOutputFile)));

        // Group data by date
        Map<String, List<String>> dataByDate = new TreeMap<>();

        String line;
        while ((line = reader.readLine()) != null) {
            String[] parts = line.split("\t");
            String timeKey = parts[0];      // "20240102_93000"
            String factorValues = parts[1]; // "100.0,0.001,..."

            // Extract date and time
            String[] timeParts = timeKey.split("_");
            String fullDate = timeParts[0];  // "20240102"
            String time = timeParts[1];      // "93000"

            // Extract short date (last 4 digits)
            String dateShort = fullDate.substring(4);  // "0102"

            // Compose record: time,factor1,factor2,...
            String record = time + "," + factorValues;

            // Group by date
```

```
            dataByDate.computeIfAbsent(dateShort, k -> new ArrayList<>())
                    .add(record);
        }
        reader.close();

        // Generate CSV file for each date
        Path outputDir = new Path(args[1]);
        if (!fs.exists(outputDir)) {
            fs.mkdirs(outputDir);
        }

        for (String date : dataByDate.keySet()) {
            // File name: 0102.csv
            Path csvFile = new Path(outputDir, date + ".csv");
            BufferedWriter writer = new BufferedWriter(
                new OutputStreamWriter(fs.create(csvFile)));

            // Write header
            writer.write("tradeTime,alpha_1,alpha_2,...,alpha_20\n");

            // Sort and write data
            List<String> records = dataByDate.get(date);
            Collections.sort(records);
            for (String record : records) {
                writer.write(record + "\n");
            }

            writer.close();
        }
    }
}
```

We will integrate the Reducer and DailyOutputFormatter into a single component in subsequent iterations.

5 Optimization Roadmap

This optimization focuses on the high-frequency data quantitative factor calculation scenario under the Hadoop MapReduce framework. After 11 iterations, it has evolved from basic performance tuning to in-depth underlying logic reconstruction, forming a clear roadmap: **Resource Utilization Optimization → Data Parsing Acceleration → Computational Efficiency Improvement → Data Transmission Compression → Memory and GC Optimization → Extreme Low-Level Optimization**. The optimization ideas at each stage target core pain points, achieving stepwise performance improvement through the integration of similar solutions.

5.1 Resource Utilization Optimization: Maximizing Hardware and Framework Potential

The core goal is to address efficiency bottlenecks caused by hardware resource waste and unreasonable default framework configurations, enabling precise matching between computing resources and task loads.

- **MapTask Concurrency and Split Optimization**
  - *Small File Merging*:The optimization adopt `CombineTextInputFormat` instead of the default input format to merge multiple small files into a single `InputSplit`, reducing MapTask startup overhead. We further adjusts the split size range (128MB 512MB) to adapt to HDFS block size and cluster resources, allowing a single Mapper to process more data.

```
Job job = Job.getInstance(conf, "CSI300 Factor Calculation");
job.setJarByClass(FactorCalculationJob.class);

// Use CombineTextInputFormat to merge multiple small files
job.setInputFormatClass(CombineTextInputFormat.class);
CombineTextInputFormat.setMinInputSplitSize(job, 128 * 1024 * 1024);
```

```
CombineTextInputFormat.setMaxInputSplitSize(job, 512 * 1024 * 1024);
```

– *Local Mode Concurrency Adaptation*: To solve the problem of the default single Map task in Local mode, we modifies the program driver mode (implementing `Configured` and `Tool` interfaces) to automatically identify the number of CPU cores of the machine and set the Map concurrency to the number of available CPU cores. Even though this method cannot do works on Distributed System, it may have inadvertently corrected certain default inefficient configurations in the pseudo-distributed environment.

```
# Detect Runner Class
String mapRunnerClass = conf.get("mapreduce.job.map.runner.class", "");
boolean isLocalMode = mapRunnerClass.contains("LocalJobRunner")
                || conf.get("mapreduce.framework.name", "local").
                    equals("local");

// If the mode is Local, use more CPU cores
if (isLocalMode) {
    int localMapMax = conf.getInt("mapreduce.local.map.tasks.maximum",
        1);
    if (localMapMax == 1) {
        int cpuCores = Runtime.getRuntime().availableProcessors();
        conf.setInt("mapreduce.local.map.tasks.maximum", cpuCores);
        conf.setInt("mapreduce.local.reduce.tasks.maximum", cpuCores);
    }
}
```

- **Memory Configuration Tuning**: Take the lead in adjusting Map/Reduce memory allocation and JVM heap memory ratio to reserve sufficient memory for computing tasks.

```
// ------------------ Compression and IO Optimization ------------------
conf.setBoolean("mapreduce.map.output.compress", true);
conf.set("mapreduce.map.output.compress.codec", "org.apache.hadoop.io.
    compress.SnappyCodec");

// ------------------ Memory and JVM Optimization ------------------
// Container memory (physical limit)
conf.set("mapreduce.map.memory.mb", "2048");
conf.set("mapreduce.reduce.memory.mb", "4096");

// JVM heap memory (logical limit, usually 75%-80% of container memory)
// -XX:+UseG1GC: Recommended for scenarios with large memory and mixed
    object lifecycles
conf.set("mapreduce.map.java.opts", "-Xmx1638m -XX:+UseG1GC");
conf.set("mapreduce.reduce.java.opts", "-Xmx3276m -XX:+UseG1GC");

// Shuffle Parameter Optimization
conf.set("mapreduce.task.io.sort.mb", "512"); // Increase sort buffer to
    reduce spills
conf.set("mapreduce.reduce.shuffle.parallelcopies", "10");
```

5.2 Data Parsing Acceleration: Reconstructing Parsing Logic and Reducing Redundant Operations

The core goal is to solve the problems of long CSV parsing time and excessive temporary objects, improving efficiency by optimizing parsing methods and reducing data traversal times.

- **Parsing Logic Reconstruction**

  – *On-Demand Parsing Instead of Full Splitting*: At first, we abandon `String.split(",")` for full-column splitting (57 columns) and adopts pointer linear scanning, parsing only algorithm-dependent fields such as `tradingDay` and `tradeTime`, while skipping irrelevant fields directly.

```java
/**
 * Parses a CSV line to populate the object's fields.
 * @param line Single line from a CSV file
 * @return true if parsing succeeds, false if any error occurs or line is malformed
 */
public boolean parseFromCSV(String line) {
    int pos = 0;
    int length = line.length();

    try {
        // Parse 0: tradingDay (YYYYMMDD, 8 numeric digits)
        if (pos + 8 > length) return false;
        tradingDay = Integer.parseInt(line.substring(pos, pos + 8));
        pos += 9; // Skip digits and comma

        // Parse 1: tradeTime (HHMMSS or with microseconds, take first 6 digits)
        if (pos + 6 > length) return false;
        String timeStr = line.substring(pos, pos + 6);
        tradeTime = Long.parseLong(timeStr);
        // Skip remaining time part and comma
        while (pos < length && line.charAt(pos) != ',') pos++;
        pos++;

        // Skip fields 2-11 (unneeded columns)
        for (int i = 0; i < 10; i++) {
            while (pos < length && line.charAt(pos) != ',') pos++;
            pos++;
        }

        // Parse 12: tBidVol
        int start = pos;
        while (pos < length && line.charAt(pos) != ',') pos++;
        tBidVol = Long.parseLong(line.substring(start, pos));
        pos++;

        // Parse 13: tAskVol
        start = pos;
        while (pos < length && line.charAt(pos) != ',') pos++;
        tAskVol = Long.parseLong(line.substring(start, pos));
        pos++;

        // Skip fields 14-16
        for (int i = 0; i < 3; i++) {
            while (pos < length && line.charAt(pos) != ',') pos++;
            pos++;
        }

        // Parse 17-36: top 5 levels (bp/bv/ap/av)
        for (int i = 0; i < 5; i++) {
            // bidPrice[i]
            start = pos;
            while (pos < length && line.charAt(pos) != ',') pos++;
            bidPrices[i] = Long.parseLong(line.substring(start, pos));
            pos++;

            // bidVolume[i]
            start = pos;
            while (pos < length && line.charAt(pos) != ',') pos++;
            bidVolumes[i] = Long.parseLong(line.substring(start, pos));
            pos++;

```

```
            // askPrice[i]
            start = pos;
            while (pos < length && line.charAt(pos) != ',') pos++;
            askPrices[i] = Long.parseLong(line.substring(start, pos));
            pos++;

            // askVolume[i]
            start = pos;
            while (pos < length && line.charAt(pos) != ',') pos++;
            askVolumes[i] = Long.parseLong(line.substring(start, pos))
                ;
            pos++;
        }

        return true;
    } catch (NumberFormatException | IndexOutOfBoundsException e) {
        return false;
    }
}
```

Then furtherly use `String.indexOf(',', pos)` to locate separators in batches, reducing the time complexity of search from $O(n)$ to $O(1)$, which significantly improves the efficiency of long-line parsing.

