Apache Hadoop –

A course for undergraduates

Homework Labs, Lecture 6

Lab: Writing a Partitioner

Files and Directories Used in this Exercise

Eclipse project: `partitioner`

Java files:

`MonthPartitioner.java` (Partitioner)

`ProcessLogs.java` (driver)

`CountReducer.java` (Reducer)

`LogMonthMapper.java` (Mapper)

Test data (HDFS):

`weblog` (full web server access log)

`testlog` (partial data set for testing)

Exercise directory: `~/workspace/partitioner`

In this Exercise, you will write a MapReduce job with multiple Reducers, and create a Partitioner to determine which Reducer each piece of Mapper output is sent to.

The Problem

In the "More Practice with Writing MapReduce Java Programs" lab you did previously, you built the code in `log_file_analysis` project. That program counted the number of hits for each different IP address in a web log file. The final output was a file containing a list of IP addresses, and the number of hits from that address.

This time, you will perform a similar task, but the final output should consist of 12 files, one each for each month of the year: January, February, and so on. Each file will contain a list of IP addresses, and the number of hits from that address *in that month*.

We will accomplish this by having 12 Reducers, each of which is responsible for processing the data for a particular month. Reducer 0 processes January hits, Reducer 1 processes February hits, and so on.

Note: We are actually breaking the standard MapReduce paradigm here, which says that all the values from a particular key will go to the same Reducer. In this example, which is a very common pattern when analyzing log files, values from the same key (the IP address) will go to multiple Reducers, based on the month portion of the line.

Write the Mapper

1. Starting with the `LogMonthMapper.java` stub file, write a Mapper that maps a log file output line to an IP/month pair. The map method will be similar to that in the `LogFileMapper` class in the `log_file_analysis` project, so you may wish to start by copying that code.

2. The Mapper should emit a Text key (the IP address) and Text value (the month). E.g.:

   Input: `96.7.4.14 - - [24/Apr/2011:04:20:11 -0400] "GET /cat.jpg HTTP/1.1" 200 12433`

   Output key: `96.7.4.14`

   Output value: `Apr`

   Hint: In the Mapper, you may use a regular expression to parse to log file data if you are familiar with regex processing (see file Homework_RegexRef.docx for reference).

   Remember that the log file may contain unexpected data – that is, lines that do not conform to the expected format. Be sure that your code copes with such lines.

Write the Partitioner

3. Modify the `MonthPartitioner.java` stub file to create a Partitioner that sends the (key, value) pair to the correct Reducer based on the month. Remember that the Partitioner receives both the key and value, so you can inspect the value to determine which Reducer to choose.

Modify the Driver

4. Modify your driver code to specify that you want 12 Reducers.

5. Configure your job to use your custom Partitioner.

Test your Solution

6. Build and test your code. Your output directory should contain 12 files named `part-r-000`*xx*. Each file should contain IP address and number of hits for month *xx*.

Hints:

- Write unit tests for your Partitioner!
- You may wish to test your code against the smaller version of the access log in the `/user/training/testlog` directory before you run your code against the full log in the `/user/training/weblog` directory. However, note that the test data may not include all months, so some result files will be empty.

This is the end of the lab.

Lab: Implementing a Custom WritableComparable

Files and Directories Used in this Exercise

Eclipse project: `writables`

Java files:

`StringPairWritable` – implements a `WritableComparable type`

`StringPairMapper` – Mapper for test job

`StringPairTestDriver` – Driver for test job

Data file:

`~/training_materials/developer/data/nameyeartestdata` (small set of data for the test job)

Exercise directory: `~/workspace/writables`

In this lab, you will create a custom WritableComparable type that holds two strings.

Test the new type by creating a simple program that reads a list of names (first and last) and counts the number of occurrences of each name.

The mapper should accepts lines in the form:

```
lastname firstname other data
```

The goal is to count the number of times a lastname/firstname pair occur within the dataset. For example, for input:

```
Smith Joe 1963-08-12 Poughkeepsie, NY
Smith Joe 1832-01-20 Sacramento, CA
Murphy Alice 2004-06-02 Berlin, MA
```

We want to output:

```
(Smith,Joe)     2
(Murphy,Alice) 1
```

> Note: You will use your custom WritableComparable type in a future lab, so make sure it is working with the test job now.

StringPairWritable

You need to implement a WritableComparable object that holds the two strings. The stub provides an empty constructor for serialization, a standard constructor that will be given two strings, a `toString` method, and the generated `hashCode` and `equals` methods. You will need to implement the `readFields`, `write`, and `compareTo` methods required by WritableComparables.

Note that Eclipse automatically generated the `hashCode` and `equals` methods in the stub file. You can generate these two methods in Eclipse by right-clicking in the source code and choosing 'Source' > 'Generate hashCode() and equals()'.

