A Cost-Benefit Analysis of Indexing Big Data with Map-Reduce

Dimitrios Siafarikas Argyrios Samourkasidis Avi Arampatzis

Department of Electrical and Computer Engineering
Democritus University of Thrace
University Campus, Kimmeria, Xanthi 67100, Greece
dimitris.siafarikas@hotmail.com, argysamo@gmail.com, avi@ee.duth.gr

Abstract—**We reflect upon the challenge a Big Data analyst faces when dealing with the complex problem of considering the approximate amount of nodes needed for a computation to be completed within a given time. We develop a formula which allows anyone, with the job of designing clusters for massive data sets, to do so. We consider the problem of Inverted Index construction which is widely used in Information Retrieval. We present the various aspects and the challenges of this problem along with details on how the system we developed works.**

Keywords: Information Retrieval, Big Data, Data Mining, Cluster, Inverted Index, Hadoop, Map-Reduce

I. INTRODUCTION

Conventional databases systems have proved their usefulness throughout the years because of their great potential as systems that provide an organized and structured place for data. The new era of information technology, however, requires advanced analytic capabilities.

Difficulties in scalability have emerged during the recent years. Companies like Facebook and Google are capable of creating Gigabytes of data per second. These enormous amounts of datasets spawn various problems and set challenges for the community. The challenges include capture, curation, storage, search, sharing, transfer, analysis and visualization of the so-called *Big Data* [2]. Big Data refers to a collection of data that is so large and complex, it becomes difficult to process using traditional database management tools or data processing applications. Another example to grasp the size of Big Data is the volume of data that is produced from the *Large Hadron Collider (LHC)* experiments at CERN. By 2012, more than 300 trillion collisions had been analyzed. The project generated approximately 27 Terabytes of raw data per day and 25 Petabytes per year.

Currently, the most popular operational approach to Big Data storage and processing is *Horizontal Scalability*, that is the ability to connect multiple hardware and software entities so that they work as a single logical unit. One of the popular such frameworks is Apache's Hadoop [3] which implements the Map-Reduce programming model [10].

In this paper, we purpose a way of building an Inverted Index using the Map-Reduce programming model. Firstly, we program an algorithm in Python in order to comprehend and grasp the idea of Inverted Indices. Then, we modify the algorithm to work on the Map-Reduce programming framework. Hadoop uses Java as the default programming language. Besides that, there is a library, called *Hadoop Streaming* that allows to develop algorithms in *any* programming language. This provides an environment for rapid algorithm development in any language the user likes.

The rest of this paper is organized as follows. Next we elaborate on inverted indices and how these are classically built. In Section III we give a short introduction to the Map-Reduce programming framework. Section IV presents our proposed implementation of inverted index construction using Map-Reduce. In Section V we attempt a cost-benefit analysis of our proposed implementation in comparison to the classical serially-implemented approach. Conclusions and directions for further research are summarized in Section VI.

II. INVERTED INDEX

A. *Information Retrieval*

Information retrieval is the activity of obtaining information relevant to an information need from a collection of information resources [7]. Searches can be based on metadata or on full-text (or other content-based) indexing. An information retrieval process begins when a user enters a query into the system. Queries are formal statements of information needs, for example search strings in web search engines. In information retrieval a query does not uniquely identify a single object in the collection. Instead, several objects may match the query, perhaps with different degrees of relevancy. For effectively retrieving relevant documents, the documents are typically transformed into a suitable representation. One of those is the inverted index, which we elaborate on next.

B. *Inverted Index*

An inverted index is a data structure storing a mapping from content (such as words) to its location in a database file or in a document or in a set of documents. The purpose of an inverted index is to allow fast full-text searches, at a cost of increased processing when a document is added to the database. It is the most popular data structure used in document retrieval systems.

