# State and DStreams

Big Data Analysis with Scala and Spark

Heather Miller

# State?

So far, we've approached Spark Streaming in the same way we have approached regular Spark.

*Assumption so far:*

**Functional transformations on immutable data.**

# State?

So far, we've approached Spark Streaming in the same way we have approached regular Spark.

*Assumption so far:*
**Functional transformations on immutable data.**

However,

**What if you need to the accumulate and aggregate the results from the start of the streaming job?**
*Which means you need to check the previous state of the RDD in order to do something with the current RDD.*

# State?

So far, we've approached Spark Streaming in the same way we have approached regular Spark.

*Assumption so far:*
**Functional transformations on immutable data.**

However,

**What if you need to the accumulate and aggregate the results from the start of the streaming job?**
*Which means you need to check the previous state of the RDD in order to do something with the current RDD.*

To handle this, Spark supports:
**Stateful Streams**

# A Stateful Streaming Example...

Let's say we want to find out when (timestamp) a user performed his or her first and last activity in a given dataset in a stream.

Given incoming event with the following fields:

```
case class UserEvent(
  user: String,
  timestamp: java.sql.Timestamp,
  activity: String)
```

# updateStateByKey

Spark Streaming provides a method, `updateStateByKey` which manages this state per key (assuming we have formed a key-value pair from our original case class).

```
def updateStateByKey[S](updateFunc: (Seq[V], Option[S]) => Option[S])
```

**updateStateByKey** requires a function which accepts:

1. `Seq[V]` - The list of new values received for the given key in the current batch
2. `Option[S]` - The state we're updating on every iteration.

# Using `updateStateByKey`

In order to define a function `updateFunc` to pass to `updateStateByKey`, we have to figure out two things.

# Using `updateStateByKey`

In order to define a function `updateFunc` to pass to `updateStateByKey`, we have to figure out two things.

1. **Define the state.** The state can be an arbitrary data type.
2. **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.

# A Stateful Streaming Example...

Let's say we want to find out when (timestamp) a user performed his or her first and last activity in a given dataset in a stream.

```
case class UserEvent(
  user: String,
  timestamp: java.sql.Timestamp,
  activity: String)
```

**What should our state look like?**

# A Stateful Streaming Example...

Let's say we want to find out when (timestamp) a user performed his or her first and last activity in a given dataset in a stream.

```
case class UserEvent(
  user: String,
  timestamp: java.sql.Timestamp,
  activity: String)
```

## What should our state look like?

In order to figure out whether the event is the first or the last for a specific user, we need to carry some state forward between batches of events. E.g.,

```
case class UserSession(
  user: String,
  var activity: String,
  var start: java.sql.Timestamp,
  var end: java.sql.Timestamp)
```

# A Stateful Streaming Example... Continued.

We need to define an `updateFunc` which updates our state based on previous events that we will pass to `updateStateByKey`. We can do this in a helper function:

```
def updateUserStateWithEvent(newEvents: Seq[UserEvent],
                    state: Option[UserSession]): Option[UserSession] = {
  newEvents.map {  input =>
    //does the activity match for the given event
    if (state.activity == input.activity) {
        if (input.timestamp.after(state.end)) {
          state.end = input.timestamp
        }
        if (input.timestamp.before(state.start)) {
          state.start = input.timestamp
        }
      } else {
       //some other activity
        if (input.timestamp.after(state.end)) {
          state.start = input.timestamp
          state.end = input.timestamp
          state.activity = input.activity
        }
      }
      //return the updated state
      state
  }
}
```

# A Stateful Streaming Example... Continued.

Now, given a `DStream` called `stream` that processes elements of type `(String, UserEvent)` where the `String` key represents `UserEvent.user` we simply do:

```
stream.updateStateByKey(updateUserEvents)
```

# Another Stateful Streaming Example

Let's say we want to maintain a running count of each word seen in a text data stream.

# Another Stateful Streaming Example

Let's say we want to maintain a running count of each word seen in a text data stream.

This is a simpler case. Nonetheless, we need to define the following:

1. **Define the state.** The state can be an arbitrary data type.
2. **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.

# Another Stateful Streaming Example

Let's say we want to maintain a running count of each word seen in a text data stream.

**Defining our state:** Here, our state is the running count, and it is just an integer.

# Another Stateful Streaming Example

Let's say we want to maintain a running count of each word seen in a text data stream.

**Defining our state:** Here, our state is the running count, and it is just an integer.

**Defining our *update function*:**

```
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)
}
```

# Another Stateful Streaming Example

Let's say we want to maintain a running count of each word seen in a text data stream.