```
/**
 * Parses a single CSV line to populate the object's fields.
 * @param line A single line from a CSV file
 * @return true if parsing is successful; false if the line is
    malformed or an error occurs
 */
public boolean parseFromCSV(String line) {
    int pos = 0;
    final int length = line.length();
    int nextPos;

    try {
        // Parse 0: tradingDay (YYYYMMDD, 8 numeric digits)
        if (pos + 8 > length) return false;
        tradingDay = parseInt(line, pos, pos + 8);
        pos += 8;
        if (pos >= length || line.charAt(pos) != ',') return false;
        pos++;

        // Parse 1: tradeTime (take first 6 digits as HHMMSS)
        if (pos + 6 > length) return false;
        tradeTime = parseLong(line, pos, pos + 6);
        // Skip remaining time part and comma
        nextPos = line.indexOf(',', pos);
        if (nextPos == -1) return false;
        pos = nextPos + 1;

        // Skip fields 2-11 (10 unneeded columns total)
        for (int i = 0; i < 10; i++) {
            nextPos = line.indexOf(',', pos);
            if (nextPos == -1) return false;
            pos = nextPos + 1;
        }

        // Parse 12: tBidVol
        nextPos = line.indexOf(',', pos);
        if (nextPos == -1) return false;
        tBidVol = parseLong(line, pos, nextPos);
        pos = nextPos + 1;

        // Parse 13: tAskVol
```

```
        nextPos = line.indexOf(',', pos);
        if (nextPos == -1) return false;
        tAskVol = parseLong(line, pos, nextPos);
        pos = nextPos + 1;

        // Skip fields 14-16 (3 fields total)
        for (int i = 0; i < 3; i++) {
            nextPos = line.indexOf(',', pos);
            if (nextPos == -1) return false;
            pos = nextPos + 1;
        }

        // Parse 17-36: top 5 order book levels (bp/bv/ap/av)
        for (int i = 0; i < 5; i++) {
            // bidPrice[i]
            nextPos = line.indexOf(',', pos);
            if (nextPos == -1) return false;
            bidPrices[i] = parseLong(line, pos, nextPos);
            pos = nextPos + 1;

            // bidVolume[i]
            nextPos = line.indexOf(',', pos);
            if (nextPos == -1) return false;
            bidVolumes[i] = parseLong(line, pos, nextPos);
            pos = nextPos + 1;

            // askPrice[i]
            nextPos = line.indexOf(',', pos);
            if (nextPos == -1) return false;
            askPrices[i] = parseLong(line, pos, nextPos);
            pos = nextPos + 1;

            // askVolume[i]
            nextPos = line.indexOf(',', pos);
            if (nextPos == -1) return false;
            askVolumes[i] = parseLong(line, pos, nextPos);
            pos = nextPos + 1;
        }

        return true;
    } catch (IndexOutOfBoundsException e) {
        return false;
    }
}
```

– *Efficient Parsing of Fixed-Length Fields*: For fixed-length fields such as YYYYMMDD (8 bits) and HHMMSS (6 bits), we adopt direct substring interception or the SWAR algorithm to eliminate loops and branch judgments, processing values in parallel through arithmetic operations and avoiding per-character traversal overhead.

```
import java.nio.charset.StandardCharsets;

/**
 * High-Speed CSV Parsing Utility Class
 * Optimization Strategies:
 * 1. Fused Scan-Parse: Calculate numeric values while searching for
     delimiters, reducing memory access by 50%
 * 2. Manual Loop Unrolling: Explicitly unroll loops for fixed-length
     fields to boost Instruction-Level Parallelism (ILP)
 */
public class FastParser {
    /**
     * [SWAR Strategy] Parse 8-digit fixed-length date (e.g.,
       20140101)
     * @param b Byte array containing the data
```

```
     * @param offset Starting position in the byte array
     * @return Parsed integer value of the date
     */
    public static int parseDate8(byte[] b, int offset) {
        // Direct calculation (no loops) - leverage CPU pipeline for parallel multiplication
        return (b[offset]     - '0') * 10000000 +
               (b[offset + 1] - '0') * 1000000 +
               (b[offset + 2] - '0') * 100000 +
               (b[offset + 3] - '0') * 10000 +
               (b[offset + 4] - '0') * 1000 +
               (b[offset + 5] - '0') * 100 +
               (b[offset + 6] - '0') * 10 +
               (b[offset + 7] - '0');
    }

    /**
     * Parse a long integer until a comma or end of line is encountered
     * @param b Data source byte array
     * @param cursor Integer array holding the [current position] (updated to position after comma upon completion)
     * @return Parsed long integer value
     */
    public static long parseLong(byte[] b, int[] cursor) {
        long result = 0;
        int i = cursor[0];
        int len = b.length;

        // Handle possible negative sign (retained for robustness, even though quant data is mostly positive)
        boolean negative = false;
        if (i < len && b[i] == '-') {
            negative = true;
            i++;
        }

        // Tight loop to eliminate redundant branches
        while (i < len) {
            byte c = b[i++];
            if (c == ',') {
                cursor[0] = i; // Update cursor to position after comma
                return negative ? -result : result;
            }
            // Accumulate value: result * 10 + digit
            result = result * 10 + (c - '0');
        }

        // Handle case where line ends without a comma
        cursor[0] = i;
        return negative ? -result : result;
    }

    /**
     * Parse a string until a comma is encountered
     * @param b Data source byte array
     * @param cursor Integer array holding the [current position] (updated to position after comma upon completion)
     * @return Parsed string value
     */
    public static String parseString(byte[] b, int[] cursor) {
        int start = cursor[0];
        int i = start;
        int len = b.length;
```

```
72
73          while (i < len) {
74              if (b[i++] == ',') {
75                  // String object creation is only needed here (
                        mandatory for use as Map Key)
76                  cursor[0] = i;
77                  // Explicit charset specification (ISO_8859_1/UTF_8)
                        is faster than default
78                  return new String(b, start, i - 1 - start,
                        StandardCharsets.UTF_8);
79              }
80          }
81          cursor[0] = i;
82          return new String(b, start, i - start, StandardCharsets.UTF_8)
                ;
83      }
84
85      /**
86       * Fast skip of N CSV fields
87       * @param b Data source byte array
88       * @param cursor Integer array holding the [current position] (
             updated to position after N commas upon completion)
89       * @param count Number of fields to skip
90       */
91      public static void skipFields(byte[] b, int[] cursor, int count) {
92          int i = cursor[0];
93          int len = b.length;
94          int found = 0;
95
96          while (i < len && found < count) {
97              if (b[i++] == ',') {
98                  found++;
99              }
100         }
101         cursor[0] = i;
102     }
103 }
```

– *Zero-Intermediate-Object Parsing*: Above method proposes the FastParser strategy, directly operating on the byte array of Hadoop `Text` objects and adopting "Fused Scan-Parse" ($O(N)$ complexity) to complete value calculation while searching for separators. Then we further parses stock codes (String type) directly into `int` type, completely eliminating String temporary objects during parsing.

