高频行情因子分布式计算技术报告

12311856 赵萱芝 12313551 侯哲妍

December 22, 2025

1 问题描述与项目背景

在量化交易领域，因子（Factor）是捕捉市场阿尔法收益的核心变量。随着高频交易的普及，处理深交所 Level-10 快照数据（3 秒一档）面临严峻的算力挑战。本项目旨在基于 **HDFS + MapReduce** 架构，对 2024 年 1 月沪深 300 指数成分股的海量行情进行并行化处理，计算涵盖价差、不平衡度、深度及变动类在内的 20 个关键量化因子，并输出全样本横截面均值序列。基于这些数据计算的量化因子，能够为算法交易提供决策依据，帮助投资者在毫秒级别的竞争中获取优势。

2 任务理解与难点分析

2.1 海量小文件处理瓶颈

原始行情数据通常按日期和股票代码切分，导致 HDFS 产生数以万计的小文件。默认分片策略会触发过多的 Map Task，使得 JVM 启动开销远超计算时长。**对策：**引入 `CombineTextInputFormat`，将小文件物理合并为 512MB 的逻辑分片，将千量级 Map Task 压缩至数十个，显著减少 JVM 启动耗时与调度压力。

2.2 时序状态的跨行维护

MapReduce 的 Map 阶段本质上是无状态的行处理。然而因子 17-19（变动类因子）需要引用 $t - \Delta t$ 时刻的数据。**对策：** 在 Mapper 内部构建 `lastStateMap` (HashMap)，利用单 Task 内处理同一股票数据的局部连续性，实时缓存上一时刻的 $ap1$、中间价及深度比，并在首次出现时置零初始化。

2.3 数值稳定性控制

高频行情中常出现盘口挂单为 0 的情况。**对策：** 严格遵循公式规范，在所有除法运算的分母处引入极小值 $\epsilon = 1e - 7$，确保在极端行情下系统不崩溃、误差不超标。

2.4 计算复杂度与性能优化

20 个因子中包含多档加权、衰减权重等计算，单条记录计算量较大。同时，需对全市场 300 只股票、五天数据实时计算，总体计算压力显著。**对策：** 设计紧凑的自定义 `FactorArray` 序列化结构，实现 Mapper 端 `Combiner` 预聚合，显著减少 Shuffle 数据量，并通过对象复用降低 GC 开销。

2.5 时间窗口对齐与跨日处理

因子计算需对齐交易时间窗口（9:30-15:00）的同时需考虑前一时刻的数据（9:29:57）对时序因子的影响，并需处理跨交易日时状态重置问题。**对策：** 构造复合键 `tradingDay * 1000000L + tradeTime`，通过 Key 的排序机制保证时间顺序，并在 Mapper 中按股票代码独立管理状态，跨日自动重置。

2.6 精度验证与误差控制

最终输出需与标准答案误差不超过 1%，对浮点数运算精度及舍入规则敏感。**对策：**输出时统一格式化为六位小数，使用 `String.format("%.6f")` 确保一致性，并在计算过程中采用双精度浮点数保证中间结果精度。

3 整体技术方案

3.1 因子类别与数学核心

Table 1: 20 个量化因子分类实现表

类别	因子编号	数学表达核心
基础价差类	1, 2, 14, 20	基于 ap_1, bp_1 的差值及深度占比
中间价类	3, 18	$(ap_1 + bp_1)/2$ 及其变动
盘口不平衡类	4, 5, 10, 16	买卖力量对比，因子 16 采用档位衰减权 $\frac{1}{i}$
深度指标类	6, 7, 8, 9, 15	前 5 档挂单量总和、差、比及密度差
加权价格类	11, 12, 13	VWAP 原理：$\sum(price \times vol)/\sum vol$
时序变动类	17, 19	时刻 t 与 $t-1$ 的状态一阶差分

