Big Data Streaming with Spark

Ankita Bansal, Roopal Jain and Kanika Modi

Abstract A stream is defined as continuously arriving unbounded data. Analytics of such real-time data has become an utmost necessity. This evolution required a technology capable of efficient computing of data distributed over several clusters. Current parallelized streaming systems lacked consistency, faced difficulty in combining historical data with streaming data, and handling slow nodes. These needs resulted in the birth of Apache Spark API that provides a framework which enables such scalable, error tolerant streaming with high throughput. This chapter introduces many concepts associated with Spark Streaming, including a discussion of supported operations. Finally, two other important platforms and their integration with Spark, namely Apache Kafka and Amazon Kinesis are explored.

Keywords Spark streaming · D-Streams · RDD · Operations · Structured streaming · Kafka · Kinesis · Pipeline · Integration with spark

1 Overview

Big data is defined for enormous complex data sets that require more sophisticated techniques apart from traditional data computing technologies. Big data and its challenges can be easily understood by five Vs: Volume, Velocity, Veracity, Variety, and Value. Big data is generated rapidly and is highly unstructured. This raw data is useless until it is converted into beneficial insights. There are two ways to process this

A. Bansal (✉)
Department of Information Technology, Netaji Subhash Institute of Technology, New Delhi, India
e-mail: ankita.bansal06@gmail.com

R. Jain · K. Modi
Department of Computer Engineering, Netaji Subhash Institute of Technology, New Delhi, India
e-mail: roopal96@gmail.com

K. Modi
e-mail: 96kanu@gmail.com

© Springer Nature Singapore Pte Ltd. 2019

M. Mittal et al. (eds.), *Big Data Processing Using Spark in Cloud*,
Studies in Big Data 43, https://doi.org/10.1007/978-981-13-0550-4_2

data: Batch Processing and Streaming (or stream processing). In batch processing, the data collected over time is processed in batches, whereas processing is done in real time under stream processing.
Chapter Objectives

- To introduce streaming in Spark, its basic fundamentals
- Illustrating architecture of Spark
- To explore several types of operations supported on Spark API
- To describe various data sources where data can be ingested from
- To explore Kafka and Kinesis and their integration with spark
- To build a real-time streaming pipeline.

1.1 Streaming

A gigantic amount of data created by thousands of sensors and sending those data records simultaneously are defined as streaming data. This data requires processing on a record-by-record basis to draw valuable information. The analytics can be filtered, correlated, sampled, or aggregated. This analysis is in a form useful for various business and consumer aspects. For example, the industry can track most popular products amongst consumers based on corresponding comments and likes on social streams or can track sentiments based on some incidents, and take timely intuitive measures. Over the time, steam processing algorithms are applied to further refine the insights.

1.2 Real-Time Use Cases

Several interesting use cases include:

- Stock price movements are tracked in the real time to evaluate risks and portfolios are automatically balanced
- Improving content on websites through streaming, recording, computing, and enhancing data with users' preferences, to provide relevant recommendations and improved experiences
- Online gaming platforms collect real-time data about game-player intercommunication and analyze this data to provide dynamic actions and stimulus to increase engagement
- Streams to find the most trending topic/article on social media platforms, news websites
- Track recent modifications on Wikipedia
- View traffic streaming through networks

1.3 Why Apache Spark

Spark Streaming is becoming a popular choice for implementing data analytic solutions for the Internet of Things (IoT) sensors [1, 2]. Some of the benefits of using Apache Spark Streaming include:

- Improved usage of resources and balancing of load over traditional techniques
- It recovers quickly from stragglers and failures
- The streaming data can be combined with interactive queries and static data
- Spark is 100 times faster than MapReduce for batch processing
- Spark allows integration with advanced libraries like GraphX, Machine Learning, and SQL for distributed computing

Some of the most interesting use cases of Spark Streaming in the real life are:

1. Pinterest: Startup that provides tool for visual bookmarking uses Apache Kafka and Spark Streaming to gain insights about user engagement around the world
2. Uber: Employs Spark Streaming to collect terabytes of data from continuous streaming ETL pipeline of mobile users
3. Netflix: Uses Spark Streaming and Kafka to construct real-time data monitoring solution and online movie recommendation that processes billions of records each day from various sources
4. To find threats in real-time security intelligence operations
5. Triggers: To detect anomalous behavior in real time

1.4 Cloud-Based Apache Spark Platform

The increasing popularity of Apache Spark is leading to emergence of solutions built on cloud around Apache Spark:

- Hadoop based platforms provide support for Spark additionally to MapReduce [3, 4].
- Microsoft had added Spark support to its cloud-hosted version of Hadoop.
- Amazon Elastic Compute Cloud (EC2) can run Spark applications in Java, Python, and Scala [5].

2 Basic Concepts

This section covers fundamentals of Spark Streaming, its architecture, and various data sources from which data can be imported. Several transformation API methods available in Spark Streaming that are useful for computing data streams are described. These include operations similar to those available in Spark RDD API and Streaming Context. Various other operations like DataFrame, SQL, and machine learning

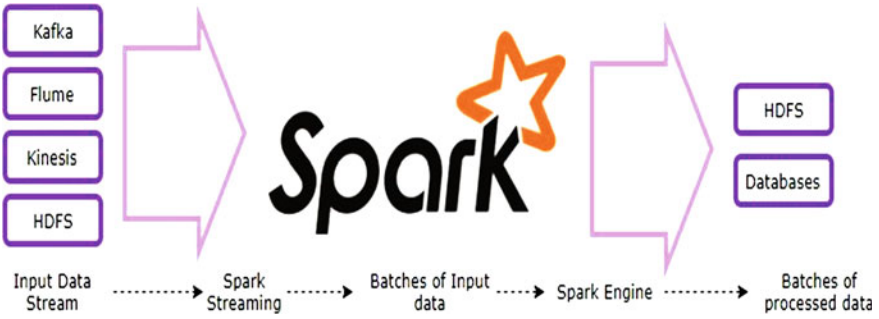

Fig. 1 Spark streaming

algorithms which can be applied on streaming data are also discussed. It also covers accumulators, broadcast variables, and checkpoints. This section concludes with a discussion of steps indispensable in a prototypical Spark Streaming program [6–9].

