Databricks Spark Knowledge Base

databricks

Published with GitBook

Table of Contents

Databricks Spark Knowledge Base

The contents contained here is also published in Gitbook format.

Best Practices

Avoid GroupByKey

Let's look at two different ways to compute word counts, one using `reduceByKey` and the other using `groupByKey`:

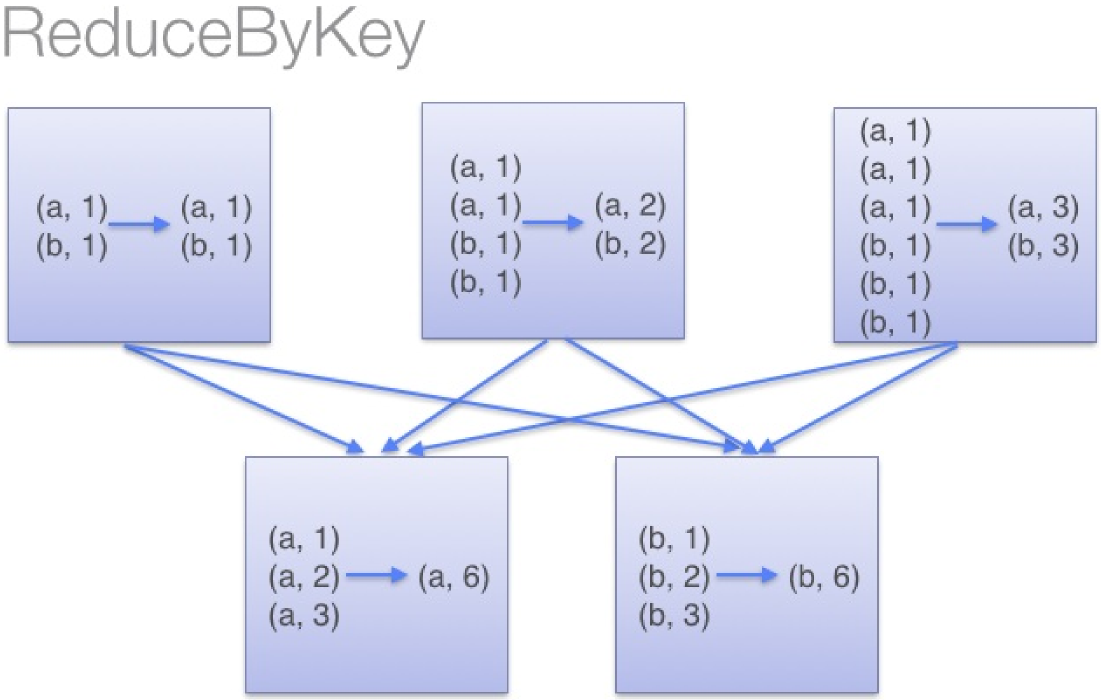
```
val words = Array("one", "two", "two", "three", "three", "three")
val wordPairsRDD = sc.parallelize(words).map(word => (word, 1))

val wordCountsWithReduce = wordPairsRDD
  .reduceByKey(_ + _)
  .collect()

val wordCountsWithGroup = wordPairsRDD
  .groupByKey()
  .map(t => (t._1, t._2.sum))
  .collect()
```