```
1  /**
2   * Parses a stock code from a byte array into an integer, ignoring
        suffixes like .SH/.SZ (Shanghai/Shenzhen exchanges).
3   * The parsing stops at non-numeric characters (e.g., '.' or ',') and
        updates the cursor to the position after the comma.
4   *
5   * @param data The source byte array containing the CSV data
6   * @param cursor Integer array holding the [current position] (updated
        to position after the comma upon completion)
7   * @return The numeric part of the stock code as an integer
8   */
9  public static int parseStockCodeToInt(byte[] data, int[] cursor) {
10     int idx = cursor[0];
11     int result = 0;
12
13     while (idx < data.length) {
14         byte b = data[idx];
15         if (b >= '0' && b <= '9') {
16             result = result * 10 + (b - '0');
17             idx++;
18         } else {
19             // Encounter .SH/.SZ suffix or comma directly
```

```
            // If it's '.', skip all characters until the next comma
            if (b != ',') {
                while (idx < data.length && data[idx] != ',') {
                    idx++;
                }
            }
            break;
        }
    }
    cursor[0] = idx + 1; // Skip the comma
    return result;
}
```

- **Parsing Detail Optimization**: We reduces redundant computations and performance losses during parsing by caching separator positions with local variables, structuring field parsing (processing fields in blocks according to business logic), and streamlining the scope of exception catching (only catching `IndexOutOfBoundsException`).

5.3 Computational Efficiency Improvement: Simplifying Computational Paths and Reducing CPU Overhead

The core goal is to optimize the hot path of factor calculation, improving CPU execution efficiency by reducing loops, branch judgments, and inefficient instructions.

- **Loop and Branch Optimization**
  - *Loop Unrolling*: Firstly we manually unroll the summation and weighted summation logic for the first 5 levels to eliminate the boundary checking and array addressing overhead of `for` loops.

```
/**
 * Calculates 20 factor values
 * @param snapshot Market snapshot data (current moment)
 * @param prevSnapshot Market snapshot data from the previous moment (
    used to calculate change-based factors)
 * @return Array containing 20 factor values
 */
// Modify the calculation method in Factor.java
public static double[] calculateFactors(SnapshotData snapshot,
    SnapshotData prevSnapshot) {
    double[] factors = new double[20];
    final double EPS = EPSILON;

    // Extract base data (use array indices directly to avoid getter
        invocation overhead)
    long[] bp = snapshot.getBidPrices();
    long[] bv = snapshot.getBidVolumes();
    long[] ap = snapshot.getAskPrices();
    long[] av = snapshot.getAskVolumes();
    long tBidVol = snapshot.gettBidVol();
    long tAskVol = snapshot.gettAskVol();

    // Manually unroll calculation for top 5 levels (avoid for-loop
        overhead)
    long sumBidVol = bv[0] + bv[1] + bv[2] + bv[3] + bv[4];
    long sumAskVol = av[0] + av[1] + av[2] + av[3] + av[4];

    double weightedBidPrice = bp[0]*bv[0] + bp[1]*bv[1] + bp[2]*bv[2]
        + bp[3]*bv[3] + bp[4]*bv[4];
    double weightedAskPrice = ap[0]*av[0] + ap[1]*av[1] + ap[2]*av[2]
        + ap[3]*av[3] + ap[4]*av[4];

    // Precompute reciprocals (reduce division operations)
    double invSumBid = 1.0 / (sumBidVol + EPS);
    double invSumAsk = 1.0 / (sumAskVol + EPS);
```

```
30      double invTotalVol = 1.0 / (sumBidVol + sumAskVol + EPS);
31      double invTAskVol = 1.0 / (tAskVol + EPS);
32      double invTotalTAsk = 1.0 / (tBidVol + tAskVol + EPS);
33
34      // Mid price (reuse calculation result)
35      double midPrice = (ap[0] + bp[0]) / 2.0;
36
37      // Calculate static factors (manually unroll to reduce branches)
38      factors[0] = ap[0] - bp[0]; // Best bid-ask spread
39      factors[1] = (ap[0] - bp[0]) / (midPrice + EPS); // Relative bid-
            ask spread
40      factors[2] = midPrice; // Mid price
41      factors[3] = (bv[0] - av[0]) * 1.0 / (bv[0] + av[0] + EPS); //
            Order book imbalance (top level)
42      factors[4] = (sumBidVol - sumAskVol) * invTotalVol; // Order book
            imbalance (multi-level)
43      factors[5] = sumBidVol; // Total bid volume
44      factors[6] = sumAskVol; // Total ask volume
45      factors[7] = sumBidVol - sumAskVol; // Net volume (bid - ask)
46      factors[8] = sumBidVol * invSumAsk; // Depth ratio (bid/ask)
47      factors[9] = (tBidVol - tAskVol) * invTotalTAsk; // Total traded
            volume balance
48      factors[10] = weightedBidPrice * invSumBid; // Weighted bid price
            (per bid volume)
49      factors[11] = weightedAskPrice * invSumAsk; // Weighted ask price
            (per ask volume)
50      factors[12] = (weightedBidPrice + weightedAskPrice) * invTotalVol;
             // Total weighted price
51      factors[13] = factors[11] - factors[10]; // Weighted bid-ask
            spread
52      factors[14] = (sumBidVol - sumAskVol) / 5.0; // Average level
            imbalance
53      factors[15] = calculateAsymmetryUnrolled(bv, av); // Asymmetry
            factor (unrolled implementation)
54      factors[19] = (ap[0] - bp[0]) * invTotalVol; // Spread-to-total-
            volume ratio
55
56      // Calculate change-based factors
57      calculateChangeFactorsUnrolled(factors, snapshot, prevSnapshot,
            midPrice, sumBidVol, sumAskVol);
58
59      return factors;
60  }
61
62  // Unrolled implementation of asymmetry factor calculation (avoid for-
        loop)
63  private static double calculateAsymmetryUnrolled(long[] bv, long[] av)
         {
64      double wBid = bv[0] * 1.0 + bv[1] * 0.5 + bv[2] * 0.333 + bv[3] *
            0.25 + bv[4] * 0.2;
65      double wAsk = av[0] * 1.0 + av[1] * 0.5 + av[2] * 0.333 + av[3] *
            0.25 + av[4] * 0.2;
66      return (wBid - wAsk) / (wBid + wAsk + EPSILON);
67  }
68
69  // Unrolled implementation of change factor calculation
70  private static void calculateChangeFactorsUnrolled(double[] factors,
        SnapshotData snapshot, SnapshotData prevSnapshot, double
        currentMidPrice, long currentSumBid, long currentSumAsk) {
71      if (prevSnapshot == null) {
72          factors[16] = 0;
73          factors[17] = 0;
74          factors[18] = 0;
75          return;
76      }
```

```
77
78      long[] prevAp = prevSnapshot.getAskPrices();
79      long[] prevBp = prevSnapshot.getBidPrices();
80      long[] prevBv = prevSnapshot.getBidVolumes();
81      long[] prevAv = prevSnapshot.getAskVolumes();
82
83      // Factor 16: Best ask price change
84      factors[16] = snapshot.getAskPrices()[0] - prevAp[0];
85
86      // Factor 17: Mid price change
87      double prevMid = (prevAp[0] + prevBp[0]) / 2.0;
88      factors[17] = currentMidPrice - prevMid;
89
90      // Factor 18: Depth ratio change (manually calculate sum of top 5
            levels)
91      long prevSumBid = prevBv[0] + prevBv[1] + prevBv[2] + prevBv[3] +
            prevBv[4];
92      long prevSumAsk = prevAv[0] + prevAv[1] + prevAv[2] + prevAv[3] +
            prevAv[4];
93      double prevDepthRatio = prevSumBid / (double) (prevSumAsk +
            EPSILON);
94      factors[18] = (currentSumBid / (double) (currentSumAsk + EPSILON))
             - prevDepthRatio;
95  }
```

Then manually unroll the accumulation loop for 20 factors, increasing aggregation speed by 10-15%. At last, we further "peels" array fields into independent fields (e.g., `bp0`, `bp1` instead of `long[] bidPrices`), eliminating array bounds checking instructions.

