BIRLA INSTITUTE OF TECHNOLOGY, MESRA, RANCHI

CA550 MINOR PROJECT REPORT

REAL-TIME STREAM PROCESSING APPLICATION USING APACHE SPARK

SUBMITTED BY:
SHAIVI SAKSHI
ROLL NO.- MCA/10025/20

SUBMITTED TO:
DR ABHIJIT MUSTAFI

INDEX

OBJECTIVE

Today's data is generated by an infinite amount of sources - IoT sensors, servers, security logs, applications, or internal/external systems. It’s almost impossible to regulate structure, data integrity, or control the volume or velocity of the data generated.

While traditional solutions are built to ingest, process, and structure data before it can be acted upon, streaming data architecture adds the ability to consume, persist to storage, enrich, and analyze data in motion.

As such, applications working with data streams will always require two main functions: storage and processing. Storage must be able to record large streams of data in a way that is sequential and consistent. Processing must be able to interact with storage, consume, analyze and run computation on the data.

This also brings up additional challenges and considerations when working with legacy databases or systems. Many platforms and tools are now available to help companies build streaming data applications.

Real-time stream processing is the process of taking action on data at the time the data is generated or published. Historically, real-time processing simply meant data was “processed as frequently as necessary for a particular use case.” But as stream processing technologies and frameworks are becoming ubiquitous, real-time stream processing now means what it says. Processing times can be measured in microseconds (one millionth of a second) rather than in hours or days.

Streaming data is data that is continuously generated and delivered rather than processed in batches or micro-batches. This is often referred to as “event data,” since each data point describes something that occurred at a given time.

Batch Processing vs Real-Time Streams

Batch data processing methods require data to be downloaded as batches before it can be processed, stored, or analyzed, whereas streaming data flows in continuously, allowing that data to be processed simultaneously, in real-time the second it's generated.

Today, data arrives naturally as never ending streams of events. This data comes in all volumes, formats, from various locations and cloud, on-premises, or hybrid cloud.

With the complexity of today's modern requirements, legacy data processing methods have become obsolete for most use cases, as it can only process data as groups of transactions collected over time. Modern organizations need to act on up-to-the-millisecond data, before the data becomes stale. This continuous data offers numerous advantages that are transforming the way businesses run.

The main objective of this project is to integrate Apache Spark with Apache kafka to implement the idea of stream processing at a small level. Real- time login records are sent through an Apache Kafka topic which is then read by Apache spark streaming application and processed in the form of micro batches. After complete execution of each micro-batch a table in the Apache Cassandra database is updated with new data in real time. This project basically elaborates the join mechanics of a stream dataframe and a static dataframe.

INTRODUCTION

APACHE SPARK STREAMING

Apache Spark Streaming is a scalable fault-tolerant streaming processing system that natively supports both batch and streaming workloads. Spark Streaming is an extension of the core Spark API that allows data engineers and data scientists to process real-time data from various sources including (but not limited to) Kafka, Flume, and Amazon Kinesis. This processed data can be pushed out to file systems, databases, and live dashboards. Its key abstraction is a Discretized Stream or, in short, a DStream, which represents a stream of data divided into small batches. DStreams are built on RDDs, Spark's core data abstraction. This allows Spark Streaming to seamlessly integrate with any other Spark components like MLlib and Spark SQL.

Spark Streaming is different from other systems that either have a processing engine designed only for streaming, or have similar batch and streaming APIs but compile internally to different engines. Spark's single execution engine and unified programming model for batch and streaming lead to some unique benefits over other traditional streaming systems.

Instead of processing the streaming data one record at a time, Spark Streaming discretizes the streaming data into tiny, sub-second micro-batches. In other words, Spark Streaming's Receivers accept data in parallel and buffer it in the memory of Spark's worker nodes. Then the latency-optimized Spark engine runs short tasks (tens of milliseconds) to process the batches and output the results to other systems. Note that unlike the traditional continuous operator model, where the computation is statically allocated to a node, Spark tasks are assigned dynamically to the workers based on the locality of the data and available resources. This enables both better load balancing and faster fault recovery, as we will illustrate next.