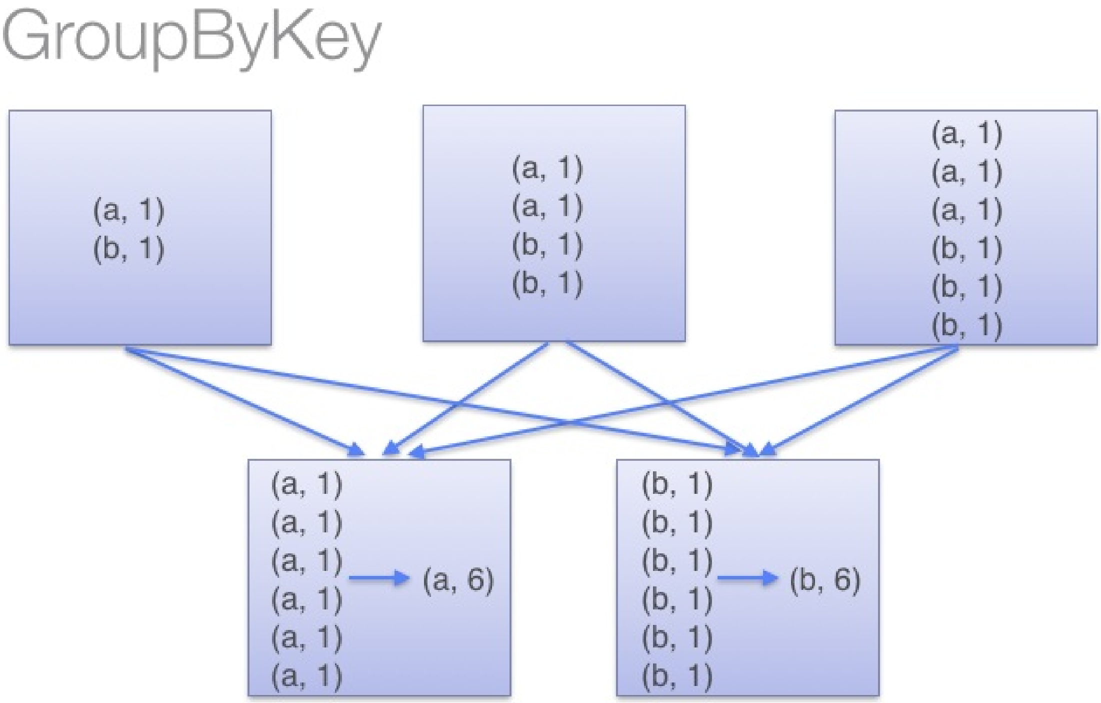
While both of these functions will produce the correct answer, the `reduceByKey` example works much better on a large dataset. That's because Spark knows it can combine output with a common key on each partition before shuffling the data.

Look at the diagram below to understand what happens with `reduceByKey`. Notice how pairs on the same machine with the same key are combined (by using the lamdba function passed into `reduceByKey`) before the data is shuffled. Then the lamdba function is called again to reduce all the values from each partition to produce one final result.



On the other hand, when calling `groupByKey` - all the key-value pairs are shuffled around. This is a lot of unnessary data to being transferred over the network.

To determine which machine to shuffle a pair to, Spark calls a partitioning function on the key of the pair. Spark spills data to disk when there is more data shuffled onto a single executor machine than can fit in memory. However, it flushes out the data to disk one key at a time - so if a single key has more key-value pairs than can fit in memory, an out of memory exception occurs. This will be more gracefully handled in a later release of Spark so the job can still proceed, but should still be avoided - when Spark needs to spill to disk, performance is severely impacted.



You can imagine that for a much larger dataset size, the difference in the amount of data you are shuffling becomes more exaggerated and different between `reduceByKey` and `groupByKey` .

Here are more functions to prefer over `groupByKey` :

- `combineByKey` can be used when you are combining elements but your return type differs from your input value type.
- `foldByKey` merges the values for each key using an associative function and a neutral "zero value".

Don't copy all elements of a large RDD to the driver.

If your RDD is so large that all of it's elements won't fit in memory on the drive machine, don't do this:

```
val values = myVeryLargeRDD.collect()
```

Collect will attempt to copy every single element in the RDD onto the single driver program, and then run out of memory and crash.

Instead, you can make sure the number of elements you return is capped by calling `take` or `takeSample`, or perhaps filtering or sampling your RDD.

Similarly, be cautious of these other actions as well unless you are sure your dataset size is small enough to fit in memory:

- `countByKey`
- `countByValue`
- `collectAsMap`

If you really do need every one of these values of the RDD and the data is too big to fit into memory, you can write out the RDD to files or export the RDD to a database that is large enough to hold all the data.

Gracefully Dealing with Bad Input Data

When dealing with vast amounts of data, a common problem is that a small amount of the data is malformed or corrupt. Using a `filter` transformation, you can easily discard bad inputs, or use a `map` transformation if it's possible to fix the bad input. Or perhaps the best option is to use a `flatMap` function where you can try fixing the input but fall back to discarding the input if you can't.

