Real-Time Data Streaming for Stock Trading

System Architecture for a Low-Latency, Reliable, and Scalable Solution

Oskari Helosuo  
 School of Science  
 Aalto University  
 Espoo, Finland  
 oskari.helosuo@aalto.fi

ABSTRACT

In the fast-paced world of trading, access to high-quality data and speed are critical for gaining a competitive edge. This project presents a theoretical proof of concept (POC) for a data processing application designed to analyze real-time trading event data. The application consists of three main components: a Streaming Engine for ingesting and streaming data, a Processing Engine for analyzing the data, and a Visualization Engine for alerting users to breakout patterns. The system is built on a scalable framework to ensure low-latency, reliability, and scalability. It demonstrates the potential of combining Apache Kafka, Apache Flink, and Docker to create a robust data-analytics platform for traders.

1 Introduction

Trading is a precise part of the business world where data and speed play a critical component. A trader with access to high quality data faster than the competition has a profitable advantage. While the source data from trading networks is released simultaneously for all traders, it must be processed to reveal the insights that lead to advantageous trades. In addition, mistakes or interference in the data processing pipeline can be costly, as traders are given incorrect or misleading analytics. Hence, a resilient, low-latency, scalable data-analytics platform is a key part of any successful trading attempt.

This project features a proof of concept (POC) of a data processing application for analyzing real-time trading event data. The application comprises of a data streaming component (Streaming Engine) to ingest the trading data, a data processing component (Processing Engine) to process the ingested data and perform analysis, and a visualization component (Visualization Engine) to alert the user of breakout patterns detected in the trading data. The POC has also been built on a scalable framework, allowing the whole application to scale both horizontally or vertically to keep up with demand and workload. The goal is to demonstrate how a combination of these three components can be built to provide a reliable, low-latency, and scalable solution for traders to make key trades as fast as possible.

As a note, due to poor time management and planning, the technical implementation of this project is merely theoretical. There were many attempts to implement the system the way it is described in this report, but the entire pipeline was never able to function correctly, hence the lack of a provided GitHub repository. This report is then, for all intents and purposes, a theoretical POC of what could be done.

2 System Architecture

The POC features three distinct components, the Streaming Engine, the Processing Engine, and the Visualization Engine. Each choice when planning the systems architecture of each component has been based the following three criteria.

Criteria 1: Low latency.

Criteria 2: Reliability.

Criteria 3: Scalability.

2.1 Streaming Engine

The Streaming Engine plays a critical part in the data pipeline. It receives the data which then must be sent to the Processing Engine as quickly as possible. Latency here is measured by calculating the time difference between receiving the data from the source and passing it onwards to the Processing Engine. The data in for this application is generated in real-time and the volume is very large, around hundreds of millions per day of trading. Hence, the Engine must be able to stream large quantities of data in a continuous manner. For this application, Apache Kafka[[1]](#footnote-2) was a reasonable and well-fitting choice.

The first design decision to was to employ an Engine that can take advantage of continuous data processing for real-time data streams. As opposed to batch processing, continuous processing ensures that data is sent to the Processing Engine very quickly after it has been received by the Streaming Engine, instead of waiting for a specified number of trade events to flow in before sending the forwards. Continuous data processing is therefore lower latency and a better choice for our application.

Scalability and reliability often go hand in hand. A horizontally scalable service, meaning scaling through the addition of more nodes to the server, is also able to use its modular network to provide fault tolerance. Apache Kafka functions by distributing its workload across nodes, called brokers, over a configured network. Brokers are easy to add, meaning an increase in data throughput can be scaled according to needs. Kafka also takes advantage of its network of brokers to provide message replication, meaning that the network can be set up to retain copies of the streamed data across multiple brokers in case one or more of them go offline, increasing the application’s reliability.

Another stream processor that was considered for this project was RabbitMQ. Both Kafka and RabbitMQ provide robust fault tolerance through message replication and allow various scalability solutions, but their primary focus is slightly different. RabbitMQ is more suited for applications requiring complex message routing, whereas Kafka is focused on real-time high-output stream processing. Kafka is able to perform better in the context of this project, as it outperforms RabbitMQ in message transmission capacity. Taking advantage of sequential disk I/O, Kafka is able to send millions of messages per second[[2]](#footnote-3). In contrast, RabbitMQ performs in the range of thousands of messages per second, which is much slower and would certainly become a bottleneck in this project. As an added benefit, Kafka also integrates well with Apache’s data processing offering, Apache Flink, also used in this project.

2.2 Processing Engine

The Processing Engine is the heart of the application. It receives the data from the Streaming Engine and performs analysis to find breakout patterns that get presented using the Visualization Engine. Latency for the Processing Engine is measured by calculating the time difference between receiving the data from the Streaming Engine and passing it onwards to the Visualization Engine. The data in for this application is streamed continuously in real-time and the volume is very large. Hence, the Engine must be able to process analysis for large quantities of data in a continuous manner. For this application, Apache Flink[[3]](#footnote-4) was a reasonable and well-fitting choice.

To minimize latency across the Processing Engine, stream processing is preferred over batch processing. Flink is built with native stream processing at its core[[4]](#footnote-5) and works well with time windows, a requirement of this project. Processing in Flink is divided into tasks across a cluster, allowing the Flink network to be scaled both horizontally and vertically. Processing happens in parallel, so scaling the processing capabilities of the Engine is simple. Distribution across a cluster also provides reliability through fault-tolerance, as distributed “snapshots” capture the state of the system which allow Flink to revert to a previous state globally in case of disruption.