2.1 D-Streams

Earlier computational models for distributed streaming were less consistent, low level, and lacked fault recovery. A new programming model was introduced known as discretized streams (D-streams) that provide efficient fault recovery, better consistency, and high-level functional APIs [10–12]. Spark Streaming converts input live stream into batches which are later processed by Spark engine to produce output in batches. D-streams are a high-level abstraction provided by Spark Streaming. A D-Stream represents a continuous stream of data which supports a new recovery mechanism that enhances throughput over traditional duplication mitigates stragglers and parallel recovery of lost state.

Discretized streams are a sequence of partitioned datasets (RDDs) that are immutable and allow deterministic operations to produce new D-streams.

D-streams execute computations as a series of short, stateless, deterministic tasks. Across different tasks, states are represented as Resilient Distributed Datasets (RDDs) which are fault-tolerant data structures [13]. Streaming computations are done as a "series of deterministic batch computations on discrete time intervals".

The data received in every timestretch forms input dataset for that stretch and is stored in clusters. After the batch interval is completed, operations like map, reduce, and groupBy are applied on the dataset is processed to produce new datasets. The newly processed dataset can be a transitional state or a program outputs as shown in Fig. 1. These results are stored in RDDs that avoid replication and offer fast storage recovery (Fig. 2).

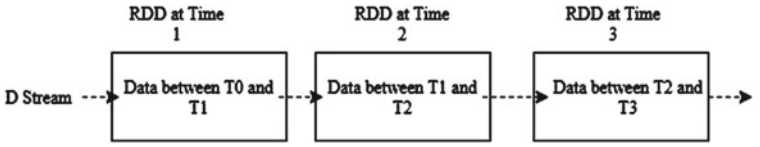

Fig. 2 Lineage graph for RDD

The following example will illustrate the idea of Spark Streaming. The program receives a stream from a server connected on a TCP socket and counts number of words in text data.

```
import org.apache.spark.streaming._
import org.apache.spark._
import org.apache.spark.storage._

# Create a local StreamingContext with two working thread and batch inter-
val of 1 second
val sparkContext = new SparkConf().setMaster("local[2]").setAppName(“
"WordCount")
batchIntervalSeconds = 1
streamingContext = StreamingContext(sc, batchIntervalSeconds)

val dStream = streamingContext.socketTextStream("localhost",8888)

val wordsDataset = dStream.flatMap(_.split(“ ”))
val pairDataset = wordsDataset.map(wordsDataset => (word, 1))
val wordCountsDataset = pairDataset.reduceByKey(_+_)

wordCountsDataset.print()

streamingContext.start()
streamingContext.awaitTermination()
```

2.1.1 Input D-Streams and Receivers

D-Streams that store stream of input data from streaming sources are known as Input D-Streams.An object that is processed in Spark's memory and stores data received from a streaming source is known as a Receiver.

Each input D-Streams is associated with a Receiver. For each data stream, a receiver is created and which runs on a worker node. Spark Streaming allows parallel processing of data streams by connecting to multiple data streams [4]. Receivers are further categorized as:

- Reliable receiver: It sends an acknowledgment to the sender (reliable data source) on receiving data and replicates it in Spark storage.
- Unreliable receiver: Unlike reliable receiver, it does not send any acknowledgment to a data source but replicates in Spark storage.

Selection of a receiver is done on the basis of chosen data sources and complexity of acknowledgment.

2.2 Data Sources

There are two kinds of built-in streaming sources in Spark Streaming:

- Basic Sources: StreamingContext API provides methods to create D-Streams from directly available sources. Only Spark Streaming library needs to be linked to the streaming application. These data sources are:
  - TCP Socket: Allows reading of data from a TCP source, hostname, and port need to be mentioned.
  - File Streams: Useful for reading files on file systems accordant to HDFS API (i.e., NFS, S3, HDFS, etc.). Files must not be altered once moved, as the newly appended data is not read.
  - Queue of RDDs: D-Streams can be created based on a queue of RDDs. Every RDD in the queue is processed and considered as a batch of data in D-Streams.
- Advanced Sources: Sources that are accessible via additional utility classes and require linking. These sources are not available in the shell. Corresponding dependencies are required to use them in Spark shell.
  - Some of the advanced sources are Apache Kafka [14], Amazon Kinesis [15], Twitter, Apache Flume, etc.
- Custom Resources: Implementation of the user-defined receiver is required.

2.3 Importing Data

In this section, an approach to obtain and export data in a production environment is discussed. Apache Spark was created to handle large volumes of data. There are two ways of batch importing data into Spark:

1. Batch Data Import: The entire dataset is loaded at once either from files or databases.
2. Streaming Data Import: Data is obtained continuously from streams.

2.3.1 Batch Import

In batch import, data is loaded from all the files at once. In production systems, Spark can be used to compute a batch job at fixed time intervals to process and publish statistics. Batch importing covers:

- Importing from files: The data should be available to all the machines on the cluster in order to facilitate batch import. Files only available on one worker node are not readable to others will cause failures. Following file systems are popular choices for massive datasets:
  - NFS or network file system assures that each of the machines can access the same files without replication. NFS is not fault-tolerant.
  - HDFS (Hadoop Distributed File System) gives every machine on the cluster access to files. The files are stored as a sequence of files and to make the system fault-tolerant, replication of blocks of a file is done.
  - S3 (Simple Storage Service) is an Amazon Web Services (AWS) solution to store large files in the cloud.

Example: A sales record for some product is collected time to time by a company using S3 file system to store data. In a similar manner, data is collected for all the products. The company needs to know revenue generated each day. Thus the solution is to run Spark jobs daily on batch created by importing data for that particular day.

2.3.2 Streaming Import

Streaming has replaced the need for daily batch jobs that used to run on servers. There are two ways for stream importing in Spark:

- Built-In methods for Streaming Import: StreamingContext provides several built-in methods for importing data.
  - socketTextStream is used to read data from TCP sockets
  - textFileStream reads data from any Hadoop Compatible FileSystem Directory
- Kafka: In socket streaming, the files are copied after HTTP request. This ensures auto-refresh but is not perfectly real time. Kafka is a high efficiency distributed system that sends data immediately.
- Kinesis
- Twitter.