Let's consider the json strings below as input:

```
input_rdd = sc.parallelize(["{\"value\": 1}",  # Good
                            "bad_json",  # Bad
                            "{\"value\": 2}",  # Good
                            "{\"value\": 3"  # Missing an ending brace.
                            ])
```

If we tried to input this set of json strings to a sqlContext, it would clearly fail due to the malformed input's.

```
sqlContext.jsonRDD(input_rdd).registerTempTable("valueTable")
# The above command will throw an error.
```

Instead, let's try fixing the input with this python function:

```
def try_correct_json(json_string):
  try:
    # First check if the json is okay.
    json.loads(json_string)
    return [json_string]
  except ValueError:
    try:
      # If not, try correcting it by adding a ending brace.
      try_to_correct_json = json_string + "}"
      json.loads(try_to_correct_json)
      return [try_to_correct_json]
    except ValueError:
      # The malformed json input can't be recovered, drop this input.
      return []
```

Now, we can apply that function to fix our input and try again. This time we will succeed to read in three inputs:

```
corrected_input_rdd = input_rdd.flatMap(try_correct_json)
sqlContext.jsonRDD(corrected_input_rdd).registerTempTable("valueTable")
sqlContext.sql("select * from valueTable").collect()
# Returns [Row(value=1), Row(value=2), Row(value=3)]
```

General Troubleshooting

- Job aborted due to stage failure: Task not serializable:
- Missing Dependencies in Jar Files
- Error running start-all.sh - Connection refused
- Network connectivity issues between Spark components

Job aborted due to stage failure: Task not serializable:

If you see this error:

```
org.apache.spark.SparkException: Job aborted due to stage failure: Task not serializable: java.io.NotSerializableException: ...
```

The above error can be triggered when you intialize a variable on the driver (master), but then try to use it on one of the workers. In that case, Spark Streaming will try to serialize the object to send it over to the worker, and fail if the object is not serializable. Consider the following code snippet:

```
NotSerializable notSerializable = new NotSerializable();
JavaRDD<String> rdd = sc.textFile("/tmp/myfile");

rdd.map(s -> notSerializable.doSomething(s)).collect();
```

This will trigger that error. Here are some ideas to fix this error:

- Serializable the class
- Declare the instance only within the lambda function passed in map.
- Make the NotSerializable object as a static and create it once per machine.
- Call rdd.forEachPartition and create the NotSerializable object in there like this:

```
rdd.forEachPartition(iter -> {
  NotSerializable notSerializable = new NotSerializable();

  // ...Now process iter
});
```

Missing Dependencies in Jar Files

By default, maven does not include dependency jars when it builds a target. When running a Spark job, if the Spark worker machines don't contain the dependency jars - there will be an error that a class cannot be found.

The easiest way to work around this is to create a *shaded* or *uber* jar to package the dependencies in the jar as well.

It is possible to opt out certain dependencies from being included in the uber jar by marking them as `<scope>provided</scope>`. Spark dependencies should be marked as provided since they are already on the Spark cluster. You may also exclude other jars that you have installed on your worker machines.

Here is an example Maven pom.xml file that creates an uber jar with all the code in that project and includes the common-cli dependency, but not any of the Spark libraries.:

```xml
<project>
  <groupId>com.databricks.apps.logs</groupId>
  <artifactId>log-analyzer</artifactId>
  <modelVersion>4.0.0</modelVersion>
  <name>Databricks Spark Logs Analyzer</name>
  <packaging>jar</packaging>
  <version>1.0</version>
  <repositories>
    <repository>
      <id>Akka repository</id>
      <url>http://repo.akka.io/releases</url>
    </repository>
  </repositories>
  <dependencies>
    <dependency> <!-- Spark -->
      <groupId>org.apache.spark</groupId>
      <artifactId>spark-core_2.10</artifactId>
      <version>1.1.0</version>
      <scope>provided</scope>
    </dependency>
    <dependency> <!-- Spark SQL -->
      <groupId>org.apache.spark</groupId>
      <artifactId>spark-sql_2.10</artifactId>
      <version>1.1.0</version>
      <scope>provided</scope>
    </dependency>
    <dependency> <!-- Spark Streaming -->
      <groupId>org.apache.spark</groupId>
      <artifactId>spark-streaming_2.10</artifactId>
      <version>1.1.0</version>
      <scope>provided</scope>
    </dependency>
    <dependency> <!-- Command Line Parsing -->
      <groupId>commons-cli</groupId>
      <artifactId>commons-cli</artifactId>
      <version>1.2</version>
    </dependency>
  </dependencies>
  <build>
    <plugins>
      <plugin>
        <groupId>org.apache.maven.plugins</groupId>
        <artifactId>maven-compiler-plugin</artifactId>
        <version>2.3.2</version>
        <configuration>
          <source>1.8</source>
          <target>1.8</target>
        </configuration>
      </plugin>
      <plugin>
        <groupId>org.apache.maven.plugins</groupId>
        <artifactId>maven-shade-plugin</artifactId>
        <version>2.3</version>
        <executions>
          <execution>
            <phase>package</phase>
            <goals>
              <goal>shade</goal>
            </goals>
          </execution>
        </executions>
        <configuration>
          <filters>
            <filter>
              <artifact>*:*</artifact>
              <excludes>
                <exclude>META-INF/*.SF</exclude>
                <exclude>META-INF/*.DSA</exclude>
                <exclude>META-INF/*.RSA</exclude>
              </excludes>
            </filter>
          </filters>
          <finalName>uber-${project.artifactId}-${project.version}</finalName>
        </configuration>
      </plugin>
    </plugins>
  </build>
</project>
```

