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.
- Data can be ingested from many sources like Kafka, Flume, 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 (micro batches), which are then processed by the Spark engine to generate the final stream of results in batches.
- The continuous streaming of data are captured for a particular interval of short duration such as 3 seconds or 5 seconds which is configurable and then its processed as a micro batch.
- Spark Streaming provides a high-level abstraction called ***discretized stream*** or ***DStream***, which represents a continuous stream of data.
- It receives input from various sources like Kafka, Flume, Kinesis, or TCP sockets.
- It can also be a data stream generated by transforming the input stream.
- At its core, **DStream is a continuous stream of RDD (Spark abstraction).**
- DStreams can be created either from input data streams from sources such as Kafka, Flume, and Kinesis, or by applying high-level operations on other DStreams. **Internally, a DStream is represented as a sequence of RDDs.**

For example:
Lets see the below basic code.

```
import org.apache.spark._
import org.apache.spark.streaming._
import org.apache.spark.streaming.StreamingContext._ // not necessary since Spark 1.3
```

The above import statements are required, so that we can refer its classes/methods to perform steaming functionalities.

The below couple of statements are important to understand streaming.

```
val conf = new SparkConf().setMaster("local[2]").setAppName("NetworkWordCount")
val ssc  = new StreamingContext(conf, Seconds(3))
```

- Streaming is not part of SparkSession. Its part of a separate jar which is imported at the top.
- In the above commands, we are creating instance for SparkConf() class along with master and name.
- Then here we create StreamingContext() which will accept stream of inputs for given time frame.
- In the above example streamingContext is created with couple of inputs.
    - Configuration such as master and app name.
    - The time frame to accumulate data and process it as batch.
    - In above example, it accumulates data for each 3 seconds and process it as micro batch.

Once the StreamingContext() is created, we need to create a **Dstream** by which we can decide from where we are going to stream the data.

```
val lines = ssc.textFileStream("file:///home/hduser/sparkdata/streaming/")
```

(or)

```
val lines = ssc.socketTextStream("localhost", 9999)
```

(or)

```
val stream = KafkaUtils.createDirectStream[String, String]
                                    (ssc,   referConsistent,Subscribe[String, String]
                                                        (topics, kafkaParams))
```

- In the above examples, we have created three different types of Dstream.
- **textFileStream** is the kind of Dstream which accumulates streaming data from the files created at continuous real time at specified location.
- **socketTextStream** is the kind of Dstream which accumulates data from socket memory of a specific IP address and its specific port.
- **createDirectStream** is the Dstream which we create to accumulate data from Kafka topic.
    - The topic and Kafka broker information are passes as input to this Dstream. (in above example topics and kafkaParams)

Note that when these lines are executed, Spark Streaming only sets up the computation it will perform

when it is started, and no real processing has started yet. To start the processing after all the transformations have been setup, we finally call

```
ssc.start()             // Start the computation
ssc.awaitTermination()  // Wait for the computation to terminate
```

Once the above command is executed **"ssc.start()",** then only the StreamingContext initiates its process and will run continuously till any manual termination is applied to the streaming which will captured by the class **"ssc.awaitTermination()"**

Every input DStream (except file stream, discussed later in this section) is associated with a **Receiver** (Scala doc, Java doc) object which receives the data from a source and stores it in Spark's memory for processing.

Spark Streaming provides two categories of built-in streaming sources.

- **Basic sources:** Sources directly available in the StreamingContext API. Examples: file systems, and socket connections.
- **Advanced sources:** Sources like Kafka, Flume, Kinesis, etc. are available through extra utility classes. These require linking against extra dependencies as discussed in the linking section.

Points to remember

- When running a Spark Streaming program locally, do not use "local" or "local[1]" as the master URL. Either of these means that only one thread will be used for running tasks locally. If you are using an input DStream based on a receiver (e.g. sockets, Kafka, Flume, etc.), then the single thread will be used to run the receiver, leaving no thread for processing the received data. Hence, when running locally, always use "local[*n*]" as the master URL, where *n* > number of receivers to run.
- Extending the logic to running on a cluster, the number of cores allocated to the Spark Streaming application must be more than the number of receivers. Otherwise the system will receive data, but not be able to process it.

