

# InvertedIndex

## Workshop Exercise: Inverted Index

This project is designed to get you familiar with the MapReduce environment by having you write a one-pass MapReduce program to calculate an **inverted index** over a set of documents.

Given a body of input text, an offset indexer uses Hadoop to produce an index of all the words in the text. For each word, the index has a list of all the locations where the word appears, and optionally a text excerpt of each line where the word appears. Running the line indexer on the complete works of Shakespeare, the following input lines:

```
lucrece.txt, offset 38624: To cipher what is writ in learned books,  
orlando.txt, offset 66001: ORLANDO Which I take to be either a fool or a  
cipher.
```

would produce the following output for the word "cipher:"

```
cipher      lucrece.txt@38624,orlando.txt@66001,...
```

The goal of this exercise is to develop the inverted index program and run it in Hadoop on a set of documents. You may use the language of your choice for this project. Interpreters for Perl, PHP, Python, and Ruby have been installed, as well as Java (of course).

If you are going to attempt this project in Java, follow the [Java instructions](#). If you're going to use a scripting language, follow the [Streaming instructions](#).

Note: If you get stuck, there are example solutions in `~/solutions`.

## Java Instructions


### Step 1: Start Eclipse

Start Eclipse (via the icon on the desktop). A project has already been created for you called `LineIndexer`. This project is preloaded with a source

code "skeleton" for the activity. This workspace also contains another project containing all the Hadoop source code and libraries. This is required as a build dependency for LineIndexer; it's also there for your reference and perusal.

## Step 2: Create the Mapper Class

Open the `LineIndexMapper` class and implement the `map()` method.

 **Review:** The map function takes four parameters which by default correspond to:

1. `LongWritable key` - the byte-offset of the current line in the file
2. `Text value` - the line from the file
3. `OutputCollector` - output - this has the `.collect` method to output a `<key, value>` pair
4. `Reporter reporter` - allows us to retrieve some information about the job (like the current filename)

The map function should output `<"word", "filename@offset">` pairs. Despite the name of the task (Line Indexer) we will actually be referring to locations of individual words by the **byte offset** at which the line starts – not the "line number" in the conventional sense. This is because the line number is actually not available to us. (We will, however be indexing a line at a time-- thus the name "Line Indexer.") Large files can be broken up into smaller chunks which are passed to mappers in parallel; these chunks are broken up on the line ends nearest to specific byte boundaries. Since there is no easy correspondence between lines and bytes, a mapper over the second chunk in the file would need to have read all of the first chunk to establish its starting line number – defeating the point of parallel processing!

Your map function should extract individual words from the input it is given, and output the word as the key, and the current filename & byte offset as the value. You need only output the byte offset where the line starts. Do not concern yourself with intra-line byte offsets.

Hints:


1. Since in this example we want to output `<"word", "filename@offset">` pairs, the types will both be `Text`.
2. To extract individual words from a string of text, see `java.util.StringTokenizer`.
3. To get the current filename, use the following code snippet:

```
FileSplit fileSplit = (FileSplit)reporter.getInputSplit();
String fileName = fileSplit.getPath().getName();
```

## Create the Reducer Class

Open the `LineIndexReducer` class in the project.

The line indexer Reducer takes in all the `<"word", "filename@offset">` key/value pairs output by the Mapper for a single word. For example, for the word "cipher", the pairs look like: `<cipher, shakespeare.txt@38624>`, `<cipher, shakespeare.txt@12046>`, `<cipher, ... >`. Given all those `<key, value>` pairs, the reduce outputs a single value string. For the line indexer problem, the strategy is simply to concat all the values together to make a single large string, using `,` to separate the values. The choice of `,` is arbitrary – later code can split on the `,` to recover the separate values. So for the key "cipher" the output value string will look like `"shakespeare.txt@38624,shakespeare.txt@12046,shakespeare.txt@34739,.."`. To do this, the Reducer code simply iterates over values to get all the value strings, and concatenates them together into our output String.

