Big Data Technologies

Agenda

- Spark Streaming
- Apache Kafka

Spark Streaming Frameworks

- Spark is originally designed for micro-batch processing.
- Spark Streaming APIs
  - Spark DStream
  - Spark Structured Streaming

Spark DStream

- Micro-batches of RDDs (Small RDDs).
- Developed in 2012. Most popular Streaming framework in 2016.
- RDD based programming

```
from pyspark import SparkConf, SparkContext
from pyspark.streaming import StreamingContext

conf = SparkConf()\
    .setMaster('local[2]')\
    .setAppName('StreamingWordCount')

sc = SparkContext(conf=conf)

# stream processing with micro-batch duration = 10 seconds
ssc = StreamingContext(sc, batchDuration=10)
```

```
# define the source = socket client (that receive data from server)
stream = ssc.socketTextStream("localhost", 4444)

# stream operations (transformations)
result = stream.map(lambda line: line.lower())\
    .flatMap(lambda line: line.split())\
    .map(lambda word: (word, 1))\
    .reduceByKey(lambda a,b: a + b)

# define the sink = console
result.pprint()

# streaming action
ssc.start()

# wait for stream processing to execute
print('waiting for termination...')
ssc.awaitTermination()

# stop streaming and spark context
ssc.stop(True)
```

- Limitations
  - Based on RDDs (in JVM). Not efficient in Python.
  - No support for event time processing.
  - Only micro-batch processing.
- Examples
  - Twitter stream

Spark Structured Streaming

- Developed in 2016.
  - Spark Dataframe is wrapper on Spark RDD.
  - Spark Structured Streaming is NOT wrapper on Spark DStreams.

    - Spark Structured Streaming a new framework developed from scratch.
- Stable in Spark 2.2.
- Spark Structured Streaming is based on dataframes.
- Works seamlessly with other Spark APIs i.e. Spark SQL & Spark ML.
- Spark Structured Streaming data is considered as infinite Dataframe i.e. new data gets appended at the dataframe. Hence most of dataframe operations are applicable on structured streaming.
- Advantages
    - Optimized (Catalyst engine)
    - Event time processing is supported
    - Support for continuous processing
    - Same query/code works for batch processing & stream processing
    - Exactly once processing mode is available
    - Fault tolerance

QuickStart -- Example

```python
from pyspark.sql import SparkSession

spark = SparkSession.builder\
            .appName('demo01')\
            .getOrCreate()

# source of the data
data = spark.readStream\
            .format('socket')\
            .option('host', '127.0.0.1')\
            .option('port', '4444')\
            .load()

data.printSchema()

# processing on the data
```

```
result = data\
            .selectExpr("EXPLODE(SPLIT(LOWER(value), '[^a-z]')) AS word")\
            .where("word NOT IN ('', 'you', 'is', 'are', 'a', 'the', 'shall', 'for', 'of')")

# sink data
query = result.writeStream\
            .trigger(processingTime='10 seconds')\
            .format('console')\
            .outputMode('append')\
            .start()

query.awaitTermination()

spark.stop()
```

Transformations

- Transformations are same as spark dataframe.
- Few transformations are not yet supported.
- e.g. select(), where(), withColumn(), groupBy().agg(), etc.

Actions

- Action is starting the stream.
- e.g. writeStream.start()

Output modes

- Every mode is not supported for every type of query.
- **append** -- only if no aggregations in the processing (without watermark).
    - result is processed and appended to the sink.
    - mainly used for data cleaning/filtering.
    - e.g. twitter tweets -- sentiment analysis i.e. tweet --> score.

- **complete** -- works with aggregation operations.
  - the full aggregate result is displayed each time.
  - e.g. live poll with 4 options -- live counting -- final result is not too big.
- **update** -- works with or without aggregation operations.
  - only updated/modified aggregate result is displayed each time.
  - e.g. monitor cabs (GPS) movement -- only modified results to be displayed.

Input sources

- socket -- server socket -- used only for testing (not in production)
- files (csv, json, ...) -- get data from files in given directory
- flume -- hadoop application to get the live data (like tweets, sensors, ...)
- kinesis -- AWS service (mainly used for IoT sensors data collection)
- kafka -- message broker
- rate -- auto-generated test data for testing

Output sinks

- console -- used only for testing
- memory -- output is hold in memory and can be queried -- for testing
- files (csv, json, ...) -- output is written into the files
- flume -- to write the live data to hadoop
- kafka -- message broker
- foreach -- custom sink (user defined processing)

Triggers

- By default, micro-batches are processed one after another.
- Trigger can specify time duration after which each batch is to be processed.
  - .trigger(processingTime='10 seconds') -- micro-batch processing - 10 second batch
  - .trigger(continuous='1 seconds') -- continuous processing
  - .trigger(once=True) -- execute only once like batch processing

Event time processing

- Time at which event is generated at source, is "event time".
- Can process out-of-order data.
- Watermark feature is used to define for how much time data should be considered (how much time should wait before processing data).