Error running start-all.sh Connection refused

If you are on a Mac and run into the following error when running start-all.sh:

```
% sh start-all.sh
starting org.apache.spark.deploy.master.Master, logging to ...
localhost: ssh: connect to host localhost port 22: Connection refused
```

You need to enable "Remote Login" for your machine. From System Preferences, select Sharing, and then turn on Remote Login.

Network connectivity issues between Spark components

Network connectivity issues between Spark components can lead to a variety of warnings / errors:

- **SparkContext <-> Spark Standalone Master**:

  If the SparkContext cannot connect to a Spark standalone master, then the driver may display errors like

  ```
  ERROR AppClient$ClientActor: All masters are unresponsive! Giving up.
  ERROR SparkDeploySchedulerBackend: Spark cluster looks dead, giving up.
  ERROR TaskSchedulerImpl: Exiting due to error from cluster scheduler: Spark cluster looks down
  ```

  If the driver is able to connect to the master but the master is unable to communicate back to the driver, then the Master's logs may record multiple attempts to connect even though the driver will report that it could not connect:

  ```
  INFO Master: Registering app SparkPi
  INFO Master: Registered app SparkPi with ID app-XXX-0000
  INFO: Master: Removing app app-app-XXX-0000
  [...]
  INFO Master: Registering app SparkPi
  INFO Master: Registered app SparkPi with ID app-YYY-0000
  INFO: Master: Removing app app-YYY-0000
  [...]
  ```

  In this case, the master reports that it has successfully registered an application, but if the acknowledgment of this registration fails to be received by the driver, then the driver will automatically make several attempts to re-connect before eventually giving up and failing. As a result, the master web UI may report multiple failed applications even though only a single SparkContext was created.

Recomendations

If you are experiencing any of the errors described above:

- Check that the workers and drivers are configured to connect to the Spark master on the exact address listed in the Spark master web UI / logs.
- Set `SPARK_LOCAL_IP` to a cluster-addressable hostname for the driver, master, and worker processes.

Configurations that determine hostname/port binding:

This section describes configurations that determine which network interfaces and ports Spark components will bind to.

In each section, the configurations are listed in decreasing order of precedence, with the final entry being the default configuration if none of the previous configurations were supplied.

SparkContext actor system:

Hostname:

- The `spark.driver.host` configuration property.
- If the `SPARK_LOCAL_IP` environment variable is set to a hostname, then this hostname will be used. If `SPARK_LOCAL_IP` is set to an IP address, it will be resolved to a hostname.
- The IP address of the interface returned from Java's `InetAddress.getLocalHost` method.

Port:

- The `spark.driver.port` configuration property.

- An ephemeral port chosen by the OS.

Spark Standalone Master / Worker actor systems:

Hostname:

- The `--host`, or `-h` options (or the deprecated `--ip` or `-i` options) when launching the `Master` or `Worker` process.
- The `SPARK_MASTER_HOST` environment variable (only applies to `Master`).
- If the `SPARK_LOCAL_IP` environment variable is set to a hostname, then this hostname will be used. If `SPARK_LOCAL_IP` is set to an IP address, it will be resolved to a hostname.
- The IP address of the interface returned from Java's `InetAddress.getLocalHost` method.

