Apache Spark Certification Preparation & Developer Course Review

Databricks Training
July 2015
The purpose of this module is to prepare students for the topics in the O’Reilly Developer Certification for Apache Spark.¹
Exam Preparation

• By exam preparation we mean an overview of *learning objectives* which are tested by the exam itself.
• There are no lab exercises in this module.
• This is meant to be a review of topics only.
O’Reilly Recommends

O’Reilly recommends watching this Spark video\(^1\) from Paco Nathan:
O’Reilly Recommends

• O’Reilly also recommends reading this Spark book\(^1\) by Matei Zaharia, et al:
PySpark

• PySpark is just the Python Spark API\(^1\)
• While there are some differences, mostly RDDs have the same methods as Scala API equivalents.\(^2\)
• Python functions, collections, and lambdas are used:

```python
logData = sc.textFile(logFile).cache()
errors = logData.filter(lambda line: "ERROR" in line)
def is_error(line):
    return "ERROR" in line
errors2 = logData.filter(is_error)
error_keywords = ["Exception", "Error"]
def is_also_error(line):
    return any(keyword in line for keyword in error_keywords)
errors3 = logData.filter(is_also_error)
```
/* Base RDD lives at the “base” of the DAG. */

String DNApath1 = "~/Desktop/Dmel_Release5/na_armX.dmel.RELEASE5";

JavaRDD<String> baseRDD = sc.textFile(DNApath1);

JavaRDD<Tuple2<String, String>> pairsRDD = baseRDD.flatMap(new DNAFlatMapFunction("na_armX.dmel.RELEASE5"));
Transformation and Action

/* Transformation returns an RDD.
* Action does something, returns void or a number */

```
JavaRDD<Tuple2<String, String>> pairsRDD = DNARDD1.flatMap(
    new DNAFlatMapFunction(
        "na_armX.dmel.RELEASE5")
);
pairsRDD.count();
pairsRDD.foreach(new AccumulatorFunction(taAccumulator));
```

List<Tuple2<String, String>> pairsList = pairsRDD.take(50);
map(...) & flatMap(...) (1 of 2)

• Use map(...) and flatMap(...) and distinguish between the two transformations:

  • **map(Function f)**
    - There is only one method\(^1\) in the Function:
      - R call(T t)
      - Use when your call method must return a single value.

  • **flatMap(FlatMapFunction f)**
    - There is only one method in the FlatMapFunction:
      - Iterable<R> call(T t)
      - Use when your call method must return a Collection of values.
map(...) & flatMap(...) (2 of 2)

- Both transformations return an RDD<R>

- But flatMap(...) first “flattens” the collections returned into one collection in this case the resulting RDD.

- Remember in this “Reactive Extension” pattern:
  - RDDs push their contents into the map(...) or flatMap(...) functions and you decide what is the incoming type “T” is and the return type “R”

```java
class DNAFlatMapFunction implements FlatMapFunction<String, Tuple2<String, String>>, Serializable {
    public Iterable<Tuple2<String, String>> call(String line) {
        /* code */
    }
}
```
Lazy Evaluation of DAG

- When RDDs are created and then transformed using the RDD transformation methods, a logical Directed Acyclic Graph\(^1\) of Spark operations is created.
  - The RDDs are not actualized or created until an action is called to force the evaluation of the graph of operations.
    - This is what is meant by “lazy evaluation.”
    - There is no eager mode.
- This is done to maximize the efficiency of what is being requested.
- Spark core analyzes the graph and decides which data is required.