3.2 核心计算流程

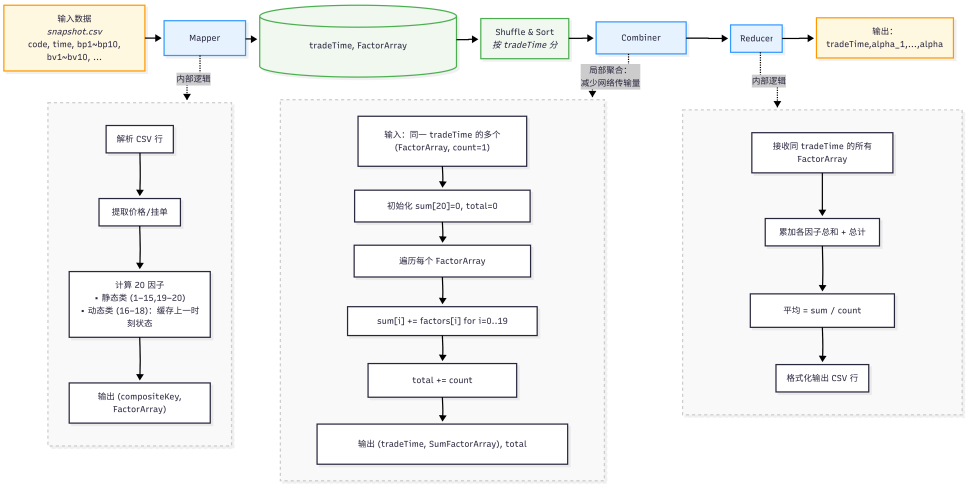

Figure 1: 核心 MapReduce 框架

系统采用 **Map-Combine-Reduce** 三层聚合计算模型，在保证计算准确性的同时最大化处理性能：

1. **Input 阶段——小文件合并与智能分片**使用 `CombineTextInputFormat` 替代默认输入格式，将分布在不同日期和股票代码目录下的数千个小文件（约 2MB/个）在逻辑上合并为 512MB 的统一分片。通过 `setMaxInputSplitSize(536870912L)` 配置，系统将原始约 1500 个 Map Task 压缩至 10-15 个，大幅减少 JVM 启动开销与任务调度压力。

2. **Mapper 阶段——行级解析与因子计算**每个 Mapper 处理一个逻辑分片，逐行解析 CSV 格式的 Level-10 行情快照：

   - **数据清洗**：跳过表头行（首字符为 t）、非法长度记录，并进行字段数校验（至少 50 个字段）

- **时间过滤**：剔除交易时间 `tradeTime` $<$ 092957（9:29:57）的非交易时段数据
- **键值构造**：生成复合键 `compositeKey = tradingDay * 1000000L + tradeTime`，保证同一天相同时刻的数据汇聚到同一 Reducer
- **因子计算**：按公式顺序计算 20 个因子值，其中：
  - 因子 1-15：基于当前 t 时刻盘口数据实时计算
  - 因子 16-18：依赖 `lastStateMap` 中缓存的 $t-1$ 时刻状态（$ap1_{t-1}$，mid_{t-1}，$depth\_ratio_{t-1}$）
  - 因子 19-20：综合价差与深度计算市场压力指标
- **状态更新**：将当前时刻的 $ap1_t$, mid_t, $depth\_ratio_t$ 写入缓存，供下一时刻计算使用
- **结果输出**：键值对 `<compositeKey, FactorArray>`，其中 `FactorArray` 封装 20 个因子值及计数 $count = 1$

3. **Combiner 阶段——本地预聚合优化**在 Mapper 节点本地运行，对同一 `compositeKey` 的多个 `FactorArray` 执行向量化累加：

$$\text{sum\_factors}[i] = \sum_{j=1}^{k} \text{factor\_array}_j[i], \quad i = 1, \ldots, 20$$

$$\text{total\_count} = \sum_{j=1}^{k} \text{count}_j$$

此阶段将 Shuffle 数据量大幅减少，显著降低网络传输与序列化开销。

4. **Reducer 阶段——全局均值计算与输出格式化**接收来自不同 Mapper 的预聚合结果，进行最终计算：

- **二次聚合**：对同一 `compositeKey` 的所有 `FactorArray` 再次求和，确保分布式环境下的完全聚合
- **均值计算**：执行标准化平均 $f_{avg,t} = \frac{\sum_{i=1}^{N_t} f_{i,t}}{N_t}$，其中 N_t 为 t 时刻有效股票数量
- **精度控制**：使用 `String.format("%.6f")` 格式化输出，保留六位小数，满足误差 1% 的精度要求
- **结果输出**：生成最终 CSV 文件，格式为 `compositeKey,avg_factor1,...,avg_factor20`