In addition, each batch of data is a Resilient Distributed Dataset (RDD), which is the basic abstraction of a fault-tolerant dataset in Spark. This allows the streaming data to be processed using any Spark code or library.

Four Major Aspects of Spark Streaming

- Fast recovery from failures and stragglers
- Better load balancing and resource usage
- Combining of streaming data with static datasets and interactive queries
- Native integration with advanced processing libraries (SQL, machine learning, graph processing)

APACHE KAFKA

Apache Kafka is a distributed data store optimized for ingesting and processing streaming data in real-time. Streaming data is data that is continuously generated by thousands of data sources, which typically send the data records in simultaneously. A streaming platform needs to handle this constant influx of data, and process the data sequentially and incrementally.

Kafka provides three main functions to its users:

- Publish and subscribe to streams of records
- Effectively store streams of records in the order in which records were generated
- Process streams of records in real time

Kafka is primarily used to build real-time streaming data pipelines and applications that adapt to the data streams. It combines messaging, storage, and stream processing to allow storage and analysis of both historical and real-time data.

Kafka combines two messaging models, queuing and publish-subscribe, to provide the key benefits of each to consumers. Queuing allows for data processing to be distributed across many consumer instances, making it highly scalable. However, traditional queues aren't multi-subscriber. The publish-subscribe approach is multi-subscriber, but because every message goes to every subscriber it cannot be used to distribute work across multiple worker processes. Kafka uses a partitioned log model to stitch together these two solutions. A log is an ordered sequence of records, and these logs are broken up into segments, or partitions, that correspond to different subscribers. This means that there can be multiple subscribers to the same topic and each is assigned a partition to allow for higher scalability. Finally, Kafka's model provides replayability, which allows multiple independent applications reading from data streams to work independently at their own rate.

APACHE CASSANDRA

Apache Cassandra is a distributed database management system that is built to handle large amounts of data across multiple data centers and the cloud. Key features include:

- Highly scalable
- Offers high availability
- Has no single point of failure

Written in Java, it's a NoSQL database offering many things that other NoSQL and relational databases cannot.

A NoSQL, often referred to as "not only SQL", database is one that stores and retrieves data without requiring data to be stored in tabular format. Unlike relational databases, which require a tabular format, NoSQL databases allow for unstructured data.

WORKING METHODOLOGY

In this project, apache spark, apache kafka and apache cassandra are combined to create a highly scalable and fault tolerant data pipeline for a real-time data stream.

In this project, streaming dataframe is integrated with static dataframe. It is assumed that in a bank there is a vast database with different kinds of data and some important information is stored in Apache Cassandra database.

The attributes of the Cassandra database table looks like {Login Id, Username, Last login Timestamp}.