C. Building an Inverted Index

To gain the speed benefits of indexing at retrieval time, we have to build the index in advance. The major steps in this are [7]:

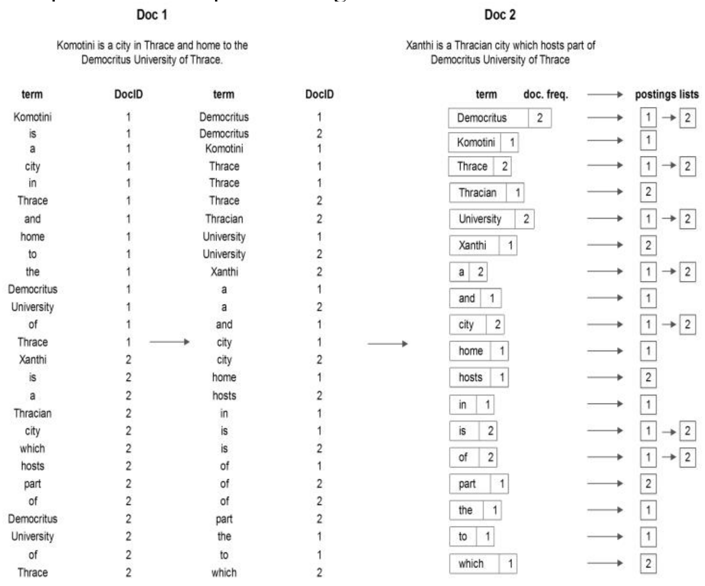
1) Collect the documents to be indexed.

2) Tokenize the text, turning each document into a list of tokens.

3) Do linguistic preprocessing (e.g. stemming or lemmatization), producing a list of normalized tokens, which are the indexing terms.

4) Index the documents that each term occurs in by creating an inverted index, consisting of a dictionary and postings.

The procedure is depicted in Figure 1.



Doc 1

Komotini is a city in Thrace and home to the Democritus University of Thrace.

Doc 2

Xanthi is a Thracian city which hosts part of Democritus University of Thrace

term	DocID
Komotini	1
is	1
a	1
city	1
in	1
Thrace	1
and	1
home	1
to	1
the	1
Democritus	1
University	1
of	1
Thrace	1
Xanthi	2
is	2
a	2
Thracian	2
city	2
which	2
hosts	2
part	2
of	2
Democritus	2
University	2
of	2
Thrace	2

term	DocID
Democritus	1
Democritus	2
Komotini	1
Thrace	1
Thrace	1
Thrace	2
Thracian	2
University	1
University	2
Xanthi	2
a	1
a	2
and	1
city	1
city	2
home	1
hosts	2
in	1
is	1
is	2
of	1
of	2
of	2
part	2
the	1
to	1
which	2

term	doc. freq.	postings lists
Democritus	2	1 → 2
Komotini	1	1
Thrace	2	1 → 2
Thracian	1	2
University	2	1 → 2
Xanthi	1	2
a	2	1 → 2
and	1	1
city	2	1 → 2
home	1	1
hosts	1	2
in	1	1
is	2	1 → 2
of	2	1 → 2
part	1	2
the	1	1
to	1	1
which	1	2

Figure 1. Postings are lists of DocIDs

III. MAP-REDUCE

Map-Reduce is a programming model for processing large datasets with a parallel, distributed algorithm on a cluster. The most popular open-source implementation is Apache's Hadoop cluster system [3]. A Map-Reduce program mainly consists of two functions: a Map function and a Reduce function. In fact the only thing one has to do is write those two functions, while the system manages the parallel execution in a fault-tolerant way, meaning that it also deals with the possibility of failures on nodes' hardware or network's infrastructure upon execution. The tasks that process those functions are called *Map Task* and *Reduce Task*, respectively.

Hadoop is actively collaborated with Hadoop Distributed File System (HDFS), a distributed file system specifically designed to store massive datasets distributed along multiple servers on a cluster. HDFS is based on the Google File System [4]. HDFS is highly fault-tolerant and can be deployed on low-cost hardware. Specifically, small portions, called *chunks*, of the data are spread across different nodes and maybe different racks of a cluster. Each of those chunks is replicated among the nodes at a factor of 3, by default. In the event of a single node failure, no data are lost; there are two other copies of those chunks at a different node. The system then perceives that a node is down and then replicates to another node the data to match the replication factor of three. The administrator can increase this factor to get a more redundant deployment.

A. Map Tasks

The input files for the Map task can be any type; a tuple or a document for example. These inputs are called *elements*. A chunk is a collection of elements and no element is stored across two chunks. The Map task takes an element as an input argument and as an output produces a number of key-value pairs. The output of the task could be zero pairs, too. Keys do not have to be unique. A Map task can produce several key-value pairs with the same key, even from the same input element.

Summarizing, the Map function takes a series of key-value pairs, processes each, and generates zero or more output key-value pairs. The input and output types of the map can be (and often are) different from each other. For example, if the application is doing a word count, the map function would break the line into words and output a key/value pair for each word. Each output pair would contain the word as the key and the number of instances of that word in the line as the value.