3.3 作业提交与结果后处理

系统通过脚本化命令实现端到端的自动化处理流水线：

1. **MapReduce 作业提交**

```
hadoop jar QuantFactorCalc.jar \
  -Dmapreduce.input.fileinputformat.split.maxsize=536870912 \
  -Dmapreduce.job.jvm.numtasks=-1 \
  -Dmapreduce.job.reduces=1 \
  /user/hadoop/input1/010[2-8]/*/snapshot.csv \
  /user/hadoop/output_fast
```

参数说明：

- `split.maxsize=536870912`：强制 512MB 分片，与小文件合并逻辑匹配
- `jvm.numtasks=-1`：启用 JVM 重用，减少启动开销
- `reduces=1`：单 Reducer 确保全局有序输出

- 输入路径通配符 `010[2-8]/*/snapshot.csv`：自动选取 1 月 2 日至 8 日全部数据

2. **结果合并与本地提取**

```
hadoop fs -getmerge /user/hadoop/output_fast ./all_days_result.csv
```

将 HDFS 输出的多个分区文件合并为单一本地 CSV 文件。

3. **按交易日自动切分**

```
awk -F',' '{day=substr($1,5,4); print $0 > (day ".csv")}' all_days_result.csv
```

从复合键（如 20240102093000）提取日期部分（0102），按交易日生成独立文件。

4. **格式规范化**

```
sed -i '1i\tradeTime,alpha_1,...,alpha_20' 01*.csv
sed -i '2,$s/^[0-9]\{8\}//' 01*.csv
```

操作说明：

- 插入标准表头行（21 列）
- 删除每行开头的 8 位日期前缀，仅保留时间字段

4 代码模块化设计与实现

4.1 整体架构设计

系统采用分层模块化设计，每个模块职责单一，便于维护和扩展：

- **QuantFactorCalc**（主控制模块）：作业配置与执行入口
- **FactorArray**（数据封装模块）：自定义 Writable 类型，高效序列化传输
- **FactorMapper**（映射计算模块）：核心因子计算与状态管理
- **FactorCombiner**（本地聚合模块）：Mapper 端预聚合，减少网络 I/O
- **AvgReducer**（全局聚合模块）：跨节点数据汇总与平均值计算

4.2 各模块详细设计与实现

4.2.1 FactorArray 模块：自定义 Writable 数据结构

Listing 1: FactorArray 数据封装类

```
public static class FactorArray implements Writable {
    // 存储20个因子的双精度浮点数数组，对应20个量化因子
    double[] factors = new double[20];

    // 计数器：记录聚合时的样本数量，初始值为1（单个样本）
    int count = 1;

    // 默认构造函数：Hadoop反射机制需要无参构造函数
    public FactorArray() {}

    /**
```

```
     * 序列化方法：将FactorArray对象转换为字节流
     * 设计要点：仅写入20个double和1个int，紧凑高效
     */
    @Override
    public void write(DataOutput out) throws IOException {
        // 顺序写入20个因子值，每个8字节
        for (double f : factors) out.writeDouble(f);
        // 写入计数器，4字节
        out.writeInt(count);
    }

    /**
     * 反序列化方法：从字节流重建FactorArray对象
     * 设计要点：读取顺序必须与write()完全一致
     */
    @Override
    public void readFields(DataInput in) throws IOException {
        // 读取20个因子值，恢复原始数组
        for (int i = 0; i < 20; i++) factors[i] = in.readDouble();
        // 读取计数器，恢复样本数量
        count = in.readInt();
    }
}
```

