Spark Streaming

Daniel Hagimont

https://www.google.fr/search?q=daniel+hagimont+home+page

1

This lecture is about the Spark Streaming framework from Google.

This is basically an extension from Spark that provides support for handling streams of data that are arriving continuously and should be handled in real time.

Many sources of data stream can be considered. The data stream may come from a file in HDFS, but this is not the typical case.

Spark streaming is often used with a distributed streaming platform which routes messages between endpoints. An example of popular streaming platforms is Apache Kafka (a sort of scalable JMS service).

Spark Streaming is able to connect with such a streaming platform in order to receive a stream of data. Received data is then processed in real time and the results can then be displayed on a dashboard or stored in a database or filesystem.

Internally, Spark Streaming is slicing the input data stream into batches of data. Each of these data batches are then considered by the Spark engine as datasets (which may be quite large) which are processed as in the previous lecture. The processing of each batch produces a result batch, thus generating a stream of result batches.
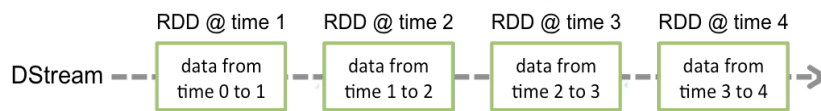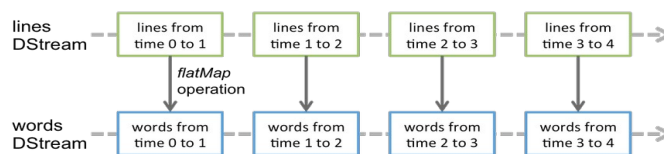
While Spark was relying on RDDs, Spark Streaming is relying on DStreams (discretized streams).

A DStream is a stream of data, sliced into a sequence of batches, each batch being a RDD.

Given a time interval T (a period), Spark Streaming stores all received data during the last interval in a RDD. Therefore, a DStream becomes a stream of RDDs.

When you program the processing of the stream, you program operations of DStreams, and this implies that these operations are applied to each RDD in the DStream.

For instance, if you receive a stream of text, the stream will be sliced into parts stored in RDDs and the wordcount application will count the words in each part/RDD. The result will be a wordcount table every time interval T.

WordCount example

```
SparkConf sparkConf = new SparkConf().setAppName("WordCountStreaming").setMaster("local[2]");
JavaStreamingContext jsc = new JavaStreamingContext(sparkConf, Durations.seconds(1));
JavaReceiverInputDStream<String> lines = jsc.socketTextStream("localhost", 9999);
JavaDStream<String> words = lines.flatMap(s -> Arrays.asList(s.split(" ")).iterator());
JavaPairDStream<String, Integer> wordCounts = words.mapToPair(s -> new Tuple2<>(s, 1))
                                        .reduceByKey((i1, i2) -> i1 + i2);
wordCounts.print();
jsc.start();
jsc.awaitTermination();
```

- Requires at least 2 threads/cores
  - One for processing / one per input receiver
- Receiver
  - Batch interval of 1 second
  - Data are received from TCP address localhost/9999
- Transformations/actions are about the same
  - Applied to each RDD in the Dstream

4

Here is the implementation of the WordCount example, applied to an incoming stream of text.

As with Spark, an initial step is to create a configuration object and then create a context object (here a JavaStreamingContext). Notice here that the context object is initialized with a duration (period) of 1 second. This means that Spark Streaming will create batches (RDDs) every second.

Then, we need to define where the data stream is coming from. Spark Streaming allows defining basic sources (file, socket), and advanced sources (like Kafka) require additional classes to be installed.

Here, we use socketTextStream() which creates a DStream which takes its data from a TCP source (localhost, 9999). The returned DStream is a JavaReceiverInputDStream. This DStream is a receiver, which means that it requires to be allocated a thread/core for functioning (so if you run the application locally, you need to specify local[2], one core for the receiver and one for processing. In a distributed setting, the receiver is in the Driver program and computations are on slave nodes).

