The distributed cache in MapReduce is almost always required for any complex assignment involving dependent libraries and code. One very common operation is passing cache files for use in each map/reduce task JVM. This recipe will use the MapReduce API and the distributed cache to mark any lines in the news archive dataset that contain one or more keywords denoted in a list. We will use the distributed cache to make each mapper aware of the list location in HDFS.

Getting ready

This recipe assumes you have a basic familiarity with the Hadoop 0.20 MapReduce API. You will need access to the `news_archives.zip` dataset supplied with this book. Inside the ZIP file, you will find `rural.txt` and `science.txt`. Place both in a single HDFS directory. Additionally, inside the ZIP file you will find `news_keywords.txt`. You will need to place this file in an HDFS directory with the absolute path `/cache_files/news_archives.txt`. Feel free to add any additional words to this file, so long as they each appear on a new line.

You will need access to a pseudo-distributed or fully-distributed cluster capable of running MapReduce jobs using the newer MapReduce API introduced in Hadoop 0.20.

You will also need to package this code inside a JAR file that is to be executed by the Hadoop JAR launcher from the shell. Only the core Hadoop libraries are required to compile and run this example.

How it works...

First, we set up our imports and create a public class `LinesWithMatchingWordsJob`. This class implements the Hadoop `Tool` interface for easy submission using the `ToolRunner`. Before the job is submitted, we first check for the existence of both input and output parameters. Inside the `run()` method, we immediately call the `DistributedCache` static helper method `addCacheFile()` and pass it a hardcoded reference to the HDFS cache file at the absolute path `/cache_files/news_keywords.txt`. This file contains the keywords, separated by newline characters, that we are interested in locating within the news archives corpus. We pass the helper method a URI reference to this path and the `Configuration` instance.

Now we can begin configuring the rest of the job. Since we are working with text, we will use the `TextInputFormat` and `TextOutputFormat` classes to read and write lines as strings. We will also configure the `Mapper` class to use the public static inner class `LineMarkerMapper`. This is a map-only job, so we set the number of reducers to zero. We also configure the output key type to be `LongWritable` for the line numbers and the output value as `Text` for the words, as we locate them. It's also very important to call `setJarByClass()` so that the `TaskTrackers` can properly unpack and find the `Mapper` and `Reducer` classes. The job uses the static helper methods on `FileInputFormat` and `FileOutputFormat` to set the input and output directories respectively. Now we are completely set up and ready to submit the job.

The `Mapper` class has two very important member variables. There is a statically compiled regex pattern used to tokenize each line by spaces, and a wordlist `Set` used to store each distinct word we are interested in searching for.

The `setup()` method in the Mapper is told to pull the complete list of cache file URIs currently in the distributed cache. We first check that the URI array returned a non-null value and that the number of elements is greater than zero. If the array passes these tests, grab the keywords file located in HDFS and write it to the temporary working directory for the task. Save the contents in a local file named `./keywords.txt`. Now we are free to use the standard Java I/O classes to read/write off the local disk. Each line contained in the file denotes a keyword that we can store in the keywords' HashSet. Inside our `map()` function, we first tokenize the line by spaces, and for each token, we see if it's contained in our keyword list. If a match is found, emit the line number it was found on as the key and the token itself as the value.

Note

Use the distributed cache to pass JAR dependencies to map/reduce task JVMs

Very frequently, your map and reduce tasks will depend on third-party libraries that take the form of JAR files. If you store these dependencies in HDFS, you can use the static helper method `DistributedCache.addArchiveToClassPath()` to initialize your job with the dependencies and have Hadoop automatically add the JAR files as classpath dependencies for every task JVM in that job.

Distributed cache does not work in local jobrunner mode

If the configuration parameter `mapred.job.tracker` is set to `local`, the `DistributedCache` cannot be used to configure archives or cache files from HDFS.