2.4 Supported Operations

This section covers various operations that are supported by streaming data. These include transformations, window operations, output operations, and lastly Spark DataFrame Spark SQL operations.

2.4.1 Transformation Operations on D-Streams

These operations allow modifications on the data in the input D-Streams. Although D-Streams functions RDD methods have same name but they are different (Table 1).

2.4.2 Window Operations

Spark Streaming allows transformations to be applied in sliding window manner. These computations are known as windowed computations. Window Operations

Table 1 Some of the transformation operations allowed on D-streamss

Function	Utility
map(function)	Returns a new D-Streams by mapping each source element
flatmap(function)	Each input entry can be mapped to multiple output entries
filter (function):	Selects records from source D-Streams on basis of function and returns a new D-Streams
repartition(numberOfPartitions)	change number of partitions and control level of parallelism
union(diffStream)	Combines source D-Streams with diffStream and returns a new D-Streams
count()	Counts number of entries in each RDD and returns a new D-Streams of RDDs containing a single element
reduce(function)	Using a commutative and associative functio, a new D-Streams is returned which contains aggregated elements of each source D-Streams RDD
countByValue()	A new D-Streams is returned in which each entry represents key and its frequency
reduceByKey(function, [numberOfTasks])	A new D-Streams is returned in which the given reduce function is used to evaluate aggregate values for each key
join(diffStream, [numberOfTasks])	A new D-Streams is returned with every pair for each key
transform(function)	This method is used to apply a RDD-to-RDD operations to each RDD in the source D-Streams
updateStateByKey(function)	This method is useful for maintaining arbitrary state data for all keys. A new D-Streams is returned where the previous state of each key is updated using given function

are: useful to find out what has happened in some given time frame for example in the last one hour and those statistics are required to be refreshed every minute. The source RDDs falling within the window are merged to form RDD of that particular window which is used for computation purposes.

All window operations take *slideInterval(sI)* and *windowLength(wL)* as parameters (Table 2).

One interesting application of window operation can be to generate word counts every 5 s over the last 40 s (Table 3).

Table 2 Common window operations

Transformation	Meaning
window(wL, sI)	A new D-Streams is returned containing windowed batched of original D-Streams
countByWindow(wL, sI)	Sliding window count of elements is returned
reduceByWindow(function, wL, sI)	Using a commutative and associative function that can be computed in parallel, a new D-Streams is returned which contains aggregated elements of in-stream over sliding window
reduceByKeyAndWindow(function, wL, sI, [numberofTasks])	A new D-Streams is returned in which the given reduce function is used to evaluate aggregate values over batches in the sliding window
reduceByKeyAndWindow(function, inverseFunc, wL, sI, [numberofTasks])	This method is enhanced version of reduceByKeyAndWindow() which calculates the reduce value of each window incrementally, making use of reducing values of preceding window
countByValueAndWindow(wL, sI, [numberOfTasks])	A new D-Streams is returned in which each entry represents key and its frequency in the sliding window

Table 3 Output operations available in spark

OutPut Operation	Utility
print()	Useful for debugging and development. First ten elements of each batch of D-Streams are printed on the driver node which is used for running the application
saveAsTextFiles(prefix,[suffix])	D-Streams's content is saved as text files. At each batch interval, on the basis of prefix and suffix, file name is generated
saveAsObjectFiles(prefix,[suffix])	D-Streams's content is saved as SequenceFiles of serialized Java objects. At each batch interval, on the basis of prefix and suffix, file name is generated
saveAsHadoopFiles(prefix,[suffix])	D-Streams's content is saved as Hadoop files. At each batch interval, on the basis of prefix and suffix, file name is generated
foreachRDD(function)	This function is used to push the data to an external system available in each RDD. External systems can be writing over the network to database or saving the RDD to files

2.4.3 Join Operations

Spark Streaming allows joining operations between D-Streams. Allowed four join operations are: join, leftOuterJoin, rightOuterJoin, fullOuterJoin
Stream-Stream Join

```
val s1: DStream[String, String] = …..
val s2: DStream[String, String] = …..
val streamJoin = s1.join(s2)
```

The RDD generated by stream2 will be combined with the RDD generated by stream1 in each batch interval. Spark Streaming also allows performing joins over windows of streams.

```
val windowStream1 = s1.window(60)
val windowStream2 = s2.window(80)
val windowJoin = windowStream1.join(windowStream2)
```

Stream-dataset Join

```
val dataset = … # Some RDD
val stream = … # Some Stream

val finalDStream= stream.transform { dataset => dataset.join(dataset) }
```

Windowed Streams can be joined in a similar manner. Datasets to be joined can be dynamically changed as for every batch interval, function provided to transform is evaluated.

```
val windowstream = stream.window(30)
val joinedDstream = windowStream.transform{ dataset => dataset.join(dataset) }
```

2.4.4 Output Operations

D-Stream's data can be consumed by external systems like file systems or databases. Output operations trigger the execution of all D-Streams transformations as they allow the transformed data to be pushed out to external systems. Just like RDDs, D-Streams are computed lazily by the output operations. Output operations are executed one-at-a-time by default and in the same order they are defined in the program. Following output operations are available in Spark:

SaveAsObjectFiles() and SaveAsHadoopFiles() are not available in Python API.

2.4.5 SQL and DataFrame Operations

Spark SQL and DataFrame operations can be easily applied on streaming data. This requires creating a SparkSession using the SparkContext used by the StreamingContext. A lazily instantiated singleton instance of SparkSession is created to ensure restarting on driver failures. Every RDD is converted to a DataFrame for application of SQL queries.

```
val user_rdd = DStream[String]  #RDD containing some user information using a website
val user_df = rdd.toDF("user_rdd")
val count_df = spark.sql("select user, count(*) as total from user group by user")
count_df.show()
```

SQL queries can also be run on tables defined on streaming data asynchronously to the running StreamingContext. Batch duration of the StreamingContext should be set enough to remember data for query to be run. Otherwise, some data gets deleted before running query is complete.