Port:

- The `--port`, or `-p` options when launching the `Master` or `Worker` process.
- The `SPARK_MASTER_PORT` or `SPARK_WORKER_PORT` environment variables (only apply to `Master` and `Worker`, respectively).
- An ephemeral port chosen by the OS.

Performance & Optimization

How Many Partitions Does An RDD Have?

For tuning and troubleshooting, it's often necessary to know how many paritions an RDD represents. There are a few ways to find this information:

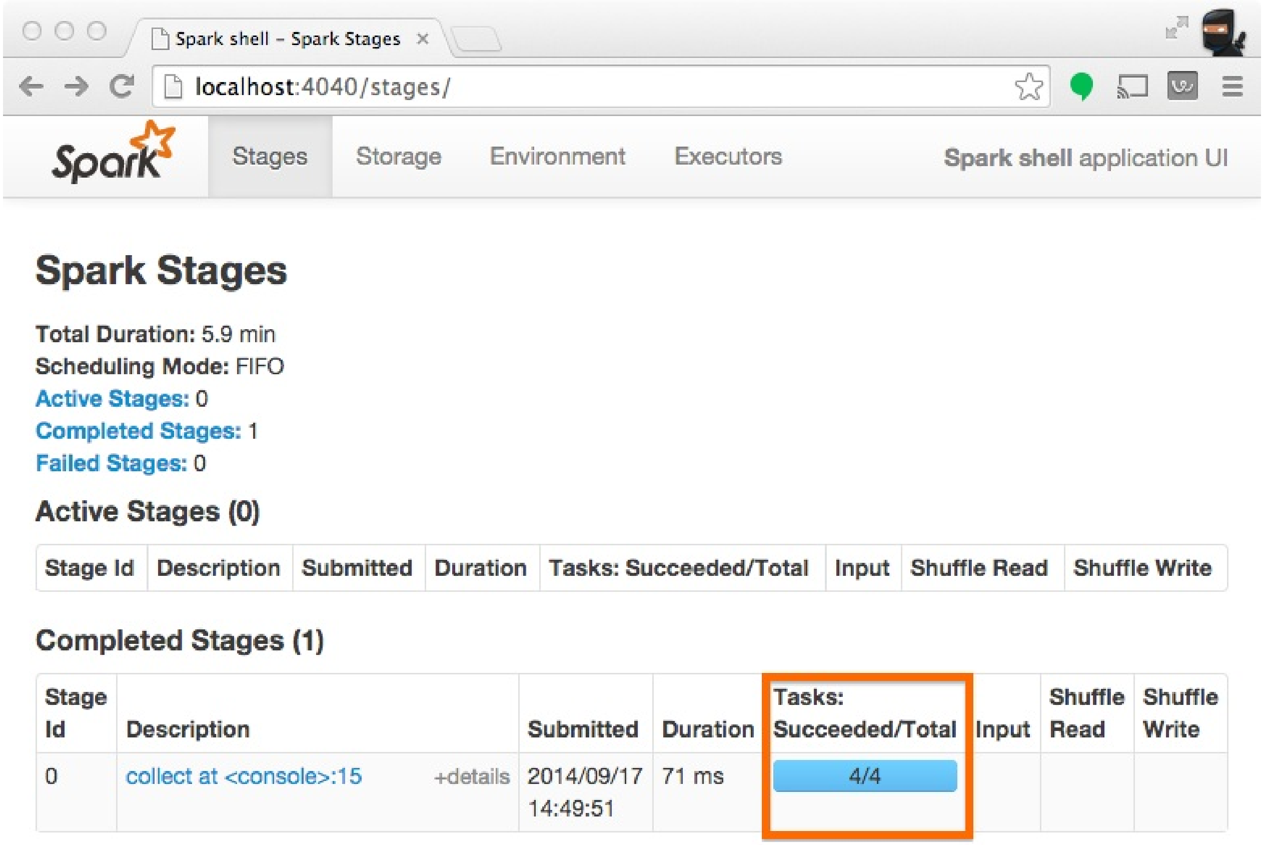
View Task Execution Against Partitions Using the UI