Spark knows what it’s doing, even if it’s lazy, we need to trust that…
Lazy Evaluation of DAG

/* Actions force DAG evaluation */

pairsRDD.collect()
pairsRDD.count()
pairsRDD.countByValue()
pairsRDD.first()
pairsRDD.fold(...)
pairsRDD.reduce(...)
pairsRDD.take(num)  // variations takeX(...)
pairsRDD.top(num)
pairsRDD.foreach(...)
Transformations and Shuffle

• We try to avoid the following transformations which may cause a Spark “Shuffle” operation…

• RDD and **DataSet** transformations:
  
  • `repartition(num)`
  • `coalesce(num)`
  • `groupByKey(...)`
  • `reduceByKey(...)`
  • `cogroup(...) // overloaded`
  • `join(...)`
  • `cartesian(...)`
Aggregation: \texttt{reduceByKey}(\ldots)

- For use with a \texttt{PairRDD} of some type…
- Similar to a “combiner” in Hadoop.
- The \texttt{reduceByKey}(\ldots) \footnote{wide transformation} combines values which are associated with the same key:
  - RDD \texttt{Before}: \{(1, 2), (3, 4), (3, 6)\}
  - RDD transformation\footnote{\texttt{transformation}}:
    - \texttt{rdd.reduceByKey( (x, y) => x + y )}
  - RDD \texttt{After}: \{(1, 2), (3, 10)\}
Persistence and Cache (1 of 3)

• RDD persist(...) method caches RDDs:
  • Now it doesn’t have to be recomputed.
  • More efficient (up to 10x faster).
  • Cache is fault-tolerant (automatically rebuilt)\(^1\)

• It needs to be at the right “storage level.”
• It needs to be at the right location in the DAG.

• The `cache()` method just calls `persist(StorageLevel.MEMORY_ONLY)`\(^2\)
  • This is the `default` storage level.
  • Other storage level options are shown on the next slide...
Persistence and Cache (2 of 3)

• More persistence Storage Levels:
  • **StorageLevel.MEMORY_ONLY_2**
    - Same as MEMORY_ONLY except replicate each partition on two cluster nodes
  • **StorageLevel.MEMORY_AND_DISK**
    - Store RDD as deserialized Java objects in the JVM. If the RDD does not fit in memory, store the partitions that don't fit on disk, and read them from there when they're needed.
  • **StorageLevel.MEMORY_AND_DISK_2**
    - And replicate each partition on two cluster nodes
  • **StorageLevel.MEMORY_ONLY_SER**
    - Store RDD as serialized Java objects (one byte array per partition). This is generally more space-efficient than deserialized objects, especially when using a fast serializer, but more CPU-intensive to read.
Persistence and Cache (3 of 3)

- **StorageLevel.MEMORY_AND_DISK_SER**
  - Similar to **MEMORY_ONLY_SER**, but spill partitions that don't fit in memory to disk instead of recomputing them on the fly each time they're needed.

- **StorageLevel.DISK_ONLY**
  - Store the RDD partitions only on disk.

- **StorageLevel.OFF_HEAP** (experimental)
  - Uses Tachyon\(^1\)
  - See: [http://tachyon-project.org](http://tachyon-project.org)
Broadcast Variables

- **Broadcast Variables:**
  - Are `broadcast(...)` by the SparkContext as read only.
  - Must be `Serializable`.
  - With increased cluster scale broadcast performs sub-linearly, using a highly efficient (O log n) algorithm.
  - After broadcast pass the variable to your Function:

```scala
Broadcast<GeneticCodeDNA> bvDNA = sc.broadcast(new GeneticCodeDNA());
JavaRDD<String> DNARDD1 = sc.textFile(DNAPath1);
JavaRDD<Tuple2<String, String>> pairs = DNARDD1.flatMap(new DNAFlatMapFunction(bvDNA));
```
Partitioning (1 of 3)

• Controlling the partitioning of your RDDs:
  • Reduces network communication:
  • Much more efficient.
  • Useful for multiple reading of the same RDD in key oriented reads (like a JOIN).