Name Count Test Job

The test job requires a Reducer that sums the number of occurrences of each key. This is the same function that the SumReducer used previously in wordcount, except that SumReducer expects Text keys, whereas the reducer for this job will get StringPairWritable keys. You may either re-write SumReducer to accommodate other types of keys, or you can use the LongSumReducer Hadoop library class, which does exactly the same thing.

You can use the simple test data in `~/training_materials/developer/data/nameyeartestdata` to make sure your new type works as expected.

You may test your code using local job runner or by submitting a Hadoop job to the (pseudo-)cluster as usual. If you submit the job to the cluster, note that you will need to copy your test data to HDFS first.

This is the end of the lab.

Lab: Using SequenceFiles and File Compression

Files and Directories Used in this Exercise

Eclipse project: `createsequencefile`

Java files:

`CreateSequenceFile.java` (a driver that converts a text file to a sequence file)

`ReadCompressedSequenceFile.java` (a driver that converts a compressed sequence file to text)

Test data (HDFS):

`weblog` (full web server access log)

Exercise directory: `~/workspace/createsequencefile`

In this lab you will practice reading and writing uncompressed and compressed SequenceFiles.

First, you will develop a MapReduce application to convert text data to a SequenceFile. Then you will modify the application to compress the SequenceFile using Snappy file compression.

When creating the SequenceFile, use the full access log file for input data. (You uploaded the access log file to the HDFS `/user/training/weblog` directory when you performed the "Using HDFS" lab.)

After you have created the compressed SequenceFile, you will write a second MapReduce application to read the compressed SequenceFile and write a text file that contains the original log file text.

Write a MapReduce program to create sequence files from text files

1. Determine the number of HDFS blocks occupied by the access log file:

   a. In a browser window, start the Name Node Web UI. The URL is `http://localhost:50070`

   b. Click "Browse the filesystem."

   c. Navigate to the `/user/training/weblog/access_log` file.

   d. Scroll down to the bottom of the page. The total number of blocks occupied by the access log file appears in the browser window.

2. Complete the stub file in the `createsequencefile` project to read the access log file and create a SequenceFile. Records emitted to the SequenceFile can have any key you like, but the values should match the text in the access log file. (Hint: You can use Map-only job using the default Mapper, which simply emits the data passed to it.)

   Note: If you specify an output key type other than `LongWritable`, you must call `job.setOutputKeyClass` – *not* `job.setMapOutputKeyClass`. If you specify an output value type other than `Text`, you must call `job.setOutputValueClass` – *not* `job.setMapOutputValueClass`.

3. Build and test your solution so far. Use the access log as input data, and specify the `uncompressedsf` directory for output.

4. Examine the initial portion of the output SequenceFile using the following command:

```
$ hadoop fs -cat uncompressedsf/part-m-00000 | less
```

   Some of the data in the SequenceFile is unreadable, but parts of the SequenceFile should be recognizable:

   - The string `SEQ`, which appears at the beginning of a SequenceFile
   - The Java classes for the keys and values

- Text from the access log file

5. Verify that the number of files created by the job is equivalent to the number of blocks required to store the uncompressed SequenceFile.

Compress the Output

6. Modify your MapReduce job to compress the output SequenceFile. Add statements to your driver to configure the output as follows:
   - Compress the output file.
   - Use block compression.
   - Use the Snappy compression codec.
7. Compile the code and run your modified MapReduce job. For the MapReduce output, specify the `compressedsf` directory.
8. Examine the first portion of the output SequenceFile. Notice the differences between the uncompressed and compressed SequenceFiles:
   - The compressed SequenceFile specifies the `org.apache.hadoop.io.compress.SnappyCodec` compression codec in its header.
   - You cannot read the log file text in the compressed file.
9. Compare the file sizes of the uncompressed and compressed SequenceFiles in the `uncompressedsf` and `compressedsf` directories. The compressed SequenceFiles should be smaller.

Write another MapReduce program to uncompress the files

10. Starting with the provided stub file, write a second MapReduce program to read the compressed log file and write a text file. This text file should have the same text data as the log file, plus keys. The keys can contain any values you like.

11. Compile the code and run your MapReduce job.

    For the MapReduce input, specify the `compressedsf` directory in which you created the compressed SequenceFile in the previous section.

    For the MapReduce output, specify the `compressedsftotext` directory.

12. Examine the first portion of the output in the `compressedsftotext` directory.

    You should be able to read the textual log file entries.

Optional: Use command line options to control compression

13. If you used ToolRunner for your driver, you can control compression using command line arguments. Try commenting out the code in your driver where you call. Then test setting the `mapred.output.compressed` option on the command line, e.g.:

```
$ hadoop jar sequence.jar \
 stubs.CreateUncompressedSequenceFile \
 -Dmapred.output.compressed=true \
 weblog outdir
```

14. Review the output to confirm the files are compressed.

This is the end of the lab.