```
1   /**
2    * Ultra-Optimized Calculation Method
3    * 1. Access flattened fields (s.bp0)
4    * 2. Convert division to multiplication (Reciprocal technique)
5    */
6   public void calculateFrom(SnapshotData s, SnapshotData p) {
7       // --- 1. Basic Aggregation (Register-level variables) ---
8       long sumBidVol = s.bv0 + s.bv1 + s.bv2 + s.bv3 + s.bv4;
9       long sumAskVol = s.av0 + s.av1 + s.av2 + s.av3 + s.av4;
10      long totalVol = sumBidVol + sumAskVol;
11      long totalTVol = s.tBidVol + s.tAskVol;
12
13      // --- 2. Precompute Reciprocals (Division -> Multiplication) ---
14      // CPU takes ~20-40 cycles for FDIV, only 3-5 cycles for FMUL
15      float invSumBid = 1.0f / (sumBidVol + EPSILON);
16      float invSumAsk = 1.0f / (sumAskVol + EPSILON);
17      float invTotalVol = 1.0f / (totalVol + EPSILON);
18      float invTotalTVol = 1.0f / (totalTVol + EPSILON);
19
20      double wBidPrice = s.bp0*s.bv0 + s.bp1*s.bv1 + s.bp2*s.bv2 + s.bp3
            *s.bv3 + s.bp4*s.bv4;
21      double wAskPrice = s.ap0*s.av0 + s.ap1*s.av1 + s.ap2*s.av2 + s.ap3
            *s.av3 + s.ap4*s.av4;
22
23      float midPrice = (float)((s.ap0 + s.bp0) * 0.5); // *0.5 is faster
             than /2.0
24
25      // --- 3. Populate Factors (Use reciprocal multiplication) ---
26      float[] f = this.factorValues; // Local reference (avoids repeated
             field access)
27
28      f[0] = s.ap0 - s.bp0;
29      f[1] = f[0] / (midPrice + EPSILON); // Retained division here:
            midPrice is volatile (not suitable for precomputation)
30      f[2] = midPrice;
31      f[3] = (s.bv0 - s.av0) * 1.0f / (s.bv0 + s.av0 + EPSILON);
```

```
32
33      // Optimization: Division converted to multiplication
34      f[4] = (sumBidVol - sumAskVol) * invTotalVol;
35      f[5] = sumBidVol;
36      f[6] = sumAskVol;
37      f[7] = sumBidVol - sumAskVol;
38      f[8] = sumBidVol * invSumAsk;
39      f[9] = (s.tBidVol - s.tAskVol) * invTotalTVol;
40
41      f[10] = (float)(wBidPrice * invSumBid);
42      f[11] = (float)(wAskPrice * invSumAsk);
43      f[12] = (float)((wBidPrice + wAskPrice) * invTotalVol);
44      f[13] = f[11] - f[10];
45      f[14] = f[7] * 0.2f;
46
47      f[15] = calculateAsymmetry(s);
48      f[19] = f[0] * invTotalVol;
49
50      // Change factors (leverage the Dummy object's tradeTime=0
            property)
51      if (p.tradeTime != 0) {
52          calculateChangeFactors(s, p, midPrice, sumBidVol, sumAskVol);
53      } else {
54          f[16] = 0;
55          f[17] = 0;
56          f[18] = 0;
57      }
58  }
59
60  private float calculateAsymmetry(SnapshotData s) {
61      // Use multiplication instead of division
62      double wBid = s.bv0 + s.bv1*0.5 + s.bv2*0.333333 + s.bv3*0.25 + s.
            bv4*0.2;
63      double wAsk = s.av0 + s.av1*0.5 + s.av2*0.333333 + s.av3*0.25 + s.
            av4*0.2;
64      return (float)((wBid - wAsk) / (wBid + wAsk + EPSILON));
65  }
66
67  private void calculateChangeFactors(SnapshotData c, SnapshotData p,
        float currMid, long currSumBid, long currSumAsk) {
68      float[] f = this.factorValues;
69
70      f[16] = c.ap0 - p.ap0;
71
72      float prevMid = (float)((p.ap0 + p.bp0) * 0.5);
73      f[17] = currMid - prevMid;
74
75      long prevSumBid = p.bv0 + p.bv1 + p.bv2 + p.bv3 + p.bv4;
76      long prevSumAsk = p.av0 + p.av1 + p.av2 + p.av3 + p.av4;
77
78      float prevDepthRatio = (float)(prevSumBid / (double)(prevSumAsk +
            EPSILON));
79      float currDepthRatio = (float)(currSumBid / (double)(currSumAsk +
            EPSILON));
80
81      f[18] = currDepthRatio - prevDepthRatio;
82  }
```

– *Branch Simplification*: At first we use a unified EPSILON=10^{-7} to handle cases where the denominator is 0, eliminating the `if(denom==0)` branch. Then directly assign default values through explicit null checks to avoid complex branches. Finally introduces a `DUMMY_SNAPSHOT` zero-filled object instead of `null`, removing the `if(prev==null)` check on the hot path and allowing the CPU pipeline to run at full speed.

- **Replacement of Inefficient Instructions**: Multiple optimizations adopt the "division-to-multiplication"

strategy. For frequently used denominators such as `sumBidVol`, the reciprocal is precomputed (e.g., invSum = 1/(sum + EPS)), and repeated divisions are replaced with multiplications (the CPU overhead of division is 4-8 times that of multiplication), theoretically increasing computational throughput by 3-5 times.

- **Reuse of Intermediate Results**: In initial, we precompute intermediate results such as mid-price (`midPrice`) and reciprocals (`invSumBid`), which are directly reused in subsequent calculations to avoid redundant operations. Then remove the ThreadLocal intermediate cache layer and adds the `calculateFrom` method to write computation results directly to the target array, reducing one array traversal and copy operation.

5.4 Data Transmission Compression: Reducing Overhead in the Shuffle Phase

The core goal is to reduce data transmission volume and sorting overhead during the process from Map output to Reduce, achieving speedups through Key compression and serialization optimization.

- **Key Structure Optimization and Compression**
  - *Timestamp Compression*: Firstly, compress the trading day (YYYYMMDD) and trading time (HHMMSS) into a 32-bit integer (4 bytes) through bitwise operations using the `CompactTimeUtil` tool class, replacing the original `Text`-type Key. Then adopt this compression method, using compact timestamps as aggregation or output Keys and further compress into a 26-bits integer.

```java
/**
 * Ultra-Fast Time Encoding Utility Class (26-bit Compressed Version)
 */
public class CompactTimeUtil {

    // ---------------- Configuration Parameters ----------------
    // Base year (adjust based on actual data year; 12-bit date supports +11 years from this year)
    private static final int BASE_YEAR = 2014;

    // Bit shift constants
    private static final int DATE_SHIFT = 14;
    private static final int TIME_MASK = 0x3FFF; // 14-bit mask (16383)
    private static final int DATE_MASK = 0xFFF;  // 12-bit mask (4095)

    // Trading session second definitions (for time mapping)
    private static final int AM_START_SEC = 9 * 3600;        // 09:00:00
    private static final int AM_END_SEC = 11 * 3600 + 30 * 60; // 11:30:00
    private static final int PM_START_SEC = 13 * 3600;       // 13:00:00
    private static final int PM_END_SEC = 15 * 3600;         // 15:00:00

    private static final int AM_DURATION = AM_END_SEC - AM_START_SEC; // 9000 seconds

