NAAN MUDHALVAN

BIG DATA ANALYTICS

ASSESSMENT – IX

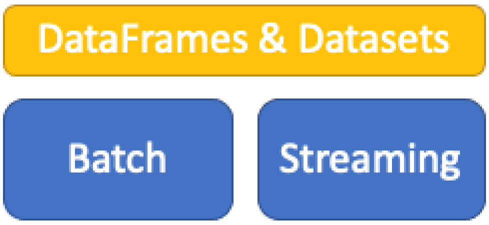
Kafka and Spark Streaming Homework

1.What is Apache Spark streaming?

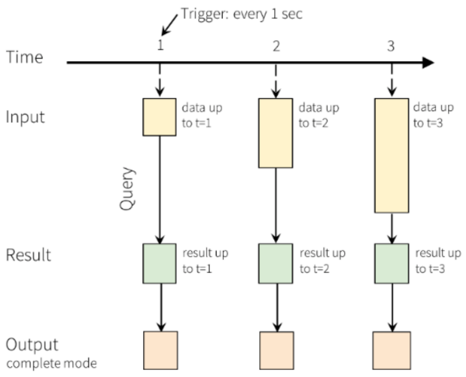
Spark Structured Streaming makes it easy to build streaming applications and pipelines with the same and familiar Spark APIs. Spark Structured Streaming abstracts away complex streaming concepts such as incremental processing, checkpointing, and watermarks so that you can build streaming applications and pipelines without learning any new concepts or tools.

```
spark
  .readStream
  .select($"value".cast("string").alias("jsonData"))
  .select(from_json($"jsonData",jsonSchema).alias("payload"))
  .writeStream
  .trigger("1 seconds")
  .start()
```

Spark Structured Streaming provides the same structured APIs (DataFrames and Datasets) as Spark so that you don’t need to develop on or maintain two different technology stacks for batch and streaming. In addition, unified APIs make it easy to migrate your existing batch Spark jobs to streaming jobs.

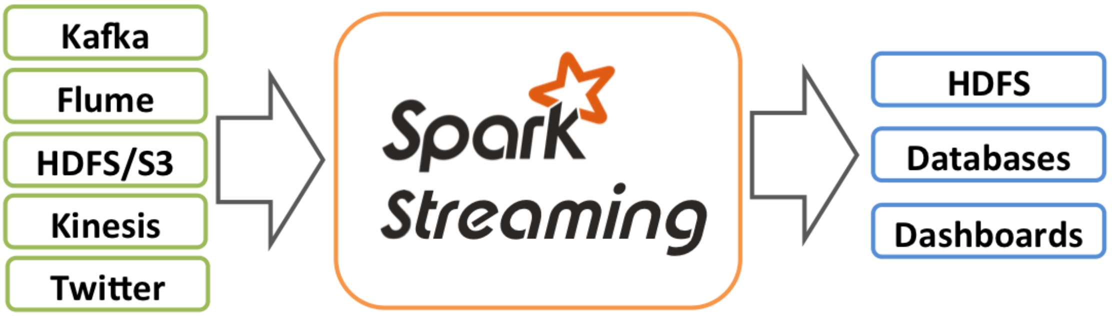


Spark Structured Streaming uses the same underlying architecture as Spark so that you can take advantage of all the performance and cost optimizations built into the Spark engine. With Spark Structured Streaming, you can build low latency streaming applications and pipelines cost effectively.

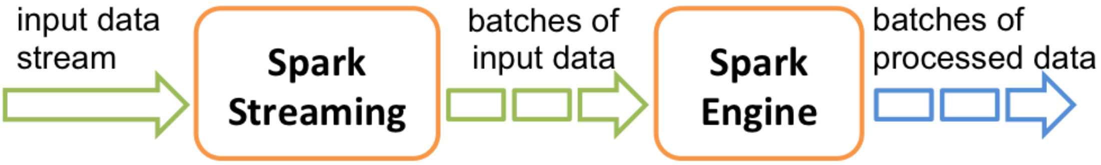


2. Describe how Spark Streaming processes data?

Spark Streaming is an extension of the core Spark API that enables scalable, high-throughput, fault-tolerant stream processing of live data streams. Data can be ingested from many sources like Kafka, Kinesis, or TCP sockets, and can be processed using complex algorithms expressed with high-level functions like map, reduce, join and window. Finally, processed data can be pushed out to filesystems, databases, and live dashboards. In fact, you can apply Spark's machine learning and graph processing algorithms on data streams.



Internally, it works as follows. Spark Streaming receives live input data streams and divides the data into batches, which are then processed by the Spark engine to generate the final stream of results in batches.