B. Reduce Tasks

The framework calls the application's Reduce function once for each unique key in the sorted order. The Reduce function takes as an input a pair of a key and its list of associated values. The output of the Reduce function is a sequence of zero or more key-value pairs. These key-value pairs can be of any type from those sent from Map tasks but it is usually the same type. A Reduce task receives one or more keys and their associated value lists. The output of all the Reduce tasks are merged into a single file. In the word count example, the Reduce function takes the input values, sums them and generates a single output of the word and the final sum.

C. Execution

In brief, a map-reduce job is executed as follows:

1) A number of Map tasks are assigned to one or more chunks of data from the distributed file system. These chunks are converted to a sequence of key-value pairs. The way the pairs are produced is up to the developer of the Map function.

2) The key-value pairs that come out of the Map tasks are then collected by a master controller and sorted by key. Then, each batch of key-value pairs with the same key, is given to a Reduce task.

3) The Reduce tasks then, combine all the values associated with that key in some way determined by the developer of the Reduce function.

The execution flow is showed in Figure 2.

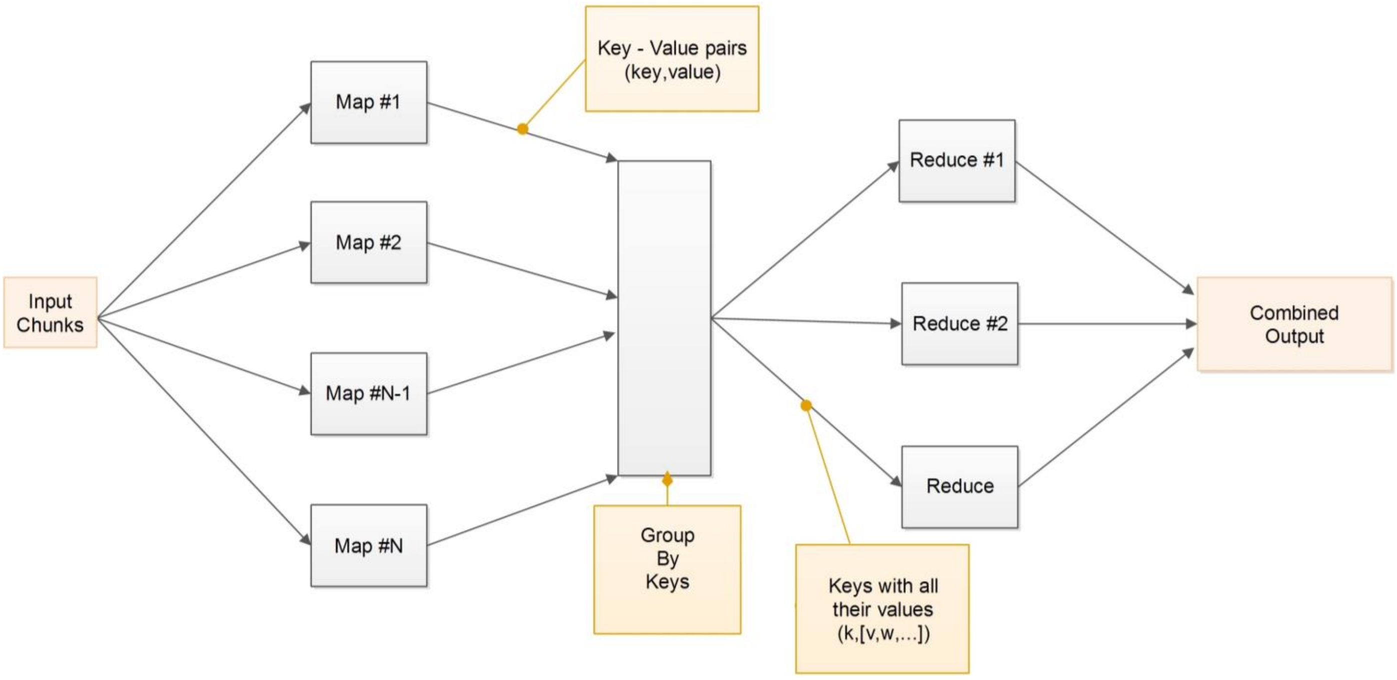


Figure 2. Execution Diagram

IV. BUILDING A NON-POSITIONAL INVERTED INDEX WITH MAP-REDUCE

A. *Map function*

In our implementation, we feed each Map task with one line of data at a time. Specifically, each word of the line that is separated with a space, is transformed to a new string in the format "*word \t DocId",* where \t is the Tab character and *DocID* is the name of the file that this particular word was found in. Also the Map function eliminates all unnecessary characters, like "!@#$%^&*" etc., for the purposes of the Inverted Index.