**We can now compute the running count of each word as follows**:

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

*This is applied on a DStream containing words, e.g., (word, 1) pairs.*

The update function will be called for each word, with `newValues` having a sequence of 1's (from the (word, 1) pairs) and the `runningCount` having the previous count.

# Problems with `updateStateByKey`

Caveats of `updateStateByKey`:

## Performance

For each new incoming batch, the transformation **iterates the entire state store**, regardless of whether a new value for a given key has been consumed or not.

This can effect performance especially when dealing with a large amount of state over time.

## No built-in timeouts

What would happen in our example if the event signaling the end of the user session was lost, or hadn't arrived for some reason?

One upside to the fact updateStateByKey iterates all keys is that we can implement such a timeout ourselves, but this should definitely be a feature of the framework.

# Introducing `mapWithState`

`mapWithState` is an alternative to `updateStateByKeys`. `mapWithState` comes with features we've been missing from `updateStateByKey`:

# Introducing `mapWithState`

`mapWithState` is an alternative to `updateStateByKeys`. `mapWithState` comes with features we've been missing from `updateStateByKey`:

1. **Built in timeout mechanism.** We can tell mapWithState the period we'd like to hold our state for in case new data doesn't come. Once that timeout is hit, `mapWithState` will be invoked one last time with a special flag (which we'll see shortly).
2. **Partial updates.** Only keys which have new data arrived in the current batch will be iterated. This means no longer needing to iterate the entire state store at every batch interval, which is a great performance optimization.
3. **Choose your return type.** We can now choose a return type of our desire, regardless of what type our state object holds.
4. **Initial state.** We can select a custom RDD to initialize our stateful transformation on startup.

# Introducing mapWithState

mapWithState is an alternative to updateStateByKeys.

```
def mapWithState[StateType, MappedType]
  (spec: StateSpec[K, V, StateType, MappedType]): DStream[MappedType]
```

# Introducing mapWithState

mapWithState is an alternative to updateStateByKeys.

```
def mapWithState[StateType, MappedType]
  (spec: StateSpec[K, V, StateType, MappedType]): DStream[MappedType]
```

**What is this StateSpec thing?**

# Introducing mapWithState

mapWithState is an alternative to updateStateByKeys.

```
def mapWithState[StateType, MappedType]
  (spec: StateSpec[K, V, StateType, MappedType]): DStream[MappedType]
```

### What is this StateSpec thing?

You put all of the things into StateSpec that you need for updating the state.

- The update function.
- An initial state as an RDD.
- Number of partitions.
- Partitioner.
- Timeout. This will ensure that keys whose values are not updated for a specific period of time will be removed from the state.

# Streaming word count with `mapWithState`

Let's start with implementing the function for updating the state in our streaming word count example.

# Streaming word count with `mapWithState`

Let's start with implementing the function for updating the state in our streaming word count example.

**Shape of our update function:**
The update function is called on a paired (key-value) DStream. The update function is called for every element in the paired DStream. The function takes the following input parameters:

- The current Batch Time
- The key for which the state needs to be updated
- The value observed at the 'Batch Time' for the key.
- The current state for the key.

The function should return the new (key, value) pair where value has the updated state information

# Streaming word count with `mapWithState`

Let's start with implementing the function for updating the state in our streaming word count example.

```
// In this example:
// - key is the word.
// - value is '1'. Its type is 'Int'.
// - state has the running count of the word. It's type is Long.
// - return value is the new (key, value) pair where value is the updated count.

def trackStateFunc(batch: Time, key: String, value: Option[Int], state: State[Long]):
  Option[(String, Long)] = {
  val sum = value.getOrElse(0).toLong + state.getOption.getOrElse(0L)
  val output = (key, sum)
  state.update(sum)
  Some(output)
}
```

# `mapWithState`'s State Specification

Now that we have an update function, we can define our state specification.

Remember, we can also set the following in the creation of our state specification:

- An initial state as an RDD.
- Number of partitions.
- Partitioner.
- Timeout.

# mapWithState's State Specification

Now that we have an update function, we can define our state specification.

```
val initialRDD = sc.parallelize(List(("dummy", 100L), ("source", 32L)))
val stateSpec = StateSpec.function(trackStateFunc)
                         .initialState(initialRDD)
                         .numPartitions(2)
                         .timeout(Seconds(60))
```

