The Pennsylvania State University

The Graduate School

School of Information Science and Technology

Towards Trusted Computational Services: Result Verification Schemes for MapReduce

A Thesis in

Information Science and Technology

by

Chu Huang

Submitted in Partial Fulfillment
of the Requirements
for the Degree of

Master of Science

December 2011

ABSTRACT

Recent development in Internet-scale data applications and services, combined with the proliferation of cloud computing, has created a new computing model for data intensive computing best characterized by the MapReduce paradigm. The MapReduce computing paradigm, pioneered by Google in its Internet search application, is an architectural and programming model for efficiently processing massive amount of raw unstructured data. With the availability of the open source Hadoop tools, applications built based on the MapReduce computing model are rapidly growing.

This thesis focuses on a unique security concern on the MapReduce architecture given its loosely-coupled computational resources. We study the potential security risks from lazy or malicious servers involved in a MapReduce task. We introduce innovative mechanisms for detecting cheating services under the MapReduce environment based on watermark injection and random sampling methods. Results of these new detection schemes are expected to significantly reduce the cost of verification overhead. Also, we believe that the research reported is an important step towards trusted computational services and will assist in directing avenues for future research. In practical applications, we hope the results reported here will hopefully further promote the secure adoption towards trusted MapReduce services and therefore help to bring profits to MapReduce service providers with increasing number of potential clients.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

ACKNOWLEDGEMENTS

I would like to thank my adviser Professor Sencun Zhu for his help and invaluable advice, co-advisor Professor Dinghao Wu for his general guidance and comments during my work. They both have supported me throughout the thesis with their patience and knowledge. Without their help, this thesis would never have been possible. I also want to thank my committee for their assistance and my fellow graduate students for all of their help and support.

Chapter 1

Introduction

Recently, a new wave of large-scale data processing technologies has emerged in various research and business areas, such as genomics analysis, image archiving, visualization, simulation and business intelligence. Due to the ever increasing demands in those fields, more and more people and organizations have come to realize the increasing needs of computational resources. MapReduce is first proposed by Google in 2004 [2]. It has attracted a lot of attention ever since its development. As a compensation mechanism to make up for individual's lack of computation resources, the MapReduce framework enables huge data processing by dynamically building up the computation environment which consists of a large number of computers. Moreover, the divide-and-concur approach enables the MapReduce model to process the huge input within only a limited period of time. In addition to its benefits in data processing, MapReduce also provides clients services at low cost with flexibility. Based on its underlying cloud computing concept, MapReduce programming model is also built in a pay-as-you-go manner. Service providers can now sell their computation resources as utilities. Meanwhile, clients are also able to process their large dataset in a timely manner by just paying the "computational fees" to achieve a win-win situation, which to some extent frees the clients from the burden of IT services. Considering its benefits, MapReduce has been widely adopted by a number of large companies, such as Google, Yahoo, Amazon, Facebook and AOL[15]. An open source implementation of MapReduce called Hadoop [3], was then developed by Yahoo. The ease of application development using Hadoop further encourages wide adoption of MapReduce.

1.1 Problem Statement and Motivation

Despite its benefits, MapReduce also brings some privacy and security concerns to its clients. Unlike the traditional paradigm, computational services provided by MapReduce applications are running on a number of distributed machines which may scatter in multiple locations. It is this open distributed environment that puts client’s data into potential dangers. MapReduce clients have no control over their data once it gets fed into the distributed applications and many of whom are either facing or concerning about possible privacy threats and security risks, such as sensitive data disclosure and violations of data integrity. For instances, financial data provider would concern about potential data leakage from the service providers to their competitors. Similarly, medical clients also would be worried about intentionally data miscalculation, including both tampering the computation result and providng fake results without actual calculation.

Given that the MapReduce model has emerged only within the recent years, only a limited number of studies have been targeted on its security and privacy issues under the context of MapReduce. To protect data from potential threats, security problems of distributed systems have been studied by many researchers. Although existing approaches usually can achieve nice results with acceptable detection rates, they still have limitations of high computational costs, given that they need to either fully or partially replicate the original data, and the calculation involves in some extra works. The research goal of this thesis is to develop a new verification method for the detection of untrusted MapReduce services with lower computational cost.

1.2 Research Scope

It is obviously impossible to cover all types of security threats under the MapReduce environment in one research. This thesis focuses on only the methods to ensure the MapReduce data processing services, specifically detecting cheating behaviors occurred on both lazy and malicious workers. Besides, given that MapReduce is now heavily adopted by major search engine companies in providing their information retrieval services, the scope of this thesis in particular focuses on cheater detection involved in the word count, word index and page rank tasks. The methods proposed in this thesis can be further generalized beyond these applications .

1.4 Thesis Outline

The rest of the thesis is organized as follows. Chapter 2 provides the background information of MapReduce and its security issues and describes the motivations of this study. Chapter 3 covers a comprehensive literature review, outlining the previous and current related research works. Chapter 4 describes the system model and cheating model of this work. Chapter 5 presents the actual method for cheating detection. Chapter 6 shows the experimental evaluation conducted with simulations and describes the experimental results and analysis in details. Chapter 7 discusses the limitation and conclusion will be drawn.

Chapter 2

Background Information on MapReduce and Security

2.1 A Historical Overview

This section provides a brief overview on the history of computing evolutions in recent decades. By comparing and differentiating a number of related concepts, we have a better understanding of the MapReduce model, which will be further discussed in the following sections.

At the very beginning of this computational age, computers are not as powerful as what we have today. The earliest computer system is characterized as serial processing, given the instructions are executed sequentially, one after another [23]. Serial computing works fine for small problems, whereas when encountered with very large scale computational tasks, it then becomes far less efficient than expected. Increasing transistor density to some extent alleviates this problem. However, it still cannot completely remove the resource limitation of serial computing as people expected.

The advent of multi-core processor announced the end of the serial computing era. As an evolution of the serial computing, parallel computing was first proposed to solve those high-cost problems, which cannot be efficiently performed with serially running computers. Adopting the divide-and-conquer paradigm, parallel computing enables the simultaneous usage of multiple computational resources to perform a single job. By breaking single jobs into multiple parts which can be run concurrently, computational recourses parallelizing has made many complex problems easy to solve. Meanwhile, parallel computing high performance also significantly reduces the time required to perform each task.

Parallel computing was favored in the 80s and early 90s. Started from the late 90s, the cluster architectures became increasingly dominant in the computational field [24]. The term “distributed computing” shares a lot of overlap with “parallel computing,” given that there is no clear distinction between these two concepts for a long period. However, according to Ghosh, Basu and Das [25], comparing to parallel systems, distributed computing adopts distributed memory instead of shared ones when performing computational tasks. So, in that sense, distributed computing systems can be viewed as a loosely-coupled form of parallel computing. Instead of asking one machine to perform all the tasks in parallel using shared memory, distributed systems separate the job and assign them to a number of processors, each with their own private memory [25]. Compared with parallel computing, distributed computing provides more scalable and reliable computing service.

During the recent years, the concept of cloud computing has attracted a lot of attentions. However, it is not very clear when the term cloud computing was first introduced. According to the definition of Foster et al. [26], cloud computing is a pool of “abstracted, virtualized, dynamically-scalable, managed computing power, storage, platforms, and services” which enabling the delivery of services to external internet customers. Given that cloud computing dynamically adopts geographically dispersed computational resources, either physical or virtual, it allows the users to accommodate their computational needs without having to add or remove any infrastructures, personnel, or any other facilities. This is very beneficial because predicting future need is difficult, especially when the demand is changing constantly. Large companies such as Amazon, Google and Yahoo, are using cloud to provide large-scale, cost-effective, reliable and scalable services and resources for public use on the Internet. A large number of services on the infrastructure, platform, and software have been developed and provided by cloud computing providers on a pay-as-you-go manner. Users rent the computation and storage resources they need by simply plugging in to the cloud and are free from the burden of IT service management. For small companies that may not have that much money to purchase resources, setting up data centers and building and maintaining business application, the cloud offers an alternative choice. In this new computing paradigm, a client may choose to outsource its data storage, data processing, or even the whole IT infrastructure to a cloud service provider so that it can focus more on their core business and leave the work to the cloud.

2.2 MapReduce and Hadoop

The emergence of MapReduce in 2007 [2] makes it one of the most popular programming models for cloud computing. By splitting the input files into several sub-pieces, a MapReduce master application starts first by distributing those smaller chunks of input each to a mapper. The mapper is responsible for transforming the input records into some intermediate states by performing the computational tasks on the subset of data and returning a <*key, value*> pairs as the output of the job. In contrast to the mapper, a reducer takes the <*key, value*> pairs and integrate the records into a final output. In that case, MapReduce enables the programmers to achieve their computational goals without needs to know details about the hidden operational complexity of the parallel execution across dispersed computing servers [27]. The flexibility of MapReduce makes it an attractive programming model for large scale parallel computation involving distributed data sets. It has been applied widely in various areas including machine learning, large scale semantic annotation, multi-core programming and other data-intensive applications.

Currently, one of the well-known implementations of MapReduce is the Apach Hadoop developed by Yahoo. According to Yahoo’s own report, its Hadoop platform now covers a total number of 25,000 usable servers, 25 petabytes of application data, and with the largest cluster with 3500 servers [28]. Hundreds of organizations worldwide have reported using Hadoop in their production, including Facebook, Amazon, Last.fm, and the New York Times. Besides its well adoption in the industries, Hadoop has also become the focus for a number of academia researches, being covered recently in a large volume of literatures []. So in that way, Hadoop in some sense is becoming a standard for current MapReduce framework.

The Hadoop platform consists of two main components: Map-Reduce and Hadoop Distributed File System (HDFS). The Map-Reduce component is very straight-forward and is the only task needs to be accomplished by the users. Programs written in functional style are naturally parallelized and fit into the cloud computing environment. The other very important component HDFS is responsible for providing input data storage for the MapReduce framework.

Applications that run on HDFS usually have very large-scale datasets. Thus HDFS is tuned to support large files by providing high aggregate data bandwidth and scale to hundreds of nodes in a single cluster.

2.3 Security Issues in MapReduce

As demonstrated above, MapReduce does have the ability to offer its clients powerful resources for performing their computational services in a cost-effective manner. However, as an integration of many other technologies such as utility computing, distributed computing, and more recently, web services [29], MapReduce suffers from security threats which also happen to many of those technologies. In that sense, security and privacy issues therefore become a major concern that prevents more widely adoption of MapReduce.

Often operated in an open environment, MapReduce faces a number of regular communication threats. With emphasis on wide accessibility, Mapreduce provides flexible and scalable computing services for its clients. However, this also leaves MapReduce vulnerable. Ssensitive data, such as stock exchanges, transaction information, and military data that are transmitted between Mappers and Reduces would be easily exposed to danger. In addition, MapReduce also faces security threats of data integrity, which happens when the attacker compromises a server within the MapReduce framework. Adversary can compromise servers in the MapReduce system and tamper data during the data processing process. Besides the security risks, MapReduce may also encounter privacy threats such as sensitive data disclosure under semi-honest environments.