2.4.6 Machine Learning Operations

Machine learning Library (MLlib) provides a large class of the machine learning algorithms that can be trained using historical data and can be applied on streaming data online. There are few streaming machine learning algorithms that are capable of training simultaneously from the streaming data and test the model on streaming data. One such streaming algorithm is Streaming KMeans which updates the cluster dynamically as the new data arrives. Another streaming algorithm is Streaming Linear Regression which updates the parameters in the real time. The following example shows the usage of Streaming Linear Regression on streaming data.

```
val totalFeatures = 3
val model = new StreamingLinearRegressionWithSGD()
  .setInitialWeights(Vectors.zeros(totalFeatures))

model.trainOn(data_train)
model.predictOnValues(data_test.map(lp => (lp.label, lp.features))).print()

streamingContext.start()
streamingContext.awaitTermination()
```

```
import
org.apache.spark.mllib.regression.StreamingLinearRegressionWithSGD
import org.apache.spark.mllib.regression.LabeledPoint
import org.apache.spark.mllib.linalg.Vectors

val sparkContext = new SparkConf().setMaster("local[2]").setAppName(“
"LinearRegression")

val data_train= ssc.textFileStream(args(0)).map(LabeledPoint.parse).cache()
val data_test = ssc.textFileStream(args(1)).map(LabeledPoint.parse)
```

The program reads input data from a text file and model is initialised by setting the weights to zero. Total number of features is three.

2.5 Steps in Spark Streaming Program

A classic Spark Streaming program involves following steps:

- Initialize StreamingContext object with the parameters SparkContext and sliding interval time. The sliding interval is used to set update window. Once initialization is done, new computations cannot be defined to already existing context.
- Next, input D-Stream is created for specifying input data sources.
- After input D-Streams is created, computations are defined using Spark Streaming transformation APIs.
- Once streaming computation logic is defined, processing starts using *start* method in StreamingContext
- Finally, processing is terminated using StreamingContext method *awaitTermination.*

3 Real-Time Streaming

3.1 Structured Streaming

As the streaming system became more complex, Apache Spark came up with the idea of Structured Streaming in Spark version 2.0. Structured streaming is not a huge change to Spark itself but a collection of additions. The fundamental concepts remain same; the core transformation is from RDD to DataFrames as before. In this section, aspects of Structured Streaming are discussed.

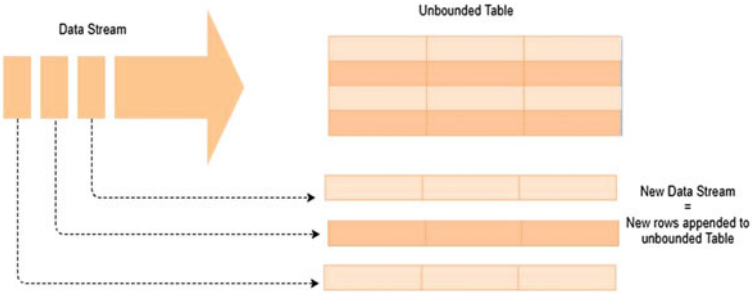

Fig. 3 Fundamentals of structured streaming

3.1.1 Basic Concepts

Structured Streaming introduces the concept of *unbounded table.* Consider the input stream as an Input table. As the data keeps on arriving, new rows are being appended to the Input table (Fig. 3).

In Structured Streaming, DataFrames and Datasets represent streaming, unbounded data as well as static, bounded data. Streaming DataFrames/Datasets are created using the same entry point for static DataFrames/Datasets and can be transformed or operated in exact same manner as batch DataFrames using user-defined functions (UDFs). Structured streaming provides unified API for both streaming and batch sources.

The new model can be explained by the following terms:

- Input: Data from source as an append-only table
- Trigger: Frequency of checking input for new data
- Query: Operations on input like map/reduce/filter/window/session operations
- Result: Final operated table is updated every trigger interval
- Output: Which part of output to be written to data sink after every trigger
  - Complete output: Result table as an output is written every time
  - Delta output: Only modified rows are written
  - Append Output: Only new rows are appended.

3.1.2 Supported Operations

Almost all operations like RDD operations, SQL operations, etc., can be applied on streaming DataFrames/Datasets. Some of the basic operations are

- Aggregation, Projection, Selection
- Window Operations
- Join Operations
- Arbitrary Stateful operations like mapGroupWithState. These operations allow user-defined code to be applied to update user-defined state.

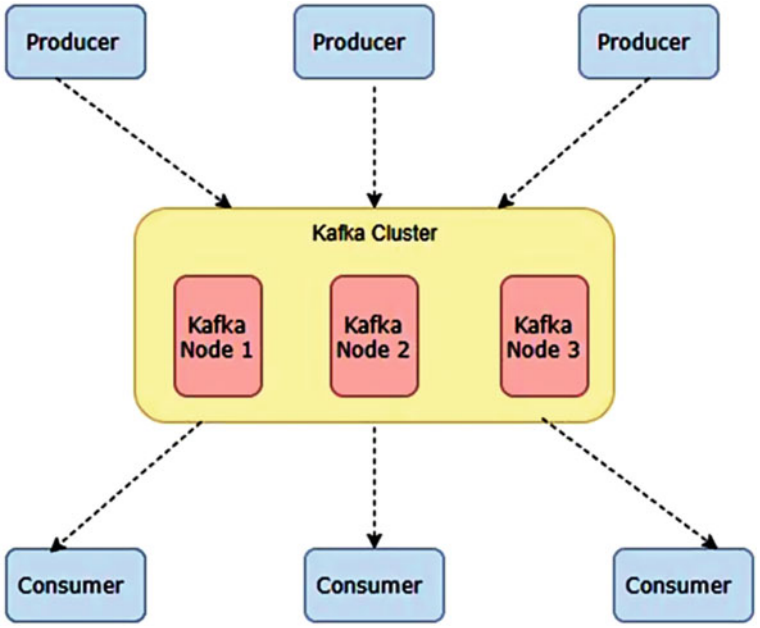

Fig. 4 Kafka

- Handling Late data

However, there are some operations that Streaming DataFrames/Datasets do not support some operations. Few of them are:

- Chain of aggregations on a streaming DataFrame
- Distinct Operations
- Sorting operations are only supported after aggregation in Complete Output Mode.
- Any join between two streaming datasets is not yet supported.
- Outer joins between a static and streaming Datasets are conditionally supported
- Some Dataset methods that run queries immediately and return result like count(), show(), foreach() do not work on streaming Datasets.