Then, the treatments are almost the same as for Spark.

- flatMap() allows transforming a DStream of lines into a DStream of words. Remember that within a DStream, the operation are applied to each RDD.

- mapToPair() allows generating a JavaPairDStream of <word, 1>

- reduceByKey() allows obtaining the wordcount (continuously for each RDD)

- print() allows printing the result DStream (i.e. each RDD within the DStream)

Notice that before we start(), no data was read nor processed. The execution of all the statements before start() only registered the operations that are to be performed on incoming data (RDDs).

Execution

```
$  spark-submit --class WordCountStreaming --master local[2] wc.jar

-------------------------------------------
Time: 1513517145000 ms
-------------------------------------------


-------------------------------------------
Time: 1513517146000 ms
-------------------------------------------
(hello,2)
(bye,2)
(world,1)

-------------------------------------------
Time: 1513517147000 ms
-------------------------------------------


-------------------------------------------
Time: 1513517148000 ms
-------------------------------------------
(hello,1)
(again,1)
```

```
$ nc -lk 9999
hello world hello
bye bye
...
hello again
```

To execute the application, we first need to start a TCP server which will accept connections and send data. We use "nc" which implements such a server, the sent data being what is entered on STDIN.

We observe that every second, the Spark Streaming application display the result of wordcount on each RDD. Notice that a separate wordcount is computed on each RDD.

Input Dstreams: receivers

- Input Dstreams
  - Basic: file system, socket ...
  - Advanced: Kafka ...
- Input Dstreams are associated with a receiver
  - Must be allocated a thread/core
  - Except file streams
- Can create custom receivers

6

Input DStreams may be

- basic (file, socket), builtin in Spark Streaming

- advanced (e.g. Kafka) requiring the installation of specific extensions (drivers)

Custom receivers can be created to connect with specific streaming platforms.

Transformations on DStreams

map(func)	Return a new DStream by passing each element of the source DStream through a function func.
filter(func)	Return a new DStream by selecting only the records of the source DStream on which func returns true.
flatMap(func)	Similar to map, but each input item can be mapped to 0 or more output items.
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.

Operations that can be invoked on DStreams are about the same as those of RDDs.

Transformations on DStreams

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.

8

Some other operations.

An interesting operation is the last one in this list.

As we have seen, the wordcount application computes a wordcount for each RDD in the initial DStream.

The updateStateByKey() operation allows to manage a global wordcount for the application (details in the next slide).

WordCount: another implementation

```
Function2<List<Integer>, Optional<Integer>, Optional<Integer>> updateFunction =
                    (values, state) -> {
                                Integer newSum = state.or(0);    // the value of state if defined, else 0
                                for (Integer i : values) newSum += i;
                                return Optional.of(newSum);    // create an Optional
                    };
SparkConf sparkConf = new SparkConf().setAppName("WordCountStreaming").setMaster("local[2]");
JavaStreamingContext jsc = new JavaStreamingContext(sparkConf, Durations.seconds(1));
jsc.checkpoint(".");

JavaReceiverInputDStream<String> lines = jsc.socketTextStream("localhost", 9999);
JavaDStream<String> words = lines.flatMap(s -> Arrays.asList(s.split(" ")).iterator());
JavaPairDStream<String, Integer> wordCounts = words.mapToPair(s -> new Tuple2<>(s, 1))
                                .updateStateByKey(updateFunction);
wordCounts.print();
jsc.start();
jsc.awaitTermination();
```

- Manage a global wordcount state (not per RDD in the stream)
  - Optional is a container for any data type
  - Checkpointing must be activated

9

We describe here how to use updateStateByKey() to manage a global wordcount.

To be able to use updateStateByKey(), we need to enable checkpointing. Checkpointing is a service which stores data (metadata or data, i.e. RDD) to stable storage (e.g. HDFS) in order to be able to resume the application in case of failure. Here we don't describe the recovery procedure (described later), we only enable checkpointing. This is done with checkpoint(), the parameter being the directory where checkpointed data are stored.