B. *Combiners*

Without any combiners, the output of the Map tasks input directly to the Reduce tasks. Because of the Map functions taking as an input more than one line, coming from multiple files, we are able to reduce the amount of data that have to be transferred across nodes of the network. This is done as follows. When the same word appears in more than one line, then the Combiner function combines the result to the string with the format:

word \t DocId1:WordFreq1 DocId2:WordFreq2 …

where DocFreq, the occurrence frequency of the particular word. To sum up, the Combiner scans for identical words coming from the same Map task.

C. *Reduce function*

The output of each Combiner constitutes a small subset of the final Inverted Index. Therefore, our goal is to aggregate all those subsets to a final set. So, this function takes as an input the combined data and as an output, makes a final record i.e. the final Inverted Index. The format of the output is the following:

Word \t DocID1:WordFreq1 | DocID2:WordFreq2 | …

Figure 3 summarizes the whole procedure of building an Inverted Index in Map-Reduce.

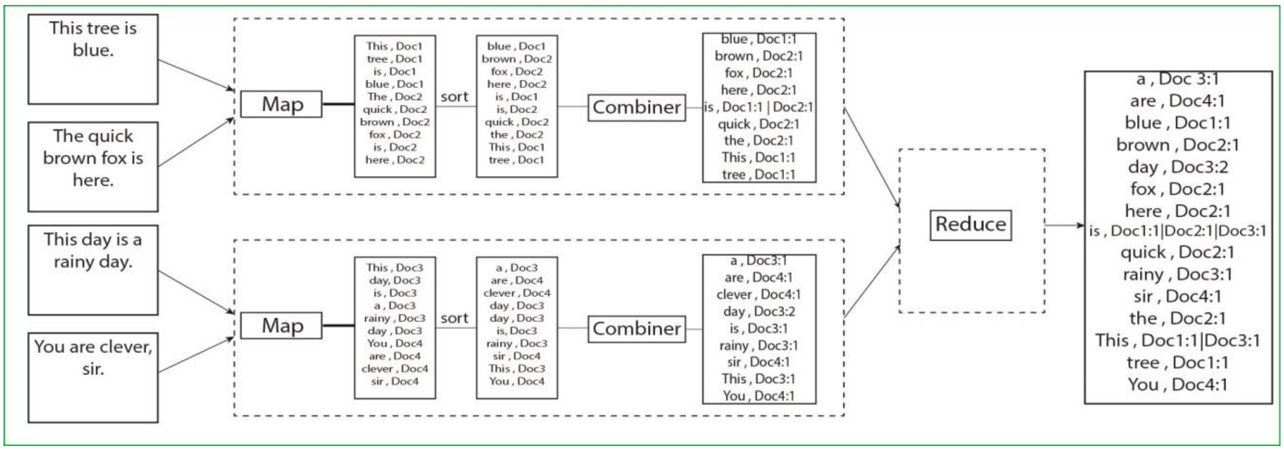


Figure 3. Data Flow

V. COST ANALYSIS

In this section, we attempt to come up with a formula that estimates the total amount of time needed for an indexing job to complete on a cluster of nodes. For the sake of simplicity, a few assumptions must be made first, otherwise the problem becomes too complex and difficult to handle.

First of all, we assume that each computing node has one central processing unit (CPU). All the nodes are connected to the same switch in order to avoid further calculations regarding connections via different routers and switches. Also, each node spawns one Map task and the total number of Reduce tasks is less than then number of total Map tasks. Furthermore, we do not take into account the size of the main memory on each node. This size could make a huge difference on the amount of time needed for a job to execute. That is because, the more data are stored in the main memory (multiple times faster than secondary memory), the less data are requested from the secondary (slow) drive. Hence the execution is faster.

For our scenario, we assume having at our disposal:

- A single computer, that is able to build an Inverted Index for the whole dataset. This can be seen as equivalent to executing one Map followed by one Reduce (at least the functionality of those) in a single node. The total time needed to complete is t_S . In particular, the equivalent of the Map function should take t_M, and the equivalent of the Reduce function should complete in time t_R. Namely:

$$t_S = t_M + t_R$$

Since, in our implementation, Combiners essentially do the same processing as Reducers, we can safely assume that this processing is all done in either of the two, in this case, in the Reducer.

