Anthony Poerio ([adp59@pitt.edu](mailto:adp59@pitt.edu))

Andrew Masih ([anm226@pitt.edu](mailto:anm226@pitt.edu))

CS1699 – Cloud Computing

Tiny Google – Spark

Final Report

**TinyGoogle - Spark**

**Overview**

For our final project, we created a second implementation of the **Tiny Google Search Engine** written to run on the **Spark** Cloud Computing framework, using **Scala** as the implementation language.

Our goals for this document are as follows:

* We will outline the overarching design for our **Tiny Google** search engine.
* We will discuss the implementation details and major decisions made to ensure that Tiny Google runs correctly and efficiently on the Scala/Spark framework.
* We will share output for a series of example keyword searches.
* Finally, we will compare and contrast our original **Python/ Hadoop Streaming** solution with our new **Scala / Spark** implementation, in order to make a determination about which method is preferable – both for programmers and for end-users.

**Design**

**User Input**

Similar to our original version of **Tiny Google**, we decided to give the Spark implementation a command line interface.

The interface works as follows:

1. **To INDEX our data**
   1. Pass in the keyword argument **-i**, and the directory containing your dataset
   2. Example:  *`scala tinyGoogle.scala –i “/my/path/to/books/dataset/”`*
2. **To SEARCH an index, using keywords**
   1. Pass in the keyword argument **–s**, and a set of keywords
   2. Example: `*scala tinyGoogle.scala –s “this is an example search”`*

If a user decides to search before an index is created, we will build an index and then search, in sequence.

Output is passed to the command line console, and we show the **Top-3 results** for EACH keyword, in order of user-entry.

**Indexing Algorithm & Data Structure**

In order to make the most of Spark’s newest and most optimized features, we decided to use a **DataFrame** to store our index.

For a programmer, the main advantages of this decision are:

* Spark DataFrames are easy to use and make
* They provide a fast and efficient SQL-like querying interface
* And the resultant code is easy to understand and modify

For an end-user the advantage is speed. First introduced in Spark Version 1.6, queries on DataFrames are optimized by Spark’s internal code, ensuring that functions are executed in the most efficient order. This allows us to search a created index with high-efficiency – all while offloading the optimizations themselves to Spark.

The chief challenge here is designing an efficient DataFrame format (columns) that provides quick access to all necessary information, using SQL-like queries. A secondary, manipulating data such that it can be parsed by the DataFrame API. And for that RDD’s were used. More details on these transformations will be outlined below, in the implementation details section.

We chose the following column format for the DataFrame we created.

+--------------------+----------+-----+

| title| word|count|

+--------------------+----------+-----+

| N/A|starter\_df| 1|

|AdventuresOfHuckl...| widows| 2|

|AdventuresOfHuckl...| reforming| 3|

|AdventuresOfHuckl...| morals| 2|

|AdventuresOfHuckl...| county| 2|

|AdventuresOfHuckl...| doing| 30|

|AdventuresOfHuckl...| clock| 18|

and so on

With this format in hand, we are able to write simple FILTER and GROUPBY queries at search time, quickly finding results for the end-user.

In fact, Spark is able to optimize our queries, precisely because the data we are searching through is in a DataFrame format.

**Search Algorithm & Displaying Results**

Our search algorithm works by showing the **TOP 3** relevant books for each search term, ordered **by occurrence counts**.

We achieve this by using the Spark DataFrame API to query the index built in the previous step.

We are also able to display context by finding the FIRST occurrence of the target keyword in any given book, and indexing directly to that line, showing the surrounding data.

Because we have all of this data encoded in our DataFrame index, and because Spark helps us with optimizations for each search query on DataFrames, we are able to perform these queries quickly and easily.

Here are examples….

**Implementations**

**User Interface**

We were able to quickly and easily handle user input, and the overall interface by parsing command line arguments……

**Indexing**

We indexed our dataset by loading each file from the user’s target directory and running it through a series of RDD transformations.

Once we have an RDD in a format that is roughly similar to our DataFrame’s column design, we pass each tuple within the RDD to a Scala Case Class. This ensures that the DataFrame API is able to parse all of our data in in a consistent way. Here’s an example …..

Finally, we perform this same set of operations for EACH target file, and union the resulting DataFrame with an accumulator, which holds data for the entire directory.

Once these operations are complete, we save the index in JSON format, using functions built into Spark directly.

**Searching**

Andrew to talk me through this…. We indexed our dataset by loading each file from the user’s target directory and running it through a series of RDD transformations.

….

….

….

**Example Runs**

**Three Searches**

Here are three example searches, showcasing our outputs……

**Comparison Analysis**

**Ease of Implementation**

Here are three example searches, showcasing our outputs……

**Index Time**

Here are three example searches, showcasing our outputs……

**Search Time**

We were able to quickly and easily handle user input, and the overall interface by parsing command line arguments

**Conclusions**

**Final Decision**

We suggest…. Primarily… because…. Here are three example searches, showcasing