In addition to all the aforementioned common risks, MapReduce users also have some unique security concerns about lazy or malicious servers involving in a MapReduce task. A lazy server can be defined as computation providers who left their computational tasks undone in economic considerations of eliminating the high computing expenses required. On the other hand, a malicious server can be defined as service providers who intentionally tamper the computation process and return back an incorrect result. In general, two main motivations lie behind this malicious service providing, including attacks by arbitrary purpose and attacks by strategic purpose.

Chapter 3

Related Research Works

Security of distributed systems has been studied by many researchers, but only a few of them really focus on MapReduce. The Airavat system proposed by Roy et al. [11] provides strong security and privacy guarantees for sensitive data computation of MapReduce. This work focuses on protecting privacy issues of untrusted code of data-mining and data-analysis algorithms executed on MapReduce, such as clustering and classification. They deployed mandatory access control (MAC) provided by the operation system SELinux to prevent data leakage via files, sockets and other storage channels. To prevent the output of computation leaks the sensitive information about the inputs, they enforced differential privacy by adding exponentially distributed noise to the output of the computation. Although Airavat demonstrates strong security and privacy guarantee, it has some limitations. It cannot confine malicious mappers leaking information through its output keys, which provides a channel for data leakage. Wei et al. [10] proposed a secure scheme called secureMR which aims at protecting the computation integrity issue of Mapreduce. They detect misbehavior of mappers by sending same tasks to multiple mappers, and check consistency of results. If inconsistency is found, master is able to identify malicious mappers. SecureMR decentralized the replica verification among reducers that participate in the computation. Thus, reducers are trusted according to their assumption. Existing security frameworks for MapReduce have the same limitation: reducers and master need to be trusted. This assumption might not be practical in real world.

Several results checking techniques have been proposed in different areas other than MapReduce. Some of these schemes are specialized and they have constrained on particular types of computation. Golle and Mironov [14] proposed a Ringer scheme which is specialized. It ensures client that most of the work are done correctly with high probability and protect against coalitions of lazy cheaters. They focus on the computations involving inversion of a one-way function (IOWF) which is to find a pre-image x_0 of a distinguished value $f(x_0)$ under a one-way function $f: D \rightarrow R$. Their solution is to select ringers which are randomly chosen elements of corresponding subdomain and the participants need to find out all pre-images that maps to those pre-computed result (ringers). They used game theoretic arguments to measure the efficacy of their strategies. Different levels of detection assurance could be achieved by varying the number of ringers in each subdomain. But this scheme is restricted to tasks that involve the IOWF and cannot be used to address generic computation problems. Szajda et al.[21] extended this ringer scheme, and presented a probabilistic verification scheme to prevent against malicious behavior in grid computing. They provided two strategies for non-sequential and sequential computations, respectively. But this work still cannot be extended to all general computations.

A generic verification scheme based on redundancy was presented by Golle and Sutbblebine [17]. They described a security framework for commercial distributed computing by simple task redundancy. Their strategy relies on probabilistic redundant verification to guarantee protection against collusion. Szajda et al. [20] presented another redundancy-based strategy. This work requires fewer resources compared to Golle and Sutbblebine's. It can achieve the same effective cheating detection rate.

Du et al. [12] proposed a different framework of commitment-based sampling scheme. To detect cheating behavior, the supervisor randomly selected and verified tasks executed by the participant. Specifically, before computing a task x_k, a participant makes a commitment for $f(x_k)$ to prevent the participant from only correctly computing the results for those sampled after it learns which inputs are samples. This scheme requires a host to maintain a huge Merkel hash tree to serve the verification purpose. The leaf nodes of the tree are commitments for all data. This scheme can be used for general computation in grid computing. However, it cannot guarantee the correctness of the final result for the grid application. Sarmenta [16] presents a spot-checking technique to address the problem of protecting volunteer computing systems from malicious participant submitting erroneous results. He integrated the idea of majority voting which efficiently limits the direct use of redundancy. In spot-checking, master randomly assigns worker a task with correct result is already known. Straightforwardly, if the result from worker cannot match with the correct result, bad worker will be caught. Task scheduling by supervisor is based on credibility of each worker. The more spot-checks it passes, the higher credibility a worker will get. Correspondingly, bad worker will be blacklisted by supervisor. However, this scheme is not practical since in real world supervisor has no prior knowledge about the percentage of malicious workers in the system. Moreover, it still cannot guarantee the correctness of the final result. Another generic approach was proposed by Zhao et al. [13]. This result verification scheme is called Quiz for peer-to-peer grid computing. The basic idea of this scheme is to inject indistinguishable tasks into a job with other normal tasks. The verifier only checks the correctness of all quiz results. If all results of quizzes are valid, they assume results for real tasks are correct.

Another work presented by Monrose, Wyckoff, and Rubin [22] addressed the problem of cheating participants in distributed environment do not perform the tasks it was assigned. Instead of verifying the correctness of the result, they deal with this problem by checking whether a participant honestly executes the code via a challenge-response protocol. They provided a remote audit mechanism to trace the execution of tasks by recording checkable state point of program execution. Specifically, participants send the result and proof of execution to a verifier. This verifier checks the state points by running a portion of the execution. One deficiency of this method is the need to re-compute result.

Most of the existing studies are based on replication technique that tasks are duplicated and sent to two different participants to process. If the result is consistent, the computation is accepted and the participants are assumed to be honest with high probability. If inconsistency is detected, then one of the two participants must be cheater. Such redundant task waste resources in the system, and is lack of efficient way to detect collude participant. Other result verification schemes are restricted to certain types of computation and not feasible in the context of MapReduce. We provide a scheme based on watermark injection and weighted sampling technique. Our scheme can efficiently detect cheating behavior of MapReduce computing in the context of text processing problem.

Chapter 4

System Model

In this chapter, we formally define the problem of uncheatable computation using MapReduce.

4.1 MapReduce Programming Model

The MapReudce framework consists of a single master JobTracker and more than one slave Task Trackers. Master is responsible for task assignment, scheduling and management (i.e., it queries the distributed file system about the location of data blocks and assigns each task to the TaskTracker) and re-executes any failed tasks. A Slave that contributes to the computation resources is also called worker in this model. Workers execute the tasks as directed by Master. A typical MapReduce system will include a single master and multiple slave nodes (which we will call worker in the rest of this thesis). Besides master and workers, the other important entity in MapReduce system is the distributed file system. Typically, a file in HDFS is gigabytes to terabytes in size. Thus HDFS provides high aggregate data bandwidth and can be scaled to hundreds of nodes in a single cluster and support tens of millions of files in a single instance. Our work will follow the Hadoop implementation of MapReduce.

The process of MapReduce data processing can be divided into two phases: a map phase and a reduce phase. In the beginning of the map phase, input data is first sliced into *n* splits. Then master assigns these splits to different workers for paralleled processing. Workers on the map phase are called mappers. In this phase, each mapper processes one split, produces intermediate result and outputs intermediate result in the *(key, value)* key-value pair format to its local disk. The intermediate results from each mapper are then partitioned into *m* parts by a partition function. In the reduce phase workers read partitioned intermediate result from mapper in the preceding map phase, process it independently, and merge all intermediate result associated with the same key into the final result. Worker runs the reduce function are called reducer correspondingly. What to follow for reducers are storing final result to their local disks. Eventually, the final result will be stored in the distributed file system. Note that all data is processed on key-value pairs.

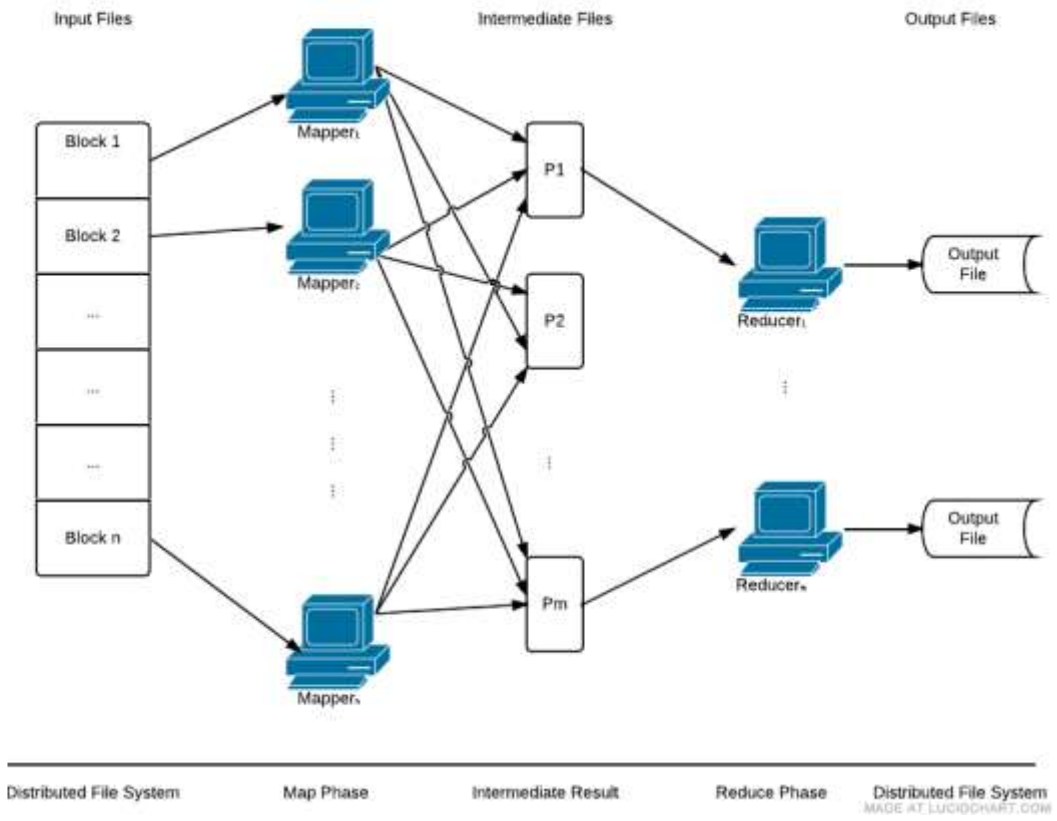


Figure **4-1**: The MapReduce processing model.

The MapReduce programming model is illustrated in Figure 4-1. As in this figure, the input file is divided into *n* blocks. Then master assigns the *n* blocks b_1, b_2, …, b_n to *n* different mappers. After processing the block, a mapper generates the intermediate result and writes the result on its local disk. The intermediate result is then partitioned into *m* parts using the partition function. The number of reducer should be the same number of partitions, which is *m* in this case. After receiving the notification of complement of the map task from the master, reducer reads data from the local disk of the mappers. For example, reduce R_1 reads partition p_1 of the intermediate result from each mapper. Similarly, reducer R_2 reads partition p_2 of the intermediate result from each mapper. When all the tasks have been completed, the reducer finally output the final result to distributed file system.

4.2 Model of Cheaters

In MapReduce computation model, we consider an attack model in which the adversary is rational and economically motivated. We define a cheater as worker who either not performs the computation it was assigned to or does its job following the specification but manipulates the final result. In our context, cheating is meaningful only if it successfully saves up computation resources or manipulates results without being detected. We classify cheating behaviors using two different adversary models. We assume the input domain for MapReduce is *U*, and the task for MapReduce is to compute $F: f_2(f_1(x))$ for all the input domain, where f_1 is map function while f_2 is reduce function.