Dstream Operations:

Like RDD, Spark DStream also support two types of Operations: Transformations and output Operations-

1. Transformation

There are two types of transformation in DStream:

- **Stateless Transformations**
- **Stateful Transformations**

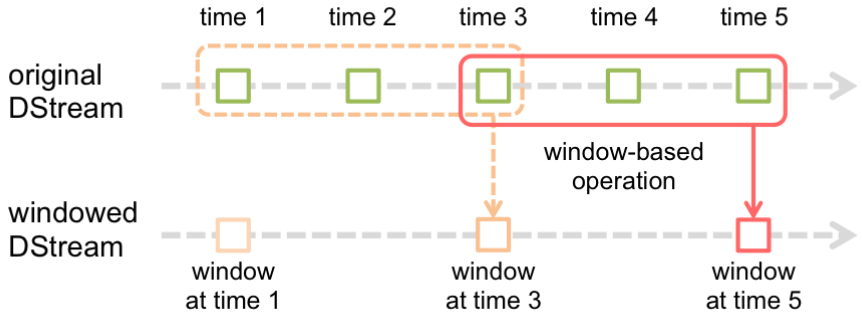
Stateless Transformations:

- **The processing of each batch has no dependency on the data of previous batches**. **Stateless transformations are simple RDD transformations.** It applies on every batch meaning every RDD in a DStream. It includes common RDD transformations like map(), filter(), reduceByKey() etc.
  Although these functions seem like applying to the whole stream, each DStream is a collection of many RDDs (batches). As a result, each stateless transformation applies to each RDD.
- Stateless transformations are capable of combining data from many DStreams within each time step. For example, key/value DStreams have the same join-related transformations as RDDs— cogroup(), join(), leftOuterJoin() etc.
- We can use these operations on DStreams to perform underlying RDD operations on each batch.

- If stateless transformations are insufficient, DStreams comes with an advanced operator called transform(). transform() allow operating on the RDDs inside them. The transform() allows any arbitrary RDD-to-RDD function to act on the DStream. This function gets called on each batch of data in the stream to produce a new stream.
- DStreams support many of the transformations available on normal Spark RDD's. Some of the common ones are as follows.
    - **map(func)**
        - Return a new DStream by passing each element of the source DStream through a function func.
    - **flatMap(func)**
        - Similar to map, but each input item can be mapped to 0 or more output items.
    - **filter(func)**
        - Return a new DStream by selecting only the records of the source DStream on which func returns true.
    - **repartition(numPartitions)**
        - Changes the level of parallelism in this DStream by creating more or fewer partitions.
    - **union(otherStream)**
        - Return a new DStream that contains the union of the elements in the source DStream and otherDStream.
    - **count()**
        - Return a new DStream of single-element RDDs by counting the number of elements in each RDD of the source DStream.
    - **reduce(func)**
        - Return a new DStream of single-element RDDs by aggregating the elements in each RDD of the source DStream using a function func (which takes two arguments and returns one). The function should be associative and commutative so that it can be computed in parallel.
    - **countByValue()**
        - When called on a DStream of elements of type K, return a new DStream of (K, Long) pairs where the value of each key is its frequency in each RDD of the source DStream.
    - **reduceByKey(func, [numTasks])**
        - When called on a DStream of (K, V) pairs, return a new DStream of (K, V) pairs where the values for each key are aggregated using the given reduce function. Note: By default, this uses Spark's default number of parallel tasks (2 for local mode, and in cluster mode the number is determined by the config property spark.default.parallelism) to do the grouping. You can pass an optional numTasks argument to set a different number of tasks.
    - **join(otherStream, [numTasks])**
        - When called on two DStreams of (K, V) and (K, W) pairs, return a new DStream of (K, (V, W)) pairs with all pairs of elements for each key.
    - **cogroup(otherStream, [numTasks])**
        - When called on a DStream of (K, V) and (K, W) pairs, return a new DStream of (K, Seq[V], Seq[W]) tuples.
    - **transform(func)**
        - Return a new DStream by applying a RDD-to-RDD function to every RDD of the source DStream. This can be used to do arbitrary RDD operations on the DStream.
    - **updateStateByKey(func)**
        - Return a new "state" DStream where the state for each key is updated by applying the given function on the previous state of the key and the new values for the key. This can be used to maintain arbitrary state data for each key.