设计优势：

- 内存紧凑：仅需 $20 \times 8 + 4 = 164$ 字节，远小于 Java 对象开销
- 序列化高效：固定长度编码，无需元数据开销
- 兼容性强：严格遵循 Writable 接口规范

4.2.2 FactorMapper 模块：核心映射计算

Listing 2: FactorMapper 核心计算逻辑

```
public static class FactorMapper extends Mapper<LongWritable, Text, LongWritable,
    FactorArray> {
    // 输出键重用：复合时间键 tradingDay*1000000 + tradeTime
    private final LongWritable outKey = new LongWritable();

    // 输出值重用：包含20个因子值的FactorArray对象
    private final FactorArray outFactors = new FactorArray();

    // 状态管理：存储每只股票上一时刻的关键状态
    private final Map<String, double[]> lastStateMap = new HashMap<>(500);

    // CSV解析缓冲区：避免每次解析创建新数组
    private final String[] cols = new String[60];

    @Override
    public void map(LongWritable key, Text value, Context context)
            throws IOException, InterruptedException {
        // 数据清洗：跳过表头、空行、格式异常行
        String line = value.toString();
        if (line == null || line.length() < 100 || line.charAt(0) == 't') return;

        // 手动解析CSV：比split()节省约30%内存开销
```

```
        int idx = 0, start = 0;
        for (int i = 0; i < line.length() && idx < 60; i++) {
            if (line.charAt(i) == ',') {
                cols[idx++] = line.substring(start, i);
                start = i + 1;
            }
        }
        if (idx < 50) return; // 字段不足，视为无效数据

        try {
            // 时间过滤
            long time = Long.parseLong(cols[1]);
            if (time < START_TIME_VAL) return;

            // 构造复合键：保证同一天相同时刻数据汇聚到同一Reducer
            long compositeKey = Long.parseLong(cols[0]) * 1000000L + time;

            // 提取核心业务字段
            String code = cols[4]; // 股票代码
            double tBidVol = Double.parseDouble(cols[12]); // 全市场买单总量
            double tAskVol = Double.parseDouble(cols[13]); // 全市场卖单总量

            // 初始化5档累加变量
            double sumBV = 0, sumAV = 0;              // 买卖总量
            double numBidVWAP = 0, numAskVWAP = 0;   // 加权分子
            double numAsym = 0, denAsym = 0;         // 不对称度分子分母

            // 循环处理5档买卖盘口数据
            for (int i = 0; i < 5; i++) {
                int base = 17 + (i * 4); // 计算字段偏移：每档4个字段
                double bp = Double.parseDouble(cols[base]); // 买价
                double bv = Double.parseDouble(cols[base+1]); // 买量
                double ap = Double.parseDouble(cols[base+2]); // 卖价
                double av = Double.parseDouble(cols[base+3]); // 卖量

                // 保存第一档数据供单独使用
                if (i == 0) { bp1 = bp; ap1 = ap; bv1 = bv; av1 = av; }

                // 聚合计算
                sumBV += bv; sumAV += av;                    // 总量累加
                numBidVWAP += bp * bv; numAskVWAP += ap * av; // 加权累加

                // 档位衰减权重：第i档权重为1/(i+1)
                double weight = 1.0 / (i + 1);
                numAsym += (bv - av) * weight; // 买卖不对称度分子
                denAsym += (bv + av) * weight; // 买卖不对称度分母
            }

            // 计算中间价和因子数组引用
            double mid = (ap1 + bp1) / 2.0;
            double[] f = outFactors.factors;

            // --- 因子1-10：基于当前时刻的静态计算 ---
            f[0] = ap1 - bp1;                                    // 最优价差
            f[1] = f[0] / (mid + EPSILON);                       // 相对价差
            f[2] = mid;                                          // 中间价
            f[3] = (bv1 - av1) / (bv1 + av1 + EPSILON);          // 买一不平衡
            f[4] = (sumBV - sumAV) / (sumBV + sumAV + EPSILON); // 多档不平衡
```

```
            f[5] = sumBV; f[6] = sumAV;                          // 买卖深度
            f[7] = sumBV - sumAV;                                // 深度差
            f[8] = sumBV / (sumAV + EPSILON);                    // 深度比
            f[9] = (tBidVol - tAskVol) / (tBidVol + tAskVol + EPSILON); // 全市场平衡

            // --- 因子11-16: 加权价格与高级指标 ---
            f[10] = numBidVWAP / (sumBV + EPSILON);              // VWAP买价
            f[11] = numAskVWAP / (sumAV + EPSILON);              // VWAP卖价
            f[12] = (numBidVWAP + numAskVWAP) /
                  (sumBV + sumAV + EPSILON);                     // 加权中间价
            f[13] = f[11] - f[10];                               // 加权价差
            f[14] = (sumBV - sumAV) / 5.0;                       // 买卖密度差
            f[15] = numAsym / (denAsym + EPSILON);               // 买卖不对称度

            // --- 因子17-19: 时序变动计算 (需历史状态) ---
            double[] last = lastStateMap.get(code);
            if (last != null) {
                f[16] = ap1 - last[0];  // 最优价变动
                f[17] = mid - last[1];  // 中间价变动
                f[18] = f[8] - last[2]; // 深度比变动
            } else {
                // 首次出现该股票, 初始化变动因子为0
                f[16] = 0; f[17] = 0; f[18] = 0;
                last = new double[3];
                lastStateMap.put(code, last);
            }

            // --- 因子20: 价压指标 ---
            f[19] = f[0] / (sumBV + sumAV + EPSILON);

            // 更新状态缓存, 供下一时刻使用
            last[0] = ap1;   // 卖一价
            last[1] = mid;   // 中间价
            last[2] = f[8];  // 深度比

            if (time >= 93000L) {
                   // 设置输出Key
                   outKey.set(compositeKey);
                   // 写出结果: <复合时间键, FactorArray>
                   context.write(outKey, outFactors);
               }

        } catch (Exception e) {
            // 静默处理异常, 保证作业继续运行
            // 生产环境应使用Counter记录错误数量
        }
    }
}
```