- Lazy Cheating Model: In this model, the cheating worker follows the master's computation with two exceptions: 1) it will either drop a task at any point before finishing the task. The extreme case is that the cheaters directly drop the task without performing any computation and return random assign value. For the first case, we assume that every machine in the computation system is possible to cheat on the task with the same possibility p_c. 2) for every $x \in \check{U} \subset U$, MapReduce uses $G: g_2(g_1(x))$ as the result instead of $F: f_2(f_1(x))$ and function *G* is usually much less expensive than function *F*. The goal of this kind of cheaters is to save computation resources and maximize their profit by performing more tasks in the same period of time. Defending against the lazy cheating worker is the main purpose of our verification scheme.

- Malicious Cheating Model: the behavior of the malicious worker is arbitrary. For any specific $x \in \check{U} \subset U$, instead of returning $F(x) = (k, v)$, it would manipulate the final result and return (k', v'). In other words, the worker intentionally returns wrong results to client. This type of attack can be further divide into two categories: arbitrary attacker and strategic attacker. For arbitrary cheating model, cheaters intentionally returns wrong result for the purpose of disrupting the computation. For strategic cheating model, adversary falsifies only a tiny little part of the result with a purpose (e.g., improve its website rank for search engines), and therefore is difficult to identify.

Below we illustrate a few scenarios where our verification scheme would be used for cheating detections in the calculation of PageRank scores. In scenario 1, we assume that in order to maximize its profit gain, MapReduce performs the task using other ranking algorithm such as performing indegree ranking which is much less expensive than PageRank, and returns wrong PageRank scores. While in scenario 2, we assume a merchandiser boosts its traffic by increasing its PageRank value during the calculation process.

Scenario 1

Given the high performance provided by the cloud infrastructure, AliceSearch, which is a big search engine company, decided to pay for the cloud resources to conduct their calculation of PageRank scores, while BobCloud is a company which provides computation resources to AliceSearch for computing PageRank using MapReduce. In order to save up computation resources and gain greater profit from AliceSearch, BobCloud ranks pages by their indegree and assigns reasonable ranking scores for each page. By miscalculating the pagerank values, BobCloud returns the wrong PageRank values to AliceSearch. With our cheating detection mechanism, AliceSearch scanned through part of the returned results and detected the miscalculation and prevented the disruption from returning the unrelated results to the end users.

Scenario 2

A small retailer company called HackShop wants to increase its annual sales by drawing more traffic to its website from the search engine returned pages. Through various channels, HackShop buys off one accompany called TrudyCom to intrude MapReduce system and manipulates final PageRank results the AliceSearch. Assuming the attack is successful, HackShop increases the PageRank score of its own page. HackShop assumes AliceSearch wouldn’t sense such miscalculation given the huge amount of web pages. However, with our verification scheme, AliceSearch catches such cheating and prevents returning low quality pages to the search engine users.

The main risks for AliceSearch in both two scenarios is the fact that the computation provider is intrinsic untrustworthy. It might either be driven by benefits or compromised by attackers therefore messes up the final results. Such risks can put AliceSearch in a commercial and reputational disadvantage.

Our primary goal is to defend against these two types of cheating workers. We assume a lazy worker probably not start processing its assigned task in the beginning but at any point of the file and process it sequentially, or stop anywhere before the task finishes, or directly drop the task without computation. Based on this assumption, we provide a secure scheme to prevent the computation provider from cheating the client by claiming that they have done the job that they actually did not, and also to protect from malicious behavior of MapReduce. Note that a primary requirement is that the verification overhead must take much less time than the outsourced computation.

Chapter 5

Result Verification Schemes for MapReduce

5.1 Watermark Injection

Watermark injection scheme is an application-specific verification scheme. The basic idea of watermark injection is to inject indistinguishable watermark into normal documents with verifiable results only known to the client, who has conducted the watermark injection. A watermark is actually a normal element in the task document. Task executers are unable to distinguish watermark within the document when they perform the computation. Using the watermark injection scheme, the client can decide whether to accept or reject the returned task results from task executers on the basis of the correctness of the injected items. We assume that if the marks we injected are all computed correctly, the workers have executed the tasks honestly. Otherwise, workers, one or many, have not conducted the task honestly. Thus, this watermark injection method realizes the verification purpose without overloading the clients.

We outline this watermark injection scheme as follows.

- **Setup phase (Watermark Injection)**: Before handing the data to the computation provider, the verifier pre-processes the input files by inserting a number of watermarks into the task documents.
- **Computation**: MapReduce performs computation tasks on the documents and return final results to the client.
- **Verification and Recovery phase**: To ensure that MapReduce has honestly executed the task, the client verifies the correctness of the results and accepts the result if no

inconsistency is found. Then it recovers the injected documents and removes the impact of the watermark. Otherwise, the client discards the results.

- **Security and accuracy**: Because marks are randomly injected and cannot be distinguished from other data items in the task documents, it is infeasible for MapReduce to identify them in an easy manner. Consequently, if some task executer cheats on a task that contains marks we injected, corresponding computation result will be inconsistent with what we pre-computed. Provided that the verifier has injected enough amounts of watermarks, this approach can detect malicious cheating behavior of Mapreduce with high probability.

Figure 5-1 shows the process of this watermark injection scheme.

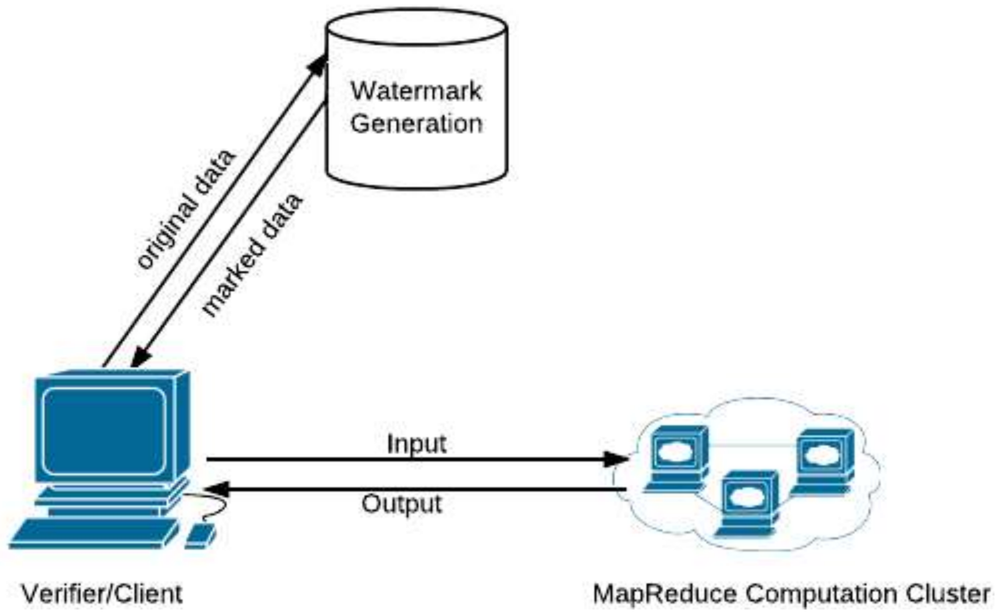


Figure **5-1**: Watermark-based verification scheme. A watermark generation module pre-processes raw data and injects watermarks. Upon received output from MapReduce, the verifier performs verification. If the computation from MapReduce is accepted, the verifier will call a recovery module to remove the effect introduced by injecting watermarks.

An advantage of the watermark injection scheme is that the client does only little injection work once per job. The percentage of documents to be injected and the percentage of words to be modified do not need to be big to maintain the high probability of cheating behavior detection. Next section introduces the application of the watermarked injection scheme to detect cheating behaviors while using MapReduce for text processing problems.

5.1.1 Text Processing Problem

Many computation tasks are now delivered to cloud computing systems, such as MapReduce, to be performed. MapReduce has been implemented by Google and Yahoo now. Anyone can rent a cluster from computation service providers for large data processing. The text processing field can also take advantage of MapReduce for complex computations. The watermark injection scheme we have introduced above can be adopted to detect cheating behaviors. In this section, we focus on the applying the scheme to two problems: word frequency and inverted index.

5.1.1.1 Word Frequency

Word frequency plays an important role in internet search and information retrieval. It counts the frequency of word *w* within a collection of documents. From practical point of view, word frequency provides researchers with useful data to understand the structure of underlying documents. It is commonly used in semantic analysis, information retrieval and nature language processing.

Suppose we are interested in performing a computation task on a large set of documents and counting all occurring words using MapReduce in the cloud. Then the problem can be defined as: given a document, workers need to build a list of distinct words with their frequency.

In MapReduce computation model, the mapper processes each input document. By scanning each document, the occurring word is stored as an intermediate key with a value of 1, emitting a new pairs of *(w, 1)*. When all mappers are done, MapReduce gathers the emitted pairs and passes the intermediate results to the reducer who sums the frequencies for each word and outputs the results. Figure 5-2 displays an example of the MapReduce counting process.

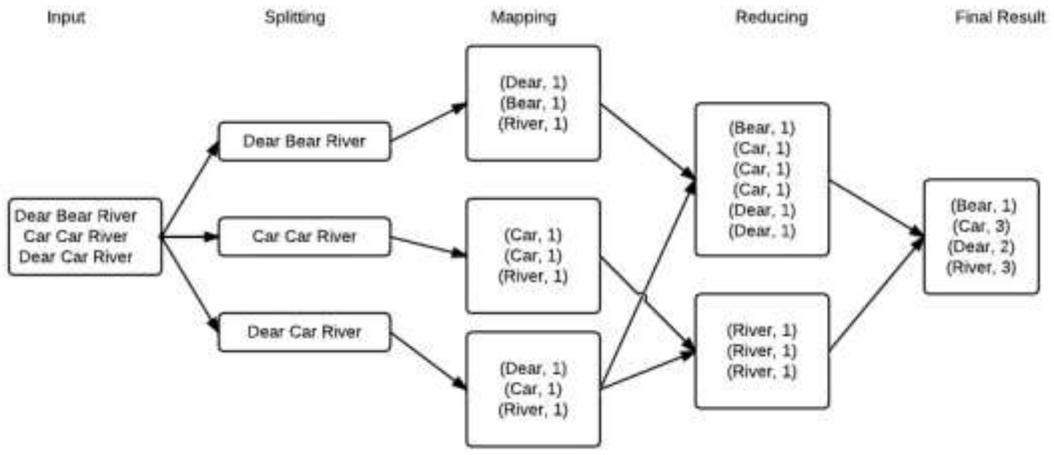


Figure **5-2:** MapReduce working process of word frequency

MapReduce allows the programmer to express the word count algorithm in a straight forward manner. We use the following algorithm[10], and implement the computation in Java.