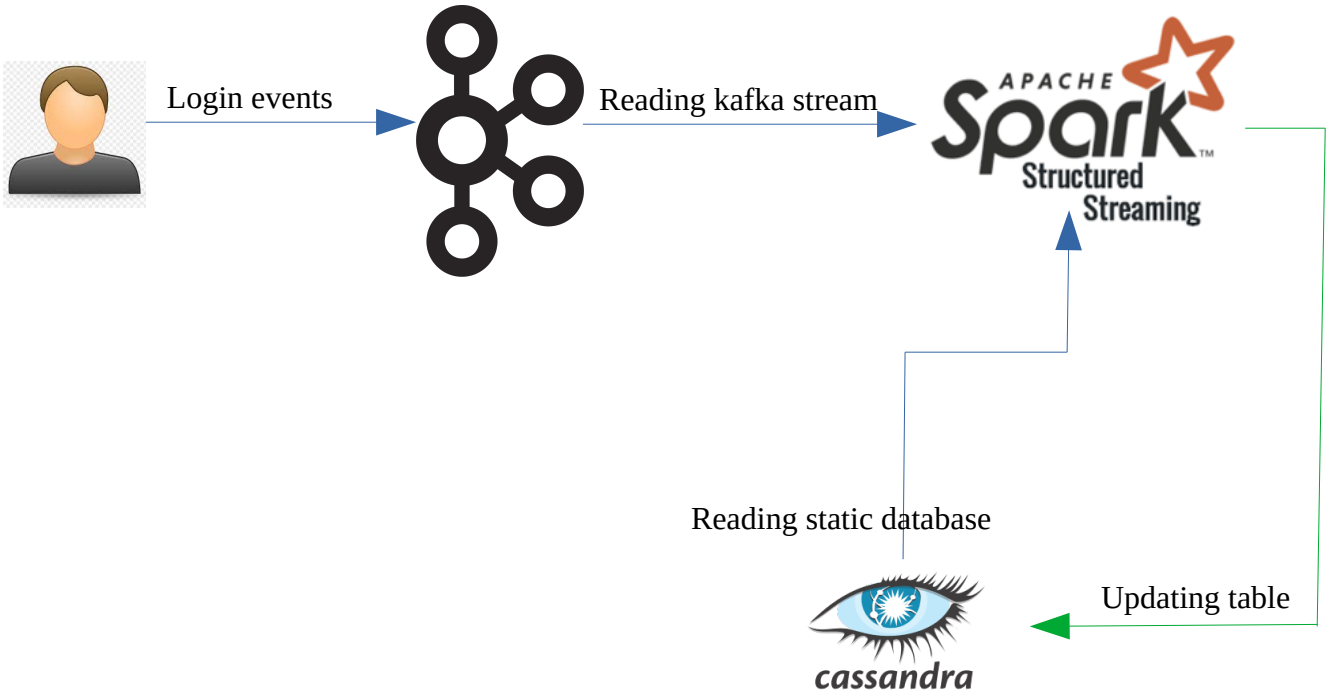
On the other side when the user logs into the banking website/app, the system is generating a login event and each event is sent to an Apache Kafka topic.

Each login event looks like {“Login Id”: “---”, “created_time”: “---”}.

The requirement is to read the login event and update the timestamp attribute in Cassandra table in real-time using spark structured streaming.

A simple approach of achieving this is by creating two dataframes in the spark application. The first dataframe is streaming dataframe for reading login events from kafka and the second dataframe is static dataframe for reading user details from Cassandra table.
Both the dataframes are joined over the common login Id in the spark application and then the older login timestamp is replaced with the new timestamp of the login event. Finally after this the information is sinked back to Cassandra table and the table is updated with new timestamp against the mentioned login Id.

The spark application is created using python. A vitrual machine instance from Google Cloud Platform is used to achieve the functionality.



HARDWARE AND SOFTWARE REQUIREMENTS

The basic hardware used is a Google Cloud compute engine instance. The various features of the virtual machine instance are:

- Operating system – Ubuntu 20.04 LTS
- Machine type – e2 standard 2
- Storage – 50 GB
- RAM – 8 GB dual core

SOFTWARE/TECHNOLOGY IN USE

- JAVA 8 or above
- Python 3.7
- Apache Spark 3.1.2
- Scala version 2.12.10
- Hadoop version 3.2
- Apache Kafka version 3.0.0
- Apache Cassandra version 4.0

INTEGRATION AND IMPLEMENTATION PROCEDURE

The first step is to create the Google cloud compute engine instance, after the virtual machine is up and running the next major step is to install the softwares that are Apache Cassandra, Apache Spark and Apache Kafka.

SPARK INSTALLATION

- Apache Spark runs on java and hadoop. Since in this project ubuntu operating system is being used, there is no need to install hadoop separately, only Java 8 or above is installed in the system.
- To install Apache Spark, spark binaries were downloaded from spark.apache.org page.
- Before running spark a few environment variables are needed to be set such as : SPARK_HOME, PYSPARK_PYTHON, PYTHONPATH (since we are building the application using python).
- We can test the installation status using the spark-shell command.

KAFKA INSTALLATION

- The first step is to download kafka binaries from the apache.kafka.org page. The version of scala associated with the kafka binary needs to be checked before installation.
- Before running Apache Kafka, the KAFKA_HOME environment variable needs to be set.
- To start running kafka , the first step is to start the zookeeper and then start the kafka server.

Now that Apache spark and Apache Kafka are installed and set, Apache Cassandra is installed on the system and a database is created under which a table is created that looks like: {Login Id, Username, Last login Timestamp}.

