Working With Partitions

The Data

Review the data in sparkdev/data/activations. Each XML file contains data for all the devices activated by customers during a specific month.

- Copy this data to HDFS: hfs bfs -put activations

The Task

- Your code should go through a set of activation XML files and output the top *n* device models activated.
- Start with the TopModels stub script. Note that for convenience you have been provided with functions to parse the XML, as that is not the focus of this Exercise. Copy the stub code into the Spark Shell.
- Read the XML files into an RDD, then call toDebugString on that RDD. This will display the number of partitions, which will be the same as the number of files that were read
- **This will display the lineage (the list of dependent RDDs; this will be discussed more in the next chapter). The one you care about here is the current RDD, which is at the top of the list.**

The Task

- Use mapPartitions to map each partition to an XML Tree structure based on parsing the contents of that partition as a string. You can call the provided function getactivations by passing it the partition iterator from mapPartitions; it will return an array of XML elements for each activation tag in the partition. For example:

- Map each activation tag to the model name of the device activated using the provided getmodel function. Assign this transformation to a named RDD.

The Task

- Call toDebugString on the new RDD and note that the partitioning has been maintained: one partition for each file.
- Count the number of occurrences of each model and display the top few. (Refer to earlier chapters for a reminder on how to use MapReduce to count occurrences if you need to.)
- Use the top(*n*) method to display the 10 most popular models. Note that you will need to key the RDD by count.

Note:Leave the Spark Shell running for the next exercise.

Viewing Stages in the spark Application UI

Viewing Stages in the Spark Application UI

In the last Exercise, you wrote a script in the Spark Shell to parse XML files containing device activation data, and count the number of each type of device model activated. Now you will review the stages and tasks that were executed.

- Make sure the Spark Shell is still running from the last Exercise. If it is not, or if you did not complete the last Exercise, restart the shell and paste in the code from the solution file for the previous Exercise.
- In a browser, view the Spark Application UI: http://localhost:4040/
- Make sure the Stages tab is selected.

Viewing Stages in the Spark Application UI

- Look in the Completed Stages section and you should see 4. the stages of the exercise you completed.
  - Things to note:
  - The stages are numbered, but numbers do not relate to the order of execution. Note the times the stages were submitted.
  - The number of tasks in the first stage corresponds to the number of partitions, which for this example corresponds to the number of files processed.
  - The Shuffle Write column indicates how much data that stage copied between tasks. This is useful to know because copying too much data across the network can cause performance issues.

Viewing Stages in the Spark Application UI (2)

- Click on the stage description to view details about that stage. Things to note:
  - The Summary Metrics area shows you how much time was spent on various steps. This can help you narrow down performance problems.
  - The Tasks area lists each task. The Locality Level column indicates whether the process ran on the same node where the partition was physically stored or not. Remember that Spark will attempt to always run tasks where the data is, but may not always be able to, if the node is busy.
  - In a realNworld cluster, the executor column in the Task area would display the different worker nodes that ran the tasks. (In this singleNnode cluster, all tasks run on the same host.)

Note:,Leave,the,Spark,Shell,running,for,the,next,exercise.,

Caching RDDs

Caching RDDs

The easiest way to see caching in action is to compare the time it takes to complete an operation on a cached and uncached RDD.

- Make sure the Spark Shell is still running from the last exercise. If it isn't, restart it and paste in the code from the solution file.
- Execute a count action on the RDD containing the list of activated models

- Take note of the time it took to complete the count operation.

- Now cache the RDD

Caching RDDs

- Execute the count again. This time should take about the same amount of time the last one did. It may even take a little longer, because in addition to running the operation, it is also caching the results.
- Re-execute the count. Because the data is now cached, you should see a substantial reduction in the amount of time the operation takes.
- In your browser, view the Spark Application UI and select the **Storage** tab. You will see a list of cached RDDs (in this case, just the system-generated name of the models RDD you cached above). Click the RDD name to see details about partitions and caching.
- Click on the **Executors** tab and take note of the amount of memory used and available for our one worker node.
  Note that the classroom environment has a single worker node with a small amount of memory allocated, so you may see that not all of the dataset is actually cached in memory. In the real world, for good performance a cluster will have more nodes, each with more memory, so that more of your active data can be cached.

Caching RDDs

- Click into the Storage tab and note that the Storage Level is currently “Memory Deserialized 1.x Replicated”. Back in the Spark Shell, set the RDD’s persistence level to StorageLevel.DISK_ONLY and compare both the compute times and the storage report in the Spark Application Web UI. (Hint: Because you have already persisted the RDD at a different level, you will need to unpersist first before you can set a new level.)

Checkpointing RDDs

Create an iterative RDD that results in a stack overflow

- Create an RDD by parallelizing 1. an array of numbers.
- Loop 200 times. Each time through the loop, create a new RDD based on the previous iteration's result by adding 1 to each element.
- Collect and display the data in the RDD.
- Show the final RDD using toDebugString(). Note that the base RDD of the lineage is the parallelized array, e.g. ParallelCollectionRDD[1]

Create an iterative RDD that results in a stack overflow

- Repeat the loop, which adds another 200 iterations to the lineage. Then collect and display the RDD elements again. Did it work?
- Keep adding to the lineage by repeating the loop, and testing by collecting the elements. Eventually, the collect() operation should give you an error indicating a stack overflow.
  - In Python, the error message will likely report “excessively deep recursion required”.
  - In Scala the base exception will be StackOverflowError, which you will have to scroll up in your shell window to see; the immediate exception will probably be related to the Block Manager thread not responding, such as “Error sending message to BlockManagerMaster”.
- Take note of the total number of iterations that resulted in the stack overflow.