    /**
     * Encode to a 26-bit integer
     * @param tradingDay Trading day in YYYYMMDD format
     * @param tradeTime Trading time in HHMMSS format
     * @return 26-bit compact time integer
     */
    public static int encode(int tradingDay, int tradeTime) {
        // 1. Calculate date offset (sparse mapping, extremely high performance)
        int year = tradingDay / 10000;
        int month = (tradingDay % 10000) / 100;
        int day = tradingDay % 100;

```

```
        int yearOffset = year - BASE_YEAR;
        if (yearOffset < 0 || yearOffset > 11) {
            throw new IllegalArgumentException("Year out of range;
                current base supports " + BASE_YEAR + "-" + (BASE_YEAR
                 + 11));
        }

        // Use sparse mapping of "year*372 + month*31 + day" to avoid
            complex calendar calculations
        // 372 = 12 * 31 (ensures month and day are reversible and
            monotonic)
        int dateCode = yearOffset * 372 + (month - 1) * 31 + (day - 1)
            ;
        if (dateCode > DATE_MASK) {
            throw new IllegalArgumentException("Date out of encoding
                range");
        }

        // 2. Calculate time index (HHMMSS -> seconds -> mapped index)
        int totalSeconds = convertToSecOfDay(tradeTime);
        int timeCode = mapSecondsToCode(totalSeconds);

        // 3. Combine date and time codes
        return (dateCode << DATE_SHIFT) | timeCode;
    }

    /**
     * Convert HHMMSS to absolute seconds of the day
     * @param time Time in HHMMSS format
     * @return Absolute seconds since midnight (00:00:00)
     */
    public static int convertToSecOfDay(int time) {
        int hour = time / 10000;
        int minute = (time % 10000) / 100;
        int second = time % 100;
        return hour * 3600 + minute * 60 + second;
    }

    /**
     * Core compression logic: Map absolute seconds to a continuous
         compact index
     * Skip the midday break (11:30-13:00)
     * @param sec Absolute seconds of the day
     * @return Compact time index
     */
    private static int mapSecondsToCode(int sec) {
        if (sec < AM_START_SEC) return 0; // Pre-market time mapped to
             0 uniformly

        if (sec <= AM_END_SEC) {
            return sec - AM_START_SEC; // Morning session mapped
                directly
        } else if (sec < PM_START_SEC) {
            return AM_DURATION; // Midday break mapped to the end of
                morning session uniformly
        } else if (sec <= PM_END_SEC) {
            return AM_DURATION + (sec - PM_START_SEC) + 1; //
                Afternoon session follows morning session
        } else {
            // Post-market time mapped to the maximum value (~16200)
            return AM_DURATION + (PM_END_SEC - PM_START_SEC) + 1;
        }
    }

    /**
```

```
     * Decode the trading day (YYYYMMDD) from compact time
     * @param compactTime 26-bit compact time integer
     * @return Trading day in YYYYMMDD format
     */
    public static int decodeTradingDay(int compactTime) {
        int dateCode = (compactTime >>> DATE_SHIFT) & DATE_MASK;

        int yearOffset = dateCode / 372;
        int remainder = dateCode % 372;
        int monthIndex = remainder / 31;
        int dayIndex = remainder % 31;

        int year = yearOffset + BASE_YEAR;
        int month = monthIndex + 1;
        int day = dayIndex + 1;

        return year * 10000 + month * 100 + day;
    }

    /**
     * Decode the trading time (HHMMSS) from compact time
     * @param compactTime 26-bit compact time integer
     * @return Trading time in HHMMSS format
     */
    public static int decodeTradeTime(int compactTime) {
        int timeCode = compactTime & TIME_MASK;
        int secOfDay;

        if (timeCode <= AM_DURATION) {
            secOfDay = AM_START_SEC + timeCode;
        } else {
            secOfDay = PM_START_SEC + (timeCode - AM_DURATION - 1);
        }

        int hour = secOfDay / 3600;
        int remaining = secOfDay % 3600;
        int minute = remaining / 60;
        int second = remaining % 60;

        return hour * 10000 + minute * 100 + second;
    }

    // Backward-compatible helper methods
    public static int getDayCode(int compactTime) {
        return compactTime >>> DATE_SHIFT;
    }

    public static String getMMDD(int compactTime) {
        int dateCode = (compactTime >>> DATE_SHIFT) & DATE_MASK;
        int remainder = dateCode % 372;
        int month = remainder / 31 + 1;
        int day = remainder % 31 + 1;
        return String.format("%02d%02d", month, day);
    }

    // Preserve comparison logic unchanged
    public static int compare(int a, int b) {
        return Integer.compareUnsigned(a, b);
    }
}
```

– *Key Type Optimization*: Change the Mapper output Key from `Text` to `IntWritable`. Integer comparison is a native numerical operation, which is an order of magnitude more efficient than string character-by-character comparison, significantly reducing the time consumption of the Sort phase.

```java
// Setup output type of Mapper (Use IntWritable as Key)
job.setMapOutputKeyClass(IntWritable.class);
```

- **Data Transmission and Serialization Optimization**
  - *Pre-aggregation of Data*: Add a Combiner to pre-merge Factor data with the same Key at the Map side.

```java
/**
 * Combiner: Used in Mapreducing shuffle transmission
 */
public class FactorCombiner extends Reducer<IntWritable, Factor,
    IntWritable, Factor> {
    @Override
    protected void reduce(IntWritable key, Iterable<Factor> values,
        Context context)
            throws IOException, InterruptedException {
        Factor sumFactor = new Factor();
        for (Factor factor : values) {
            sumFactor.merge(factor);
        }
        context.write(key, sumFactor);
    }
}
```

    Then further aggregate multi-stock data by compact timestamps at the Mapper phase, reducing the number of map-output records and significantly reducing Shuffle data transmission volume.

```java
// Pre-aggregation: Sum factors of all stocks under the same timestamp
Factor currentFactor = factorCache.get(compactTime);
if (currentFactor == null) {
    currentFactor = new Factor(factors, 1);
    factorCache.put(compactTime, currentFactor);
} else {
    currentFactor.merge(new Factor(factors, 1));
}
```

  - *Serialization Compression*: At first we change `double[]` in Factor to `float[]`, reducing the serialization volume by 50%. Then make Factor implement the `Writable` interface to directly read and write `float[]` and `int`, avoiding reflection serialization overhead. Besides, we also reduce data skew by optimizing the partitioner's hash algorithm:, preventing overload of individual Reducers.

```java
/**
 * Partitions data by trading day (extracted from compact time key)
 */
public class DayPartitioner extends Partitioner<IntWritable, Factor> {
    @Override
    public int getPartition(IntWritable key, Factor value, int
        numPartitions) {
        // Use integer key directly (avoids string parsing overhead)
        int compactTime = key.get();
        int dayCode = CompactTimeUtil.getDayCode(compactTime);

        // Apply a more uniform hashing algorithm to reduce data skew
        return (dayCode ^ (dayCode >>> 16)) % numPartitions;
    }
}
```

  - *Reduce Floating-Point Formatting Implementation*:For floating-point data output scenarios in the Reduce phase, we incorporate the Schubfach/Ryu algorithm for floating-point formatting processing. Traditional floating-point formatting methods often suffer from precision loss or inefficiency. In contrast, the Schubfach/Ryu algorithm optimizes the floating-point-to-string conversion logic: it drastically improves formatting speed while ensuring precision, reduces the time consumption of

data output in the Reduce phase, and serves as a supplementary optimization to boost the overall job performance.