Apache Kafka

- Kafka is a distributed messaging system.
- Developed at LinkedIn and open sourced in 2011.
- Developed in Scala.
- Used by LinkedIn, Twitter, Uber, Yahoo, Airbnb, …
- References
    - https://www.slideshare.net/jhols1/kafka-atlmeetuppublicv2
    - https://www.slideshare.net/AkashVacher/kafka-meetup-ppt-1

Advantages

- High throughput
- Disk based
- Durable
- Scalable
- Low latency
- Finite retention
- Strong ordering
- Exact once delivery

Applications

- Stream processing
- Log processing
- Notifications
- Customer tracking

Kafka Terminologies

- Broker
  - Each node in kafka cluster is called as "kafka broker".
  - Each broker have its own id (configured in $KAFKA_HOME/config/server.properties).
  - Broker is JVM process -- Kafka -- running on port 9092.
- Topic
  - The messages in kafka are logically divided under topics.
  - The topics are configured (e.g. replication, ...) as per need of application.
- Partitions
  - Topics are physically divided under partitions.
  - The messages data is actually stored into log files (.log) for each partition under /tmp/kafka-logs (default directory).
  - Each partition each message is numbered (indexed) from 0,1,2,... called as "offsets".
  - Partitions have replication factor. If replication factor is 3, partition data will be copied on two more brokers.
  - In case of replication of partitions, one of the partition is leader while others will be followers/replicas. These replicas are always try to be in sync with leader.
  - The replicas which are not too behind the leader and said to "in-sync replicas" (ISRs).
- Producer
  - Client process of kafka which send messages to the broker, is said to be producer.
  - These client processes can be implemented in Java, Python, C/C++, ...
  - Producer publish data to the topics, which is sent to partitions, so that partitions are load balanced. It may use round-robin algorithm or may do key-based division.
  - Producer always write data to the leader of the partition.
- Consumer
  - Client process of kafka which receive messages from the broker, is said to be consumer.
  - These client processes can be implemented in Java, Python, C/C++, ...
  - Clients subscribe to the kafka topics.
  - It will read the data from any of the replica, which have data required for that client.
  - The data in a partition is guaranteed to be received in sequence (of writing). However data in different partitions is not guaranteed to read in sequence of writing.
- Consumer groups
  - Each topic can have one or more consumers -- divided into one or more consumer groups.

- Consumer group -- kafka ensure than only one consumer from a consumer group will get a message.
- Example 1: Three consumer groups as follows:
  - G1 -- C1a, C1b, C1c
  - G2 -- C2a, C2b
  - G3 -- C3a, C3b, C3c, C3d
  - If message "m0" is received into kafka, it will be received by exactly one consumer from each group.
- Example 2 (Broadcast system): Three consumer groups as follows:
  - G1 -- C1
  - G2 -- C2
  - G3 -- C3
  - Each message is received by each consumer.
- Example 3: One consumer group as follows:
  - G1 -- C1a, C1b, C1c
  - Each message is received by only one consumer.
  - Parallel processing of records.

Kafka Installation and Usage

- Download and extract Kafka.
- In ~/.bashrc
  - export KAFKA_HOME=/path/to/kafka
  - export PATH=$KAFKA_HOME/bin:$PATH
- In $KAFKA_HOME/config/server.properties
  - Uncomment line -> listeners=PLAINTEXT://:9092
- Start Kafka
  - terminal> zookeeper-server-start.sh $KAFKA_HOME/config/zookeeper.properties
  - terminal> kafka-server-start.sh $KAFKA_HOME/config/server.properties
- Using Kafka
  - Create topic
    - terminal> kafka-topics.sh --zookeeper localhost:2181 --create --topic test --replication-factor 1 --partitions 2
    - Verify the log directory: terminal> ls /tmp/kafka-logs
  - List topics

        - terminal> kafka-topics.sh --zookeeper localhost:2181 --list
    - Write to topic (producer)
        - terminal> echo "one" | kafka-console-producer.sh --broker-list localhost:9092 --topic test
    - Read from topic (consumer)
        - terminal> kafka-console-consumer.sh --bootstrap-server localhost:9092 --topic test
        - terminal> kafka-console-consumer.sh --bootstrap-server localhost:9092 --partition 0 --offset 2 --topic test
        - terminal> kafka-console-consumer.sh --bootstrap-server localhost:9092 --from-beginning --topic test
    - Consumer Groups
        - terminal> kafka-console-consumer.sh --bootstrap-server localhost:9092 --topic test --group 1
        - terminal> kafka-console-consumer.sh --bootstrap-server localhost:9092 --topic test --group 1
        - terminal> kafka-console-consumer.sh --bootstrap-server localhost:9092 --topic test --group 2

Kafka Python Clients

- step 1

```
pip3 install kafka-python

# python 3.12
pip3 install kafka-python-ng
```

- step 2: create new topic

```
kafka-topics.sh --zookeeper localhost:2181 --create --topic iot --replication-factor 1 --partitions 2

kafka-topics.sh --zookeeper localhost:2181 --list
```

- step 3: Implement kafka producer in python
    - Demo: Python App (Random values) --> Kafka Topic
- step 4: Testing consumer

```
kafka-console-consumer.sh --bootstrap-server localhost:9092 --topic iot
```

- step 5: Implement kafka consumer in python
    - Demo: Kafka Topic --> Python App (& print)

Uber Big Data Architecture

- https://www.uber.com/en-IN/blog/uber-big-data-platform/