Fix the stack overflow problem by checkpointing the RDD

- Exit and restart the Spark Shell following the error in the previous section.
- Enable checkpointing by calling sc.setCheckpointDir("checkpoints")
- Paste in the previous code to create the RDD.
- As before, create an iterative RDD dependency, using at least the number of iterations that previously resulted in stack overflow.
- Inside the loop, add two steps that are executed every 10 iterations:
  - Checkpoint the RDD
  - Materialize the RDD by performing an action such as count.
- After looping, collect and view the elements of the RDD 13. to confirm the job executes without a stack overflow.
- Examine the lineage of the RDD; note that rather than going back to the base, it goes back to the most recent checkpoint.

Writing and Running a Spark Application

Write a Spark application in Python

- A simple stub file to get started has been provided: sparkdev/stubs. This stub imports the required Spark class and sets up your main code block. Copy this stub to your work area and edit it to complete this exercise.
- Set up a SparkContext using
- In the body of the program, load the file passed in to the program, count the number of JPG requests, and display the count. You may wish to refer back to the “Getting Started with RDDs” exercise for the code to do this.
- Run the program, passing the name of the log file to process, e.g.:
  - spark-submit CountJPGs.py weblogs/*
  - spark-submit —class CountJPGs target/countjpgs-1.0.jar weblogs/*

Write a Spark application in Python

- By default, the program will run locally. Re-run the program, specifying the cluster master in order to run it on the cluster: spark-submit —master spark://localhost:7077 —class CountJPGs target/countjpgs-1.0.jar weblogs/*

- Visit the Standalone Spark Master UI and confirm that the program is running on the cluster.

Spark Application in Scala

- A Maven project to get started has been provided: projects/countjpgs (or one supplied by your instructor).
- Edit the Scala code in src/main/scala/stubs/CountJPGs.scala.
- Set up a SparkContext using the following code:
- In the body of the program, load the file passed in to the program, count the number of JPG requests, and display the count. You may wish to refer back to the “Getting Started with RDDs” exercise for the code to do this.
- Perform a Maven Package to create the jar file.

Spark Application in Scala

- If the build is successful, it will generate a JAR file called countjpgs-1.0.jar in countjpgs/target. Run the program from the ~/exercises/projects/countjpgs directory using the following command:

- By default, the program will run locally. Re-run the program, specifying the cluster master in order to run it on the cluster:

- Visit the Standalone Spark Master UI and confirm that the program is running (or completed running) on the cluster.

Configuring Spark Applications

Set configuration options at the command line

- Rerun the CountJPGs Python or Scala program you wrote in the previous exercise, this time specifying an application name.
- Visit the Standalone Spark Master UI (http://localhost:18080/) and note the application name listed is the one specified in the command line.
- *Optional*: While the application is running, visit the Spark Application UI and view the **Environment** tab. Take note of the spark.* properties such as master, app.name, and driver properties.'

Set configuration options in a configuration file

- Change directories to your exercise 4. working directory. (If you are working in Scala, that is the countjpgs project directory.)
- Using a text editor, create a file in the working directory called myspark.conf, containing settings for the properties shown below:

  spark.app.name My Spark App
  spark.ui.port 4141
  spark.master spark://localhost:7077

- Re-run your application, this time using the properties file instead of using the script options to configure Spark properties:

  **--properties-file myspark.conf **

- While the application is running, view the Spark Application UI at the alternate port you specified to confirm that it is using the correct port: http://localhost:**4141**

Set configuration options in a configuration file(1)

8. Also visit the Standalone Spark Master UI to confirm that the application correctly ran on the cluster with the correct app name

Set configuration properties programmatically

9. Following the example from the slides, modify the CountJPGs program to set the application name and UI port programmatically.
   a) First create a SparkConf object and set its spark.app.name and spark.ui.port properties
   b) Then use the SparkConf object when creating the SparkContext.

Set Logging Levels

- Copy the template file $SPARK_HOME/conf/log4j.properties.template to log4j.properties in your exercise working directory.
- Edit log4j.properties . The first line currently reads: **log4j.rootCategory=INFO, console** Replace INFO with DEBUG: **log4j.rootCategory=DEBUG, console**
- Rerun your Spark application. Because the current directory is on the Java classpath, your log4.properties file will set the logging level to DEBUG.

Set Logging Levels

- Notice that the output now contains both the INFO messages it did before and DEBUG messages

  Debug logging can be useful when debugging, testing, or optimizing your code, but in most cases generates unnecessarily distracting output.

Set Logging Levels

- Edit the log4j.properties file to replace DEBUG with WARN and try again.
  This time notice that no INFO or DEBUG messages are displayed, only WARN messages.
- You can also set the log level for the interactive Spark Shell by placing the log4j.properties file in your working directory before starting the shell. Try starting the shell from the directory in which you placed the file and note that only WARN messages now appear.

- Note: During the rest of the exercises, you may change these settings depending on whether you find the extra logging messages helpful or distracting.