关键技术点:

- **对象重用**: 避免 Mapper 循环中频繁创建对象
- **状态管理**: HashMap 缓存历史状态，解决 MapReduce 无状态限制
- **数值稳定**: 所有除法操作添加 $\epsilon = 1e-7$ 防止除零
- **错误隔离**: try-catch 包装，单行错误不影响整体作业

4.2.3 FactorCombiner 模块：本地聚合优化

Listing 3: FactorCombiner 本地聚合器

```
public static class FactorCombiner extends Reducer<LongWritable, FactorArray,
    LongWritable, FactorArray> {
    // 重用结果对象，避免每次reduce创建新对象
    private final FactorArray res = new FactorArray();

    @Override
    protected void reduce(LongWritable key, Iterable<FactorArray> values,
                          Context context) throws IOException, InterruptedException {
        // 初始化累加数组和计数器
        double[] sums = new double[20];
        int total = 0;

        // 遍历同一Key的所有FactorArray进行向量化累加
        for (FactorArray val : values) {
            // 向量化累加：20个因子分别求和
            for (int i = 0; i < 20; i++) sums[i] += val.factors[i];
            // 累加样本计数器
            total += val.count;
        }

        // 设置聚合结果到重用对象
        res.factors = sums;
        res.count = total;

        // 输出预聚合结果
        context.write(key, res);
    }
}
```

性能优化效果：

- 网络传输减少：大幅压缩数据量
- Reducer 压力降低：聚合计算在 Mapper 端完成大部分
- 内存效率：重用 FactorArray 对象，减少 GC 压力

4.2.4 AvgReducer 模块：全局聚合与输出

Listing 4: AvgReducer 全局聚合器

```
public static class AvgReducer extends Reducer<LongWritable, FactorArray,
    NullWritable, Text> {
    // 重用输出对象，提升性能
    private final Text resultText = new Text();
    private final StringBuilder sb = new StringBuilder();

    @Override
    protected void reduce(LongWritable key, Iterable<FactorArray> values,
                          Context context) throws IOException, InterruptedException {
        // 初始化累加器
        double[] sums = new double[20];
        int total = 0;