Algorithm 1: Pseudo-code of word frequency

```
Procedure Map
Input: docid: unique  id of documents; path: document location
Output: key-value piar

  for all term w  doc d do
        emit (term w, count 1)

Procedure Reduce
Input: word; counts list: list of count 1for a word
Output: word, total counts
  sum ← 0
```

```
for all count c counts[$c_1$, $c_2$, ...] do
sum $\leftarrow$ sum+c
emit (term w, counts s)
```

Security issues are serious concerns for cloud computing. When using MapReduce, we should be aware that computation providers may cheat for their own benefit. One aspect is that the computation is on a web-scale collection that contains billions of words. The large size may incentivize computation service provider to cheat for saving both storage and computational resources. One way to cheat is to skip the computation and return wrong (randomly picked) result from its own vocabulary. For strategic cheaters, they might only pick certain words and manipulate their corresponding frequency.

To detect and hence prevent cheating phenomenon, we introduce the watermark injection scheme. Let $D = \{D_1, D_2, \ldots, D_k\}$ be the input corpora. In addition, let $d = \{d_1, d_2, \ldots, d_{k'}\}$ be a collection of watermark files. We outline the watermark generation process as follows:

- First, the verifier randomly selects *k'* input documents $d_1, d_2, \ldots, d_{k'}$ among all the input documents before handing over to mapreduce
- Second, for each selected documents d_i *(i=1 2, …, k')*, verifier conducts a simple alphabet substitution on every ε words. A substitution table *T* should be pre-defined by the verifier. The verifier also needs to store the information about injection distance ε, which word has been encoded. It should also count the frequency of every encoded word.

The substitution approach is similar to the process of performing a simple Caesar cipher. Watermark injection is done by encoding words on the files. Only the client knows which files are marked and MapReduce cannot distinguish them from other files. The percentage of marked files among all input files is $p_b = k'/k$, a security parameter in our scheme and set by client. The verifiable computation scheme for word count is performed as follow:

1. Before handing over the data to computation provider, the verifier pre-processes the data and inserts watermarks by encoding every ε words of randomly selected files, according to substitution table *T*. Meanwhile, the verifier stores information about which words are encoded and counts all encoded words.
2. Computation provider counts the frequency of word using MapReduce. After completing the computation, it returns the final results to the client
3. The verifier checks the results from MapReduce by 1) whether all encoded words are included in the result from MapReduce; 2) comparing the frequency of each distinct encoded word with the pre-computed frequency of all selected documents. If two results match, the verifier accepts the result form MapReduce. Otherwise, the verifier concludes MapReduce has cheated and discards the results.
4. If the result if accepted, verifier needs to recover these words that have been "marked." This recovery process is the reversion of Caesar cipher. If a recovered word *w'* is the match word *w*, its associated counting value is added to the corresponding match word's frequency.

Figure 5-3 shows MapReduce computation process after watermark injection, corresponding to the MapReduce task shown in Figure 5-2.

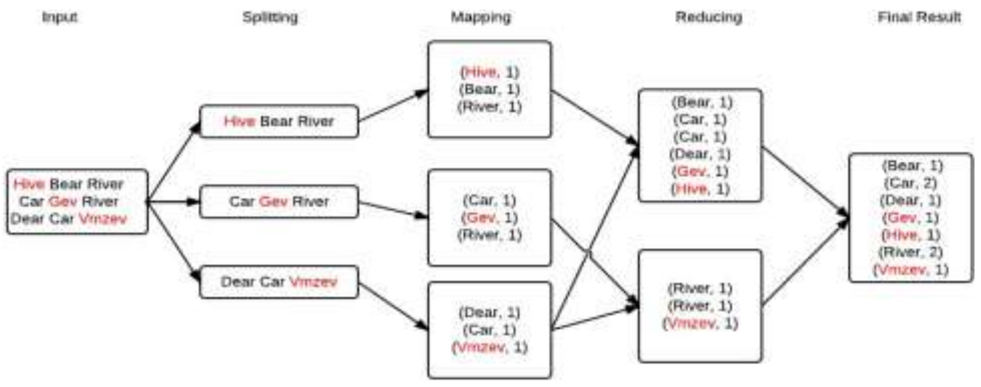


Figure **5-3**: Watermark-based injection scheme for word frequency with *injection distance = 3* and *key of substitution table= 3*. Therefore *Dear* is encoded to *Hiew*, *Car* is encoded to *Gev*, and *River* is encoded to *Vmzev*.

Algorithm 2: Pseudo-code for verification and recovery

```
Input: $_1$: list of encoded words with frequency; $_2$: result from MapReduce
Output: pass or fail
  i ← 0
  j ← 0
  z ← 0
temp ← new String

  for each  i  from 1 to length($_1$)
  If (!( j = search($_1$[i], $_2$))) then
        return fail
 else if ($_2$[j].value ==$_1$[i].value)
        temp = recover($_2$[j])
        if (z = search( temp, $_2$))  then
              $_2$[z].value += $_2$[j].value
                    remove($_2$[j])
              else
              replace(temp, $_2$[j])

   else
        return fail
  sort($_2$)
  return pass
```

Security analysis: The security of the watermark injection scheme is based on the assumption that workers of MapReduce cannot determine whether the input file has been marked or not. The fixed number of positions for alphabet shifting is the *key* and should be kept by the client for recovering purpose. For example, word “watermark” with a *key = 3* would be encoded “aexivqevo”. To decode “aexivaevo” after verification, verifier needs to reverse it back to “watermark”. It is not necessarily to use Caeser cipher for watermark injection; any encoding method can be used. One requirement is that words after encoding needs to be unique from other words in real world and there is no collision between two different encoded words. Otherwise, there may exist false positive (Type I error) during verification. Therefore, if Caeser cipher is not secure enough, hash functions can be used.

Suppose MapReduce already get to know our scheme and think of a way to escape from being detected. The only way to escape detection is to correctly compute all encoded words. To determine whether a word is actually encoded or not, lazy workers of MapReduce needs to perform brute force search over all words in its dictionary (assuming it contains all words in real world). The computation overhead of this process is much heavier than that of frequency counting. If the purpose of the cheating workers were to save computation resource, they would lose the incentive to cheat. If lazy workers do not process the data, or give wrong results by random guess, counting of encoded words would be inconsistent with the pre-computed value. Misbehavior is going to be caught with high probability as long as they ever cheat on the computation.

The marked files should not be too dense. Otherwise, it will introduce too much extra computation overhead to the computation system. The verifier needs to perform a heavy task on the verification and recovery as well. On the other hand, in order to achieve high detection rate, the number of the marked files needs to be sufficiently large.

From the verifier’s point of view, lazy cheating behavior or any other arbitrary malicious behavior will lead to similar consequences---part of the returned results is corrupted. However, the detection rate of the watermark injection scheme for word frequency problems would vary on two different situations. If a worker (mapper or reducer) cheats by not conducting the task, we cannot know which part is cheated and suspect the honesty of the whole MapReduce system. The probability that a worker cheats on average affects the detection rate. In another case, whole MapReduce system has honestly executed all the computation tasks but some machines have been attacked by adversary from outside. Portion of machines in MapReduce are compromised. If the purpose of attacker is to modify the values associated with certain keys rather than arbitrarily mess up computation result, it is very hard for the watermark verification scheme to detect.

To convince readers the high accuracy of the watermark injection scheme for word frequency problems, a brief theoretical derivation on a simplified scenario. Let $D = \{D_1, D_2, \ldots, D_n\}$ represent the set of documents to be sent to computation providers. We randomly select p_1 percent of documents, np_1, to inject watermarks. In each document, we encode p_2 percentage of words to do the watermark injection. If we assume that each document contains roughly the same number of words, *w*, the total number of watermarked word would be np_1wp_2. Suppose we have *m* mappers and *r* reducers in a MapReduce system. The client (verifier) divides all *n* task documents evenly and distributes them to the *m* mappers. Thus each mapper gets *n/m* documents. As long as $p_1 > 0$, we can guarantee that there is at least one mapper which gets watermarked documents. Usually we will have more mappers receiving these documents. When one mapper decides to cheat, unless this mapper does not receive any watermarked documents, the cheating will be detected. Thus, the probability of this situation is

$$Prob(Not\ detected \mid One\ mapper\ cheats) = (1 - p_1)^{n/m}$$

$$Prob(Detected \mid One\ mapper\ cheats) = 1 - (1 - p_1)^{n/m}$$

A similar analysis should be applied to the reducer step. The *nw* words are randomly assigned to *r* reducers and each reducer receives on average *nw/r* words. The probability of that a word is watermarked is p_1p_2. When one reducer decides to cheat, it escapes detection only if this reducer does not receive any watermarked words. Thus, the probability of this situation is

$$Prob(Not\ detected \mid One\ reducer\ cheats) = (1 - p_1p_2)^{n*w/r}$$

$$Prob(Detected \mid One\ reducer\ cheats) = 1 - (1 - p_1p_2)^{n*w/r}$$

When there are more mappers or more reducers cheating, the probability of detection rate is even higher. Say, x mappers and y reducers cheat, the probability of detection is

$$Prob(Detected|x,y) = 1 - (1-p_1)^{x*\frac{n}{m}}(1-p_1p_2)^{y*n*\frac{w}{r}}$$

Let's denote $(1-p_1)^{x*\frac{n}{m}}(1-p_1p_2)^{y*n*\frac{w}{r}}$ as $f(x,y)$, describing the probability that the verification scheme fails when x mappers and y reducers have cheated. The overall conditional probability of detection, given that at least one mapper or one reducer cheats, can be calculated. Assume each mapper and each reducer may cheat with a fixed probability, p_m and p_d, respectively. The probability that x mappers and y reducers cheat is

$$Prob(x,y) = \frac{m!}{x!\,(m-x)!}\,(1-p_m)^{m-x}p_m^x\,\frac{r!}{y!\,(r-y)!}(1-p_r)^{r-y}p_r^y$$

Thus, the conditional probability of detection is

$$Prob(Detected \mid Cheating) = \sum_{[0,m]}^{x}\sum_{[0,r]}^{y} Prob(x,y)Prob(Detected|x,y)$$

Using what we have,

$$Prob(Detected \mid Cheating) = 1 - \sum_{[0,m]}^{x}\sum_{[0,r]}^{y} Prob(x,y)f(x,y)$$

Thus,

$$Prob(Detected \mid Cheating)$$
$$= 1 - \left(1-p_m+p_m(1-p_1)^{\frac{n}{m}}\right)^m \left(1-p_r+p_r(1-p_1p_2)^{n*w/r}\right)^r$$

Let's suppose $m = 15, p_m = 0.10$ and $r = 15, p_r = 0.10$ and look at the detection rate of this verification system. Assume $n = 1500$ and $w = 1500$ and we can plot the detection rate vs. $p1$ and $p2$. Before proceeding, we examine the equation first. We have $\frac{n}{m} = 100$ and $(1-p_1)^{100} = 0.9^{100}$ that is almost 0. Similarly, we have $(1-p_1p_2)^{n*w/r}{\sim}0$. Thus, the equation is approximately:

$$Prob(Detected \mid Cheating) = 1-(1-p_m)^m(1-p_r)^r.$$

This means that when each mapper or each reducer has to deal with a large number (say, 100) of documents or words and the system cheats, the probability that the verifier detects this cheating is independent of the percentage of the watermarked files/words (as long as they are not too small). For the simple scenario here, we have $m = 15, p_m = 0.10$ and $r = 15, p_r = 0.10$. Consequently,

$$Prob(Detected \mid Cheating) = 0.9576$$

When there are a large number of documents to be processed, using a large p_1 is not practical and we may need to use small values such as $p_1 = 0.01$ and because $0.99^{100} = 0.366$, we cannot ignore $p_m(1-p_1)^{\frac{n}{m}}$ in the equation. Using MatLab, we obtain the following results for small values of p_1 and p_2.