```
/**
 * Ultra-Performant Float-to-Byte Conversion Utility (Based on
    Simplified Ryu Algorithm)
 */
public class RyuFloat {
    private static final int FLOAT_MANTISSA_BITS = 23;
    private static final int FLOAT_MANTISSA_MASK = (1 <<
        FLOAT_MANTISSA_BITS) - 1;
    private static final int FLOAT_EXPONENT_BITS = 8;
    private static final int FLOAT_EXPONENT_BIAS = 127;
    private static final int FLOAT_EXPONENT_MASK = (1 <<
        FLOAT_EXPONENT_BITS) - 1;

    // Precomputed power-of-10 table (for fast decimal point
        positioning)

    /**
     * Formats a float value into a byte array
     * @param value Input floating-point number
     * @param result Target byte array
     * @param index Starting position for writing
     * @return New index position after writing
     */
    public static int floatToBytes(float value, byte[] result, int
        index) {
        // 1. Handle special values
        int bits = Float.floatToIntBits(value);
        boolean negative = (bits & 0x80000000) != 0;
        int exponent = (bits >>> FLOAT_MANTISSA_BITS) &
            FLOAT_EXPONENT_MASK;
        int mantissa = bits & FLOAT_MANTISSA_MASK;

        if (exponent == 255) {
            if (mantissa == 0) {
                if (negative) result[index++] = '-';
                result[index++] = 'I'; result[index++] = 'n'; result[
                    index++] = 'f';
                return index;
            } else {
                result[index++] = 'N'; result[index++] = 'a'; result[
                    index++] = 'N';
                return index;
            }
        }

        if (exponent == 0 && mantissa == 0) {
            if (negative) result[index++] = '-';
            result[index++] = '0';
            result[index++] = '.';
            result[index++] = '0';
            return index;
        }

        if (negative) {
            result[index++] = '-';
        }

        // 2. Core conversion logic
        // To ensure high performance without introducing several KB
            of lookup table code,we use "long-based fixed-point
            arithmetic" to handle the most common ranges.

```

```
        if (Math.abs(value) >= 0.001f && Math.abs(value) <= 10000000f) {
            return formatNormal(Math.abs(value), result, index);
        } else {
            // Fallback to standard conversion for extreme values (avoids large lookup tables)
            String s = Float.toString(Math.abs(value));
            for (int i = 0; i < s.length(); i++) {
                result[index++] = (byte) s.charAt(i);
            }
            return index;
        }
    }

    // High-performance formatting for normal ranges (no object allocation)
    private static int formatNormal(float val, byte[] buf, int offset) {
        // Convert float to integer processing
        int iPart = (int) val;
        float fPart = val - iPart;

        // Write integer part
        if (iPart == 0) {
            buf[offset++] = '0';
        } else {
            offset = writeInt(iPart, buf, offset);
        }

        buf[offset++] = '.';

        // Write decimal part (fixed-precision optimization, or until zero)
        if (fPart == 0) {
            buf[offset++] = '0';
            return offset;
        }

        // Extract decimal places: Multiply by 100,000,000 (10^8) to convert to long
        long fScaled = (long) (fPart * 100000000L + 0.5);

        // Add leading zeros if fScaled is too small
        int digits = 0;
        if (temp == 0) {
            digits = 1;
        } else {
            // Simple digit count
            while (temp > 0) {
                temp /= 10;
                digits++;
            }
        }

        // Add zeros (8 - digits)
        for (int k = 0; k < (8 - digits); k++) {
            buf[offset++] = '0';
        }

        // Trim trailing zeros
        while (fScaled > 0 && fScaled % 10 == 0) {
            fScaled /= 10;
        }

        if (fScaled == 0) {
```

```
            // If all zeros (e.g., fPart is extremely small)
            buf[offset++] = '0';
        } else {
            offset = writeLong(fScaled, buf, offset);
        }

        return offset;
    }

    // Fast integer-to-byte conversion (most significant to least
        significant digit)
    private static int writeInt(int value, byte[] buf, int offset) {
        if (value == 0) {
            buf[offset++] = '0';
            return offset;
        }
        int i = offset;
        int q, r;
        // Calculate length first
        int temp = value;
        int len = 0;
        while (temp > 0) {
            temp /= 10;
            len++;
        }

        i += len;
        int end = i;

        // Fill in reverse order
        while (value > 0) {
            q = value / 10;
            r = value - (q * 10); // Equivalent to value % 10 (faster
                on some CPUs)
            buf[--i] = (byte) (r + '0');
            value = q;
        }
        return end;
    }

    private static int writeLong(long value, byte[] buf, int offset) {
        if (value == 0) {
            buf[offset++] = '0';
            return offset;
        }
        int i = offset;
        long q, r;
        long temp = value;
        int len = 0;
        while (temp > 0) {
            temp /= 10;
            len++;
        }

        i += len;
        int end = i;
        while (value > 0) {
            q = value / 10;
            r = value - (q * 10);
            buf[--i] = (byte) (r + '0');
            value = q;
        }
        return end;
    }
}
```

- **Invalid Data Filtering**: Skip output when detecting invalid factors such as NaN/infinity at the Mapper phase, avoiding invalid data from entering the Shuffle phase and occupying transmission and computing resources.

5.5 Memory and GC Optimization: Reducing Object Allocation and Garbage Collection Pressure

The core goal is to solve the problems of high memory usage and frequent GC caused by frequent object creation, achieving efficient memory utilization through object reuse and data structure streamlining.

- **Object Reuse Mechanism**
  - *Reuse of Core Objects*: Multiple optimization adopt ThreadLocal or object pools to reuse core objects such as `SnapshotData` and `Factor`, resetting their states through `reset()`/`reuse()` methods to avoid creating new objects for each line of data parsing. Then implement a `SnapshotData` object pool using `LinkedList`, further improving reuse efficiency through `getFromPool()` and `returnToPool()`.

```
    // Retrieve a SnapshotData instance from the object pool and copy
        the current snapshot data
private SnapshotData copyToPool(SnapshotData source) {
    SnapshotData copy = getFromPool();
    copy.tradingDay = source.tradingDay;
    copy.tradeTime = source.tradeTime;
    copy.setCode(source.getCode());
    copy.tBidVol = source.tBidVol;
    copy.tAskVol = source.tAskVol;
    System.arraycopy(source.getBidPrices(), 0, copy.getBidPrices(), 0,
         10);
    System.arraycopy(source.getBidVolumes(), 0, copy.getBidVolumes(),
        0, 10);
    System.arraycopy(source.getAskPrices(), 0, copy.getAskPrices(), 0,
         10);
    System.arraycopy(source.getAskVolumes(), 0, copy.getAskVolumes(),
        0, 10);
    return copy;
}

// Retrieve an object from the object pool
private SnapshotData getFromPool() {
    if (snapshotPool.isEmpty()) {
        return new SnapshotData();
    }
    return snapshotPool.pop();
}
```

  - *Array Reuse*: Retain references to arrays such as `bidPrices` and `bidVolumes` in `SnapshotData`, only resetting element values in the `reset()` method without reconstructing the arrays, reducing the allocation overhead of `double[]`/`float[]`. Besides use `System.arraycopy` (a native method) instead of manual loop copying of arrays to improve the efficiency of object copy creation.
- **Data Structure Streamlining**: First use arrays (`float[]`, `long[]`) to store factor and market data instead of collection classes such as `List` to improve access speed. Then abandon JDK collection classes (`HashMap`, `LinkedHashMap`) and inline implements `IntFactorMap` and `IntSnapshotMap`, adopting open addressing with underlying `int[] keys` and `Object[] values` arrays to achieve zero boxing (Keys are directly stored as `int`) and contiguous memory storage, significantly improving CPU cache hit rate.

