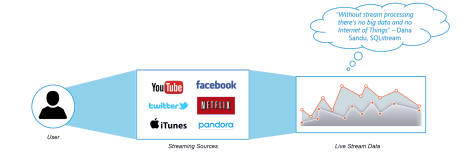
Spark streaming:

Spark Streaming is an extension of the core Spark API that enables scalable, high-throughput, fault-tolerant stream processing of live data streams. Spark Streaming can be used to stream live data and processing can happen in real time. Spark Streaming’s ever-growing user base consists of household names like Uber, Netflix and Pinterest.

**What is Streaming?**

Data Streaming is a technique for transferring data so that it can be processed as a steady and continuous stream. Streaming technologies are becoming increasingly important with the growth of the Internet.



**Why Spark Streaming?**

We can use Spark Streaming to stream real-time data from various sources like Twitter, Stock Market and Geographical Systems and perform powerful analytics to help businesses.

**Spark Streaming Overview**

*Spark Streaming* is used for processing real-time streaming data. It is a useful addition to the core Spark API. Spark Streaming enables high-throughput and fault-tolerant stream processing of live data streams.



The fundamental stream unit is DStream which is basically a series of RDDs to process the real-time data.

**Spark Streaming Features**

1. **Scaling:** Spark Streaming can easily scale to hundreds of nodes.

2. **Speed:** It achieves low latency.

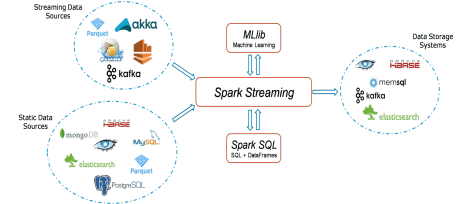
3. **Fault Tolerance:** Spark has the ability to efficiently recover from failures. 4. **Integration:** Spark integrates with batch and real-time processing.

5. **Business Analysis:** Spark Streaming is used to track the behavior of customers which can be used in business analysis.

**Spark Streaming Workflow**

Spark Streaming workflow has four high-level stages. The first is to stream data from various sources. These sources can be streaming data sources like Akka, Kafka, Flume, AWS or Parquet for real-time streaming. The second type of sources includes HBase, MySQL, PostgreSQL, Elastic Search, Mongo DB and Cassandra for static/batch streaming. Once this happens, Spark can be used to perform Machine Learning on the data through its MLlib API. Further, Spark SQL is used to perform further operations on this data. Finally, the streaming output can be stored into various data storage systems like HBase, Cassandra, MemSQL, Kafka, Elastic Search, HDFS and local file system.





**Spark Streaming Fundamentals**

1. Streaming Context

2. DStream

3. Caching

4. Accumulators, Broadcast Variables and Checkpoints

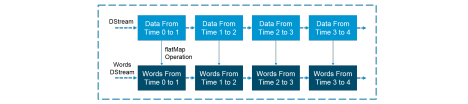
**Streaming Context**

*Streaming Context* consumes a stream of data in Spark. It registers an *Input DStream* to produce a *Receiver* object. It is the main entry point for Spark functionality.

A StreamingContext object can be created from a SparkContext object. A SparkContext represents the connection to a Spark cluster and can be used to create RDDs, accumulators and broadcast variables on that cluster.

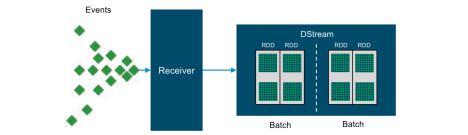
**DStream**

*Discretized Stream* (DStream) is the basic abstraction provided by Spark Streaming. It is a continuous stream of data. It is received from a data source or a processed data stream generated by transforming the input stream.



Internally, a DStream is represented by a continuous series of RDDs and each RDD contains data from a certain interval.

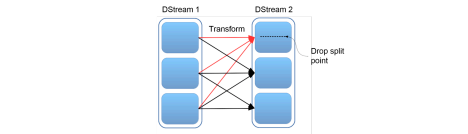
**Input DStreams:** *Input DStreams* are DStreams representing the stream of input data received from streaming sources.



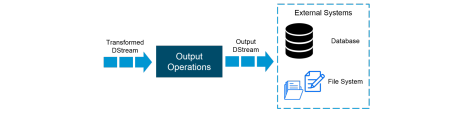
Every input DStream is associated with a Receiver object which receives the data from a source and stores it in Spark’s memory for processing.

**Transformations on DStreams:**

Any operation applied on a DStream translates to operations on the underlying RDDs. Transformations allow the data from the input DStream to be modified similar to RDDs. DStreams support many of the transformations available on normal Spark RDDs.

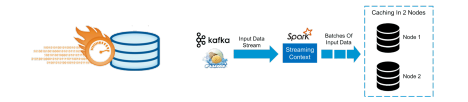
**Output DStreams:**

Output operations allow DStream’s data to be pushed out to external systems like databases or file systems. Output operations trigger the actual execution of all the DStream transformations.



**Caching**

*DStreams* allow developers to cache/ persist the stream’s data in memory. This is useful if the data in the DStream will be computed multiple times. This can be done using the *persist()* method on a DStream.



For input streams that receive data over the network (such as Kafka, Flume, Sockets, etc.), the default persistence level is set to replicate the data to two nodes for fault-tolerance.