In Section 6, we provide the results from a computational simulation for this verification system.

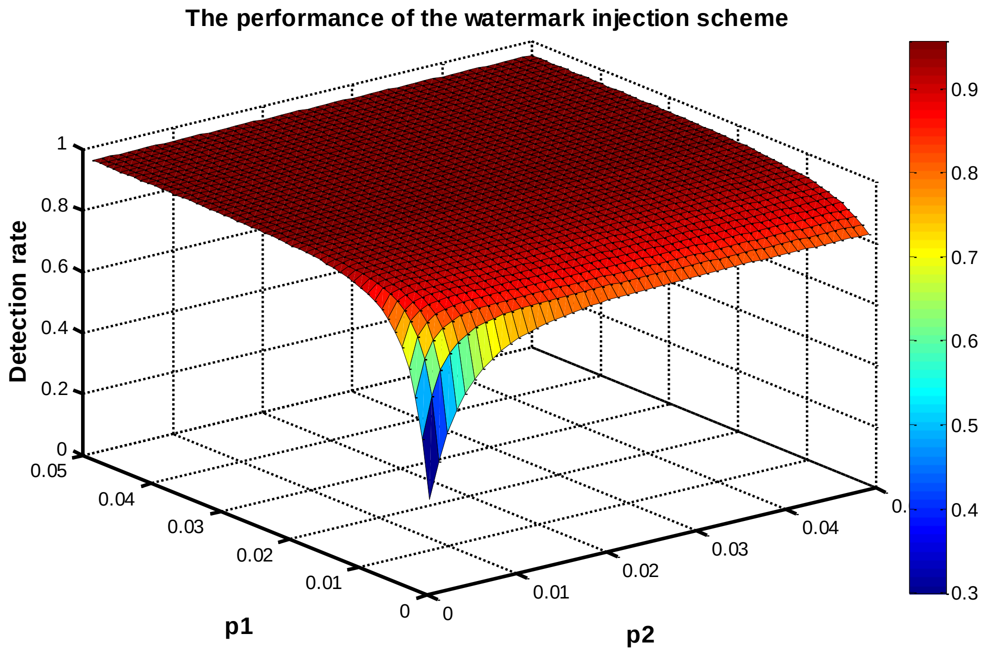


Figure **5-4**. The performance of the watermark injection scheme. p_1 is the percentage of documents selected for watermark injection, p_2 is the percentage of encoded words in those p1 documents. The detection rate is close to 0.9576 when p_1 and p_2 are large.

When no one cheats in the MapReduce system, the verifier will accept the computation results for sure. Thus, the false positive (Type I error) probability is *0*. Table 5-1 shows the effectiveness of the watermark injection scheme for the word frequency problem. The two cells represent the probability that the verifier rejects or accepts the computation results from the MapReduce system, when the system cheats or does not cheat. As shown, the accuracy is fairly high and the Type I error rate is *0*.

Table **5-1**: The effectiveness of the watermark injection scheme for the word frequency problem.

	The MapReduce system cheats	The MapReduce system does not cheat
The verifier rejects the results	$1-(1-p_m)^m(1-p_r)^r$	0
The verifier accepts the results	$(1-p_m)^m(1-p_r)^r$	1

Computation overhead: The extra computation cost of the watermark scheme includes four parts. (1) The verifier performs a simple encoding (through letter shifting) on a percentage of words in the randomly selected documentes. (2) The verifier counts the frequency of those encoded words within those documentes. (3) The verifier checks if the returned results are consistent with the pre-computed results of the watermarked words. (4) The verifier recovers the original words from the watermarked words in the computation results.

5.1.1.2 Inverted Index

Consider the case that a user wants to find all documents that contain certain keyword on a laptop. A na ïve (and heuristic) searching scans each ducument on the disk, if there is no intelligent index. Usually, a laptop stores thousands of documents and this na ïve scanning will take many minutes to finish. On the other hand, consider a real world web-scale search. Although over billions of documents exist on the web, a good search engine only takes less than a second to return valid results. The silver bullet that makes the search engine works so fast is an efficient data structure called inverted index. An inverted index is a data structure that maps terms to their locations (documents containing the terms). Using this approach, a search engine easily identifies the appropriate documents for terms, without scanning each document in the whole web. Many modern search engines on the web use an inverted index scheme for data storage.

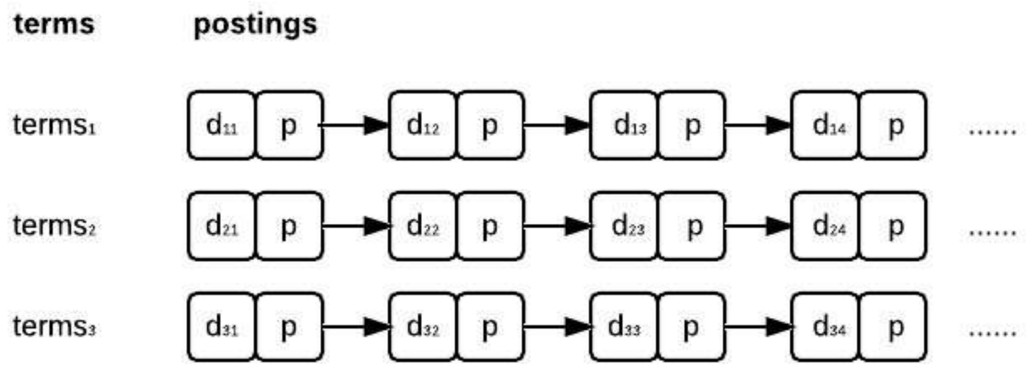


Figure **5-5**: Structure of inverted index

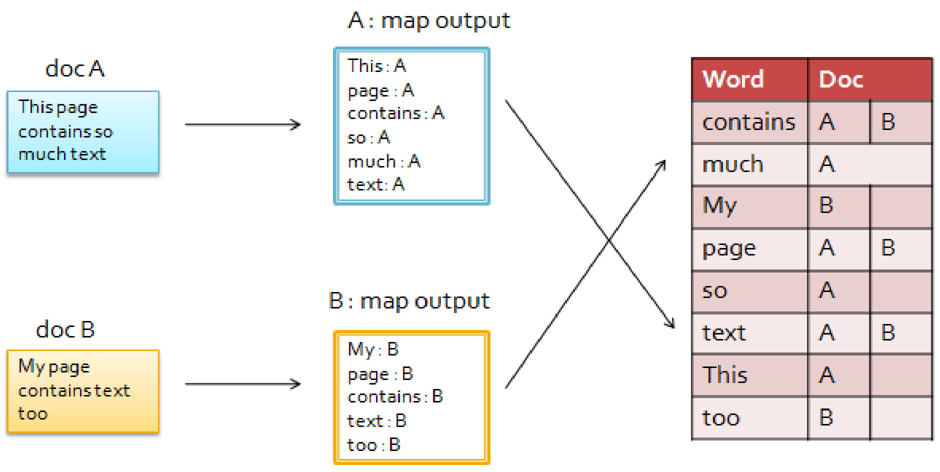
Typically, an inverted-index construction consists of two lists: a list of terms and its associated postings lists. Postings list consists of individual postings. Each posting has two components: a unique document *id* and a piece of payload information, which describes (1) at what position(s) in the document the term appears and (2) the frequency (number of appearance) of the term in the document. The position information can be expressed as the location within a sentence, section, or paragraph. Some systems do not record the payload information. Figure 6 shows the structure of an inverted-index construction.

Building such a construction is not easy. MapReduce can be utilized for this task and we can again use the watermark injection scheme for security insurance. A formal description of the problem is: given a collection of documents, use MapReduce to produce an index that maps each term in the corpora to a list of documents that contain the term. Note that each term in the list appears only once (typically terms may appear more than once in a document). For example, consider the following two short documents:

Doc A: This page contains so much text

Doc B: My page contains text too

An inverted index for these three documents is given in figure 5-6:



Word	Doc	
contains	A	B
much	A	
My	B	
page	A	B
so	A	
text	A	B
This	A	
too	B	

Figure **5-6**: Process of building inverted index using MapReduce

Suppose that we have the task to construct an inverted index without payload information. We decide to use MapReduce for this task. At the very beginning, we need to define what a “term” is and what a “document” is. In this case, we consider a term as a single word and ignore phrases. A document can be a book, a chapter of a book, an individual html document or a portion of a webpage. The granularity of ducuments does not matter. MapReduce will divide input corpora into smaller chunks (e.g. 64M or 128M) and pass them to mappers in parallel. Given a collection of documents, mappers extract individual terms from the documents and identify distinct terms. For each term, mappers output *<term, doc-id>* pairs which the term as the key and the document *id* as the value. These pairs are passed to reducers and reducers read the pairs, sort them by term and merge the results. The reducers build the posting list for each term. The following pseudo-code, Algorithm 3, shows the basic inverted index approach.

Algorithm 3: Pseudo-code of inverted index

```
Procedure Map
Input: docid n: unique  id of documents; path: document location
Output: key-value piar

  for all term t  doc d do
      emit (term t, posting<n>)


Procedure Reduce
Input: term t; postings<$n_1$, $n_2$, …>
Output: term, posting
  P ← new LIST
  for all posting a  postings[$n_1$, $n_2$, ...] do
      append(P, a)
      sort(P)
  emit (term t, postings P)
```

The watermark injecting approach for inverted index construction is similar to what we use for the word frequency problem. Let $D = \{D_1, D_2, \ldots, D_k\}$ be a collection of documents being indexed. To inject watermarks into the task, this scheme first randomly chooses a certain number of documents $d = \{d_1, d_2, \ldots, d_{k'}\}$ from all input documents where $k' < k$. The verifier then injects watermarks by encoding a percentage of words in each selected document, and records the document *id* and the corresponding words. A more formal description is as follows.