- A cluster that consists of m nodes with the aforementioned configuration. m and r correspond to the total number of Map and Reduce tasks, respectively.

The data at each node is read and processed serially. The dataset is considered already stored in a single disk in the single node setup, and in HDFS in the multi-node setup. We would like to estimate the total amount of time needed to complete a job, given the time needed for a single node to run the same job, t_{S}, and the number of nodes, m, present in the cluster.

During execution, the Hadoop framework breaks the whole dataset in smaller portions of data. Each portion is assigned to a Map node and is being processed. All the Map tasks (one per node) are executed in parallel, hence the total time for Map tasks to complete equals to the execution time of one Map task. The amount of the parallel execution of the Map tasks is

$$\frac{t_M}{m}$$

Upon completion of each Map task, the output becomes an input for the Combiner that is associated with the Map on the same node. As we said earlier, we use Combiners to reduce the data transferred across the network. For the purpose of our scenario we consider the processing time of the Combiners as negligible; as we said earlier, it is some amount of processing that has to be done in either the Combiner or the Reducer, and we can assume we load it up to the Reducer. What makes a huge difference is the amount of data that have to be transferred over the network from the Map tasks to the Reduce tasks, and here is where Combiners come into play.

The output of each Map task has to be transferred to the Reduce tasks. Next we try to get a rough estimate of the amount of data per chunk, *K*, that have to be transferred over the network with and without Combiners. We can assume that for a large number of Mappers it is safe to consider that almost all data coming out of each Map will have to get to a Reducer somewhere else on the net.

Let us assume that each chunk has the default size of 64Mbytes. We take the length of the average English word as 5.1 characters [5]. We should at least add spaces between words. Therefore, the length equals to 5.1+ 1 = 6.1 bytes per word. Also, we presume what we have pure text (no html or other mark-up in between) and the text is in 8-bit ASCII, too. So, if we divide the chunk size with the average size per word, we get around 11,001,453 non-unique words per chunk.

Without Combiners we have to transfer *"Word \t DocId"* approximately 11 million times. Specifically, *Word* is around 5.1 bytes, the tab character is 1 byte and the *DocID* takes 4 bytes assuming it is a Long Integer. Thus, the length per string is 10.1 bytes, and the total transfer is K=105.97 Mbytes per chunk.

With Combiners we need to transfer *"Word \t DocID1:WordFreq1 | DocID2:WordFreq2 ..."* per unique word in a chunk. *Word* is again 5.1 bytes. Each posting is 10 bytes, assuming 4 bytes for docIDs and 4 for frequency (both Long Integers) plus 2 bytes for the delimiters (`\t' and `:' for the first posting and `|' and `:' for all the others.) To complete the calculation, we need to estimate the number of unique words per chunk and the average number of postings per unique word.

We assume that no stop-list, stemmer or any other pre-processing that reduces the length or the number of words/tokens are used.

In linguistics, the number of unique words, *N*, in a text as a function of text length (in words) is given by the empirical law of Heaps [8]. According to Heaps' Law and assuming beta=0.5 (a widely-known empirically found value), *k*=31.6 (we take the middle of the logarithmic range [10,100] using a base of 10---this is the suggested range for *k* in the literature, for English text) and *n*=11,001,453, we estimate *N*=104,812 unique words per chunk.

Zipf's law states that given some corpus of natural language utterances, the frequency of any word is inversely proportional to its rank in the frequency table [9], i.e. the distribution of frequencies follows a power-law with an exponent of 1. It is also known that the distribution of *document frequencies* (DF, i.e. the number of documents a word occurs in) also follows a power-law, albeit with a higher exponent (steeper curve). Empirical data show that the exponent is around *s*=2 [11]. This means that the *i*th largest DF is

$$DF_i = c \cdot \frac{1}{i^2}$$

, where c is some constant. The average document length is around 1,000,000 characters [5], i.e. 0.95367 Mbytes. So, we have 64 Mbytes/0.95367 Mbytes = 67.11 documents per chunk. We assume that the most common word (i=1) appears in all chunk's documents, so c must be 67.11. Now the average DF is

$$\frac{1}{104812} \cdot 67.11 \sum_{i=1}^{104812} \frac{1}{i^2} = 0.001053227$$

Thus, empirical laws produce a very small number meaning that each unique word has almost zero postings; empirically the average is just above 1, which makes sense since most unique words occur in a single document.