Stateful Transformations:

- It uses data or intermediate results from previous batches and computes the result of the current batch. Stateful transformations are operations on DStreams that track data across time. Thus it makes use of some data from previous batches to generate the results for a new batch.
- The two main types are windowed operations, which act over a sliding window of time periods, and updateStateByKey(), which is used to track state across events for each key (e.g., to build up an object representing each user session).
- **Windowed Operation**
    - Spark Streaming also provides *windowed computations*, which allow you to apply transformations over a sliding window of data. The following figure illustrates this sliding window.



    -
    - As shown in the figure, every time the window slides over a source DStream, the source RDDs that fall within the window are combined and operated upon to produce the RDDs of the windowed DStream. In this specific case, the operation is applied over the last 3 time units of data, and slides by 2 time units. This shows that any window operation needs to specify two parameters.
        - window length - The duration of the window (3 in the figure).
        - sliding interval - The interval at which the window operation is performed (2 in the figure).
    - These two parameters must be multiples of the batch interval of the source DStream..

```
// Reduce last 30 seconds of data, every 10 seconds
val windowedWordCounts = pairs.reduceByKeyAndWindow((a:Int,b:Int) => (a + b), Seconds(30), Seconds(10))
```

    - **window(windowLength, slideInterval)**
        - Return a new DStream which is computed based on windowed batches of the source DStream.
    - **countByWindow(windowLength, slideInterval)**
        - Return a sliding window count of elements in the stream.
    - **reduceByWindow(func, windowLength, slideInterval)**
        - Return a new single-element stream, created by aggregating elements in the stream over a sliding interval using func. The function should be associative and commutative so that it can be computed correctly in parallel.
    - **reduceByKeyAndWindow(func, windowLength, slideInterval, [numTasks])**
        - When called on a DStream of (K, V) pairs, returns a new DStream of (K, V) pairs where the values for each key are aggregated using the given reduce function func over batches in a sliding window. Note: By default, this uses Spark's default number of parallel tasks (2 for local mode, and in cluster mode the number is determined by the config property spark.default.parallelism) to do the grouping.

You can pass an optional numTasks argument to set a different number of tasks.

- **reduceByKeyAndWindow(func, invFunc, windowLength, slideInterval, [numTasks])**
  - A more efficient version of the above reduceByKeyAndWindow() where the reduce value of each window is calculated incrementally using the reduce values of the previous window. This is done by reducing the new data that enters the sliding window, and "inverse reducing" the old data that leaves the window. An example would be that of "adding" and "subtracting" counts of keys as the window slides. However, it is applicable only to "invertible reduce functions", that is, those reduce functions which have a corresponding "inverse reduce" function (taken as parameter invFunc). Like in reduceByKeyAndWindow, the number of reduce tasks is configurable through an optional argument. Note that checkpointing must be enabled for using this operation.
- **countByValueAndWindow(windowLength, slideInterval, [numTasks])**
  - When called on a DStream of (K, V) pairs, returns a new DStream of (K, Long) pairs where the value of each key is its frequency within a sliding window. Like in reduceByKeyAndWindow, the number of reduce tasks is configurable through an optional argument.

- **UpdateStateByKey Operation**
  - 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.
  - Let's illustrate this with an example. Say you want to maintain a running count of each word seen in a text data stream. Here, the running count is the state and it is an integer. We define the update function as:

```
def updateFunction(newValues: Seq[Int], runningCount: Option[Int]): Option[Int] = {
   val newCount = ...  // add the new values with the previous running count to get the new count
   Some(newCount)
}

val runningCounts = pairs.updateStateByKey[Int](updateFunction _)
```

2. Output Operation

- Once we get the data after transformation, on that data output operation are performed in Spark Streaming.
- After the debugging of our program, using output operation we can only save our output. Some of the output operations are print(), save() etc..
- The save operation takes directory to save file into and an optional suffix. The print() takes in the first 10 elements from each batch of the DStream and prints the result.