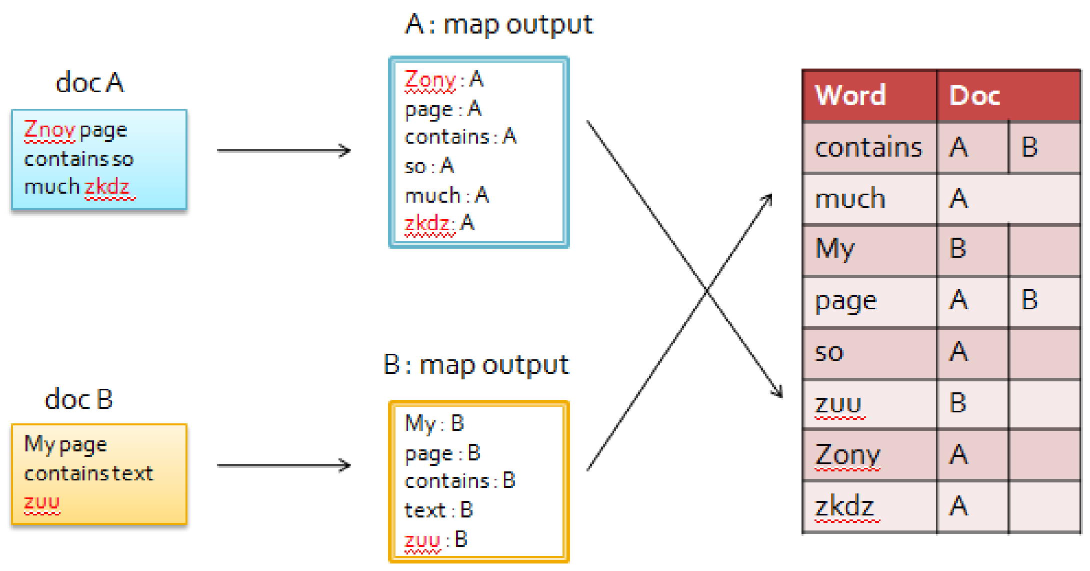
- The verifier randomly selects *k'* documents $d_1, d_2, \ldots, d_{k'}$ from input documents.
- For each selected documents, the verifier conducts an alphabet substitution on every ε words through the document through each documents d_i $(i=1, \ldots, k')$ according to a substitution table *T*, records every encoded words and their corresponding document *id*. The verifier needs to store the information about injection distance, encoded words and corresponding document *id*.
- For each encoded term in document collection $d = \{d_1, d_2, \ldots, d_{k'}\}$, verifier builds a posting list. The computation overhead for building a small inverted index *I'* for all encoded terms is far lower than building a complete inverted index I $(I' \epsilon I)$.

Given security parameter ε, *k'*, only the client knows which documents have been chosen and which words have been "marked". Task executers cannot distinguish them from other normal words or documents. The verifier then carries out the following steps to cloud compute the task and verify the result.

1. The verifier pre-processes the data before uploading it to MapReduce. It injects watermarks into input files and constructed an index *I'* of encoded (watermark) words
2. MapReduce performs the task of constructing an inverted index *I* and returns *I* to client
3. The verifier examines the results from MapReduce, identifies those posting associated with the watermarked words and compares them with the pre-computed result *I'*. Specifically, the verifier first checks whether all encoded terms appearing in the dictionary of *I'* are all included in the dictionary of *I*. Second, the verifier compares the posting list of each term in *I'* with the results from the MapReduce system. If both steps

return a matching conclusion, the verifier accepts the results. Otherwise, the verifier considers a cheating behavior has happened and rejects the results.

4. If no cheating is detected, the verifier needs to recover the accepted results. In order to do so, the verifier needs to decode those words based on the same substitution table *T*. The decoding process is the reversion of encoding in which alphabet shift back *x* positions. For each term in the index *I'*: if corresponding document *id* id_x in its postings lists of *I*, the doing nothing; otherwise, inserted a posting list with id_x as document id into index *I*.



Word	Doc	
contains	A	B
much	A	
My	B	
page	A	B
so	A	
zuu	B	
Zony	A	
zkdz	A	

Figure **5-7**: Watermark-based injection for inverted index with *injection distance = 4* and *key of substitution table= 5*. Therefore *This* is encoded to *Zony*, *text* is encoded to *zkdz*, and *too* is encoded to *zuu*.

Algorithm 4: Pseudo-code of verification and recovery for inverted index

---

```
Input: : index of encoded words; : index from MapReduce
Output: pass or fail
  i ← 0
  j ← 0
```

```
z ← 0
temp ← new String

  for each  i   from  1 to length(I')
  If ( j = search(I'[i], I)) then
          return fail
else if (match(I'[i].postinglist, I[j].postinglist))
          temp = recover(I[j])
          if (z = search( temp, I)) then
                  append(I'[i].postinglist, I[z].postinglist)
                          remove(I[j])
                  else
                  replace(temp, [j])
                          distinct_insert_sort(I'[i].postinglist, I[z].postinglist)
             else
          return fail
  return pass
```

Security analysis: Similar to the word frequency scheme, security guarantee of this scheme relies on the fact that lazy workers have no way to tell marks from normal words. But verifier on client side is able to identify the "marks" in the final result and verify it for consistency.

The misbehavior of lazy workers will cause losing of terms or links from certain terms to documents that contain them. Some words are so common especially the words "the", "a", "and", "of", "in", "for" … etc. Usually these words need to be filtered out and will not appear in the index. For simplicity, we treat them as normal words. Even after removing all these "noisy" words, in real world it is very likely that a word appears more than once within a document. Therefore we need to "mark" words to make them identifiable in the result from MapReduce.

At the end of the computation, the client only needs to check whether the mapping is correct based on *I'*. Misbehavior is detected when inconsistency is found during the verification process. Similar to the previous scheme, as long as the number of marks is large enough, we will detect cheating behavior with high probability. But at the same time, we want to keep it within a range so that it will not introduce too much computation overhead. Detection rate and computation over head of this scheme is the same as the previous one. So we will not repeat it here.

5.2 Random Sampling Scheme for PageRank

5.2.1 PageRank

PageRank is a well know link analysis algorithm[6,7]. In general, it calculates the probability that each webpage is visited by a casual web surfer who randomly clicks links to visit on the Internet. Consider the whole Web as a directed graph, in which directed links from node *A* to node *B* indicates that webpage *A* has a link directing to webpage *B*. If some webpage is can be linked from a great number of webpages, i.e., the in-degree is very high, the probability that this webpage gets visited is very high. If a webpage, *W*, has a lot of links itself, i.e., the out-degree is high, each link on *W* has an equal probability to be clicked if the casual web surfer is visiting *W* now. Thus, the value of *W* to any of its out-nodes (the nodes *W* is pointing to) is equal and is not significant because the total value (say, 1) is divided into a lot of independent parts. In reality, if many webpages all point to one single page, it is highly possible that that this single page plays an important role in the system, such as the home directory of an organization and the contact page. PageRank considers these facts and uses an interation algorithm to calculate a numerical value for each webpage to represent its "importance" – the probability that it will be visited by a random Web surfer. The theory of ranking introduced by Page and Brin[6,7] is based on the aforementioned thought that the most important pages on the Internet are the pages with the most links leading to them. A link from page *u* to page *v* suggests that page *v* is important. If we think of link as a vote, page *u* links to page *v* is like casting a vote to *v*. Because Web designers tend to make links on their webpage to direct to relevant contents, useful resources, such as fundamental applications, contact information, and help pages, usually have a high in-degree value. The algorithm also puts a weight on the contribution of page *u* to page *v*, giving that *u* links to *v*.

Simply speaking, if a webpage has hundreds of links on it, it does not provide much useful help to users to identify important resources or information. Moreover, just like a recommendation system, recommendations from authorities are more convincing than that from the lay people. Links on a “good” webpage tend to point to good webpages as well. Thus, the weight of this contribution depends on the number of out-links of this webpage and the “significance’’ value of this webpage.

In the following, I first introduce the representation of graphs and some PageRank models.

5.2.1.1 Graph representations

A graph consists of two elements: the vertices and edges. Vertices usually refer as nodes and edges are also named links or connections. A graph can be called a network, representing the relations between items (nodes). The relations can be directed or undirected. Formally, we define a graph $G = (V, E)$, in which $V = \{n_1, n_2, \ldots n_k\}$ is the set of all nodes, and $E = \{(n_{i1}, n_{j2}), (n_{i2}, n_{j2}), \ldots, (n_{is}, n_{js})\}$ is the set of all relations. In the simple example shown in Figure 5-8, it is a directed graph, and we have:

$V = \{n_1, n_2, n_3, n_4, n_5\}$

$E = \{(n_1, n_5), (n_2, n_1), (n_2, n_5), (n_3, n_2), (n_3, n_5), (n_4, n_1), (n_4, n_3), (n_5, n_2), (n_5, n_4)\}$

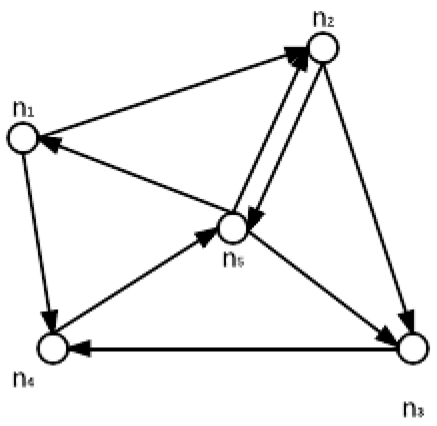
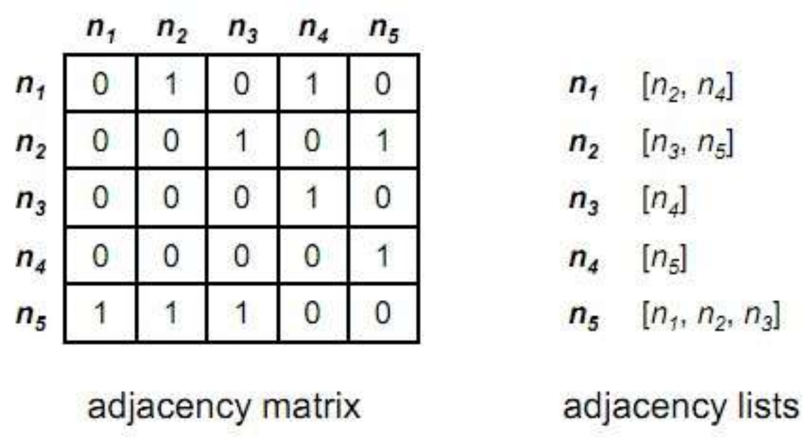


Figure **5-8**: A simple example of directed graph

One way to represent a graph is using an adjacency matrix and the matrix elements describes the relations. For a graph with *n* vertices, adjacency matrix of this graph is an *n* x *n* matrix *A* with rows and columns labeled by vertices. $a_{i,j} = 1$in the matrix indicates an edge from node n_i to node n_j while $a_{i,j} = 0$ indicates there is no edge from n_i to n_j. According to this definition, the adjacency matrix for an undirected graph is symmetric in the sense that $a_{i,j} = a_{j,i}$. For the graph shown in Figure 7, we should have a 5*5 matrix. Using 1 and 0 to represent the relations, we obtain the following matrix (Figure 8 Left).



	n_1	n_2	n_3	n_4	n_5
n_1	0	1	0	1	0
n_2	0	0	1	0	1
n_3	0	0	0	1	0
n_4	0	0	0	0	1
n_5	1	1	1	0	0

adjacency matrix

n_1	$[n_2, n_4]$
n_2	$[n_3, n_5]$
n_3	$[n_4]$
n_4	$[n_5]$
n_5	$[n_1, n_2, n_3]$

adjacency lists

Figure **5-9**: Example of adjacency matrix and adjacency list

The right part of Figure 8 shows an adjacency list, another graph representation manner, which corresponds to the same graph *represented by the matrix.* . In this list, the left column shows each node in the network, and the right column provides a set of nodes that the node in the left column points to. Sometimes, we use numeric numbers not only 0 and 1 to represent a graph in the adjacency matrix. The numbers can be the weight or other properties of the corresponding link,

5.2.1.2 PageRank Algorithm

The web pages and hyperlinks on the Web can be modeled as nodes and edges in a directed graph *G = (V, E)*. I introduce a basic PageRank model here. Let N be the total number of web pages and use 1, 2, …, N to index these N web pages. Let *L(u)* be the number of out-links from page *u* where *u=1,…, N*. The corresponding adjacency matrix is defined as follows:

$$A_{u,v} = \begin{cases} \frac{1}{L(u)} & \text{if there is an edge from u to v} \\ 0 & \text{otherwise} \end{cases}$$

Users should note that the value of A{u,v} can be any number between 0 and 1 in this case. The reason of this definition will become clear in the following explanations. The PageRank value for a page *v* is given as follows [6]:

$$PR(v) = \frac{1-d}{N} + d \sum_{u \in L(v)}^{n} \frac{PR(u)}{L(u)} \qquad (1)$$

Thus, the PageRank value of one webpage depends on the PageRank value of all its in-nodes, as well as their out-degrees. The damping factor [7] d in the equation is used to represent the probability that a Web surfer follows the random behavior pattern (randomly select a hyperlink on the webpage to click). Correspondingly, *1-d* is the probability that a web surfer will go somewhere else. The value of *d* is usually set to 0.85.