 **Important:** The following Java programming advice is important for the performance of your program. Java supports a very straightforward string concatenation operator:

```
String newString = s1 + s2;
```

which could just as easily be:

```
s1 = s1 + s2;
```

This will create a new string containing the contents of `s1` and `s2`, and return it to the reference on the left. This is  $O(|s1| + |s2|)$ . If this is performed inside a loop, it rapidly devolves into  $O(n^2)$  behavior, which will make your reducer take an inordinately long amount of time. A linear-time string append operation is supported via the `StringBuilder` class; e.g.:

```
StringBuilder sb = new StringBuilder();
sb.append(s1);
sb.append(s2);
sb.toString(); // return the fully concatenated string at the end.
```

## Examine the driver program

Open the `LineIndexer` class. This is the main "driver" for the program, which configures the MapReduce job. This class has already been written for you. You may want to examine its body so you understand how MapReduce jobs are created and dispatched to Hadoop.

## Run JUnit tests


In the Package Explorer in Eclipse, open the `test/index` directory. Right click on `AllTests.java`, select "Run as..." and "JUnit test." This will use the Eclipse JUnit plugin to test your Mapper and Reducer. The unit tests have already been written in `MapperTest.java` and `ReducerTest.java`. These use a library developed by Cloudera to test mappers and reducers, called MRUnit. The source for this library is included in your workspace.

## Compile your system

Open a terminal window, navigate to `~/workspace/LineIndexer/`, and compile your Java sources into a jar:

```
$ cd ~/workspace/LineIndexer
$ ant
```

This will create a file named `indexer.jar` in the current directory. If you're curious about the instructions ant followed to create the compiled object, read the `build.xml` file in a text editor.

 You can also run JUnit tests from the ant buildfile by running `ant test`

## Run your program

In the previous exercise, you should have loaded your sample input data into Hadoop. See Aaron for help if you got stuck. Now we're going to run our program over it.

Run:

```
$ hadoop jar indexer.jar index.LineIndexer
```

This will read all the files in the `input` directory in HDFS and compute an inverted index. It will be written to a directory named `output`.

View your results by using `hadoop fs -cat filename`. You may want to pipe this into `less`.

If your program didn't work, go back and fix places you think are buggy. Recompile using the steps above, and then re-run the program. Before you run the program again, though, you'll need to remove the output directory that you already created:

```
$ hadoop fs -rmr output
```

... or else Hadoop will refuse to run the job and print an error message ("Directory already exists").

### Testing the mapper

If you want to test just the mapper without a reducer, add the following line to `LineIndexer.java`'s `runJob()` method:

```
conf.setNumReduceTasks(0);
```

The results of the mappers will be written straight to HDFS with no processing by the reducer.

You may note that the `StringTokenizer` class doesn't do anything clever with regards to punctuation or capitalization. If you've got extra time, you might want to improve your mapper to merge these related tokens.

## Solutions

Example solution source code is available in the `~/solution` directory; the Java source is in the `index/` package subdirectory.

## Streaming Instructions

If you plan to use a scripting language to run this example, follow this set of instructions.

If you need additional reference material, see the official Streaming documentation at

<http://hadoop.apache.org/core/docs/current/streaming.html>

### Load the streaming-specific input data

In the "[Getting Familiar With Hadoop](#)" exercise, we loaded the Shakespeare corpus into HDFS. Due to internal differences in how Java MapReduce and streaming work, some important metadata will be unavailable to your mapper script. We've preprocessed the text files and inlined this metadata into the documents. You should remove the original input files from HDFS and load the streaming-specific input files as follows:

```
$ hadoop fs -rmr input
$ rm -rf input
$ tar xzf shakespeare-streaming.tar.gz
$ hadoop fs -put input input
```

## Create your mapper script

Pop open the text editor of your choice (pico, vim, emacs, and gEdit have all been loaded into this virtual machine). Create a new script program to run as the mapper.

The mapper will receive on stdin lines containing:

- a key consisting of *filename* "@" *offset*
- a tab character ('\t')
- a value consisting of the line of the input file which occurs at that offset.
- a newline character ('\n')

For example, here are a few lines from one of the files:

```
hamlet@0          HAMLET
hamlet@8
hamlet@9
hamlet@10         DRAMATIS PERSONAE
hamlet@29
hamlet@30
hamlet@31         CLAUDIUS          king of Denmark. (KING CLAUDIUS:)
hamlet@74
hamlet@75         HAMLET  son to the late, and nephew to the present king.
hamlet@131
```

Note that after the `hamlet@offset` string is a tab (\t) character.

Your map function should extract individual words from the input it is given, and output the word as the key, and the current filename & byte offset as the value. You need only output the byte offset where the line starts. Do not concern yourself with intra-line byte offsets.