When a stage executes, you can see the number of partitions for a given stage in the Spark UI. For example, the following simple job creates an RDD of 100 elements across 4 partitions, then distributes a dummy map task before collecting the elements back to the driver program:

```
scala> val someRDD = sc.parallelize(1 to 100, 4)
someRDD: org.apache.spark.rdd.RDD[Int] = ParallelCollectionRDD[0] at parallelize at <console>:12

scala> someRDD.map(x => x).collect
res1: Array[Int] = Array(1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23, 24, 25, 26, 27, 28, 29, 30, 31, 32,
```

In Spark's application UI, you can see from the following screenshot that the "Total Tasks" represents the number of partitions:



Spark Stages

Total Duration: 5.9 min
Scheduling Mode: FIFO
Active Stages: 0
Completed Stages: 1
Failed Stages: 0

Active Stages (0)

Stage Id	Description	Submitted	Duration	Tasks: Succeeded/Total	Input	Shuffle Read	Shuffle Write

Completed Stages (1)

Stage Id	Description		Submitted	Duration	Tasks: Succeeded/Total	Input	Shuffle Read	Shuffle Write
0	collect at <console>:15	+details	2014/09/17 14:49:51	71 ms	4/4			

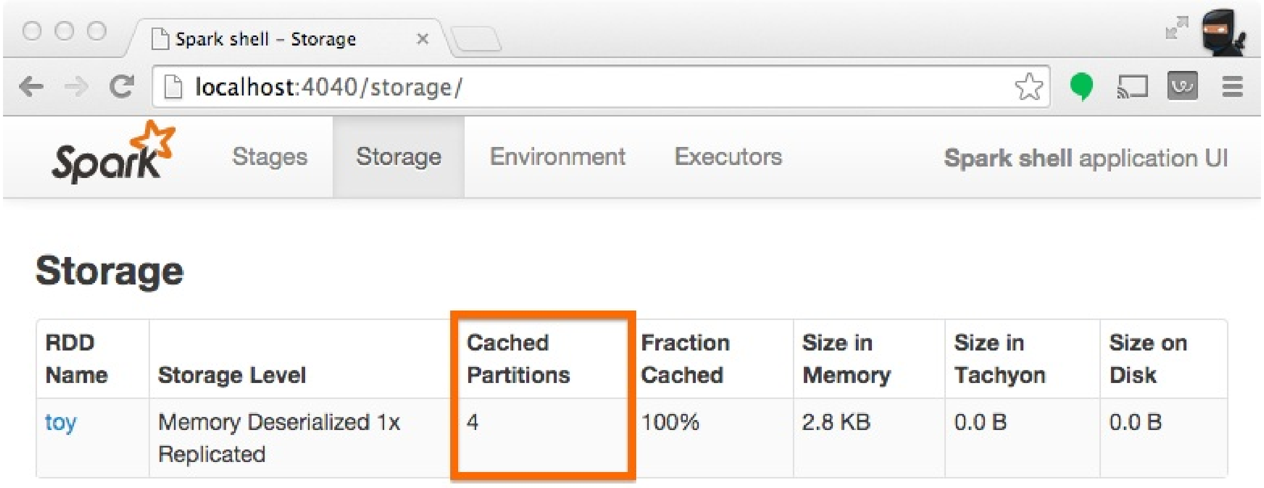
View Partition Caching Using the UI

When persisting (a.k.a. caching) RDDs, it's useful to understand how many partitions have been stored. The example below is identical to the one prior, except that we'll now cache the RDD prior to processing it. After this completes, we can use the UI to understand what has been stored from this operation.

```
scala> someRDD.setName("toy").cache
res2: someRDD.type = toy ParallelCollectionRDD[0] at parallelize at <console>:12

scala> someRDD.map(x => x).collect
res3: Array[Int] = Array(1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23, 24, 25, 26, 27, 28, 29, 30, 31, 32,
```

Note from the screenshot that there are four partitions cached.



