Using foreachRDD

Big Data Analysis with Scala and Spark

Heather Miller

Using `foreachRDD`

`foreachRDD` is an important and flexible primitive that allows data to be sent out to external systems.

Using foreachRDD

foreachRDD is an important and flexible primitive that allows data to be sent out to external systems.

```
dstream.foreachRDD { rdd =>
  val connection = createNewConnection()  // executed at the driver
  rdd.foreach { record =>
    connection.send(record) // executed at the worker
  }
}
```

What's wrong with this code?

Using `foreachRDD`

`foreachRDD` is an important and flexible primitive that allows data to be sent out to external systems.

```
dstream.foreachRDD { rdd =>
  val connection = createNewConnection()  // executed at the driver
  rdd.foreach { record =>
    connection.send(record) // executed at the worker
  }
}
```

What's wrong with this code?

This is incorrect as this requires the connection object to be serialized and sent from the driver to the worker. Such connection objects are rarely transferable across machines.

Using foreachRDD

```scala
dstream.foreachRDD { rdd =>
  rdd.foreach { record =>
    val connection = createNewConnection()
    connection.send(record)
    connection.close()
  }
}
```

What’s wrong with this code?

Using foreachRDD

```
dstream.foreachRDD { rdd =>
  rdd.foreach { record =>
    val connection = createNewConnection()
    connection.send(record)
    connection.close()
  }
}
```

What’s wrong with this code?

Typically, creating a connection object has time and resource overheads. Therefore, creating and destroying a connection object for each record can incur unnecessarily high overheads and can significantly reduce the overall throughput of the system.

A better solution is to use rdd.foreachPartition - create a single connection object and send all the records in a RDD partition using that connection.

Using `foreachRDD`

```
dstream.foreachRDD { rdd =>
  rdd.foreachPartition { partitionOfRecords =>
    val connection = createNewConnection()
    partitionOfRecords.foreach(record => connection.send(record))
    connection.close()
  }
}
```

This amortizes the connection creation overheads over many records.

Finally, this can be further optimized by reusing connection objects across multiple RDDs/batches.

Using foreachRDD

Note that the connections in the pool should be lazily created on demand and timed out if not used for a while. This achieves the most efficient sending of data to external systems.

```
dstream.foreachRDD { rdd =>
  rdd.foreachPartition { partitionOfRecords =>
    // ConnectionPool is a static, lazily initialized pool of connections
    val connection = ConnectionPool.getConnection()
    partitionOfRecords.foreach(record => connection.send(record))
    ConnectionPool.returnConnection(connection)  // return to the pool for future
  }
}
```