Overview

One approach to this project would be to launch the MapReduce and Spark processes by shelling out to the console programmatically from within their main progress. That is, launch a command line application written in Java or Scala, start a MapReduce or Scala job, use a module like java.lang.Process, and launch the Hadoop job as a subprocess.

There are a couple of weaknesses to this approach. First, the program becomes heavily dependent on the environment in which it's running. The `PATH` must contain an executable with the approperate name. No shell syntax may be used, limiting the extensibility of the command issued.

Environment variables could change underneath the application, increasing the attack surface. For example, if an attacker gained access to the server, he or she could replace the target executable in the `PATH` with a binary of his or her own choosing, allowing arbitrary code execution.

Finally, because the CLI is communicating with a subprocess over StdIn and StdOut, the output for the MapReduce and Spark programs, which very well could have been in a structured format like XML or JSON, will now to unstructured, written as arbitary text.

Of course, all of the weaknesses are fairly irrelevant for a final project in a Cloud Computing class, but they're usually convincing enough in an industry setting for this approach to be eschewed.

The primary focus of our project is to build our Tiny Google such that MapReduce and Spark jobs are submitted programmatically using a REST API instead of shelling out to the console. By doing this, we're practicing for how we would be tasked to implement Tiny Google were it to be part of our job. Using a REST API is more robust, safer, and more extensible. With this approach, Hadoop doesn't even have to be installed on the machine receiving the requests; you can communicate with any machine with an accessible IP. As a result, this server can have more finely grained security settings.

Functional Requirements

Users will access our application in a web browser. On loading the web page, a web-based terminal interface will appear, telling the user what commands it accepts. It will accept three commands: `help`, `search <terms>`, and `upload`. The `help` command will print to the terminal instructions. The `upload` command will open a file upload dialogue so the user can search their computer for a file to index. The `search` command is followed by a single term. Searching will print to the terminal a list of documents and a link to access them with the HTML `<a href="document-link">document-name</a>"`. Clicking on the document name will open a link to the document so that it can be viewed in the browser. Following the name of the document, a blerb of text from the will be printed to the browser terminal, containing the context of the word's use in the document. Finally, the time it took to geneate the list from MapReduce and from Spark will be printed to the screen, so our performance metrics will be evident from the browser. These metrics will also be available after `upload` commands.

Architecture

We will orchestrate the application with two Docker containers running under Docker Compose. The first container will run a Golang web server, which serves the static web front end. When an HTTP request from the front end is sent to the Go server, the server will launch and time two HTTP requests to the second container. The second container runs Hadoop, Spark, and MapReduce on top of a REST server. This is not an original container we developed, but a pre-existing one available as open source.

These two requests to the second container will launch a MR job and a Spark job. Each jobs reads from it's owned inverted index and writes it back out in the event of an upload. The jobs will return a JSON response back to the Go server, which will aggregate them into a payload for the client. The client will parse and display the results.

Inverted Index

Our inverted index will map fom string keys (which are the search terms) to a list of documentRef objects, which is defined below.

```
string -> List[documentRef]
```

Each document reference stores the name of the document, the path to the document, and a list of positions in the document where the search term appears.

```
object documentRef {
    name: string
    path: string
    occurrences: List[FileLocation]
}
```

Our inverted index will be seralized into a persistent format (with JSON or using Java `ObjectOutputStream`). All MapReduce jobs will read and write the inverted index to the same location on disk, sharing the index across jobs.

Spark jobs will do the same, only at a separate path, so as not to incur a race condition or share data between the two systems.

Delivery Dates

```
- Web Front End (mostly done), 12/10
- Golang Reverse Proxy and Timer (partially complete), 12/11
- Inverted Index, 12/12
- Map Reduce Jobs, 12/16
```

```
- Spark Jobs, 12/16
- Final Paper, 12/17
```