Most of the code is the same, except that instead of doing a reduceByKey() with an additioner, we call updateStateByKey() with an updateFunction.

reduceByKey() would have computed a separate wordcount for each RDD in the DStream.

updateStateByKey() groups all values behind a unique key (for instance <w1, {1,1,1}>, <w2, {1,1}>) and then calls for each key (say w1) the updateFunction which is responsible for updating (with {1,1,1}) the global state for this key.

The updateFunction has:

- first parameter : the list of values ({1,1,1}) to integrate in the global state

- second parameter : the current global state

- the returned value is the new global state

Optional<T> is a container for T, allowing the value of T to be undefined. They use Optional since the global state may not be defined if it is the first time we update it.

When executing this version of wordcount, we observe that every time slice (second), the global wordcount table is displayed.

Window operations
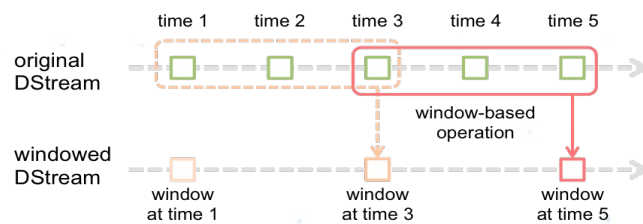
Apply transformations over a sliding window of data
- The operation is applied over the last 3 time units of data (window), and slides by 2 time units (slide)
- RDDs are fusioned

```
JavaPairDStream<String, Integer> windowedWordCounts =
       pairs.reduceByKeyAndWindow((i1, i2) -> i1 + i2,
       Durations.seconds(60), Durations.seconds(5));
```

window   slide   10

Another interesting feature of Spark Streaming is the concept of window operation.

It allows to apply operations on a sliding window of data. Every time the window slides over a source DStream, the source RDDs that fall within the window are combined to produce a new RDD in the windowed DStream.

In the figure, the window size is 3 time units, and the slide size is 2 time units.

In the wordcount example, if "pairs" is the DStream which includes <word, 1>, then the code here will compute and display a wordcount table over the window of the last minute, and it will slide of 5 seconds (sliding of about 5 RDDs as RDDs are generated every second).

Notice that data (RDDs in the original DStream) are reused many times in the RDDs of the windowed DStream.

Window operations

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.
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.

11

Many operations have their window version.

Output operations on DStreams

print()	Prints the 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.
saveAsTextFiles(prefix, [suffix])	Save this DStream's contents as 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 SequenceFiles of serialized Java objects. The file name at each batch interval is generated based on prefix and suffix: "prefix-TIME_IN_MS[.suffix]".
saveAsHadoopFiles(prefix, [suffix])	Save this DStream's contents as Hadoop files. The file name at each batch interval is generated based on prefix and suffix: "prefix-TIME_IN_MS[.suffix]".
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 an external system, such as 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.

12

Output operations on DStreams.

Checkpointing

- **Saving RDDs and metadata in reliable storage**
  - Periodic RDD checkpointing
- **Required for**
  - Statefull transformations (updateStateByKey or reduceByKeyAndWindow)
- **Programming**
  - Enabled by setting a directory in a filesystem (local, HDFS …)

  ```
  jsc.checkpoint(checkpointDirectory)
  ```

  - Re-create a StreamingContext from the checkpoint data on restart

Checkpointing allows making periodic snapshots of the state of the application, so that the state can be recovered in case of failure and the application resumed without any loss. A checkpoint saves on a reliable storage (local disk, HDFS) the state of RDDs and metadata about the execution state of the application.

Checkpointing is required if you want to use global states and windowing.

The first thing to do is to enable checkpointing after the creation of the JavaStreamingContext (jsc). The parameter indicates where checkpointing data will be saved.