This equation does not provide a direct solution to the whole graph. It only provides the relationships among the PageRank values of the nodes. To retrieve a stable solution for the PageRank value of all webpages, we can run an iteration process.

To initialize, we set the *PR(i)* value to be $\frac{1}{N}$ for $i = 1, 2, \dots, N$. Using the equation, we can update the PageRank value for all the N documents and this is one iteration of the process. We keep running iterations until stationary values are obtained or only small differences exist between the results from adjacent iterations. Figure 5-10 shows two iterations of this PageRank model, corresponding to the graph shown in Figure 5-8.

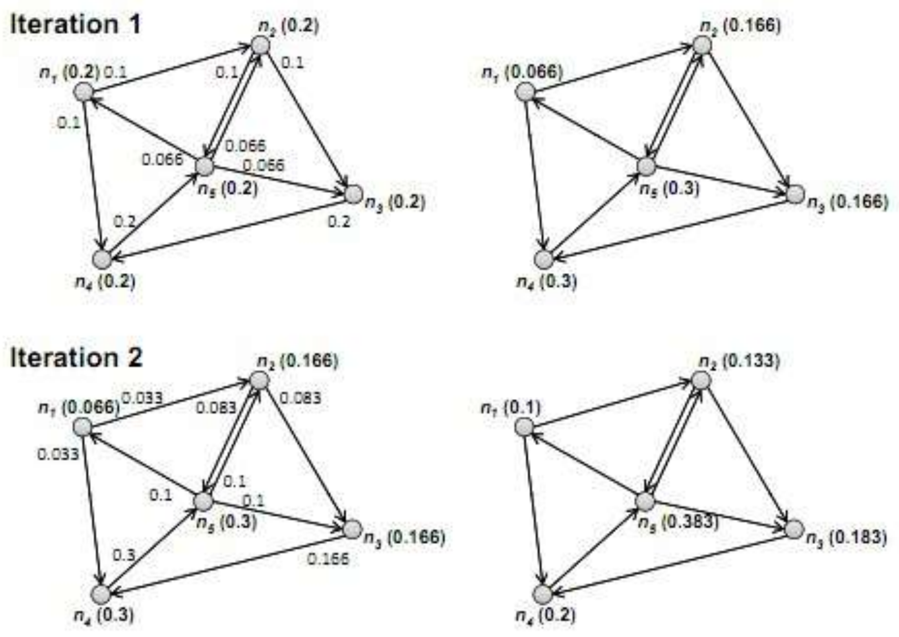


Figure **5-10**: Two iterations of the PageRank algorithm, corresponding to the graph shown in Figure **5-8**.

As we can see, during one iteration, each node passes its PageRank value to its nearest neighbors it links to. Since we could think of PageRank as a probability distribution over all web pages in the system, the iteration is the process for every page to distribute its own value via outgoing links to pages it connects to. Each page also receives contributions from all its in-nodes. The iteration stops when the values do not change any more. The consequences from distributing and receiving cancel each other out. Note that in the rest of our work, we will use the term “node” and “page” interchangeable.

5.2.2 Ranking pages with MapReduce

In the real world, the number of web pages could be very large and it is still growing exponentially. According to Hansen [8], the size of the World Wide Web contains at least 13.99 billion indexed pages. Such a worldwide graph contains billions of nodes and several billions of links. Although an adjacency matrix representing the graph is easy to manipulate, especially for some tasks such as linear algebra. It is not practical for the Web network. First, if the number of web pages is *n*, the space requirement for graph using adjacency matrix is $O(n^2)$. If we assume that each URL takes 0.5KB to store, the whole Internet, containing trillions of unique URLs, will take 400TB space. Calculating the PageRank values using the aforementioned model on such a big data set is not feasible. MapReduce provides a solution to tackle this kind of scaling challenges in a highly distributed manner.

Suppose a client rent MapReduce from computation provider to process a large size of web pages collection for finding a way to characterizing structure of the web graph and computing the importance of every web page using classic PageRank algorithm. The problem of PageRank calculation cannot be done by a single MapReduce job. This task needs multiple map and reduce phase to accomplish: the first job is initiation. The second job performs iteration of calculating pagerank score. The third job is gathering outlinks for pages. Since values needed for an iteration of a page only depend on previous page rank of its inlinks, every task sent to workers of MapReduce is independent and can be execute in parallel.

In the first module of url parser, map function is an identity function. It reads the file with the format as *fromPage* $\rightarrow$ *toPage*, mapper would output key-value pairs: *(1.html, 2.html), (1.html, 4.html), …* . Then reduce appends all oulinks from a document to a string, also needs to assign an initial value $\frac{1}{N}$ and output *(1.html, (<0.2> 2.html, 4.html))*. For the second module of pagerank iteration, mapper takes pairs from last mapreduce module as input. For each outlink of one page, mapper ouput id of that page as the key with its pagerank score and number of oulinks as values. For example, suppose the input from the last step is *(1.html, (<0.2> 2.html, 4.html>)),* mapper produces *(2.html, (1.html, <0.2>, 2))* and *(4.html, (1.html, <0.2>, 2))* and passes the intermediate result to reducer. Reducer then gathers all inlinks for a particular id and their corresponding pagerank score and number of outlinks, and computes a new pagerank score for that page. The output format of reduce is the same as the first module. This iteration will continue until convergence. The stopping criteria of iteration is that |prNew – prOid| < ε. After the second module completes its job, the last step of MapReduce is to sort all the pages according their pagerank score. The algorithm for PageRank iteration using MapReduce is given as follows:

Algorithm 5: Pseudo-code of pagerank iteration

```
Procedure Map
Input: nid n: unique id of nodes; node N
Output: key-value piar

  P ← N.pagerank/length(N.adjacencylist)
  emit (nid n, N)
  for all nodeid m ∈ N.adjacencylist do
  emit(nid m, p)

Procedure Reduce
Input: nid n: unique id of nodes; pagerank mass from inlink nodes [$p_1$, $p_2$, ...]
Output: nid m; pagerank score with m's adjacencylist
  M ← ∅
  for all p ∈ counts[$p_1$, $p_2$, ...] do
          if isNode(p) then
                  M ← p
          Else
                  s ← s+p
          M.pagerank ← s
  emit (nid m, node M)
```

We still need some sort of verification mechanism to ensure that we know when the MapReduce system has cheated and hence we can reject false results. The watermark injection scheme does not work very well under the context of PageRank calculation. Neither marking existing pages nor injecting new pages could make results verifiable. The reason is that in order to do the verification, we need to pre-calculate the PageRank values for these marked or injected pages. However, doing this calculation means we need to do the calculation for the whole data set. If so, using MapReduce to distribute the complex task is not meaningful anymore. In this paper, I propose another approach for the problem: verification by sampling. After the verifier receives results from the MapReduce system, he can randomly sample a small set of pages and check if the PageRank equation is satisfied. If not, the verifier has enough reason to believe that the results is not completely right. I will show that this approach has a high detection rate for cheating behavior in the later sections. I provide two sampling template for cheating detection for PageRank. The easiest one is the naive sampling. Although it is very simple, it still provides a good detection rate. The second method I propose here is called the weighted sampling, which improves the naive sampling to a great extent.

SCHEME I: Na ïve Random Sampling

In this na ïve random sampling scheme, the verifier randomly selects pages from the whole data set with an equal probability for each page. Using the PageRank scores returned from MapReduce, the verifier checks if the PageRank equation holds. The PageRank values of the in-nodes of the sampled nodes and the out-degree of these in-nodes need to be collected as well, according to the equation. This approach works because of the underline complexity of the PageRank task. Computation providers cannot possibly “guess” or make a reasonable solution that holds the equation for all the nodes in the network. Thus, under any type of cheating models, there must be a great number of webpages on which the equation does not hold. Thus, the detection rate of this naive sampling verification scheme is almost 1.

However, there might be cases when computation providers intentionally modify the PageRank value for certain webpages for their personal sakes. The percentage of these modified pages may take only a small portion of the whole data set. Consequently, the naive sampling approach cannot guarantee a high detection rate. Thus, I propose the weighted sampling scheme to tackle this concern. After the explanation of the method, I will theoretically compare the effectiveness (detection rate) of these two sampling schemes.

SCHEME II: In-degree Weighted Sampling

When we select one page and check if Equation (1) holds for this page, we are actually checking this page as well as all its in-nodes. If one of its in-nodes is modified, the equation won't hold and hence we detect the cheating behavior. On the basis of this fact, when we select a page that has a high in-degree, we are actually checking a number of pages in the set. Thus, an intuitive suggestion is to sample those pages with high in-degrees as much as possible. In order to prevent the computation providers figure out our verification approach, we should add a randomness degree to the sampling. In the in-degree weighted scheme, we add a weight that is proportional to *(in-degree + 1)* to each page and adopt a weighted sampling method. Thus, pages with higher in-degrees have a higher probability to be selected. The reason we have a *(+1)* here is to "help" those pages with no in-nodes out – if we don't add 1 on the weight, it is impossible for them to be selected and tested. The effect of this weighted algorithm is that the verifier is actually checking more nodes while conducting the same number of calculations (the verifier only checks if Equal 1 holds for the selected node). Thus, the detection rate, in intuition, should increase.

Security analysis of the two sampling schemes:

First, when computation providers cheat in a lazy manner, i.e., they skip the calculation for a number of pages, or they generate random numbers for the final results, or they use a approximate ranking algorithm instead of PageRank, Equation (1) won't hold for most pages in the data set. Both the naive sampling and the weighted sampling will provide us a good detection rate *(~100%)*.