Spark Streaming provides a high-level abstraction called *discretized stream* or *DStream*, which represents a continuous stream of data. DStreams can be created either from input data streams from sources such as Kafka, and Kinesis, or by applying high-level operations on other DStreams. Internally, a DStream is represented as a sequence of RDDs.

This guide shows you how to start writing Spark Streaming programs with DStreams. You can write Spark Streaming programs in Scala, Java or Python, all of which are presented in this guide.

3. What are DStream?

A Discretized Stream (DStream), the basic abstraction in Spark Streaming, is a continuous sequence of RDDs (of the same type) representing a continuous stream of data (see spark.RDD for more details on RDDs). DStreams can either be created from live data (such as, data from HDFS, Kafka or Flume) or it can be generated by transformation existing DStreams using operations such as map, window and reduceByKeyAndWindow. While a Spark Streaming program is running, each DStream periodically generates a RDD, either from live data or by transforming the RDD generated by a parent DStream. This class contains the basic operations available on all DStreams, such as map, filter and window. In addition, PairDStreamFunctions contains operations available only on DStreams of key-value pairs, such as groupByKeyAndWindow and join. These operations are automatically available on any DStream of the right type (e.g., DStream[(Int, Int)] through implicit conversions when spark.streaming.StreamingContext._ is imported.

DStreams internally is characterized by a few basic properties:

- A list of other DStreams that the DStream depends on
- A time interval at which the DStream generates an RDD
- A function that is used to generate an RDD after each time interval

Linear Supertypes:

- Logging
- Serialisable
- AnyRef

- Any

Known Subclasses:

- ConstantInputDStream
- InputDstream
- NetworkInputDstream

4. What is StreamingContext object?

Class StreamingContext

Object:org.apache.spark.streaming.StreamingContext

All Implemented Interfaces:Logging

public class StreamingContext
extends Object
implements Logging

Main entry point for Spark Streaming functionality. It provides methods used to create DStreams from various input sources. It can be either created by providing a Spark master URL and an appName, or from a org.apache.spark.SparkConf configuration (see core Spark documentation), or from an existing org.apache.spark.SparkContext. The associated SparkContext can be accessed using context.sparkContext. After creating and transforming DStreams, the streaming computation can be started and stopped using context.start() and context.stop(), respectively. context.awaitTermination() allows the current thread to wait for the termination of the context by stop() or by an exception.

5. What are some of the common transformations on DStreams supported by Spark Streaming?

a. map()

Map function in Spark passes each element of the source DStream through a function and returns a new DStream.

Spark map() example

```
[php]val conf = new SparkConf().setMaster(“local[2]”)
.setAppName(“MapOpTest”)
val ssc = new StreamingContext(conf , Seconds(1))
val words = ssc.socketTextStream(“localhost”, 9999)
val ans = words.map { word => (“hello” ,word ) } // map hello with each line
ans.print()
ssc.start() // Start the computation
ssc.awaitTermination() // Wait for termination
}
```

b. flatMap()

FlatMap function in Spark is similar to Spark map function, but in flatmap, input item can be mapped to 0 or more output items. This creates difference between map and flatmap operations in spark.

Spark FlatMap Example

```
[php]val lines = ssc.socketTextStream(“localhost”, 9999)
val words = lines.flatMap(_.split(” “)) // for each line it split the words by space
val pairs = words.map(word => (word, 1))
val wordCounts = pairs.reduceByKey(_ + _)
wordCounts.print()
```

c. filter()

Filter function in Apache Spark returns selects only those records of the source DStream on which func returns true and returns a new DStream of those records.

Spark Filter function example

```
[php]val lines = ssc.socketTextStream(“localhost”, 9999)
val words = lines.flatMap(_.split(” “))
val output = words.filter { word => word.startsWith(“s”) } // filter the words starts with letter“s”
output.print()
```

d. reduceByKey(func, [numTasks])

When called on a DStream of (K, V) pairs, ReduceByKey function in Spark returns a new DStream of (K, V) pairs where the values for each key are aggregated using the given reduce function.

Spark reduceByKey example

```
[php]val lines = ssc.socketTextStream(“localhost”, 9999)
val words = lines.flatMap(_.split(” “))
val pairs = words.map(word => (word, 1))
val wordCounts = pairs.reduceByKey(_ + _)
wordCounts.print()
```

e. countByValue()

CountByValue function in Spark is called on a DStream of elements of type K and it returns a new DStream of (K, Long) pairs where the value of each key is its frequency in each Spark RDD of the source DStream.