Apache Spark was also considered for this project. However, Spark is based on a batch processing approach to data streaming, meaning a higher latency for the use case of this project. Spark is capable of imitating stream processing with a feature called micro-batching, but its simulation of stream processing includes higher latency than the native stream support provided by Flink. In the context of this project other features, such as fault tolerance using snapshots and scaling through distributed workloads, remain quite similar between the two. However, as performance and low latency for real-time data processing is the most important component of this project, Flink is the more appropriate choice as Processing Engine. As an added benefit, Flink also integrates well with Apache’s data streaming offering, Apache Kafka, also used in this project.

2.3 Visualization Engine

For visualization and user configuration, a simple web interface is proposed as the chosen Visual Engine. The interface would be built to allow the user to select what symbols to follow and receive alerts about, as well as see those alerts. Additionally, Python would be used to plot basic graphs of recent windows for selected symbols. Data to the Visual Engine would be consumed via a sink, where the Flink instance is a producer for.

3 Implementation

Following the design principles defined during system architecture design, it is important that the application stays low-latency, reliable, and scalable. For these reasons, much of this system is built to run on the cloud using Docker containerization. Docker was used for this project, since it is one of the most popular containerization tools and is tried and tested with both Kafka and Flink. During development, the containers were run locally, but running them on a cloud service, such as Google Cloud Platform or Microsoft Azure, is entirely possible. Cloud providers promise very high reliability and uptime for their compute, which combined with the fact that the system can be ran simultaneously in multiple physical locations, adds a strong component of reliability and fault-tolerance. In addition, employing containers to run the Streaming and Processing Engines means that scaling up or down is trivial and can be done very quickly depending on the resource requirements. Purchasing more compute on the cloud and then releasing it when it is no longer necessary can be done at a moment’s notice.

In the Processing Engine, several tick and housekeeping events must take place for smooth and efficient operation. First, the incoming data from the Streaming Engine is filtered to remove invalid or empty rows from further analysis. These rows are recorded for later analysis to see what type of invalid rows can be found in the source data stream, and if anything can be done to address them. The cleanup is also done to avoid spending any resources on analyzing rows not relevant the user.

Second, data is grouped by symbol to allow for tumbling window processing. This mean that the breakout pattern recognition is computed separately and hence correctly for each symbol. The window parameters follow those highlighted in the project guidelines.

Third, the exponential moving average (EMA) is calculated for each symbol, as each window tumbles to an end. Flink divides this workload to its cluster so that the jobs can be completed in parallel.

Fourth, logs are kept of all events and calculation results. These are useful for later analysis, or in case the user wishes to include a new symbol in analysis and wants to see its historical data. Logs, however, can consume a lot of space, especially with the high quantity of events in the use case. This is why log compaction and cleanup are performed at regular intervals to vacate storage space and speed up performance. For example, logs from over 100 days ago or invalid or empty rows are not kept, as they are no longer used for calculations.

Finally, alerts are provided to the user through the Visual Engine’s web interface. Alerts are visible notifications, akin to the push notifications on mobile devices. The user can choose to react or ignore them if they wish.

4 Performance Evaluation

Due to this project never reaching a working version, all performance analysis is purely theoretical. Hence, this section of the report is not applicable.

5 Conclusion

In conclusion, developing a low-latency, reliable and scalable application for trading data streaming and analysis is possible using a combination of several free and open-source tools. Combining Docker, Apache Kafka, and Apache Flink with a cloud environment, such as Google Cloud Platform or Microsoft Azure, would result in an effective and efficient application to satisfy user needs. This project employs Kafka for streaming data, Apache for processing data, Python for plotting graphs and alerts onto a web interface, and Docker for containerization. Unfortunately, a working version of the architecture was never reached, so performance details are left to speculation and estimates based on claims from each tool used.

This project was completed alone, as a one-member team. The most important decision this team made about systems design was the core criteria on which everything was built. These three criteria: low-latency, reliability, and scalability were carefully decided and weighed and played a critical role in designing each part of the system architecture. The most challenging part of the implementation was getting everything working together. With no previous experience in either Kafka or Flink, the team made a mistake embarking on a mission to learn and implement these with limited time. Initial testing was completed using Microsoft Power Query which the team was already familiar with, and in retrospect this should have been the direction to go with to reach a working application.

A lot of mistakes were made along this project, but a lot has been learned as well. The team’s overall functionality was poor, as the person in charge of keeping the project on track and dedicating adequate time to implement learnings from the course contents did not collaborate well with the person in charge of actually implementing the features. While the team did manage to get Kafka to work, the team spent far too long attempting to implement Flink that it took time and effort out of all other aspects of the project. As a reminder, this was a solitary effort, and the project was completed as a one-member team.Conference Name:ACM Woodstock conference

Conference Short Name:WOODSTOCK’18

Conference Location:El Paso, Texas USA

ISBN:978-1-4503-0000-0/18/06

Year:2018

Date:June

Copyright Year:2018

Copyright Statement:rightsretained

DOI:10.1145/1234567890

RRH: F. Surname et al.

Price:$15.00

1. https://kafka.apache.org/intro [↑](#footnote-ref-2)
2. https://aws.amazon.com/what-is/apache-flink/ [↑](#footnote-ref-3)
3. https://flink.apache.org/what-is-flink/use-cases/ [↑](#footnote-ref-4)
4. https://www.redpanda.com/guides/event-stream-processing-flink-vs-spark [↑](#footnote-ref-5)