3.2 Apache Kafka

Apache Kafka is a scalable, fault-tolerant, real-time messaging system for transferring data in real time between server, applications, and processes. It is used in the systems where capturing user activity on stock ticker data, logs [2], and websites is required. In this section, brief overview of Kafka is given and later on, its integration with Spark is discussed (Fig. 4).

3.2.1 Overview

Kafka is a software where applications connected to the system can transfer and process messages onto a category/feed name known as topic. Byte arrays that are capable of storing any object in any format are known as messages. A message is a stream of records. Each Kafka message is organized into topics. Each record has a key, a value, and a timestamp. Kafka works like a distributed database. When a message is posted to Kafka, it is written to the disk as well as replicated to different machines in the cluster simultaneously.

Kafka Broker: One or multiple servers known as Kafka brokers run a Kafka cluster. Kafka is based on four core APIs

1. Producer API: allows an application to push messages (publish data) to one or more topics within the broker.
2. Consumer API: allows an application to pull messages from one or more Kafka topics and process the stream of records.
3. Stream API: allows an application to take input stream from one or multiple topics and creating an output stream to one or more topics. In shorter words, this API converts input streams to output streams
4. Connector API: an API to build and run of reusable consumers or producers that connect Kafka topic to current data systems or applications.

Apache Kafka provides a data transfer framework known as Kafka Connect apart from client APIs.

Kafka Connect: Apache Kafka provides a framework for streaming data between external data systems and Apache Kafka to support data pipelines in organizations. Kafka Connect can be run as a distributed service or as a standalone process. In distributed service, it uses REST API to submit the connectors to Kafka Connect cluster.

3.2.2 Kafka Topic Partition

Kafka topics are divided into partitions that contain messages in an unmodifiable sequence. In a partition, each message is assigned a unique offset which is used for its identification as well. For multiple consumers to read from a single topic in parallel, it needs to have multiple partition logs where each partition must have one or more replicas. Messages with the same key arrive at same partition. Moreover, a topic can be parallelized by splitting the data across several servers.

Consumers can read data in any order. This feature of Kafka implies that consumers can come and go without little impact on other consumers and the cluster. Every partition is replicated across a number of servers to ensure fault tolerance.

3.2.3 Kafka as a Storage System

Kafka writes data to disk and duplicates it for increasing fault tolerance. Write operation is not complete unless replication is done and producers are made to wait for an acknowledgment. Kafka allows consumers to control their read position, i.e., moving backward or forward in a partition and takes storage seriously.

3.2.4 Kafka for Streaming Processing

Apart from reading, writing, and storing streams of data, Kafka can be used for real-time processing of streams. A stream processor takes continuous streams of data from input topics and produces continuous streams to output topics.

Simple stream processing can be done through consumer and producer APIs. Nonetheless, for complex processing operations like joining streams or aggregating streams, Kafka provides integrated Stream APIs as discussed in Sect. 3.2.1.

3.2.5 Advantages of Kafka

- Every topic can do scale processing and multi subscribing simultaneously as there is no trade-off between the two.
- It ensures ordering as messages in a particular topic partition are appended in the order they are send by the producer. Message sent earlier has a lower offset.
- In traditional queues, ordering is lost as records are consumed asynchronously. On the other hand, traditional messaging systems that allow only one process to consume imply no parallel processing can take place. But Kafka ensures ordering as well as load balancing. Parallelism takes place through partition within the topics. If there are N partitions, consumer group can consume a topic with a maximum of N threads in parallel. This is the maximum degree of consumer parallelism.
- By combining low-latency and storage capabilities, Kafka is suitable for building streaming data pipelines as well as streaming applications.

3.3 Integrating Kafka with Spark Streaming

Apache Spark Streaming applications that read messages from one or more Kafka topics have several benefits. These applications gather insights very fast, lead to uniform management of Spark cluster, guarantee delivery without the need of a write-ahead log and have access to message metadata.

3.3.1 Reading from Kafka

There are two ways to determine read parallelism for Kafka

1. Number of Input D-Streams: Spark runs one task (receiver) per input D-Streams; multiple D-Streamss will parallelize read operations across multiple cores/machines.
2. Number of consumer threads per input D-Streams: Same receiver (task) runs multiple threads, thereby parallelising read operations on the same machine/core.

Option 1 is preferred in practical usage as read-throughput is not increased by running several threads on the same machine.

3.3.2 Message Processing in Spark

After receiving data from Kafka, the data needs to be processed in parallel. More threads for processing can be used rather than reading. In Spark, parallelism is defined by number of RDD partitions as one receiver is run for each RDD partition. There are two ways to control parallelism here:

1. Number of Input D-Streams: Either they can be modified or taken as it is as they are received from previous stage.
2. By applying repartition transformation: repartition method can be used to change number of partitions, hence level of parallelism.

3.3.3 Writing to Kafka

foreachRDD operator is the most generic output operator and should be used for writing to Kafka. This function should be used to push data in RDD to an external system writing it over the network or saving into a file. New Kafka producers should not be created for each partition as Spark Streaming creates several RDDs containing multiple partitions each minute.

3.3.4 Apache Kafka on AWS

Kafka application can be run on Amazon Elastic Computing Cloud (EC2) offering high scalability and enhanced performance solutions for ingesting streaming data (Fig. 5).

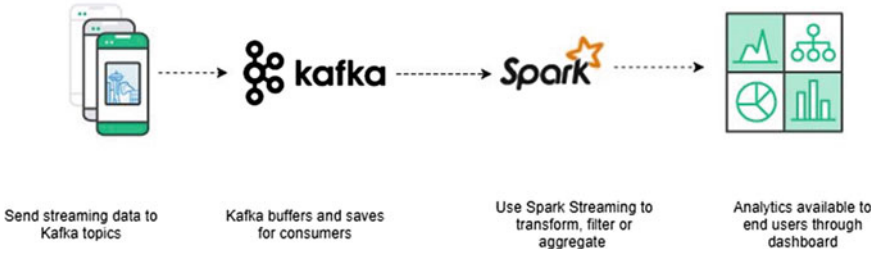

Fig. 5 Apache Kafka on AWS

3.3.5 Real-Time Spark Streaming Pipeline Using Kafka

The example discussed below reads from a Kafka topic using Twitter. Number of tweets per user is counted within each batch. Later, same is computed over a time interval. Step by step program is explained [16].