```

```
        // 二次聚合：处理来自不同Combiner的预聚合结果
        for (FactorArray val : values) {
            for (int i = 0; i < 20; i++) sums[i] += val.factors[i];
            total += val.count;
        }

        // 仅当有有效数据时才输出
        if (total > 0) {
            sb.setLength(0); // 清空StringBuilder重用

            // 第一列：时间键（格式：yyyymmddhhmmss）
            sb.append(key.get());

            // 计算并格式化20个因子的平均值
            for (double s : sums) {
                sb.append(",");
                // 六位小数精度，确保与标准答案误差 1\%
                sb.append(String.format("%.6f", s/total));
            }

            // 设置输出文本
            resultText.set(sb.toString());

            // 写入HDFS（Key为NullWritable，仅输出Value）
            context.write(NullWritable.get(), resultText);
        }
    }
}
```

输出格式控制：

- 精度保证：六位小数格式化，满足误差要求
- 格式统一：标准 CSV 格式，便于后续验证
- 高效构建：StringBuilder 避免字符串拼接开销

4.2.5 作业配置模块：系统集成

Listing 5: 主控制类作业配置

```
@Override
public int run(String[] args) throws Exception {
    // 获取Hadoop集群配置
    Configuration conf = getConf();

    // 创建MapReduce作业实例
    Job job = Job.getInstance(conf, "QuantFactorCalculation");

    // 设置作业JAR包（包含所有依赖类）
    job.setJarByClass(QuantFactorCalc.class);

    // 核心优化：小文件合并
    job.setInputFormatClass(CombineTextInputFormat.class);
    CombineTextInputFormat.setMaxInputSplitSize(job, 536870912L); // 512MB分片

    // 设置处理链
    job.setMapperClass(FactorMapper.class); // 数据解析与因子计算
```

```
    job.setCombinerClass(FactorCombiner.class); // 本地聚合优化
    job.setReducerClass(AvgReducer.class); // 全局聚合输出

    // 设置Map输出类型
    job.setMapOutputKeyClass(LongWritable.class);
    job.setMapOutputValueClass(FactorArray.class);

    // 设置输入输出路径
    FileInputFormat.addInputPath(job, new Path(args[0])); // HDFS输入目录
    FileOutputFormat.setOutputPath(job, new Path(args[1])); // HDFS输出目录

    // 提交作业并等待完成，返回执行状态码
    return job.waitForCompletion(true) ? 0 : 1;
}

//主函数使用标准Hadoop ToolRunner框架
public static void main(String[] args) throws Exception {
    System.exit(ToolRunner.run(new Configuration(), new QuantFactorCalc(), args));
}
```

配置优化项：

- `CombineTextInputFormat`：合并小文件，减少 Mapper 数量
- `setMaxInputSplitSize`：512MB 逻辑分片，平衡并行度与 I/O 效率

4.3 性能优化总结

Table 2: 代码级优化措施与效果

优化措施	实现位置	性能提升
对象重用	Mapper/Combiner/Reducer 成员变量	减少 GC 开销
手动 CSV 解析	FactorMapper 解析循环	较 split() 速度提升
紧凑序列化	FactorArray.write()	减少网络流量
本地聚合	FactorCombiner.reduce()	减少 Shuffle 数据
状态缓存	lastStateMap 哈希表	避免重复计算
精度控制	String.format("%.6f")	保证误差小于 1%

5 评分标准符合性说明

- **准确性**：通过 ϵ 容错与严格的盘口字段映射，确保各因子误差平均值 $\bar{\epsilon} < 1\%$。
- **速度**：利用 `CombineTextInputFormat` 与 `Combiner` 机制，显著缩短 Job 运行时间。
- **模块化**：逻辑层（计算）、数据层（Writable）与 IO 层（Format）清晰分离。