You emit intermediate (key, value) pairs to the reducer by writing strings of the form `"key tab value newline"` to stdout.

## Create your reducer script

You should create another program to run as the reducer. The lines sent from your mapper instances will be sent to the appropriate reducer instance. Several values with the same key will be sent to your program on stdin as successive lines of input. Each line contains a key, a tab, a value, and a newline; this is the same format for the input as received by the mapper. All the lines with the same key will be sent one after another, possibly followed by a set of lines with a different key, until the reducing process is complete.

The line indexer reducer takes in all the `<"word", "filename@offset">` key/value pairs output by the Mapper for a single word. For example, for the word "cipher", the pairs look like: `<cipher, shakespeare.txt@38624>`, `<cipher, shakespeare.txt@12046>`, `<cipher, ... >`. Given all those `<key, value>` pairs, the reduce outputs a single value string. For the line indexer problem, the strategy is simply to concat all the values together to make a single large string, using `,` to separate the values. The choice of `,` is arbitrary – later code can split on the `,` to recover the separate values. So for the key "cipher" the output value string will look like `"shakespeare.txt@38624,shakespeare.txt@12046,shakespeare.txt@34739,.."`. To do this, the Reducer code simply iterates over values to get all the value strings, and concatenates them together into our output String.

## Run unit tests

Two test scripts named `test-mapper.sh` and `test-reducer.sh` have been written; these take as arguments the name of the mapper or reducer script respectively; they determine if the output conforms to the expected format.

To run them, use:

```
$ cd ~
$ ./test-mapper.sh mapper-script-name
$ ./test-reducer.sh reducer-script-name
```

## Run the streaming program

Make sure your input scripts are executable:

```
$ chmod a+x (mapper/script/name)
```

```
$ chmod a+x (reducer/script/name)
```

You can then run your program with Hadoop streaming via:

```
$ hadoop jar $HADOOP_HOME/contrib/streaming/hadoop-0.18.3-CH-0.2.1-streaming.jar -input input -output output -file (path/to/mapper/script) -file (path/to/reducer/script) -mapper (mapper-basename) -reducer (reducer-basename) -inputformat org.apache.hadoop.mapred.KeyValueTextInputFormat
```

The `-file` arguments tell what files from the local filesystem should be loaded into the runtime environment. In this case, that's your two scripts for the mapper and the reducer.

The `-mapper` and `-reducer` arguments tell what command to run in the runtime environment to launch your process. This is just the filename of your scripts (with no paths associated with them). Alternatively, since you're running your MapReduce process inside a single VM (as opposed to a distributed environment), you could just specify the full absolute path to the mapper and reducer scripts to the `-mapper` and `-reducer` arguments, and omit the `-file` arguments.

This will read all the files in the `input` directory in HDFS and compute an inverted index. It will be written to a directory named `output`.

So, for example, if your mapper program was in `/home/training/mapper.py` and your reducer program was in `/home/training/reducer.py`, you could run:

```
$ chmod a+x ~/mapper.py
$ chmod a+x ~/reducer.py
$ hadoop jar $HADOOP_HOME/contrib/streaming/hadoop-0.18.3-CH-0.2.1-streaming.jar -input input -output output -file ~/mapper.py -file ~/reducer.py -mapper mapper.py -reducer reducer.py -inputformat org.apache.hadoop.mapred.KeyValueTextInputFormat
```

View your results by using `hadoop fs -cat output/part-00000`. You may want to pipe this into `less`.

If your program didn't work, go back and fix places you think are buggy. Recompile using the steps above, and then re-run the program. Before you run the program again, though, you'll need to remove the output directory that you already created:

```
$ hadoop fs -rmr output
```



... or else Hadoop will refuse to run the job and print an error message ("Directory already exists"). Then you can re-run your program until it gives you the results you need.

### **Debugging the mapper**

If you want to test just the output of the mapper, without being processed by your reducer, run streaming with `-reducer cat` to pass the output of the mapper straight to the output.

## **Solutions**

Example solution source code (written in python) is available in the `~/solution` directory.

---

This document contains content (c) Copyright 2008 University of Washington; redistributed under the terms of the Creative Commons Attribution License 3.0. All other contents (c) Copyright 2009 Cloudera, Inc.