Storage

RDD Name	Storage Level	Cached Partitions	Fraction Cached	Size in Memory	Size in Tachyon	Size on Disk
toy	Memory Deserialized 1x Replicated	4	100%	2.8 KB	0.0 B	0.0 B

Inspect RDD Partitions Programatically

In the Scala API, an RDD holds a reference to it's Array of partitions, which you can use to find out how many partitions there are:

```
scala> val someRDD = sc.parallelize(1 to 100, 30)
someRDD: org.apache.spark.rdd.RDD[Int] = ParallelCollectionRDD[0] at parallelize at <console>:12

scala> someRDD.partitions.size
res0: Int = 30
```

In the python API, there is a method for explicitly listing the number of partitions:

```
In [1]: someRDD = sc.parallelize(range(101),30)

In [2]: someRDD.getNumPartitions()
Out[2]: 30
```

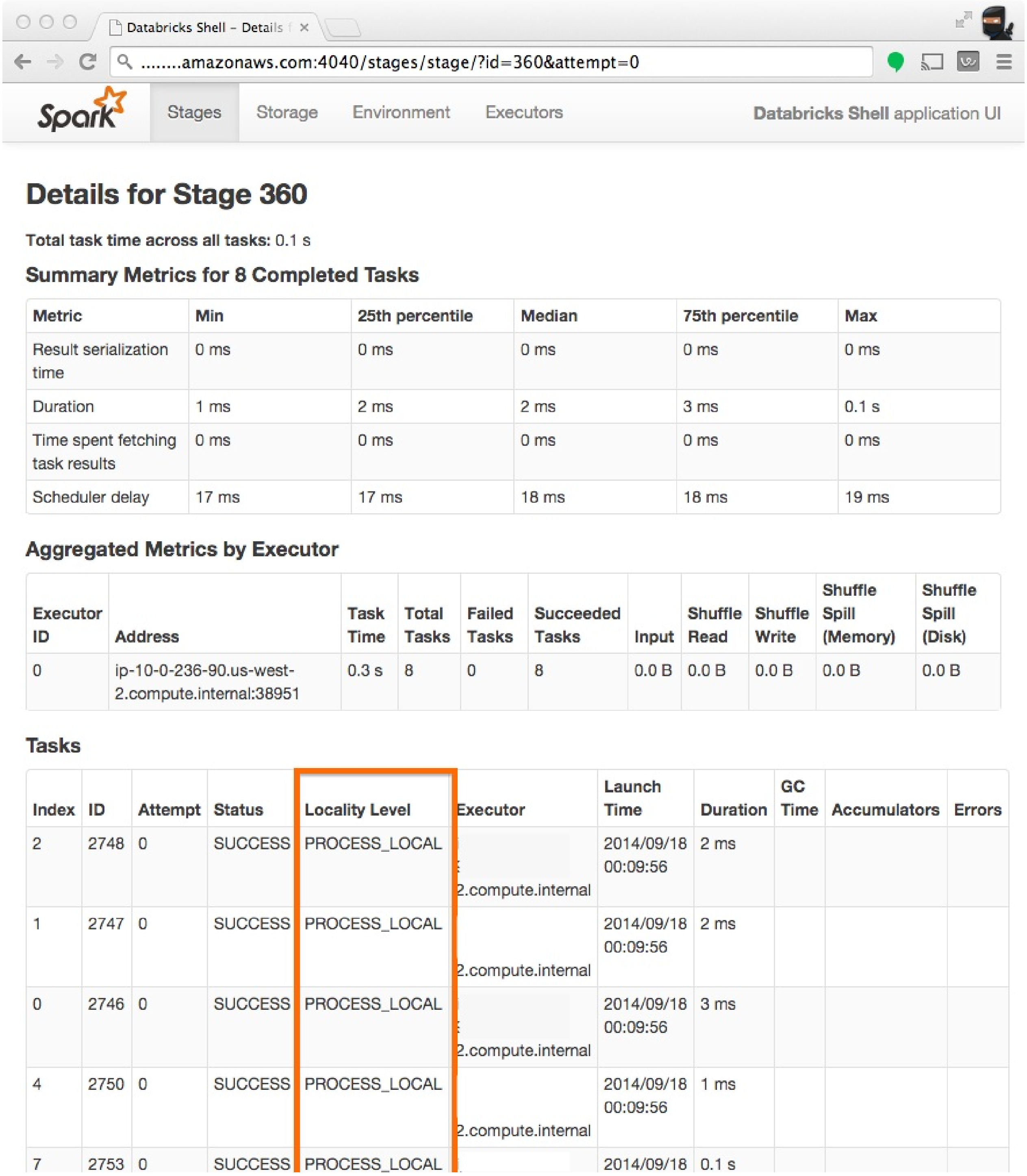
Note in the examples above, the number of partitions was intentionally set to 30 upon initialization.