SPARK STREAMING APPLICATION DEVELOPMENT APPROACH

The main task of the project is to develop the spark application. A typical spark program starts with a main entry point that looks something like,

```
if __name__ == "__main__":
    spark = SparkSession \
        .builder \
        .master("spark://cassandra-ubuntu-20-04-1-vm.us-west1-b.c.spark-streaming-326814.internal:7077") \
        .appName("Stream Table Join Demo") \
        .config("spark.streaming.stopGracefullyOnShutdown", "true") \
        .config("spark.sql.shuffle.partitions", 2) \
        .config("spark.cassandra.connection.host", "localhost") \
        .config("spark.cassandra.connection.port", "9042") \
        .config("spark.sql.extensions", "com.datastax.spark.connector.CassandraSparkExtensions") \
        .config("spark.sql.catalog.lh", "com.datastax.spark.connector.datasource.CassandraCatalog") \
        .getOrCreate()
```

A spark streaming application does these three things:

- Read a streaming source- input Dataframe
- Transform-output dataframe
- Write the output-sink

A critical requirement here is that spark streaming and kafka integration is offered by a separate package called spark-sql-kafka. The package is available in public maven repository of apache spark. The maven coordinates of the required package needs to be included either in spark-conf file (included in spark binaries) or can be included with the command that is used to submit the spark job.

The main role of including these is that when the spark application is started it will automatically download the required packages and dependencies from the maven repository.

Now to read data from kafka source a dataframe is created which looks something like,

```
kafka_source_df = spark \
    .readStream \
    .format("kafka") \
    .option("kafka.bootstrap.servers", "34.127.22.67:9092") \
    .option("subscribe", "logins") \
    .option("startingOffsets", "earliest") \
    .load()
```

The dataframe obtained from the kafka stream is then transformed according to the schema defined in the spark application.

The next step is to create another dataframe by reading the static data from the Cassandra database table. The connection details of cassandra table are included in spark configs. The code for the same looks something like;

```
user_df = spark.read \
    .format("org.apache.spark.sql.cassandra") \
    .option("keyspace", "spark_db") \
    .option("table", "users") \
    .load()
```

The next step is to perform an inner join on the two dataframe on the basis of same login id and an output dataframe is created on the basis of the join. The code for the same looks like;

```
join_expr = login_df.login_id == user_df.login_id
join_type = "inner"

joined_df = login_df.join(user_df, join_expr, join_type) \
    .drop(login_df.login_id)

output_df = joined_df.select(col("login_id"), col("user_name"),
                             col("created_time").alias("last_login"))
```

Finally the output dataframe is sinked to the cassandra table so that the table is updated with new login timestamp. The code looks like;

```
output_query = output_df.writeStream \
    .foreachBatch(write_to_cassandra) \
    .outputMode("update") \
    .option("checkpointLocation", "/home/kafka/chk-point-dir") \
    .trigger(processingTime="30 second") \
    .start()
```

The spark streaming application starts running after a spark job is submitted. For that purpose, first of all spark master needs to start. Spark Master (often written standalone Master) is the resource manager for the Spark Standalone cluster to allocate the resources (CPU, Memory, Disk etc) among the Spark applications. The resources are used to run the Spark Driver and Executors.

The command for starting the master is start-master.sh

After the master is ready, worker node is created. Workers (slaves) are running Spark instances where executors live to execute tasks. They are the compute nodes in Spark. A worker receives serialized tasks that it runs in a thread pool.

The command for creating the worker node is start-worker.sh {the ip on which master is running}

The command for submitting the spark application mainly includes maven coordinates of kafka-spark integration maven repository , coordinates of cassandra-spark connectors and ip address of master node.

The spark-submit command looks like:

spark-submit --packages org.apache.spark:spark-sql-kafka-0-10_2.12:3.1.2,com.datastax.spark:spark-cassandra-connector_2.12:3.1.0 --master spark://cassandra-ubuntu-20-04-1-vm.us-west1-b.c.spark-streaming-326814.internal:7077 --deploy-mode client stream.py

where stream.py in the name of the spark streaming python file.

STARTING THE KAFKA CLUSTER

To start the kafka stream , first of all zookeeper is started. Kafka uses ZooKeeper to manage the cluster. ZooKeeper is used to coordinate the brokers/cluster topology.