• Specify partitioning for key/value RDDs:
  • `JavaPairRDD<K,V>` for example

• These RDDs have a `partitionBy(Partitioner part)` method
  • Returns a transformed RDD partitioned using the specified partitioner.
Partitioning (2 of 3)

• Depends on the Partitioner used
  
  org.apache.spark.Partitioner is abstract base class

  • Known concrete subclasses:
    - HashPartitioner [default]
      • Guarantees keys with same hashcode value will appear on same node.
    - RangePartitioner

• For custom business domain knowledge, subclass Partitioner yourself:
  - Extend Partitioner and override\(^1\) a few methods:
    • public int getPartition(Object key)
    • public int numPartitions()
    • public boolean equals(…)

\(^1\)Override is the correct term, not “overriding.”
Partitioning (3 of 3)

• Partitioner Rules¹:

1. Each RDD has an optional `Partitioner` object.²
2. Any shuffle operation on an RDD with a `Partitioner` will respect that `Partitioner`.
3. Any shuffle operation on two RDDs will take on the `Partitioner` of one of them, if one is set.
4. Otherwise, by default use `HashPartitioner`.
Spark Cluster Sizing

• Spark Clusters are NOT limited (upward or downward) regarding how many nodes are in the cluster itself.
  
  • Largest known Spark cluster in production:
    - 8,000 nodes (China)
  
  • Smallest cluster:
    - one node 😊
Spark Streaming (1 of 2)

• Spark Streams are “discretized”\(^1\) over time:
  • Therefore known as DStreams.
  • A **StreamingContext** is needed first.
  • The **StreamingContext** Factory [GoF] methods create “DStreams”
  • A **DStream** is composed of RDDs containing data received\(^2\)
    – One RDD per period of time (batch interval) as configured in `Durations.seconds(num)`
    – DStream transformations are *stateless* by default

• Streaming also has “checkpoint” intervals whereby state of the DStream is saved to filesystem for more efficient recovery of lost data.\(^3\)
  • **Must be enabled in StreamingContext**
  • **Use-case**: Must be enabled for stateful transformations (see below)
  • **Use-case**: “Recovering from failures of the driver running the application - Metadata checkpoints are used to recover with progress information.”\(^4\)
Spark Streaming (2 of 2)

• A Spark Streaming application that is stateful leverages the ability to track data across time:
  • Uses previous data to compute current results.
  • Requires checkpointing.
  • Requires windowing, which is implemented by DStream method \texttt{window(...)} invocation\(^1\)
  • Windowed transformations are stateful.
  • Use \texttt{updateStateByKey(...)} for session building.
• DStreamLike convenience methods:
  - \texttt{reduceByWindow(...)}
  - \texttt{slice(Time fromTime, Time toTime)}
  - and more…
Error Conditions (1 of 2)

• The following checked Exception can be thrown when running a Spark operation:
  
  • org.apache.spark.SparkException: Job aborted due to stage failure: Task not serializable: java.io.NotSerializableException: ...

• Exception thrown when you initialize a variable on the driver and then try to pass it to a Function on one of the workers.
  
  • Spark will try to serialize the object and fail if the object is in fact not Serializable.
Error Conditions (2 of 2)

• Some remedies for this situation\(^\text{1}\):
  1. Make the class **Serializable**
  2. Declare the instance of that class only within the lambda function
  3. Make the not-Serializable object **static** and create it once per machine.
  4. Invoke `rdd.forEachPartition(...)` and create the not-Serializable object within this operation using a lambda.

- This allows use of that variable reference with all items “\(T\)” in the iterator for that **Partition**:

```java
void foreachPartition(VoidFunction<Iterator<T>> f)
```
Best Practices

• The `cache()` & `persist()` methods:
  • Use `persist()` to avoid re-evaluation of the RDD lineage.
  • Cache the RDDs which are most likely to be re-used in the application lower down in the DAG.
    - By lower-down we mean: children or grandchildren will be relying on that data, and we don’t want actions below to re-compute the parent RDD so we cache it.

• When NOT to invoke `collect()`:
  • When `collect()` can pull too much data from the Spark cluster back to the driver and crash the driver memory.