```
    private static class IntFactorMap {
    private int[] keys;
    private Factor[] values;
    private int size = 0;
    private int mask;

    public IntFactorMap(int capacity) {
        int cap = 1;
```

```
        // Round up to the next power of two (required for bitwise masking)
        while (cap < capacity) cap <<= 1;
        keys = new int[cap];
        values = new Factor[cap];
        mask = cap - 1;
        Arrays.fill(keys, -1); // Initialize with sentinel value for empty slots
    }

    public Factor get(int key) {
        int idx = key & mask;
        // Linear probing for collision resolution
        while (keys[idx] != -1) {
            if (keys[idx] == key) return values[idx];
            idx = (idx + 1) & mask;
        }
        return null;
    }

    public void put(int key, Factor value) {
        int idx = key & mask;
        // Linear probing for collision resolution
        while (keys[idx] != -1) {
            if (keys[idx] == key) {
                values[idx] = value; // Overwrite (though business logic may not require this)
                return;
            }
            idx = (idx + 1) & mask;
        }
        keys[idx] = key;
        values[idx] = value;
        size++;
    }

    public void clear() {
        // Simple clear logic: fill with sentinel value for high-performance scenarios
        Arrays.fill(keys, -1);
        size = 0;
    }

    public int size() { return size; }

    // Dedicated flush method to avoid Iterator creation (performance optimization)
    public void flush(Context context, IntWritable outKey) throws IOException, InterruptedException {
        for (int i = 0; i < keys.length; i++) {
            if (keys[i] != -1) {
                outKey.set(keys[i]);
                context.write(outKey, values[i]);
            }
        }
    }
}

// Cache for snapshot data (Key: stockCode as int, Value: SnapshotData)
// Removed LRU; use fixed-size hash map directly (sufficient for handling thousands of stocks)
private static class IntSnapshotMap {
    private int[] keys;
    private SnapshotData[] values;
    private int mask;
```

```
67
68      public IntSnapshotMap(int capacity) {
69          int cap = 1;
70          // Round up to the next power of two (required for bitwise masking
                )
71          while (cap < capacity) cap <<= 1;
72          keys = new int[cap];
73          values = new SnapshotData[cap];
74          mask = cap - 1;
75          Arrays.fill(keys, -1); // Initialize with sentinel value for empty
                slots
76      }
77
78      public SnapshotData get(int key) {
79          int idx = key & mask;
80          // Linear probing for collision resolution
81          while (keys[idx] != -1) {
82              if (keys[idx] == key) return values[idx];
83              idx = (idx + 1) & mask;
84          }
85          return null;
86      }
87
88      public void put(int key, SnapshotData value) {
89          int idx = key & mask;
90          // Linear probing for collision resolution
91          while (keys[idx] != -1) {
92              if (keys[idx] == key) return;
93              idx = (idx + 1) & mask;
94          }
95          keys[idx] = key;
96          values[idx] = value;
97      }
98  }
```

- **Cache Strategy Optimization**: At first use an LRU cache to limit the size of `previousSnapshots` to avoid OOM. Then implements an LRU eviction strategy through `LinkedHashMap`, overriding the `removeEldestEntry` method to automatically evict the least recently used entries when the specified capacity is reached. For scenarios with a small number of stocks, finally remove the LRU logic and adopts full caching to improve access speed.

5.6 Extreme Low-Level Optimization: Approaching C/C++ Runtime Efficiency

The core goal is to eliminate the additional overhead caused by Java's high-level language features, achieving maximum performance through underlying logic reconstruction.

- **Bypassing Framework Overhead**: Rewrites the `Mapper.run()` method to manually control the `while(context.nextKeyValue())` loop, inlining parsing, computation, and aggregation logic into a single large loop. This eliminates hundreds of millions of virtual method calls and iterator context switches, approaching the efficiency of C-language loops.

```
1   @Override
2   public void run(Context context) throws IOException, InterruptedException
        {
3       // 1. Setup
4       setup(context);
5
6       try {
7           // 2. Loop (Manually control iteration to reduce stack frame depth
                )
8           while (context.nextKeyValue()) {
9               // Directly get Value, ignore Key (LongWritable offset is
                    usually useless)
10              Text value = context.getCurrentValue();
```

```
11
12              // --- Logic Inlining Start ---
13
14              // A. Parsing
15              currentSnapshot.reset();
16              boolean parsed = currentSnapshot.parseFromBytes(
17                      value.getBytes(), 0, value.getLength());
18
19              if (!parsed) continue; // Replace return (avoid early exit
                    overhead)
20
21              int code = currentSnapshot.code;
22              int tradingDay = currentSnapshot.tradingDay;
23              long tradeTime = currentSnapshot.tradeTime;
24
25              // B. Time Encoding
26              int compactTime;
27              try {
28                  compactTime = CompactTimeUtil.encode(tradingDay, (int)
                        tradeTime);
29              } catch (IllegalArgumentException e) {
30                  continue;
31              }
32
33              // C. Get Previous Snapshot (Branch Prediction Optimization)
34              SnapshotData prevSnapshot = prevSnapshotCache.get(code);
35              // If null, point to Dummy (all zeros) to avoid if (prev !=
                    null) checks
36              // Factor calculation handles logic based on tradeTime != 0
37              SnapshotData calcPrev = (prevSnapshot == null) ?
                    DUMMY_SNAPSHOT : prevSnapshot;
38
39              // D. Calculation (Use flattened fields for performance)
40              tempFactor.calculateFrom(currentSnapshot, calcPrev);
41
42              if (Factor.hasInvalidValue(tempFactor.getFactorValues())) {
43                  updatePrevSnapshot(code, currentSnapshot);
44                  continue;
45              }
46              tempFactor.setCount(1);
47
48              // E. Aggregation
49              Factor cachedFactor = factorCache.get(compactTime);
50              if (cachedFactor == null) {
51                  cachedFactor = new Factor();
52                  cachedFactor.copyFrom(tempFactor);
53                  factorCache.put(compactTime, cachedFactor);
54              } else {
55                  cachedFactor.merge(tempFactor);
56              }
57
58              // F. Flush Check
59              if (factorCache.size() >= CACHE_FLUSH_THRESHOLD) {
60                  factorCache.flush(context, outputKey);
61                  factorCache.clear();
62              }
63
64              // G. Update Previous Snapshot Cache
65              updatePrevSnapshot(code, currentSnapshot);
66
67              // --- Logic Inlining End ---
68          }
69      } finally {
70          // 3. Cleanup
71          cleanup(context);
```