The command for starting the zookeeper is:

```
bin/zookeeper-server-start.sh config/zookeeper.properties
```

After the zookeeper is running , the kafka server is started. A Kafka server, a Kafka broker and a Kafka node all refer to the same concept and are synonyms (see the scaladoc of KafkaServer). ... A Kafka broker receives messages from producers and stores them on disk keyed by a unique offset. A Kafka broker allows consumers to fetch messages by topic, partition and offset.

The command for starting the kafka server is :

```
bin/kafka-server-start.sh config/server.properties
```

Now both zookeeper and kafka server are running, the next step is to create a kafka topic to which the spark application will subscribe.

The command for creating a kafka topic named “logins” is

```
bin/kafka-topics.sh --create --topic logins --bootstrap-server 34.127.22.67:9092 --replication-factor 1 --partitions 1
```

The bootstrap server ip is the ip address where the server is running. Replication factor defines the number of copies of a topic in a Kafka cluster. A partition is the smallest storage unit that holds a subset of records owned by a topic.

After creating the topic , a kafka producer of that topic is created. Kafka producers are the publishers responsible for writing records to topics. The login events of customers are sent to producer which is then written in kafka topics.

The command for creating kafka producer is:

```
bin/kafka-console-producer.sh --topic logins –bootstrap-server 34.127.22.67:9092
```

SETTING UP CASSANDRA DATABASE

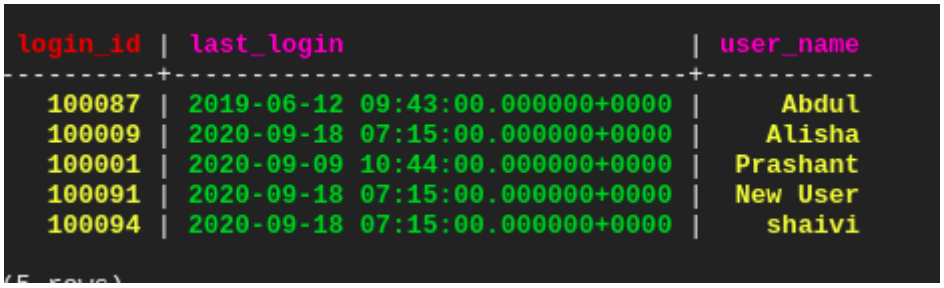
At first a keyspace in Cassandra database is created. A keyspace is a data container in Cassandra, similar to a database in relational database management systems (RDBMS). Keyspaces are entirely separate entities, and the data they contain is unrelated to each other.

The command for creating the keyspace is:

```
CREATE KEYSPACE spark_db WITH replication = {‘class’: ‘SimpleStrategy’ , ‘replication-factor’ : 1}
```

Under the keyspace spark_db, a table named ‘users’ is created.

The table looks like:



login_id	last_login	user_name
100087	2019-06-12 09:43:00.000000+0000	Abdul
100009	2020-09-18 07:15:00.000000+0000	Alisha
100001	2020-09-09 10:44:00.000000+0000	Prashant
100091	2020-09-18 07:15:00.000000+0000	New User
100094	2020-09-18 07:15:00.000000+0000	shaivi

OUTCOME

After the spark-submit command, the spark application starts running. The trigger time set in the application code is ‘30 seconds’ , so spark runs microbatches at an interval of 30 seconds. The trigger time can be reduced to 5 seconds also to obtain near real-time features.

The spark submit command deploy the spark job in ‘client’ mode which means the driver runs locally from where the application is submitted.

Once a Spark submit is done, a driver program is launched and this requests for resources to the standalone cluster manager. The entire resource allocation and the tracking of the jobs and tasks are performed by the cluster manager. Meanwhile, the tasks are split into various subtasks internally and each worker node is given the subtasks and they are all monitored by the standalone cluster manager.

After execution of each micro-batch, the table in the Cassandra database is updated. Cassandra table is being re-read in each micro-batch.

Spark maintains a checkpoint directory which maintains the records of starting and ending of each micro-batch. So, in case of any failures, spark consults the checkpoint directory automatically and resumes the processing. The checkpoint directory is also used to retain states between batches of data being processed.