For more detailed informations refer to the below link:
https://spark.apache.org/docs/2.2.0/streaming-programming-guide.html

Now let's see some examples:

Usecase 1:

File Streams: For reading data from files on any file system compatible with the HDFS API, a DStream can be created and manipulated in runtime in the given sequence of interval-

Step1:
Goto package – org.tech.streaming

Step2:
Create an object - filestream

Step3:
Replace whole content with the below code

```
package org.tech.streaming

import org.apache.spark._
import org.apache.spark.streaming._
import org.apache.spark.sql.SparkSession

object filestream {

 def main(args:Array[String])
 {

// Create the context
// val sparkConf = new SparkConf().setAppName("textstream").setMaster("local[*]")
//val sparkcontext = new SparkContext(sparkConf)
val sparkSession =
SparkSession.builder.appName("textstream").enableHiveSupport.master("local[*]").getOrCreate();
val sparkcontext = sparkSession.sparkContext;

sparkcontext.setLogLevel("ERROR")

val ssc = new StreamingContext(sparkcontext, Seconds(10))
// we create streamingcontext object with sparkcontext which is inside sparksession and the timeframe input

val lines = ssc.textFileStream("file:///home/hduser/sparkdata/streaming/")
//here we create a Dstream which is of type textFileStream specifying the source location.

/*in the below code, we apply flatmap delimited by single space to get all the words from source stream as a single column.
Then group by that column to get the count of each work. */
```

```
val courses = lines.flatMap(_.split(" "))
val coursesCounts = courses.map(x => (x, 1)).reduceByKey(_ + _)
coursesCounts.print()
ssc.start()
ssc.awaitTermination()
}
}
```

Step4:

Execute the below command in another linux terminal to create file stream in hdfs/linux.

```
cp /home/hduser/mrdata/courses.log /home/hduser/sparkdata/streaming/c1
cp /home/hduser/mrdata/courses.log /home/hduser/sparkdata/streaming/c2
cp /home/hduser/mrdata/courses.log /home/hduser/sparkdata/streaming/c3
```

Usecase 2:

Network Socket Streams:

This DStream represents the stream of data that will be received from the data server. Each record in this DStream is a line of text.

Create a DStream that represents streaming data from a TCP source, specified as hostname (e.g. localhost) and port (e.g. 9999).

Step1:

Create a package – org.tech.streaming

Step2:

Create an object - socketstream

Step3:

Replace whole content with the below code

```
package org.tech.pkg
import org.apache.spark.SparkConf
import org.apache.spark.SparkContext
import org.apache.spark.streaming.{Seconds, StreamingContext}
import StreamingContext._
import org.apache.spark.SparkContext._
import java.lang.Math
object socketstream1 {

//import the required streaming classes
```

```
 def main(args:Array[String])

 {

//val sparkConf = new SparkConf().setAppName("auctionsocketstream").setMaster("local[2]")
//val sparkcontext = new SparkContext(sparkConf)

val sparkSession =
SparkSession.builder.appName("auctionsocketstream").enableHiveSupport.master("local[*]").getOrCreate();
val sparkcontext = sparkSession.sparkContext;
//note: here we create instance for SparkSession along with command to enable hive support.
sparkcontext.setLogLevel("ERROR")

//index names
val auctionid = 0
val bid = 1
val bidtime = 2
val bidder = 3
val bidderrate = 4
val openbid = 5
val price = 6
val itemtype = 7
val daystolive = 8

   // Create the context
val ssc = new StreamingContext(sparkcontext, Seconds(20))
val lines = ssc.socketTextStream("localhost", 9999)
val auctionRDD = lines.map(line => line.split("~"))

 val items_auctionRDD = auctionRDD.map(x => (x(itemtype), 1)).reduceByKey((x, y) => x + y)

   //Identify which item has more auction response

 items_auctionRDD.print()

   //total number of items (auctions)

 val totalitems = auctionRDD.map(line => line(0)).count()
totalitems.print()
items_auctionRDD.saveAsTextFiles("hdfs://localhost:54310/user/hduser/auctionout/auctiondata")

   ssc.start()
   ssc.awaitTermination()
   //ssc.awaitTerminationOrTimeout(50000)
 }
}
```

Step4:

Open the netcat socket and start paste the below content multiple times:

nc -lk 9999

//This is the command the socket with port number 9999. And then paste the below text data.