Import necessary pySpark modules

```
#Spark Streaming
from pyspark.streaming import StreamingContext
#Kafka
from pyspark.streaming.kafka import KafkaUtils
#Spark
from pyspark import SparkContext
#json parsing
import json
```

Create Spark Context which is the starting point of a Spark Streaming program. Create StreamingContext. SparkContext is passed with batch duration of 60 s.

```
sparkContext = SparkContext(appName="KafkaSparkStreaming")
streamingContext = StreamingContext(sparkContext, 60)
```

Connection to Kafka cluster is made using native Spark Streaming capabilities. Consumer group is *spark_streaming* and topic connected is *twitter*

```
kafkaStream = KafkaUtils.createStream(streamingContext, 'cdh57-01-node-
01.moffatt.me:2181', 'spark_streaming', {'twitter':1})
```

Input Stream is a D-Streams which is used to parse input messages stored in JSON format.

```
inputDS = kafkaStream.map(lambda v: json.loads(v[1]))
```

Tweets are stored in a JSON structure. Analysis is done by author name which can be accessed through *user.screen_name*

```
#To count and print number of tweets in a batch
#pprint prints 10 values and nothing is printed until start() is called

inputDS.count().map(lambda x:'Number of tweets in the batch are: %s' %
x).pprint()

user_stream = inputDS.map (lambda tweet: tweet['user']['screen_name'])

#Count the number of tweets done by a user
user_counts = user_stream.countByValue()
user_counts.pprint()
```

Sort the author on the basis of number of Tweets. D-Streams does not have an inbuilt sort function. So, *transform* function is used to access sortBy available in pyspark.

```
sorted_user_count = user_counts.transform(\
        (lambda x :x\
                .sortBy(lambda x:( -x[1]))))

sorted_user_count.pprint()
```

To get top five users with tweet count

```
top_five_users = sorted_user_count.transform\
        (lambda rdd:sc.parallelize(rdd.take(5)))

top_five_users.pprint()
```

To get authors whose username starts with ‘r’ and who have tweeted more than once

```
query1 = user_counts.filter(lambda x: x[1]>1
                        or
                        x[0].lower().startswith('r'))
```

To get most commonly used words in tweets

```
inputDS.flatMap(lambda tweet:tweet['text'].split(" "))\
   .countByValue()\
   .transform(lambda rdd:rdd.sortBy(lambda x:-x[1]))\
   .pprint()
```

Starting the streaming context: The program starts here and results of pprint() are printed.

```
streamingContext.start()
streamingContext.awaitTermination(timeout = 180 )
```

The complete program is given below

```
from pyspark.streaming import StreamingContext
from pyspark.streaming.kafka import KafkaUtils
from pyspark import SparkContext
import json

sparkContext = SparkContext(appName="KafkaSparkStreaming")

streamingContext = StreamingContext(sparkContext, 60)

kafkaStream = KafkaUtils.createStream(streamingContext, 'cdh57-01-node-
01.moffatt.me:2181', 'spark_streaming', {'twitter':1})
inputDS = kafkaStream.map(lambda v: json.loads(v[1]))

inputDS.count().map(lambda x:'Number of tweets in the batch are: %s' %
x).pprint()

user_stream = inputDS.map (lambda tweet: tweet['user']['screen_name'])

user_counts = user_stream.countByValue()
user_counts.pprint()
sorted_user_count = user_counts.transform(\
        (lambda x :x\
                .sortBy(lambda x:( -x[1]))))

sorted_user_count.pprint()

top_five_users = sorted_user_count.transform\
(lambda rdd:sc.parallelize(rdd.take(5)))
```

For Windowed Stream Processing, complete program looks like

```
top_five_users.pprint()

query1 = user_counts.filter(lambda x: x[1]>1
                        or
                        x[0].lower().startswith('r'))

inputDS.flatMap(lambda tweet:tweet['text'].split(" "))\
   .countByValue()\
   .transform(lambda rdd:rdd.sortBy(lambda x:-x[1]))\
   .pprint()

streamingContext.start()
streamingContext.awaitTermination(timeout = 180 )
from pyspark.streaming import StreamingContext
from pyspark.streaming.kafka import KafkaUtils
from pyspark import SparkContext
import json

sparkContext                                =                                SparkCon-
text(appName="KafkaSparkStreamingWindowed")

streamingContext = StreamingContext(sparkContext, 5)

kafkaStream = KafkaUtils.createStream(streamingContext, 'cdh57-01-node-
01.moffatt.me:2181', 'spark_streaming', {'twitter':1})
inputDS = kafkaStream.map(lambda v: json.loads(v[1]))

#Number of tweets in batch
batch_count = kafkaStream.count().map(lambda x:'Number of tweets in the
batch are: %s' % x).pprint()

#number of tweets in batch
window_count       =       kafkaStream.countByWindow(60,5).map(lambda
x:('Tweets total (One minute rolling count): %s' % x))

#Get users
user_stream = inputDS.map (lambda tweet: tweet['user']['screen_name'])
#word count in batch
batch_occurence = user_stream.countByValue()\
                        .transform(lambda rdd:rdd\
                        .sortBy(lambda x:-x[1]))\
                            .map(lambda    x:"User    counts    this    batch:\tValue
%s\tCount %s" % (x[0],x[1]))
```

```
window_occurence = user_stream.countByValueAndWindow(60,5)\
                        .transform(lambda rdd:rdd\
                        .sortBy(lambda x:-x[1]))\
                            .map(lambda x:"user counts in one minute window:\tValue %s\tCount %s" %

streamingContext.start()
streamingContext.awaitTermination(timeout = 180 )
```

3.4 *Amazon Kinensis*

Amazon Kinesis [15] Streams makes real-time streaming data simpler to collect, process, and analyze, gain timely insights and make quick decisions. Kinesis makes streaming cost effective, scalable and gives control to choose tools in the best interest of the application (Fig. 6).

