CS 2510– Computer Operating Systems

Project 2 – MiniGoogle
Fall Term 2015

Project Goals

Data-intensive Computing and Cloud Computing are two emerging computing paradigms, which are poised to play an increasingly important role in the way Internet services are deployed and provided. Increasingly, large-scale Internet services are being deployed atop multiple geographically distributed data centers. These services must scale across a large number of machines, tolerate failures, and support a large volume of concurrent requests. The main objective of this project is to design and implement a **basic data-intensive application** to **index** and **search** large documents. More specifically, the goal is to design a **simple search engine**, referred to as **tiny-Google,** to retrieve documents relevant to simple **search queries** submitted by users.

This project is designed to provide students with exposure to new programming models of computing and processing of large scale data, such as MapReduce and Hadoop. It is expected that the students will learn:

- How to divide both data and work across a cluster of computing machines?
- How to design algorithms for data-intensive computing?

tiny-Google Components

The design and implementation of **tiny-Google** involves three basic components:

1. The first component consists in designing a simple **user interface** (**UI**) to allow (i) **indexing** a document and (ii) **submit** simple search **queries** and **retrieve** relevant document objects. The search query consists of a query name and a short list of keywords.
2. The second component consists of developing a data structure, referred to as **inverted index** (**II**), to support the full-text search component of an information retrieval engine. In its basic form, an II contains a posting list for each term. The posting list is a linked list of individual postings, each of which consists of a document identifier (id) and a payload. The id value uniquely identifies a document, while the payload contains information about "occurrences" in the document. In this project, the payload is reduced to the **term frequency**, defined as the number of times a given term occurs in the document.
3. The third component, **ranking and retrieval** (**RaR**), consists in retrieving the documents relevant to the query in a **ranked** order. Given a search query for a particular pattern of words, RaR returns all the documents that contain the words of the search pattern, rank-ordered in the decreasing order of the word count associated with each document.

Inverted Indexes

Given a collection of documents, $\mathcal{D}$, an inverted index is a data structure, which given a term provides access to the list of documents that contain the term. In its basic form, an inverted index consists of:

- A set of terms, $\mathcal{T}$, and
- A set of posting lists, $\mathcal{L}$, each of which is associated with a specific term in $\mathcal{T}$. A posting consists of a document identifier and a payload, for each document in $\mathcal{D}$ which contains the term associated with the posting list. Depending on the objective of the application, the payload filed may be empty, in which case the existence of the posting only indicates the presence of the term in the document, or contain additional information relevant to the frequency of occurrence of the term in the document. In this project, the payload is the number of times a term occurs in a document. A simple illustration of an inverted index is depicted in Figure 1. In this example, the inverted index contains four terms and reports on the occurrence of these terms in twelve documents.

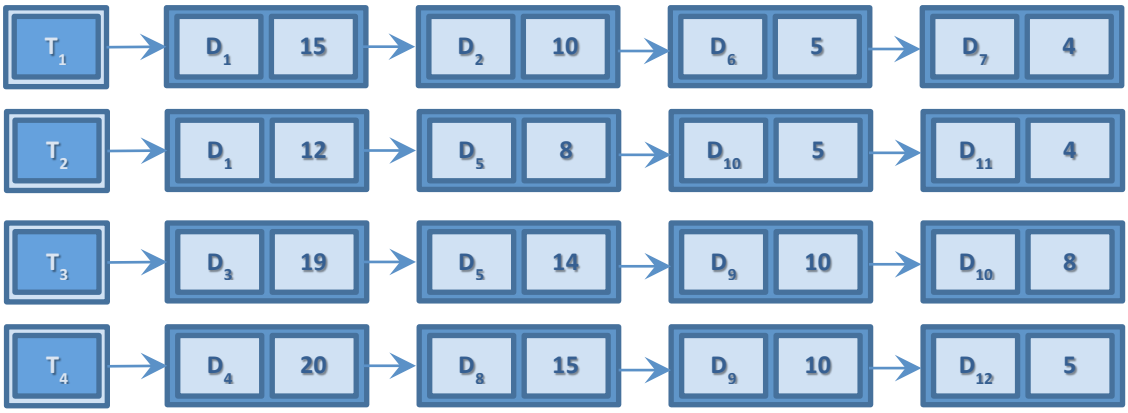

Figure 1. The figure illustrates an inverted index associated with a list of postings. Each posting consists of a document Id and a payload. The payload represents the number of times the term appears in the document.

Project Implementation

The **tiny-Google** framework will be implemented using a **socket-based client-server** computing model using a traditional file system.

Part I – Socket-based Implementation

In the first part of the project, you are to design and implement a client-server based model, using sockets in an Internet domain, to support the operations of **tiny-Google**. In this model, the client process provides a simple interface to its users to submit **indexing requests** and **search queries**. The **tiny-Google** server responds to these request and queries, as follows:

1. A client user submits an **indexing request** to the user interface, referred to as **uiShell.** The submitted request contains a **directory path name**, where the document is stored. In response, the uiShell establishes a connection to the **tiny-Google** server and uses the connection to send to the server the user's request to index the document specified by the path name. The uiShell then awaits the outcome. Upon receiving the request, the **tiny-Google** server creates a process, referred as the **indexing-master** to carry out the document indexing task; the server then returns to wait for the next request from other clients. Upon receiving the indexing query, the master selects a set of indexing **helpers**, each residing in a different machine, and divides the task of indexing the document among these helpers. Each helper receives a segment of the document and creates a **word-count** for each word in its assigned segment; it then uses the word-count outcome to update the **master inverted index**. Upon updating the master index, each helper informs the master of the failure or success of its assigned task. Upon hearing from all helpers, the master informs the uiShell of the final outcome (success or failure), which in turn informs the user.
2. The user can also issue a search query to retrieve information related to already indexed documents. The search query contains a number of items, in the form a key words, which are relevant to the search. The uiShell establishes a connection to the **tiny-Google** server; it then sends the search query request, along with the associated items, to the **tiny-Google** server and waits for the response. Upon receiving the search query, the **tiny-Google** server creates inserts the query, along the Internet address and port number of the uiShell, into the "**Work Queue"**, and if wakes up a sleeping **search-query master**, if one exits, to handle the query. Then, the **tiny-Google** server returns to listen to new requests from other clients. The search-query master selects a set of **search-quer**y **helpers**, residing in different machines, and tasks each one of them to (i) search a segment of the **master inverted index** and (ii) retrieve the name of the documents which contain all the words of the query. Upon completing task (i) and (ii), the helpers "shuffle-exchange" aggregate the partial results for each document. Upon receiving the query outcome from all the helpers, the master sorts the outcome into a final response and sends it to the uiShell client. The uiShell client lists the outcome for the user.

Project Requirement

The following are the expected deliverables and due dates for this project:

1. Implement the socket-based **tinyGoogle** search engine on a Linux cluster.
2. Submit the final code of the implementation and a final report no later than **December 13, 2015 (Midnight sharp).**
    a. Email a self contained tar ball of all the files needed to deploy your project to both the instructor and the TA.

b. Submit final report discussing your implementations, the techniques used to optimize the system, when applicable.
c. Schedule a time to demonstrate your project. Available dates TBA.