3406945791~30~0.02539~shanie713~0~9.99~232.5~palm~7
3406945791~52~0.0575~jvross524~-1~9.99~232.5~palm~7
3406945791~95~0.66248~yebo~441~9.99~232.5~ssd~7
3406945791~125~0.66262~yebo~441~9.99~232.5~mmc~7
3406945791~145~0.6628~yebo~441~9.99~232.5~tab~7
3406945791~152.5~0.66304~yebo~441~9.99~232.5~laptop~7
3406945791~175~0.76547~lilbitreading~2~9.99~232.5~laptop~7
3406945791~195~0.80953~jaguarhw~1~9.99~232.5~pc~7
3406945791~195~0.93341~sirvinsky~52~9.99~232.5~mmc~7
3406945791~200~0.93351~sirvinsky~52~9.99~232.5~palm~7
3406945791~200~1.10953~jaguarhw~1~9.99~232.5~palm~7
3406945791~215~3.68168~lilbitreading~2~9.99~232.5~palm~7
3406945791~205~4.04531~jaguarhw~1~9.99~232.5~palm~7
3406945791~225~4.04568~jaguarhw~1~9.99~232.5~palm~7
3406945791~220~6.60786~robb1069~3~9.99~232.5~palm~7
3406945791~225~6.60799~robb1069~3~9.99~232.5~palm~7
3406945791~230~6.60815~robb1069~3~9.99~232.5~palm~7
3406945791~230~6.67638~jaguarhw~1~9.99~232.5~palm~7
3406945791~232.5~6.67674~jaguarhw~1~9.99~232.5~palm~7

Usecase 3:

Kafka Streams:

This kafka consumer DStream represents the stream of data that will be received from multiple sources pushed to kafka topics directly or using tools like NIFI. Each record in this DStream is a line of text.

Create a createDirectStream kafka consumer Dstream that represents streaming data from a Kafka queue, specified with topic, broker list etc.

Step1:

Goto package – org.tech.streaming

Step2:

Create an object - kafkastream

Step3:

Replace whole content with the below code

```
package org.tech.streaming import
org.apache.spark.SparkConf import
org.apache.spark.SparkContext
import org.apache.spark.streaming.{Seconds, StreamingContext}
import StreamingContext._
import org.apache.kafka.clients.consumer.ConsumerRecord
import org.apache.kafka.common.serialization.StringDeserializer
import org.apache.spark.streaming.kafka010._
import org.apache.spark.streaming.kafka010.LocationStrategies.PreferConsistent
import org.apache.spark.streaming.kafka010.ConsumerStrategies.Subscribe

object kafkastream {

 def main(args:Array[String])
 {
val sparkConf = new SparkConf().setAppName("kafkastream").setMaster("local[*]")
val sparkcontext = new SparkContext(sparkConf)
val ssc = new StreamingContext(sparkcontext, Seconds(10))

//ssc.checkpoint("checkpointdir")
val kafkaParams = Map[String, Object](
"bootstrap.servers" -> "localhost:9092",
        "key.deserializer" -> classOf[StringDeserializer],
        "value.deserializer" -> classOf[StringDeserializer],
        "group.id" -> "kafkatest1",
        "auto.offset.reset" -> "earliest"         )

      val topics = Array("tk1")

   /*now we need to create a Dstream which is of type DirectStream which supports Kafka.
   Before creating the Dstream, we need to define the Kafka related input which is to be passed to the Dstream.
   Inputs for createDirectStream () are
   1. The Locatoin Strategy
            - PreferConsistent
                  o In most cases, we should use LocationStrategies.PreferConsistent as shown above.
                  o This will distribute partitions evenly across available executors.
            -PreferBrokers
                  o If your executors are on the same hosts as your Kafka brokers, use PreferBrokers, which will
                    prefer to schedule partitions on the Kafka leader for that partition.
            -PreferFixed
                  o Finally, if you have a significant skew in load among partitions, use PreferFixed. This allows
```

you to specify an explicit mapping of partitions to hosts (any unspecified partitions will use a consistent location).

2. **subscription parameters**

To subscribe to a particular topic in Kafka, we need to specify couple of inputs.