A classical Amazon Kinesis Streams application takes input data from Kinesis streams as data records. Processed records can be used for a variety of applications like sending data to other AWS services, advertising strategies, dynamically change pricing or generating alerts and is sent to dashboards.

3.4.1 Benefits of Using Kinesis

- Real time: Processing and analyzing of data in real-time instead of waiting to have all the data collected.
- Fully Managed: It does not require the user to manage any infrastructure or any custom code.
- Scalable: Allows changing capacity of streams by API calls or by few clicks

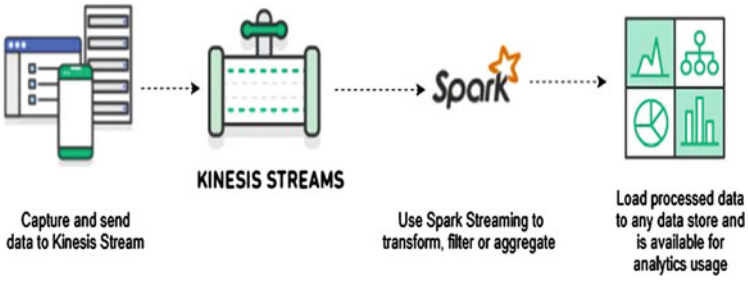

Fig. 6 Using Amazon Kinesis on spark

- Reliable: Manual configuration for replication is not required as it automatically replicates across three AWS zones.
- Integration with open-source frameworks like Apache Flink and Apache Spark and with other AWS services makes loading and processing of data easier.

3.4.2 Kinesis Use Cases

Kinesis streams are used for continuous, rapid intake and aggregation of data. The processing is lightweight as the processing and response time for data ingestion is being done in the real time. Kinesis streams offer quick data input as data is not divided into batches on the servers before submitting for input. The typical use cases of Kinesis streams are:

- Kinesis can be used for complex stream processing as Directed Acyclic Graphs (DAGs) can be created from Kinesis Stream data streams and applications.
- Parallel processing on real-time data can be used for obtaining real-time analytics. For example, analyzing site usability by processing website clickstreams in real time.
- Accelerated data intake and processing: Data pushed directly into a stream can be processed in merely seconds. This ensures logs are not lost.
- Kinesis Streams are used for obtaining real-time metrics which are used for reporting the status of system in real time.

3.5 Kinesis on Spark

Kinesis can be used as a source for streaming data. It is not necessary to use Kinesis Analytics; Spark SQL and structured APIs can also be used instead. Processing Kinesis Streams with Spark Streaming requires an Elastic MapReduce (EMR) cluster with Spark to read data from Kinesis stream.

3.5.1 Create Kinesis Data Stream

Amazon Kinesis Client Library (KCL) is used for creating an input D-Streams via Kinesis consumer.

The following command is run from command line to create a Kinesis stream in user's account.

```
aws kinesis create-stream –stream-name my_stream –shard-count1
```

3.5.2 Kinesis Spark Streaming Application

A simple streaming application is discussed below which reads a Kinesis stream that contains events in JSON format. Spark job counts the events on basis of type and these counts are aggregated into 1 min buckets.

```
import org.apache.spark.streaming.kinesis.KinesisUtils
import org.apache.spark.SparkConf
import org.apaches.spark.streaming._

val streamingSparkContext = {
        val sparkConf = new SparkConf().setAppName( config.appName )
        .setMaster(config.master)
        New StreamingContext(sparkConf, config.batchInterval)
        }
        streamingSparkContext
```

Reading from Source: reading stream using Kinesis connector.

```
val kinesisClient = KinesisUtils.setupKinesisClientConnection( con-
fig.endpointURL, config.awsProfile)
require(kinesisClient != null, “No AWS credentials found”)

val sreamingSparkContext = setupSparkContext(config)
val numberShards = KinesisUtils.getShardCount(kinesisClient, con-
fig.streamName)
val sparkStream = ( 0 until numberShards).map { i=>
        KinesisUtils.createStream(
                sc = streamingSparkContext,
                streamName = config.streamName,
                endPointURL = config.endpointUrl,
                initialPosInStream = config.initialPosition,
                checkPointInterval = config.batchInterval,
                storageLevel = config.storageLevel
                )
        }
```

Exploring and applying Transformation on Streams: After Kinesis records are mapped to DataFrames and data is loaded, Spark DataFrame and SQL APIs can be applied to obtain analytics. This is the part where insights are derived.

```
//Map phase:
val bucketEvent = streamingSparkContext.union( sparkStream)
                .map { bytes =>
                        Val e = SimpleEvent.fromJson(bytes)
                        (e.bucket, e.type)
}

//Reduce Phase
val tot_count  = bucketEvent.groupByKey
        .map { case (eventType, events) =>
        val count = events.groupBy(identity).mapValues(_.size)
        (eventType, count)
        }
```

Saving transformed Stream: Optionally, transformed data can be written somewhere. In this example, finally aggregation is done and saved into Amazon DynamoDB

```
// add import for DynaoDB
import com.amazonaws.services.dynamodbv2.document.DynamoDB
import storge.DynamoUtils

//setup Spark Connection with DynamoDB
val conn = DynamoUtils.setupDynamoClientConnection(config.awsProfile)

//aggregating and storing
tot_count.foreachRDD{ rdd=>
        rdd.foreach { case (df, aggregate) =>
                aggregates.foreach { case (type, count) =>
                        DynamoUtils.setOrUpdateCount(
                                conn,
                                config.tableName,
                                df.toString,
                                type,
                                DynamoUtils.timeNow()
                                DynamoUtils.timeNow(),
                                count.toInt
                        )
                }
        }
}
```

3.6 Apache Kafka versus Amazon Kinesis

There are number of options available to work with streaming data on cloud. If cloud being used is AWS, then using Amazon Kinesis offers several advantages over Kafka. Both Amazon Kinesis and Apache Kafka are scalable, reliable and durable data ingesting framework sharing common core concepts like application components, partitioning and replication.

However, the following points should be considered before building a real-time application.