Data Locality

Spark is a data parallel processing framework, which means it will execute tasks as close to where the data lives as possible (i.e. minimize data transfer).

Checking Locality

The best means of checking whether a task ran locally is to inspect a given stage in the Spark UI. Notice from the screenshot below that the "Locality Level" column displays which locality a given task ran with.


Databricks Shell – Details f
........amazonaws.com:4040/stages/stage/?id=360&attempt=0
Spark
Stages
Storage
Environment
Executors
Databricks Shell application UI

Details for Stage 360

Total task time across all tasks: 0.1 s

Summary Metrics for 8 Completed Tasks

Metric	Min	25th percentile	Median	75th percentile	Max
Result serialization time	0 ms	0 ms	0 ms	0 ms	0 ms
Duration	1 ms	2 ms	2 ms	3 ms	0.1 s
Time spent fetching task results	0 ms	0 ms	0 ms	0 ms	0 ms
Scheduler delay	17 ms	17 ms	18 ms	18 ms	19 ms

Aggregated Metrics by Executor

Executor ID	Address	Task Time	Total Tasks	Failed Tasks	Succeeded Tasks	Input	Shuffle Read	Shuffle Write	Shuffle Spill (Memory)	Shuffle Spill (Disk)
0	ip-10-0-236-90.us-west-2.compute.internal:38951	0.3 s	8	0	8	0.0 B	0.0 B	0.0 B	0.0 B	0.0 B

Tasks

Index	ID	Attempt	Status	Locality Level	Executor	Launch Time	Duration	GC Time	Accumulators	Errors
2	2748	0	SUCCESS	PROCESS_LOCAL	2.compute.internal	2014/09/18 00:09:56	2 ms			
1	2747	0	SUCCESS	PROCESS_LOCAL	2.compute.internal	2014/09/18 00:09:56	2 ms			
0	2746	0	SUCCESS	PROCESS_LOCAL	2.compute.internal	2014/09/18 00:09:56	3 ms			
4	2750	0	SUCCESS	PROCESS_LOCAL	2.compute.internal	2014/09/18 00:09:56	1 ms			
7	2753	0	SUCCESS	PROCESS_LOCAL		2014/09/18	0.1 s			

Adjusting Locality Confugrations

You can adjust how long Spark will wait before it times out on each of the phases of data locality (data local --> process local --> node local --> rack local --> Any). For more information on these parameters, see the `spark.locality.*` configs in the Scheduling section of the Application Configration docs.

Spark Streaming

- ERROR OneForOneStrategy

ERROR OneForOneStrategy

If you enable checkpointing in Spark Streaming, then objects used in a function called in forEachRDD should be Serializable. Otherwise, there will be an "ERROR OneForOneStrategy: ... java.io.NotSerializableException:

```
JavaStreamingContext jssc = new JavaStreamingContext(sc, INTERVAL);

// This enables checkpointing.
jssc.checkpoint("/tmp/checkpoint_test");

JavaDStream<String> dStream = jssc.socketTextStream("localhost", 9999);

NotSerializable notSerializable = new NotSerializable();
dStream.foreachRDD(rdd -> {
      if (rdd.count() == 0) {
        return null;
      }
      String first = rdd.first();

      notSerializable.doSomething(first);
      return null;
    }
);

// This does not work!!!!
```

This code will run if you make one of these changes to it:

- Turn off checkpointing by removing the `jssc.checkpoint` line.
- Make the object being used Serializable.
- Declare NotSerializable inside the forEachRDD function, so the following code sample would be fine:

```
JavaStreamingContext jssc = new JavaStreamingContext(sc, INTERVAL);

jssc.checkpoint("/tmp/checkpoint_test");

JavaDStream<String> dStream = jssc.socketTextStream("localhost", 9999);

dStream.foreachRDD(rdd -> {
      if (rdd.count() == 0) {
        return null;
      }
      String first = rdd.first();
      NotSerializable notSerializable = new NotSerializable();
      notSerializable.doSomething(first);
      return null;
    }
);

// This code snippet is fine since the NotSerializable object
// is declared and only used within the forEachRDD function.
```