- **Kafka topic**
- **Kafka config parameters.** such as
  - **Bootstrapserver**
    - This contains info about the IP address of broker and its port number.
    - Multiple brokers can be specified by separated by comma.
  - **DeSerializer**
    - While producing message into Kafka it would have been serialized.
    - We need to use the same type of format to deSerialize the message while consuming.
  - **GroupID**
    - If multiple consumers contact a particular topic with same group id, then the available partitions for the message inside that topic will be shared between those consumers. Not all partitions will be given to both consumers. It will be share between those consumers to achieve efficiency.
    - If multiple consumers with different group id contacts kafka topic, then all the available partitions of that message are given each consumer. Will not be shared between them.
  - **Auto offset reset.**
    - use auto.offset.reset to define the behavior of the consumer when there is no committed position (which would be the case when the group is first initialized) or when an offset is out of range. You can choose either to reset the position to the "earliest" offset or the "latest" offset (the default).

```
        */
    val stream = KafkaUtils.createDirectStream[String, String](ssc,
    PreferConsistent,
    Subscribe[String, String](topics, kafkaParams)
   )

val kafkastream = stream.map(record => (record.key, record.value))
val inputStream = kafkastream.map(rec => rec._2);
val words = inputStream.flatMap(_.split(" "))
val results = words.map(x => (x, 1L)).reduceByKey(_ + _)
inputStream.print();
results.saveAsTextFiles("hdfs://localhost:54310/user/hduser/kafkastreamout/outdir")
ssc.start()
ssc.awaitTermination()
 } }
```

Step4:

Check if zookeeper and kafka are running, if not use the below commands to start it and start the producer to push some random space delimited data.

Start the Zookeeper Coordination service:

zookeeper-server-start.sh -daemon /usr/local/kafka/config/zookeeper.properties

Start the Kafka server:

kafka-server-start.sh -daemon /usr/local/kafka/config/server.properties

kafka-console-producer.sh --broker-list localhost:9092 --topic tk1

Keyin some messages

Additional Usecase:

Execute the file streaming method with the integration of dataframe, DSL and Declarative queries in one program.

```
package org.tech.streaming
import org.apache.spark._
import org.apache.spark.streaming._
import org.apache.spark.sql.SparkSession

object bidsfilestream {

case class Auction(auctionid: String, bid: Float, bidtime: Float, bidder: String, bidderrate:
Integer, openbid: Float, price: Float, item: String, daystolive: Integer)

def main(args:Array[String])
{

//Old method
  //val sparkConf = new SparkConf().setAppName("textstream").setMaster("local[*]")
  //val sparkcontext = sparkSession.SparkContext(sparkConf)

  // Create the context

  val sparkSession =
SparkSession.builder.appName("textstream").enableHiveSupport.master("local[*]").getOrCreate
();
val sparkcontext = sparkSession.sparkContext;
//val sqlcontext = sparkSession.sqlContext;

sparkcontext.setLogLevel("ERROR")

val ssc = new StreamingContext(sparkcontext, Seconds(10))

//val auctionRDD = sparkcontext.textFile("file:///home/hduser/sparkdata/streaming/");

val auctionRDD = ssc.textFileStream("file:///home/hduser/sparkdata/streaming/")

// create an RDD of Auction objects

// change ebay RDD of Auction objects to a DataFrame
import sparkSession.implicits._
```

```
// Foreach rdd function is an iterator on the streaming micro batch rdds as like map function

auctionRDD.foreachRDD(rdd => {

  if(!rdd.isEmpty)
        {

   val ebay = rdd.map(_.split("~")).map(p => Auction(p(0), p(1).toFloat, p(2).toFloat,p(3),
p(4).toInt, p(5).toFloat, p(6).toFloat, p(7), p(8).toInt))

   val ebaydf = ebay.toDF;

   ebaydf.createOrReplaceTempView("ebayview");

   val auctioncnt = sparkSession.sql("select count(1) from ebayview");

   print("executing table query\n")

   print("total auctions are\n")

   auctioncnt.show(1,false);


   print("Executing dataframe\n")

   print("total distinct auctions\n")


   val count = ebaydf.select("auctionid").distinct.count

   System.out.println(count);


   print("Executing dataframe\n")

   print("Max bid, sum of price of items are\n")


   import org.apache.spark.sql.functions._

   val grpbid =
ebaydf.groupBy("item").agg(max("bid").alias("max_bid"),sum("price").alias("sp")).sort($"sp"
.desc)

   grpbid.show(10,false);
```

```
        }
})

print("streaming executing in x seconds\n")

ssc.start()
ssc.awaitTermination()
}
}
```