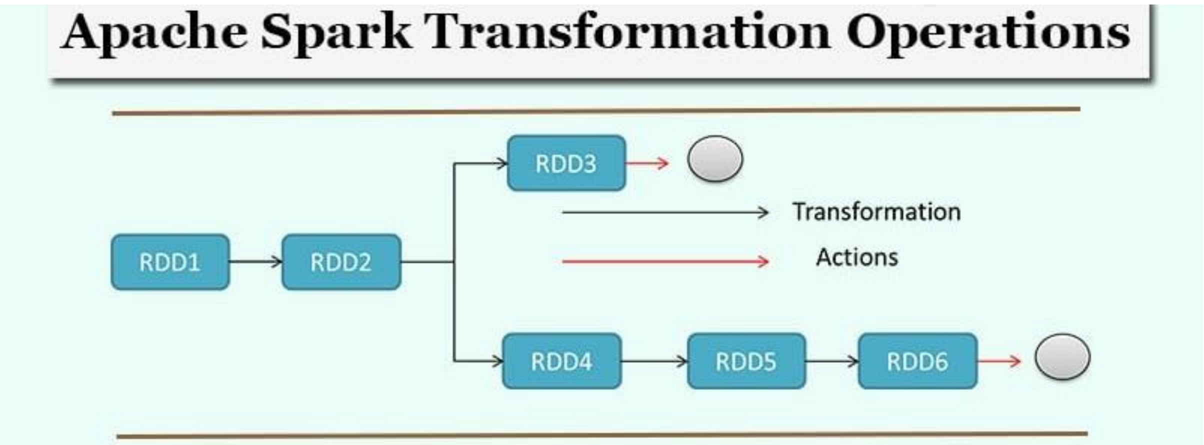
Spark CountByValue function example

```
[php]val line = ssc.socketTextStream(“localhost”, 9999)
val words = line.flatMap(_.split(” “))
words.countByValue().print()
```

f. UpdateStateByKey()

The updateStateByKey operation allows you to maintain arbitrary state while continuously updating it with new information. To use this, you will have to do two steps.
Define the state – The state can be an arbitrary data type.
Define the state update function – Specify with a function how to update the state using the previous state and the new values from an input stream.
In every batch, Spark will apply the state update function for all existing keys, regardless of whether they have new data in a batch or not. If the update function returns None then the key-value pair will be eliminated.
Spark UpdateByKey Example

```
[php]def updateFunc(values: Seq[Int], state: Option[Int]): Option[Int] = {
val currentCount = values.sum
val previousCount = state.getOrElse(0)
Some(currentCount + previousCount)
}
val ssc = new StreamingContext(conf , Seconds(10))
val line = ssc.socketTextStream("localhost", 9999)
ssc.checkpoint("/home/asus/checkpoints/") // Here ./checkpoints/ are the directory where all checkpoints are stored.
val words = line.flatMap(_.split(" "))
val pairs = words.map(word => (word, 1))
val globalCountStream = pairs.updateStateByKey(updateFunc)
globalCountStream.print()
ssc.start() // Start the computation
ssc.awaitTermination()
```



6. What are the output operations that can be performed on DStreams?

Output operations allow DStream's data to be pushed out external systems like a database or a file systems. Since the output operations actually allow the transformed data to be consumed by external systems, they trigger the actual execution of all the DStream transformations (similar to actions for RDDs). Currently, the following output operations are defined:

Output Operation	Meaning
print()	Prints first ten elements of every batch of data in a DStream on the driver node running the streaming application. This is useful for development and debugging. Python API This is called pprint() in the Python API.
saveAsTextFiles(*prefix*, [*suffix*])	Save this DStream's contents as a text files. The file name at each batch interval is generated based on *prefix* and *suffix*: *"prefix-TIME_IN_MS[.suffix]"*.
saveAsObjectFiles(*prefix*, [*suffix*])	Save this DStream's contents as a SequenceFile of serialized Java objects. The file name at each batch interval is generated based on *prefix* and *suffix*: *"prefix-TIME_IN_MS[.suffix]"*. Python API This is not available in the Python API.
saveAsHadoopFiles(*prefix*, [*suffix*])	Save this DStream's contents as a Hadoop file. The file name at each batch interval is generated based on *prefix* and *suffix*: *"prefix-TIME_IN_MS[.suffix]"*. Python API This is not available in the Python API.
foreachRDD(*func*)	The most generic output operator that applies a function, *func*, to each RDD generated from the stream. This function should push the data in each RDD to a external system, like

	saving the RDD to files, or writing it over the network to a database. Note that the function *func* is executed in the driver process running the streaming application, and will usually have RDD actions in it that will force the computation of the streaming RDDs.