CS 2510 Project 2 – TinyGoogle

Deadline Dec 12, 2018

Project Goals

Data-intensive Computing and Cloud Computing is poised to play an increasingly important role in Internet services. The services provided therein 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**, aka tiny-Google**,** to retrieve documents relevant to simple **search queries** submitted by users.

Students will be exposed to new programming models of computing and processing of large-scale data, namely 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
- How to implement these algorithms using data-centric computing and storage paradigms

tinyGoogle Components

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

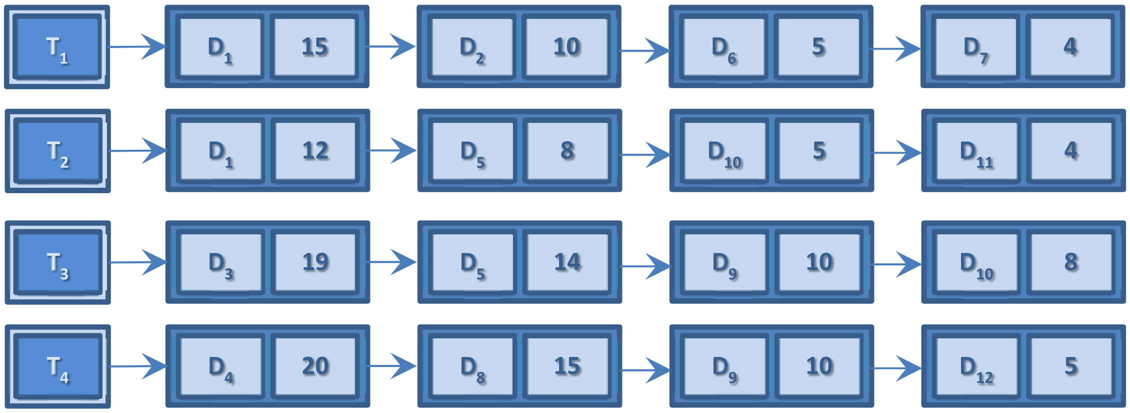
1. A simple **input/output mechanism** to allow (i) **indexing** a document and (ii) **submit** simple search **queries** and **retrieve** relevant document objects. The search query consists of a query name and a short list of keywords
2. A data structure, referred to as **inverted index** (**II**), to support the full-text search component of an information retrieval engine. In general, an II contains a *posting list* for each term, that is, a linked list of individual postings, each of which consists of a document identifier (Did) 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*, that is, the number of times a given term occurs in the document (see example below)
3. A **ranking and retrieval** (**RaR**) process, consisting in retrieving the documents relevant to the query in a ranked order. Given a search query with a set of search terms, RaR returns all the documents that contain the search terms in the decreasing order of the term frequency

Details: Inverted Indexes

Given a collection of documents, D, an II is a data structure that keeps a list of documents that contain the term. In its basic form, an II consists of:

- A set of terms, $T = T_1, T_2, \ldots, T_n,$ and
- A set of posting lists, L_i, each of which is associated with a specific term in T_i. A posting L_{ij} consists of a document identifier and a payload <Did$_{ij}$, P$_{ij}$>, for each document in D_{ij} that contains the term associated with the posting list. Depending on the objective of the application, the payload field 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 = ***T_1*, *T_2*, …, *T_4*,** and reports on the occurrence of these terms in twelve documents, ***D_1*, *D_2*, …, *D_{12}*,**.



T_1 → D_1 15 → D_2 10 → D_6 5 → D_7 4
T_2 → D_1 12 → D_5 8 → D_{10} 5 → D_{11} 4
T_3 → D_3 19 → D_5 14 → D_9 10 → D_{10} 8
T_4 → D_4 20 → D_8 15 → D_9 10 → D_{12} 5

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 two different computing platforms. The first platform is a **socket-based client-server** computing model using a traditional file system. The second platform uses the **MapReduce** computation model and **Hadoop** running on a Linux cluster.

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 the indexing requests and search queries, as follows:

1. An **indexing request** contains a **directory path** where the documents are stored. The request is sent to the **tiny-Google** server, which 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 the same or other clients. The master selects a set of indexing **helpers**, each residing in a different machine (sets of helpers may reside in the same machine), and divides the task of indexing the document among these helpers. Each helper will process a segment of each the document and will create 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 returns the outcome to the user.

2. A search query is intended to retrieve information related to already indexed documents. The search query contains a number of key words; the query is sent to the tiny-Google server; and waits for the response. Upon receiving the search query, the tiny-Google server creates inserts the query into the "**Work Queue"** and wakes up a sleeping **search-query master**, if one exits, to handle the query. 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**, each residing in different machines (sets of helpers may reside in the same machine), and tasks each one of them with (i) searching a segment of the **master inverted index** and (ii) retrieving the name of the documents that contain all the words of the query. Upon completing tasks (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 client.

Part II – Hadoop-based implementation

In the second part of the project, you are to develop a Hadoop program to implement the tiny-Google search engine. As explained in class, Hadoop is an open-source software framework that enables distributed manipulation of large volumes of data, in a reliable, efficient, and scalable way. The main components of Hadoop include:

- Hadoop Distributed File System (HDFS) stores files across storage nodes in a Hadoop cluster. From a client perspective, HDFS appears as a traditional hierarchical file system, supporting typical file management primitives, such as create, delete, move, rename, etc. In addition, HDFS supports a collection of special nodes: *NameNode* provides metadata services within HDFS, and the *DataNode*, which serves storage blocks for HDFS.
- MapReduce engine, which consists of JobTrackers and TaskTrackers.

This part of the project has the same goals of Part I, and you are required to carry out the following tasks:

1. Write a Hadoop program to develop a master inverted index structure to index a collection of documents. Each term in the inverted index is associated with a posting list, whereby each posting contains the identifier of the document and a counter of the number of times the term occurs in the document.
2. Write a Hadoop program, tiny-Google, to search for a particular word in the index file and return the names of all chapters that contain this word.
3. Experiment with at least 2 techniques to optimize the map-reduce code or the Hadoop job configuration by varying the number of mappers and reducers, for example.

Project Requirement

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

1. A design document that discusses your approach for both implementation and experimentation; include metrics (what you'll measure, like response time, CPU time, and/or something else) and experiment parameters (what you'll change, such as number of keywords, maximum number of helpers, etc).
2. Design a client-server architecture to carry out the tinyGoogle search engine operations.
3. Implement the tinyGoogle search engine from item 1 on a Linux cluster.
4. Implement the Hadoop-based tinyGoogle search engine on a Linux cluster.
5. Develop a statistically-significant experiment to compare the performance of each scheme.
6. Submit the final code of the Part I and Part II implementation and a final report no later than **December 12, 2018 anywhere on earth.**
    a. You will get intructions for how to hand in all code used for the implementation, including a Makefile and a README, as well as the report discussing your implementations and the techniques used to optimize the system. (see below)
    b. Schedule a time to demonstrate your project, for the week of **December 13, 2018**.

For full credit, you need to **design** an experiment to measure the performance of each scheme using different input and parameters of your choice. You also need to include in the final report a section to describe the experiment and compare the performance of the two implementations for the different experiments conducted in this project. This section of the report must include graphs (e.g., runtimes of the word list generator, the inverted index generator and the search for both models). **Make sure you include all references, books, tutorials, software components and blogs that you made use of to complete the project.**

You can suggest extensions for Extra Credit: both extensions and how many extra points, up to 20% of this project.