# mapWithState's State Specification

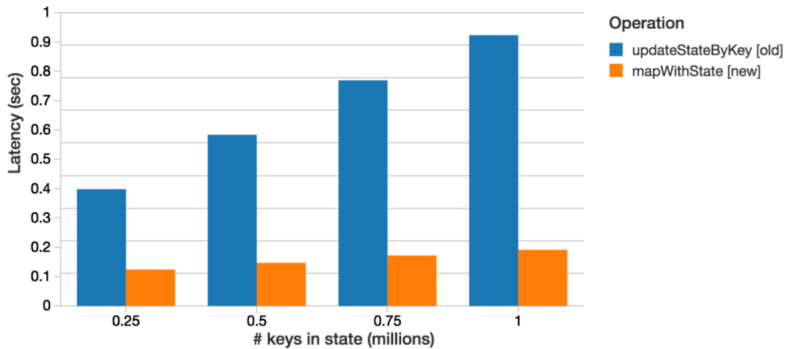
Now that we have an update function, we can define our state specification.

```
val initialRDD = sc.parallelize(List(("dummy", 100L), ("source", 32L)))
val stateSpec = StateSpec.function(trackStateFunc)
                         .initialState(initialRDD)
                         .numPartitions(2)
                         .timeout(Seconds(60))


val wordCountStateStream = wordStream.mapWithState(stateSpec)
wordCountStateStream.print()
```

# mapWithState vs updateStateByKey

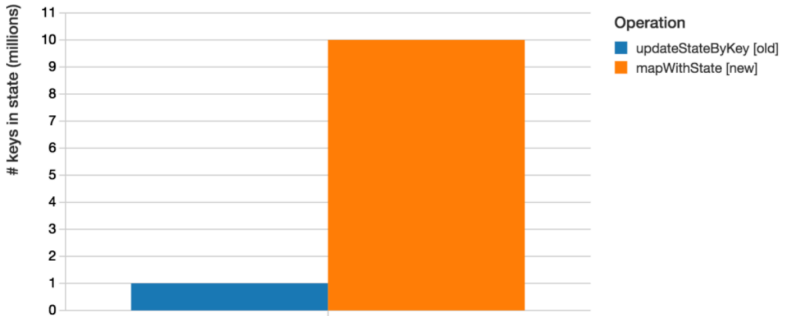
`mapWithState` has been been shown to provide can provide 6X lower latency.




*Up to 8X lower batch processing times (i.e.latency) with mapWithState than updateStateByKey*

# mapWithState vs updateStateByKey

Faster processing allows `mapWithState` to manage 10X more keys compared with `updateStateByKey` (with the same batch interval, cluster size, update rate in both cases).




*Up to 10X more keys in state with mapWithState than updateStateByKey*

# But what if our job fails?

What happens to the state that Spark is maintaining between batches if a node crashes?

Wasn't it **functional transformations on immutable data** that made Spark able to be fault-tolerant? What now, with all of this state?

# But what if our job fails?

What happens to the state that Spark is maintaining between batches if a node crashes?

Wasn't it **functional transformations on immutable data** that made Spark able to be fault-tolerant? What now, with all of this state?

To keep fault-tolerance and still enable stateful streaming, Spark supports

**Checkpointing**

# Checkpointing

A streaming application must operate 24/7 and hence must be resilient to failures unrelated to the application logic (e.g., system failures, JVM crashes, etc.)

For this to be possible, Spark Streaming needs to checkpoint enough information to a fault-tolerant storage system such that it can recover from failures.

# Checkpointing

A streaming application must operate 24/7 and hence must be resilient to failures unrelated to the application logic (e.g., system failures, JVM crashes, etc.)

For this to be possible, Spark Streaming needs to checkpoint enough information to a fault-tolerant storage system such that it can recover from failures.

**There are two types of data that are checkpointed.**

# Checkpointing

**There are two types of data that are checkpointed.**

1. **Metadata checkpointing.** Saving of the information defining the streaming computation to fault-tolerant storage like HDFS. This is used to recover from failure of the node running the driver of the streaming application. Meta-data includes:

- *Configuration.* The configuration that was used to create the streaming application.
- *DStream operations.* The set of DStream operations that define the streaming application.
- *Incomplete batches.* Batches whose jobs are queued but have not completed yet.

# Checkpointing

**There are two types of data that are checkpointed.**

2. **Data checkpointing.** Saving of the generated RDDs to reliable storage. This is necessary in some stateful transformations that combine data across multiple batches. In such transformations, the generated RDDs depend on RDDs of previous batches, which causes the length of the dependency chain to keep increasing with time. To avoid such unbounded increases in recovery time (proportional to dependency chain), intermediate RDDs of stateful transformations are periodically checkpointed to reliable storage (e.g. HDFS) to cut off the dependency chains.

# Checkpointing

**In sum:**

- **Metadata checkpointing** is primarily needed for recovery from driver failures, whereas
- **Data or RDD checkpointing** is necessary even for basic functioning if stateful transformations are used.