**What is Shared Variable in Spark**

Generally, while functions passed on, it executes on the specific remote cluster node. Usually, it works on separate copies of all the variables those we use in functions. These specific variables are precisely copied to each machine. Also, on the remote machine, no updates to the variables sent back to the driver program. Therefore, it would be inefficient to support general, read-write shared variables across tasks. Although, in spark for two common usage patterns, there are two types of shared variables, such as:

1. Broadcast Variables

**Broadcast Variables:** *Broadcast variables* allow the programmer to keep a read-only variable cached on each machine rather than shipping a copy of it with tasks. They can be used to give every node a copy of a large input dataset in an efficient manner. Spark also attempts to distribute broadcast variables using efficient broadcast algorithms to reduce communication cost.

scala> val broadcastVar1 = sc.broadcast(Array(1, 2, 3))

broadcastVar1: org.apache.spark.broadcast.Broadcast[Array[Int]] = Broadcast(0)

scala> broadcastVar1.value

res0: Array[Int] = Array(1, 2, 3)

After we create a broadcast variable, instead of using value v in any functions we should use it. By ensuring that we cannot ship v to the nodes more than once. It is also very important that no modification can take place on the object v after it is broadcast. It will help ensure that all nodes get the same value of the broadcast variable.

2. Accumulators

The variables which are only “added” through a commutative and associative operation. Also, can efficiently support in parallel. We can

use *Accumulators* to implement counters or sums. Spark natively supports programmers for new types and accumulators of numeric types. **For Example:**

In this code we are using an accumulator to add up the elements of an array:

scala> val accum1 = sc.longAccumulator("Accumulator1")

accum1: org.apache.spark.util.LongAccumulator = LongAccumulator(id: 0, name: Some(Accumulator1), value: 0)

scala> sc.parallelize(Array(1, 2, 3, 4)).foreach(x => accum.add(x))

...

10/09/29 18:41:08 INFO SparkContext: Tasks finished **in** 0.317106 s

scala> accum1.value

res2: Long = 10

Afterwards, we have seen how accumulators help to handle shared objects. What is Kafka Spark Streaming Integration?

Spark Streaming is an extension to the central application API of Apache Spark. It allows you to extract data from several sources such as Kafka, Kinesis, TCP sockets and process it using complex algorithms expressed using high level functions such as map, reduce, join and window. It optimizes the use of a discretized stream of data (**DStream**) that extends a continuous data stream for an enhanced level of data abstraction.

Spark Streaming receives a live input stream and splits the data into batches. These batches are processed by the Spark engine to produce the final batch result stream. This allows versatile integrations through different sources with Spark Streaming including Apache Kafka.

**Why is Spark Streaming and Kafka Integration Important?**

There are several benefits of implementing Spark Kafka Integration:

● By setting up the the Spark Streaming and Kafka Integration, you can ensure minimum data loss through Spark Streaming while saving all the received Kafka data synchronously for an easy recovery.

● Users can read messages from a single topic or multiple Kafka topics. ● Along with this level of flexibility, you can also access high scalability, throughput and fault-tolerance, and a range of other benefits by using Spark and Kafka in tandem.

In Apache Kafka Spark Streaming Integration, there are two approaches to configure Spark Streaming to receive data from Kafka i.e. Kafka Spark Streaming Integration.

First is by using Receivers and Kafka’s high-level API, and a second, as well as a new approach, is without using Receivers. There are different programming models for both the approaches, such as performance characteristics and semantics guarantees.

**a. Receiver-Based Approach**

Here, we use a Receiver to receive the data. So, by using the Kafka high-level **consumer** API, we implement the Receiver. Further, the received data is stored in **Spark executors**. Then jobs launched by Kafka – Spark Streaming processes the data.

Although, it is a possibility that this approach can lose data under failures under default configuration. Hence, we have to additionally enable write-ahead logs in Kafka Spark Streaming, to ensure zero-data-loss.

**b. Direct Approach (No Receivers)** After Receiver-Based Approach, new receiver-less “direct” approach has been introduced. It ensures stronger end-to-end guarantees. This approach periodically queries Kafka for the latest offsets in each topic+partition, rather than using receivers to receive data.

Also, defines the offset ranges to process in each batch, accordingly. Moreover, to read the defined ranges of offsets from Kafka, it’s simple consumer API is used, especially when the jobs to process the data are launched. However, it is similar to read files from a file system.

**Key Kafka Spark APIs**

To set up the Spark Streaming and Kafka Integration, there are namely 3 main Kafka Spark APIs:

● StreamingContext API: This API acts as the main entry point for utilizing the Spark Streaming functionality. Using the methods provided by this API, you can create DStreams from various input sources.

● SparkConf API: It represents the configuration for a Spark application. You can create a SparkConf object with the new **SparkConf()**, which will load values from any **spark.\* Java** system properties set in your application.

● **KafkaUtils API**: The KafkaUtils API is allows you to connect Kafka clusters to Spark streaming and set up the Spark Streaming and Kafka Integration . This API has an important method, **createStream.** It is used to create an input stream that pulls messages from the Kafka broker. The KafkaUtils API has another method, **createDirectStream**. It is used to create an input stream that fetches messages directly from the Kafka broker without using a receiver. This stream can guarantee that every message from Kafka participates in the conversion only once.