• When to use `saveAsTextFile(...)`:
  • When you need the RDD to be output to a single directory, as files (one per partition)\(^1\).
  • RDD may be *too large* to fit in memory on the driver machine.
public void showWordCount(String pathToFile) {
    JavaRDD<String> inputFile = sc.textFile(pathToFile);

    JavaRDD<String> words = inputFile.flatMap(new WordCountFlapMapFunction());

    JavaPairRDD<String, Integer> result =
        words.mapToPair(new WordCountPairFunction())
            .reduceByKey(new ReduceFunction<>(), 3); // num reducers
        // then work with result
}
public class WordCountFlatMapFunction implements FlatMapFunction<String, String>, Serializable {

    public Iterable<String> call(String rawLine) {
        String lowerCaseWords = rawLine.toLowerCase();
        char v = ':'; /* remove punctuation marks */
        char w = ',';
        char x = '.';
        char y = ' ';
        char z = ' ';
        String cleanerLine = lowerCaseWords
            .replace(v, z).replace(w, z)
            .replace(x, z).replace(y, z);
        return Arrays.asList(cleanerLine.split(" "));
    }
}
public class WordCountPairFunction implements PairFunction<String, String, Integer>, Serializable {
    public Tuple2<String, Integer> call(String word) {
        return new Tuple2(word, 1);
    }
}

public class ReduceFunction implements Function2<Integer, Integer, Integer>, Serializable {
    /* a is the old value (how many we found so far)
       b is the new value coming in: 1 */

    public Integer call(Integer a, Integer b) {
        return a + b; // add new value to old
    }
}
Log Filter in Java

```java
JavaRDD<String> allLines = 
    sc.textFile("hdfs://bigLogFile.txt");

Long x = allLines.filter( //transformation
    new Function<String, Boolean>() {
        Boolean call(String line) {
            return line.contains("error");
        }
    }
).count(); //action

System.out.println("This many error lines:" + x);
```
Two kinds of Virtual Machines

- Spark has two kinds of VMs:
  - Driver
  - Executor
Spark UI http://localhost:4040

<table>
<thead>
<tr>
<th>Active Jobs (1)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Job Id</td>
</tr>
<tr>
<td>--------</td>
</tr>
<tr>
<td>0</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Completed Jobs (2)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Job Id</td>
</tr>
<tr>
<td>--------</td>
</tr>
<tr>
<td>2</td>
</tr>
<tr>
<td>1</td>
</tr>
</tbody>
</table>
Spark vs. Hadoop

• Understand the relationship between a Spark application’s use of the JVM Heap Memory space, and the equivalent application in Hadoop:

  • The Spark Application *does NOT require more JVM Heap space,* than the equivalent application using Apache Hadoop… period.
Immutable RDDs or not?

- We have said, as a rule, that RDDs are **immutable**.
- Are there any exceptions to this rule? Yes:
  1. A **Partition** is lost, and the base RDD loads data from a mutable source (such as an Apache Cassandra table).
  2. The elements in an RDD are Objects – which states are modified by a `map(…)` method.

**immutable**

/ˈɪmjuːtəb(ə)l/  
**adjective**

unchanging over time or unable to be changed.  
"an immutable fact"  
synonyms: fixed, set, rigid, inflexible, permanent, established, carved in stone;
// Scala code imports not shown
val d = sc.textFile("kmeans_data.txt") // base RDD

/* reshape data into features for training a model */
val x = d.map(s => Vectors.dense(s.split(" ")).map(_.toDouble))

/* select training data set & reserve a holdout */
val y = x.sample(false, 0.8)

val numClusters = 3 // ML model parameter
val numIterations = 10
//Continued on the next slide
/* trains a distributed ML model */
val clusters = KMeans.train(y, numClusters, numIterations)

/* evaluate error estimate of the trained ML model */
val w = clusters.computeCost(y) println(w)

/* use trained ML model to score the holdout data */
val z = clusters.predict(x)
z.collect()
What else?

• This was a thorough overview, now…

• Look over the Spark API Docs for Scala, Python and Java, get familiar with them:
  • http://spark.apache.org/documentation.html
  • https://databricks.com/spark/developer-resources

• Take a closer look at Spark Streaming…
• Take a closer look at Spark SQL…
• Take a closer look at GraphX…
Thank you.

JavaRDD<String> answersRDD =
    sc.parallelize(Arrays.asList("Any", "Questions"));