3.6.1 Configuration and Setup

Amazon Kinesis is a fully managed platform that makes it easy to work with. Users do not need to be concerned about hosting the software and the resources. On the other hand, Apache Kafka was initially developed at LinkedIn and is an open-source solution. A user has to take care of installation, monitoring and security of clusters, durability, failure recovery and ensuring high availability. This can be an administrative burden.

3.6.2 Cost

Apache Kafka requires hosting and managing of the framework. Total cost of the application in this depends on prudent selection of storage capabilities and computing resources which requires capacity planning. Cumulative cost of such a system is sum of resource cost and human cost. On the contrary, human resources are significantly lower given the hosted nature of Amazon Kinesis. Nevertheless, Kafka is more cost efficient in certain data ingests patterns and should be considered carefully.

3.6.3 Architectural Differences

There are few architectural differences between Kafka and Kinesis. These include scalability models, consumption latency and end-to-end data ingest. For example, in order to scale Amazon Kinesis, either a shard can be split to increase capacity or two shards can be joined to lower cost and reduce capacity, whereas Kafka requires monitoring for hot partitions and partitions can be added or moved as per the requirement.

User applications are determined by how APIs work. So, it is essential to keep in mind the features offered by either solution while choosing which streaming platform to be used. For example, Kafka enables users to replicate data based on certain key as it has the capability to keep hold of last known message. In contrast, this feature is not available in Amazon Kinesis and has to be built using API.

3.6.4 Performance

In a reasonable price, it is tough to replicate Kafka's performance in Kinesis. Kafka is faster than Kinesis. Kinesis is costly in terms of latency and throughput as it writes each message synchronously to three different data centers.

3.6.5 Summing Up

It is clear that despite being similar, Kinesis and Kafka are meant to be used in different contexts. Pros of Kafka are its performance and integration with other big data frameworks/platforms. Kafka requires in-depth knowledge to make the infrastructure less risky.

Built-in-reliability and simplicity are Kinesis's core strength. Streams only store records for 7 days and the data gets lost, some other storage is required if data is to be kept for longer. Throughput is less as compared to Kafka.

4 Conclusion

In this chapter, various concepts related to Apache Spark Streaming API were discussed. Apache Spark API provides a framework that enables scalable, fault-tolerant streaming with high throughput. In this chapter, we started with a short overview of streaming and spark streaming. After the introduction, fundamentals of Spark streaming, its architecture were focused upon. We described several transformation API methods available in spark streaming that are useful for computing data streams. These included Spark streaming API RDD—like operations and Streaming Context. We also included in our study various other operations like DataFrame, SQL, and machine learning algorithms which can be applied on streaming data. We concluded this section with discussion of steps necessitated in a prototypical spark streaming program.

Apache Kafka is a distributed streaming platform that enables processing of streams of records in the order they occur, i.e., it can be thought of a message queue which is distributed, partitioned, and replicated commit log service. In the next section of the chapter, a brief introduction of Apache Kafka and how it is used with spark, its benefits when integrated with Spark were given. We further described in detail a real-time streaming data pipeline build using Kafka integrated with Spark. We continue this discussion with another real-time parallel data processing stream known as Amazon Kinesis Streams and its integration with spark.

Finally, we conclude with trade-offs between Apache Spark and Amazon Kinesis, focusing on building a real-time streaming pipeline using available Spark API. To make the chapter more realistic and to enhance understanding of the user, wherever possible, code snippets have been provided.

References

1. Apache Spark. https://spark.apache.org/
2. Zaharia, M., Xin, R.S., Wendell, P., Das, T., Armbrust, M., Dave, A., Meng, X., Rosen, J., Venkataraman, S., Franklin, M.J., Ghodsi, A., Gonzalez, J., Shenker, S., Stoica, I.: Apache spark: a unified engine for big data processing. Commun. ACM CACM Homepage Archive New York NY USA **59**(11), 56–65 (2016)
3. Dean, J., Ghemawat, S.: MapReduce: simplified data processing on large clusters. In: Proceedings of the 6th Conference on Symposium on Opearting Systems Design & Implementation, San Francisco, CA, pp. 10–10 (2006)
4. Logothetis, D., Trezzo, C., Webb, K. C., Yocum, K.: In-situ MapReduce for log processing. In: USENIX Annual Technical Conference (2011)
5. Scala; http://www.scala-lang.org
6. Alsheikh, M.A., Niyato, D., Lin, S., Tan, H.P., Han, Z.: Mobile big data analytics using deep learning and apache spark. IEEE Netw. **30**(3), 22–29 (2016)
7. Owen, S., Ryza, Laserson S., Wills U.: Advanced Analytics with Apache Spark. O'Reilly Media (2015)
8. Spark homepage. http://www.spark-project.org
9. Apache Hive; http://hadoop.apache.org/hive
10. Balazinska, M., Balakrishnan, H., Madden, S.R., Stonebraker, M.: Fault-tolerance in the Borealis distributed stream processing system. ACM Trans. Database Syst. **33**(1), 3–16 (2008)
11. Shah, M., Hellerstein, J., Brewer, E.: Highly available, fault-tolerant, parallel dataflows. In: Proceeding of ACM SIGMOD Conference, pp. 827–838 (2004)
12. Zaharia, M., Das, T., Li, H., Shenker, S., Stoica, I.: Discretized streams: an efficient programming model for large-scale stream processing. In: 4th USENIX Workshop on Hot Topics in Cloud Computing (2012)
13. Zaharia, M., Chowdhury, M., Das, T., Dave, A., Ma, J., McCauley, M., Franklin, M., Shenker, S., Stoica, I.: Resilient distributed datasets: a fault-tolerant abstraction for in-memory cluster computing. In: NSDI Proceedings of the 9th USENIX Conference on Networked Systems Design and Implementation, pp. 2–2 (2012)
14. Apache Kafka. https://kafka.apache.org/
15. Amazon Kinesis. http://docs.aws.amazon.com/streams/latest/dev/introduction.html
16. Nair, L.R., Shetty, S.D.: Streaming twitter data analysis using spark for effective job search. J. Theor. Appl. Inf. Technol. **80**(2), 349–353 (2015)