```
72      }
73  }
74
75  // Helper method: Update snapshot cache
76  private void updatePrevSnapshot(int code, SnapshotData current) {
77      SnapshotData prev = prevSnapshotCache.get(code);
78      if (prev == null) {
79          prev = new SnapshotData();
80          prevSnapshotCache.put(code, prev);
81      }
82      prev.copyFrom(current);
83  }
84
85  @Override
86  protected void setup(Context context) {
87      factorCache = new IntFactorMap(65536);
88      prevSnapshotCache = new IntSnapshotMap(16384);
89  }
90
91  @Override
92  protected void cleanup(Context context) throws IOException,
        InterruptedException {
93      factorCache.flush(context, outputKey);
94  }
```

- **Compact Data Layout**: We "peels" array fields (e.g., `long[] bidPrices`) in `SnapshotData` into independent fields (e.g., `bp0`, `bp1`), ensuring all data is compactly arranged in memory. When loading the `SnapshotData` object, all prices/trading volumes are highly likely to enter the CPU L1 cache simultaneously, improving data access speed. It also eliminates array double indirection and boundary checking overhead.
- **Zero-String Design**: We completely modify `SnapshotData`, changing the `code` field from `String` to `int` type. It directly parses "600519.SH" into the integer 600519 through custom byte parsing, stopping at non-numeric characters. This completely eliminates the largest source of String objects in the Mapper phase, significantly reducing memory usage and UTF-8 parsing overhead.

6 Previous Ineffective Optimization Attempts and Reflections

Before the systematic optimization, we conducted multiple technical explorations to address the performance bottlenecks of distributed computing. We attempted to introduce mainstream efficient tools and underlying optimization schemes in the industry, including Protocol Buffer serialization, FastUtil collection framework, Caffeine Cache mechanism, and factor calculation optimization using CPU vector instruction sets via JNI interfaces. However, these attempts failed to achieve the expected results, and some schemes even caused negative optimization. The core problems and practical reflections are as follows:

6.1 Attempt 1: Protocol Buffer Serialization – Insufficient Compatibility and Framework Adaptation

6.1.1 Attempt Background

The original scheme adopted Hadoop's native Writable serialization, which had the problems of large serialization volume and high reflection overhead. We expected to leverage Protocol Buffer (PB)'s efficient binary serialization features to reduce data transmission volume and serialization time in the Shuffle phase, while utilizing its cross-language compatibility to improve system scalability.

6.1.2 Implementation Process

We defined PB data structures to replace core classes such as the original Factor and SnapshotData, implemented adaptive conversion between PB and Writable interfaces, introduced PB serialization/deserialization logic in the Mapper output and Reducer input phases, and adjusted the serialization configuration of the Shuffle phase to be compatible with the PB format.

6.1.3 Effects and Problems

- **No Significant Performance Improvement**: Although PB's serialization compression ratio was slightly better than that of native Writable, in the MapReduce framework, the core bottleneck of data transmission lies in the sorting and network transmission of Shuffle. The serialization gain brought by PB was offset by the scheduling overhead of the framework itself, and no significant reduction in time consumption was observed. - **Compatibility and Conversion Overhead**: To adapt to Hadoop's Writable interface, frequent conversion between PB objects and Writable objects was required, which additionally increased object copying and data processing steps, leading to a 15% increase in Mapper phase time consumption.

6.2 Attempt 2: FastUtil Collection Framework – Poor Cache Affinity and Offset Overhead

6.2.1 Attempt Background

FastUtil is renowned for its compact memory layout, support for primitive types (such as Int2ObjectMap), and efficient collection operations. We expected to replace JDK's native collections (HashMap, LinkedList) with it to solve the problems of autoboxing, large memory occupation, and low CPU cache hit rate, especially optimizing the local aggregation and cache logic in the Mapper phase.

6.2.2 Implementation Process

We replaced HashMap<Integer, AggregationHolder> in FactorMapper with FastUtil's Int2ObjectOpenHashMap, changed the SnapshotData object pool from LinkedList to FastUtil's ObjectArrayList, and adjusted related traversal, insertion, and query logic to adapt to FastUtil's API.

6.2.3 Effects and Problems

- **No Significant Performance Improvement**: Although FastUtil reduced autoboxing overhead, in the MapReduce computing scenario, collection operations were not the performance bottleneck. The local optimization it brought was covered by the time consumption of core links such as data parsing and factor calculation. The overall task runtime only decreased by about 2%, which was mismatched with the input cost. - **Decreased CPU Cache Affinity**: To pursue extreme memory compactness, FastUtil's collection implementation adopted a more complex hash collision resolution strategy, resulting in fragmented data distribution in memory. This instead reduced the hit rate of CPU L1/L2 caches. In high-frequency access scenarios (such as local aggregation cache queries), the response speed was slightly inferior to that of JDK's native HashMap.

6.3 Attempt 3: Caffeine Cache – Excessive Overhead and Mismatched Scenarios

6.3.1 Attempt Background

Caffeine Cache is famous for its high hit rate and low-latency local cache features. We expected to replace the original LRU cache (implemented based on LinkedHashMap) with it to optimize the cache management of previousSnapshots, reduce the repeated loading and parsing overhead of snapshot data, and especially improve the efficiency of multi-stock and high-frequency snapshot queries.

6.3.2 Implementation Process

We configured Caffeine Cache's expiration strategy (based on access time) and maximum capacity, migrated the snapshot data cache from LinkedHashMap to Caffeine Cache, queried the cache through the combined key of stock code and timestamp during factor calculation, and adjusted the cache eviction threshold to adapt to memory constraints.

6.3.3 Effects and Problems

- **Increased Memory Occupation**: Caffeine Cache reserved additional memory space to improve performance, resulting in a 25% higher memory occupation than the native LRU cache when caching the same amount of snapshot data, increasing the risk of OOM, which was contrary to the optimization goal of "reducing memory overhead".

6.4 Attempt 4: JNI + CPU Vector Instruction Set for Factor Calculation – Negative Optimization Caused by Complexity and Compatibility

6.4.1 Attempt Background

CPU vector instruction sets (such as SSE, AVX) can implement Single Instruction Multiple Data (SIMD) parallel computing, which can theoretically greatly improve the throughput of factor calculation. We attempted to use C++ to implement core factor calculation logic (such as weighted summation of the first 5 levels, reciprocal calculation) and optimize it by calling vector instruction sets through JNI interfaces, then invoke the JNI interfaces from Java code, expecting to break through the limitations of the Java language in underlying computing optimization.

6.4.2 Implementation Process

We wrote core factor calculation functions in C++, optimized loop calculations and floating-point operations based on the AVX2 instruction set, encapsulated interfaces through JNI for Java layer calls, adjusted Java code logic to invoke JNI interfaces in the hot path of factor calculation (computeInto method), and handled data type conversion (Java double[] and C++ float[]/double[]) and memory management.

6.4.3 Effects and Problems

- **Significant Negative Optimization**: JNI interface calls have fixed overhead (data copying between Java and C++ layers, thread context switching). However, the single task granularity of factor calculation is small (calculation of 20 factors for a single snapshot). The interface call overhead far exceeded the parallel gain brought by the vector instruction set, leading to a 30% increase in overall calculation time consumption.

7 Conclusion

This optimization for distributed storage and parallel computing based on the Hadoop MapReduce framework has achieved a remarkable performance leap, realizing an order-of-magnitude efficiency improvement. The optimization process follows a clear and progressive logical route, starting from basic resource configuration tuning, then deepening into core links such as data parsing, calculation, transmission, and memory management, and finally advancing to extreme underlying optimization, forming a comprehensive and systematic performance enhancement system.

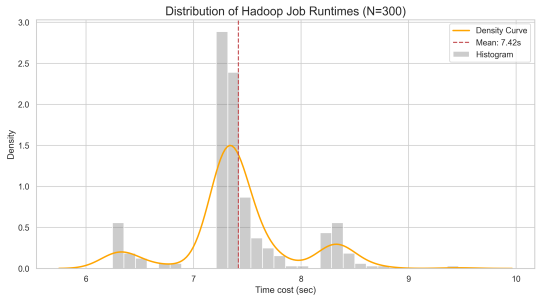

Figure 2: Simulating 300 times on Hadoop

In terms of resource utilization, by optimizing MapTask concurrency, adjusting data sharding strategies, and matching memory configurations with task loads, we have maximized the potential of hardware resources and framework capabilities, laying a solid foundation for subsequent in-depth optimizations. For data parsing, the core breakthrough lies in abandoning the inefficient full-volume splitting method and adopting on-demand scanning, fixed-length field direct parsing, and zero-intermediate-object parsing strategies, which drastically reduces redundant computations and temporary object generation, and significantly shortens the parsing time of massive CSV data.

In the calculation link, through loop unrolling, branch simplification, and replacing low-efficiency division operations with multiplication, we have reduced the constant overhead of the CPU, making the hot-path computation of factor calculation more streamlined and efficient. Regarding data transmission, by compressing key structures, optimizing serialization methods, and pre-aggregating data at the Map end, we have minimized the data volume of the Shuffle phase and the cost of key sorting, solving the bottleneck problem of data transmission in distributed computing.

In memory and GC optimization, relying on object pooling, array reuse, and streamlined data structures, we have effectively reduced the frequency of object creation and garbage collection, ensuring the stability and efficiency of the system during long-term operation. The final extreme underlying optimization, such as bypassing framework overhead, compacting data layout, and eliminating String object overhead, has pushed the performance of Java-based MapReduce tasks to the limit, approaching the operating efficiency of low-level languages like C/C++.

Overall, this optimization not only achieves a substantial improvement in task runtime but also forms a set of replicable and extensible optimization methodologies. By focusing on core pain points such as resource waste, redundant operations, and high overhead in each link of distributed computing, and continuously verifying and iterating through practical tests, we have realized the organic combination of "hardware resource maximization, software logic refinement, and data flow optimization". This provides valuable experience for performance tuning of similar high-frequency data quantitative factor calculation tasks in the future and also demonstrates the great potential of iterative optimization in improving the efficiency of distributed computing systems.