Then, when the application starts (at the beginning of the main()), you must either :

- create everything (JavaStreamingContext, DStreams …..) if this is a normal start

- recover from the last checkpoint if there is one

Checkpointing

```
// Create a JavaStreamingContext and initialize treatments
Function0<JavaStreamingContext> create = () -> {
    JavaStreamingContext jsc = new JavaStreamingContext(...);
    jsc.checkpoint(checkpointDirectory);
    JavaReceiverInputDStream<String> lines = jsc.socketTextStream(…);
        ...
    return jsc;
    }
};

// Recover JavaStreamingContext from checkpoint data or create a new one
JavaStreamingContext jsc = JavaStreamingContext.getOrCreate(checkpointDirectory,
                                                            create);

jsc.start();
```

This is the typical sequence of code at the beginning of the main(), when using checkpoints.

In the create() function which returns a JavaStreamingContext, we put everything we were doing in our application without checkpoint, except that we enable checkpointing with the checkpoint() method.

Then the beginning of the main is calling getOrCreate(checkpointDirectory, create).

This method checks whether there's a checkpoint saved in the directory (checkpointDirectory). If there's one, it recovers the application from this checkpoint. If there isn't any, it executes the create() function which creates everything.

Afterwards, in both case, the application is started (or resumed) with the start() method.

To test this, you run the wordcount application (the global state version). You kill the process and then restart the application. You should observe that the global state if restored.

NB: to make it work, you have to kill the process (kill -9). A Ctrl-C doesn't work as the interrupt/exception is caught and the application terminates normally (not like a failure) and cancels checkpointing.

Development/deployment

- In Eclipse
  - Jars
    - Must add the import of spark-streaming_2.11-2.4.3.jar
- Deployments
  - The same
    - spark-submit

Using Spark Streaming is similar to Spark.

In eclipse, you just have to add a jar.

You submit your application to Spark the same way with spark-submit.

Exercise

- **Same as in the previous lecture**
  - A set of stores
  - A log (file) of purchases (transactions)
    - storeid,productid,number,totalprice
      - storeid : the identifier of the store
      - productid : the identifier of the product
      - number : the number of products sold in the current transaction
      - price : the total price for the transaction (a product may have different prices in different stores)
- **Now you receive these records as a stream. You must compute**
  - the best selling product from the last 10 days
  - the best selling product from the beginning

16

We consider the same example as in the previous lecture.

The large log of sales from a set a stores.

But now, we receive the records (lines) as a stream.

The first question : show the best selling product from the last 10 days

Exercise

- Assuming the same function as in the previous lecture

```
class GetProductNumber implements PairFunction<String, String, Integer> {
        public Tuple2<String, Integer> call(String s) {
                …
                        return new Tuple2<String, Integer>(productid,number);
}
```

- We compute totals separately

```
JavaStreamingContext ssc = new JavaStreamingContext(conf, Durations.seconds(10));
JavaReceiverInputDStream<String> lines = ssc.socketTextStream("localhost", 9999);
JavaPairDStream<String, Integer> sells = lines.mapToPair(new GetProductNumber());
JavaPairDStream<String, Integer> counts = sells.reduceByKeyAndWindow((v1, v2) -> v1+v2,
                                        Durations.minutes(14400), Durations.minutes(1));
JavaDStream<Tuple2<String, Integer>> bestsell =
                        counts.reduce((t1,t2) -> (t1._2 > t2._2) ? t1 : t2);
bestsell.print();
ssc.start();
ssc.awaitTermination();
```

17

We assume the same function as in the previous lecture (GetProductNumber which extracts from a line a pair <productid, number>).

Then :

- mapToPair() returns a DStream of <productid, number>

- reduceByKeyAndWindow() returns a DStream of <productid, number> where each productid is unique and number is the total number of sales for that product over 10 days (14400 seconds). The window is updated every minute.

- reduce() returns a DStream of <productid, number> which gives every minute the best sale.

The second question has to use updateStateByKey().