Word Count Using Spark Structured Streaming For High-Concurrency Purpose

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. In the meantime, to accelerate the step of input processing, this project utilizes Goroutine to implement a cheaper and faster method to achieve parallelism.

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 file input
    - 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.
- Parameters
  - This project is expected to improve the performance by replacing socket communication with Kafka messaging queues, through the peak cut strategy, the system will be able to handle all high-concurrent scenarios, which minimizes the possibility of missed requests.
  - Meanwhile, the project should accelerate input processing through using a N:M Goroutine model, which allows more parallelism.
- 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 user-friendly way to implement parallelism. Meanwhile, the context switching overhead between goroutines are much more acceptable than threads.

Results: Output is stored in result.txt, the processing is based on counting words of a novel.

- The project measured running time for implementation with and without Goroutines, the one with Goroutine is much faster in producing and sending messages, achieving nearly 2 times faster in average.

```
chriscao@mbookair producer % go run kafka-go
2025/05/04 19:47:33 Server Booting...
2025/05/04 19:47:33 Creating Kafka config successful
2025/05/04 19:47:33 Creating client successful
2025/05/04 19:47:33 Creating producer successful
Time Taken (Go Producer): 2742ms
chriscao@mbookair producer % go build
chriscao@mbookair producer % go run kafka-go
2025/05/04 19:48:36 Server Booting...
2025/05/04 19:48:36 Creating Kafka config successful
2025/05/04 19:48:36 Creating client successful
2025/05/04 19:48:36 Creating producer successful
Time Taken (Go Producer): 5877ms
```

Future Works:

- Handle real-time requests
  - allowing TCP transmission as input method, meanwhile, retrieve results from Kafka as well
- Real-time sensitive word detection:
  - if input stream has “error” or “fatal”, write information to error log
- Implement more reliable running time measurement
  - current implementation replies on Kafka message content “__END__” to indicate the end of input. Due to the network fluctuations and the features of goroutine, this method isn’t effective as expected.
- 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 (Kafka library)
- https://spark.apache.org/docs/latest/structured-streaming-programming-guide.html#quick-example (Spark Structured Streaming examples)
- https://github.com/WenbinZhu/mit-6.824-labs (Distributed System Implementation based on Spark)
- “*Spark: Cluster Computing with Working Sets”* - Matei Zaharia

Source Code: Code has been attached as a zip file.