The spark streaming application code as a whole is :

```
from pyspark.sql import SparkSession
from pyspark.sql.functions import from_json, col, to_timestamp
from pyspark.sql.types import StructType, StructField, StringType

from lib.logger import Log4j


def write_to_cassandra(target_df, batch_id):
    target_df.write \
        .format("org.apache.spark.sql.cassandra") \
        .option("keyspace", "spark_db") \
        .option("table", "users") \
        .mode("append") \
        .save()
    target_df.show()


if __name__ == "__main__":
    spark = SparkSession \
        .builder \
        .master("spark://cassandra-ubuntu-20-04-1-vm.us-west1-b.c.spark-streaming-326814.internal:7077") \
        .appName("Stream Table Join Demo") \
        .config("spark.streaming.stopGracefullyOnShutdown", "true") \
        .config("spark.sql.shuffle.partitions", 2) \
        .config("spark.cassandra.connection.host", "localhost") \
        .config("spark.cassandra.connection.port", "9042") \
        .config("spark.sql.extensions", "com.datastax.spark.connector.CassandraSparkExtensions") \
        .config("spark.sql.catalog.lh", "com.datastax.spark.connector.datasource.CassandraCatalog") \
        .getOrCreate()

    logger = Log4j(spark)

    login_schema = StructType([
        StructField("created_time", StringType()),
        StructField("login_id", StringType())
    ])

    kafka_source_df = spark \
        .readStream \
        .format("kafka") \
        .option("kafka.bootstrap.servers", "34.127.22.67:9092") \
        .option("subscribe", "logins") \
        .option("startingOffsets", "earliest") \
        .load()

    value_df = kafka_source_df.select(from_json(col("value").cast("string"), login_schema).alias("value"))

    login_df = value_df.select("value.*") \
        .withColumn("created_time", to_timestamp(col("created_time"), "yyyy-MM-dd HH:mm:ss"))

    user_df = spark.read \
        .format("org.apache.spark.sql.cassandra") \
        .option("keyspace", "spark_db") \
        .option("table", "users") \
        .load()

    join_expr = login_df.login_id == user_df.login_id
    join_type = "inner"

    joined_df = login_df.join(user_df, join_expr, join_type) \
        .drop(login_df.login_id)

    output_df = joined_df.select(col("login_id"), col("user_name"),
                                 col("created_time").alias("last_login"))

    output_query = output_df.writeStream \
        .foreachBatch(write_to_cassandra) \
        .outputMode("update") \
        .option("checkpointLocation", "/home/kafka/chk-point-dir") \
        .trigger(processingTime="30 second") \
        .start()

    logger.info("Waiting for Query")
    output_query.awaitTermination()
```

CONCLUSION AND FUTURE ASPECTS

The project tried to implement spark streaming integration with kafka. The project typically aimed at creating a data pipeline between a streaming data source with a static datasource. The streaming dataframe and the static dataframe were joined over identical login id. For streaming datasource Kafka stream was used and for static datasource Cassandra database was used.

Most of the time these joins are used for stream enrichment, for example, if we have streaming records about a user and we want to pull in some more information about the user from some other static tables. So, in that case we can join the two datasource get desired results.

Thus, Spark Streaming and Kafka integration can offer some pretty robust features for data streaming requirements. It can enhance the scalability, fault-tolerance and other features for an optimized stream processing environment. Spark streaming has packages for spark-kafka integration, it has also got Cassandra connectors and many more functionality which can be leveraged to process data and generate insights.

REFERENCES

1. https://databricks.com/glossary/what-is-spark-streaming
2. https://databricks.com/blog/2015/07/30/diving-into-apache-spark-streamings-execution-model.html
3. https://aws.amazon.com/msk/what-is-kafka/
4. https://www.bmc.com/blogs/apache-cassandra-introduction/
5. https://spark.apache.org/docs/2.2.0/streaming-kafka-0-10-integration.html
6. https://www.baeldung.com/kafka-spark-data-pipeline
7. https://spark.apache.org/docs/latest/streaming-programming-guide.html
8. https://towardsdatascience.com/connecting-the-dots-python-spark-and-kafka-19e6beba6404