Apache Structured Streaming Programming
- Word Count For Streaming Data

Chris Cao

Introduction: This project addresses the problem of real-time text stream processing and propose a more scalable and fault-tolerant solution for word count streaming computation. Traditional socket-based systems, such as the one implemented using Spark Structured Streaming using a TCP connection, are limited by the transient nature, lack of buffering, and incapability of large-scale, distributed systems.

To overcome these limitations, this project replaces the socket input source with Apache Kafka, a distributed messaging queue that provides high-throughput, fault-tolerant, and durable stream ingestion. Our goal is to decouple the data producer from the stream processing engine. This enables asynchronous, parallel, and reliable communication between components.

Method: This project proposes to build a real-time word count system by combining a high-performance data ingestion system, with a robust streaming analytics backend powered by Apache Spark Structured Streaming.

- System Workflow
  - Data ingestion through TCP server
    - The server splits the input into lines and pushes each line into a Kafka topic using the IBM/samara client library. This component handles concurrency via goroutines and ensures non-blocking ingestion into Kafka.
  - Stream processing through Spark Structured Streaming
    - On the consumer end, a Spark application subscribes to the Kafka topic. Each message is parsed and tokenized into individual words. Spark then performs a groupBy + count operation over a sliding window to compute a running word count. The result is periodically output to the console or persisted to a file.
- Parameters
  - This project is expected to improve the performance by replacing socket communication with Kafka messages, through the peak cut strategy, the system will be able to handle all high-concurrent scenarios, which minimizes the possibility of missed requests.
- Interesting Point
  - For the producer end, this project uses Golang instead of Java, as it is easier to handle high-concurrent scenarios. Compared to the thread pool implementation in Java, goroutine in Go is a faster and more friendly way to implement parallelism. Meanwhile, the context switching overhead between goroutines are much more acceptable than threads.

Prompt: My current method for the project is to setup a tiny TCP server, which accepts texts input from users. Server parses each line and sends to Kafka topics. Then at Spark end, the consumer will count the words of texts from the message queue.
Elaborate on the method above.

Future Works:

- Real-time sensitive word detection:
  - if input stream has “error” or “fatal”, write information to error log
- Support for multi-topic and multi-class text analysis
  - Simultaneously listen to multiple topics in Kafka (e.g., “news”, “logs”, “chat”), respectively, to count the and compare the differences
  - Use hashing algorithm to write texts to different topics

References:

- https://github.com/IBM/sarama
- https://spark.apache.org/docs/latest/structured-streaming-programming-guide.html#quick-example
- “*Spark: Cluster Computing with Working Sets”* - Matei Zaharia