Consequently, with Combiners, we have to transfer 104,812 unique words of 6.1 bytes each associated on average with 1 posting of 10 bytes, that is K=1.61 Mbytes in total per chunk. Thus, using Combiners reduces the network load by an impressive 98.5%.

At first sight, we see that the data transferred for a single chunk, when we do not use the Combiner, is around twice the size of data without the use of it. The benefit comes when we take into account the number of the unique and the non-unique words in a chunk. As we calculated earlier, the total data need to be transferred when we use Combiners is K = 1.61 Mbytes = 0.00157 Gbytes = 0.01256 Gigabits.

To sum up, the total data that need to be transferred are:

$$\frac{Dataset\ Size}{Chunk\ Size} \cdot 0.01256 = 0.02512 \cdot Dataset \quad [Gigabits]$$

Now, even if we interconnect the nodes with an Ethernet network of one Gigabit per second, the amount of time needed to move the data across the nodes is significantly more than the time needed to process the same data. This equals to:

$$\frac{0.02512 \cdot Dataset\ Size}{1\ Gbps} = 0.02512 \cdot Dataset \quad [Seconds]$$

As with the case of the Map tasks, the time needed for all the parallel Reduce tasks to complete is:

$$\frac{t_R}{r}$$

To determine the total time for completion of the cluster job, we use the following equation:

$$Time = \frac{t_R}{r} + 0.02512 \cdot Dataset + \frac{t_M}{m} \quad [Seconds]$$

The above equation could be an asset on the process of determining the approximate number of nodes of a cluster, given the execution time on a single node that processes the data serially.

We have to underline here, the tradeoff for the benefits of parallelization, is the big communication cost between the nodes.

I. Conclusions

Map-Reduce is the state of the art for processing Big Data sets. Indexing is a parallelizable problem, so that Map-Reduce is an ideal framework for information retrieval. Since parallelization comes with an amount of overhead, it is crucial to specify when the Big Data are "big" enough in order to use a cluster for the process. Equally important is to be able to determine the approximate number of nodes that consist a cluster, in order to process Big Data at a given time.

References

[1] Anand Rajaraman and Jeffrey David Ullman, “Mining of Massive Datasets”, Cambridge University Press, 2011.

[2] Wikipedia, the free encyclopedia, “*Big Data*”, [Online]. Available: http://en.wikipedia.org/wiki/Bigdataw

[3] Wikipedia, the free encyclopedia, “Apache Hadoop”, [Online]. Available: http://en.wikipedia.org/wiki/Apache_Hadoop

[4] Sanjay Ghemawat, Howard Gobioff and Shunk-Tak Leung, “The Google File System”, Proceedings of the 19th ACM Symposium on Operating Systems Pinciples 2003 (SOSP 2003), Bolton Landing, NY, USA, October 19-22, 2003.

[5] Wolfram Alpha, computational knowledge engine, “Average English word length [Online]. Available: http://www.wolframalpha.com/input/?i=average+english+word+length.

[6] Tom White, “Hadoop: The definitive guide” 2nd editions, O’Reilly

[7] Christopher D. Manning, Prabhakar Raghavan and Hinrich Schutze, “Introduction to Information Retrieval”, Cambridge University Press, 2008.

[8] Heaps, Harold Stanley, “Information Retrieval: Computational and Theoretical Aspects”, Academic Press, 1978. Section 7.5, pp. 206-208.

[9] Wentian Li, “Random Texts Exhibit Zipf’s-Law-Like Word Frequency Distribution”, IEEE Transactions on Information Theory 38 (6): 1842-1845. Doi:10.1109/18.165464. 1992

[10] Jeffrey Dean, Sanjay Ghemawat, “MapReduce: Simplified Data Processing on Large Clusters”, 6th Symposium on Operating System Design and Implementation (OSDI 2004), San Francisco, California, December 6-8, 2004.

[11] Jimmy Lin. “The Curse of Zipf and Limits to Parallelization: A Look at the Stragglers Problem in MapReduce” , Proceedings of the 7th Workshop on Large-Scale Distributed Systems for Information Retrieval (LSDS-IR’09) at SIGIR 2009, July 2009, Boston, Massachusetts.