Second, when computation providers intentionally change the value of a small portion of pages, such as increasing the PageRank value of their own pages and decreasing the value of their opponents' pages, there might be only a small number of nodes for which Equation (1) does not hold. Let's look at the effectiveness of the two sampling schemes here. Assume *N* is the total number of pages and *er* is the error rate – the percentage of pages who do not satisfy Equation (1). The verifier selects *M* pages by one of the two sampling schemes. As mentioned, the in-degree values of these *M* pages affect the verification effectiveness. Let's use *f(x)* to describe the in-degree distribution of all pages in the data set and *x* in the following equations refers to the in-degree value. Use *c* to represent the detection rate. When we only check one page whose in-degree is *x*, we are actually checking this node and all the x in-nodes. Thus, the detection rate is $g(x) = 1 - (1 - er)^{x+1}$. The overall detection rate should be the expectation value of $g(x)$, with respects to the distribution of *x*. Thus,

$$c(Naive) = E\left(1 - (1 - er)^{M(1+x)}\right) = \int f(x)\left(1 - (1 - er)^{M(1+x)}\right)dx$$

According to Jamakovic and Uhlig [30], in a regulatory network, the in-degrees follows follow an exponential distribution. Let's assume that x follows an exponential distribution, we have,

$$f(x) = Kexp(-Kx), x \in [0, \infty)$$

Substituting *f(x)* into the detection rate equation, we obtain,

$$c(Naive) = 1 - \frac{K(1-e)^M}{1 - (1-e)^M e^K}$$

when $(1-e)^M e^K < 1$, which can be easily satisfied.

For the in-degree weighted sampling algorithm, instead of $f(x)$, we should have $(x+$ $1f(x)/A$ in the integration. $(x+1)/A$ represents the effect of the weights and $A=\int x+1fxdx$ is only a normalization coefficient. Thus,

$$c(Weighted) = E\left(1-(1-e)^{M(1+x)}\right) = \int (x+1)f(x)\left(1-(1-e)^{M(1+x)}\right)dx/A$$

For the same exponential distribution of *x*, we obtain,

$$c(Weighted) = 1 - \frac{K(1-e)^M}{(1-(1-e)^M e^K)^2 A}$$

The relationship between *A* and *K* is that

$$A = 1 + \frac{1}{K}$$

For the Internet, Jamakovic and Uhlig [30] concluded that in-degrees of the Internet follow a power law distribution $f(x) = ax^{-\gamma}, x \in [0, \infty)$. Under this circumstance, we can conduct a similar derivation and obtain the following results.

$$c(Naive) = \int \left(1-(1-er)^{M(1+x)}\right)f(x)dx = 1-(1-er)^M a \int [(1-er)^M]x \cdot x^{-\gamma}dx$$

$$c(Weighted) = \int \left(1-(1-er)^{M(1+x)}\right)xf(x)/Adx$$

$$= 1-(1-er)^M a/A \int [(1-er)^M]x \cdot x^{1-\gamma}dx$$

Table 5-2 shows the effectiveness of the weighted sampling scheme for the PageRank calculation problem. If MapReduce has not conducted any cheating behavior, the returned results will certainly pass the verification process and be accepted. Thus, the false positive probability (Type I error) is 0. However, the sampling verification scheme has a probability to fail on revealing MapReduce's cheating behavior, as calculated before. 1 minus the detection rate is the false negative probability (Type II error).

Table **5-2**: The effectiveness of the weighted sampling scheme for PageRank.

	The MapReduce system cheats	The MapReduce system does not cheat
The verifier rejects the results	$1 - \frac{K(1-er)^M}{(1-(1-er)^M er^K)^2 A}$	0
The verifier accepts the results	$\frac{K(1-er)^M}{(1-(1-er)^M er^K)^2 A}$	1

Let's set K = ¼ (the average in-degree number is 4) and e = 0.05. Let's increase M from 10 to 100 and compare the two schemes under this simple scenario. Figure 5-11 shows the plots for the detection rates. Apparently, the weighted scheme outperforms the naive one.

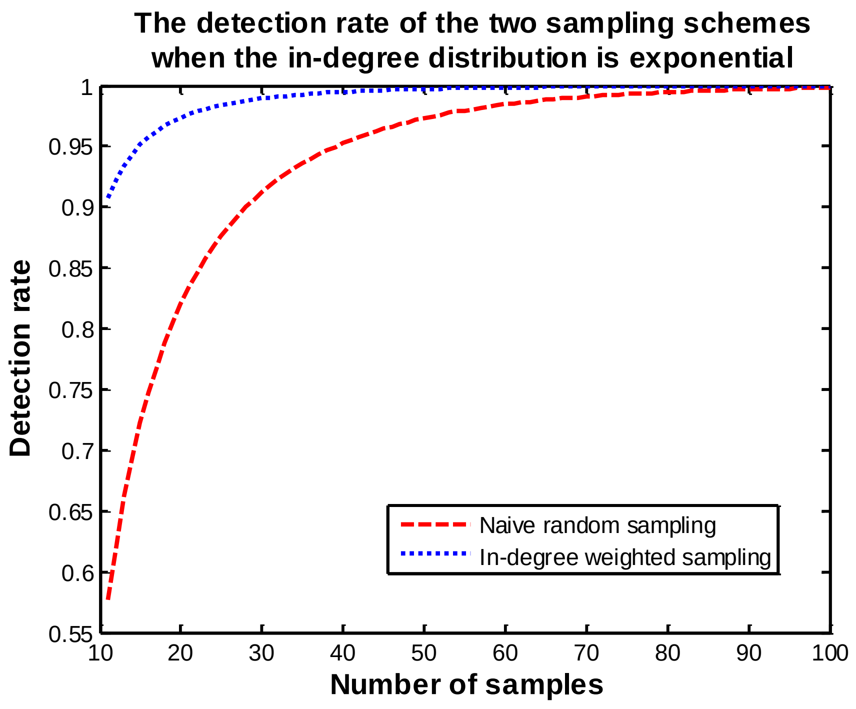


Figure **5-11**: The comparison of the two sampling schemes, given that the in-degree values of all pages in the data set follow an exponential distribution. In this plot, we assume the average in-degree value is 4 and the error rate in the returned results is 0.05. As shown, the weighted sampling scheme outperforms the naive sampling.

In the next section, we will introduce the results from computational simulations on the two applications: (1) the watermark injection approach in text processing problems, and (2) the sampling approach on the PageRank problem.

Chapter 6

Experimental Evaluation

In order to evaluate the effectiveness of our verification scheme, a prototype of watermark injection and random sampling verification schemes is implemented to test the overall method performance. Experiments conducted in this research aims to mainly measure the detection rate under different types of cheating models

6.1 Experiment Setup

To configure Hadoop, parameters on a set of configuration files need to be set. In Hadoop system, configuration is driven by two types of files: default configuration file and site-specific configuration file. Hadoop framework is controlled by these configuration files. Different configurations will affect the performance of Hadoop significantly [8]. Since this experiment is not focus on optimization of Hadoop configuration, I just use the default configuration of Hadoop.

Limited by resource condition, the experiment is executed on a single machine with a multi-node distributed Hadoop cluster where each Hadoop daemon runs in a separate process, instead of a fully-distributed cluster. Specifically, we ran the experiment on a cluster that consisted of 10 virtual machines. This host machine was running on 2.67GHz Intel Intel(R) Dual-Core(TM) i7 processor. Each virtual machine had 256MB of memory and 20GB disk and was installed Ubuntu Linux 10.04.2, Sun JDK 6 and Hadoop 0.20.2. we conducted our experiment using Hadoop WordCount application and PageRank calculation based on Google PageRank algorithm.

6.2 Performance Analysis

6.2.1 Watermark Injection

Considering the two parameters set for the word count verification scheme, 3D surf plot is used to better demonstrate the distributions of the experimental detection rates along with the percentage of watermark injected documents and words. Topographic colors are used in the 3D plot as to better distinguish different detection levels.

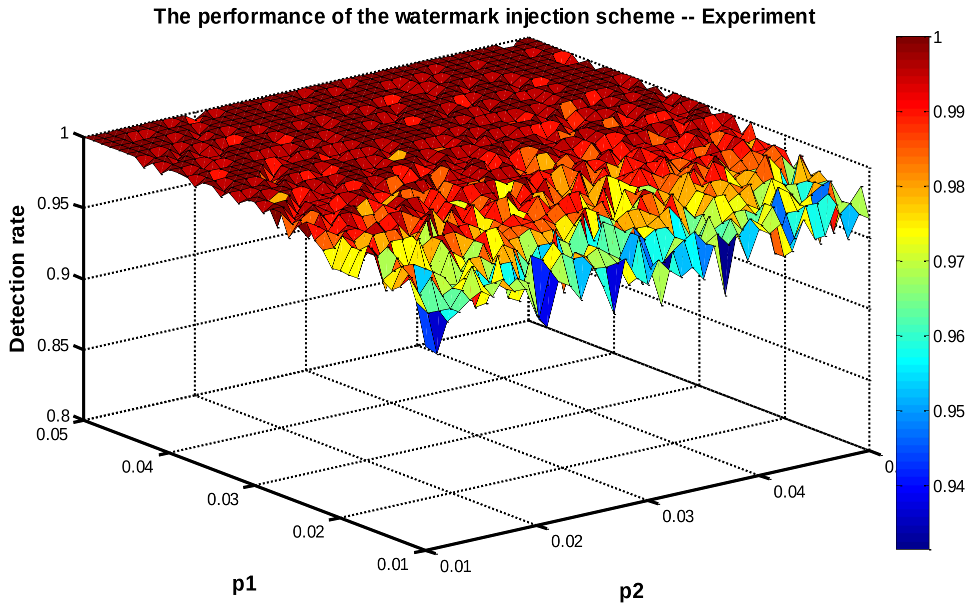


Figure **6-1**: The performance of watermark injection scheme

From figure **6-1** one can see that the watermark injection method performs effectively on detecting the cheatings occurred on the MapReduce word count tasks. Even the lowest detection rate returned in this experiment is larger than 93%. Besides, results of this watermark injection approach under the MapReduce environment are consistent with our theoretical derivations as shown in the last chapter.

As can be seen from figure **6-1**, every data point can be viewed as a test conducted with settings of varied watermark injection proportions within documents and words. Given that 200 simulations are run for each of those tests, mean values are calculated as the data points as shown in the figure. Consistent with the conceptual modeling, p_1 in this case is the percentage of documents with injected watermarks, ranging from 1% to 5%. In the same way, p_2, also with the same range values, indicates the proportion of words within each document which are encoded with watermarks. One can interpret from the above figure that among all the 1681 results points, only less than 0.5% of them have a detection rate less than 95%. Moreover, more than half of the simulations have their detection rates larger than 99%.

Besides, when considering p_1 and p_2's effect on the overall detection rates, we further find that, compared with p_1, p_2 seems to have non-significant impact on the experimental results. It seems that detection rates do not change very much along with the variances in the percentage of watermark injected words. However, if viewing across the p_1 axis, one can notice that along with the growth of the watermark injected document numbers, detection rates of cheating behaviors begin to increase until the injection percentage reaches 3%. In other words, simulations showed relatively lower detection rate when the watermark injected document proportions are less than 3%. Beyond that injection percentage, detection rates begin to reach about 100%.

6.2.2 Random Sampling

In this section, the performance of PageRank cheating detection based on both naive and weighted random sampling is presented. In order to simulate the cheating behavior of the malicious and lazy workers, intentional miscalculations of PageRank values were injected into the final evaluation dataset at an error rate of 1% to 5% respectively. The error detection results of both the naive and weighted random sampling are then extracted and compared with each other.

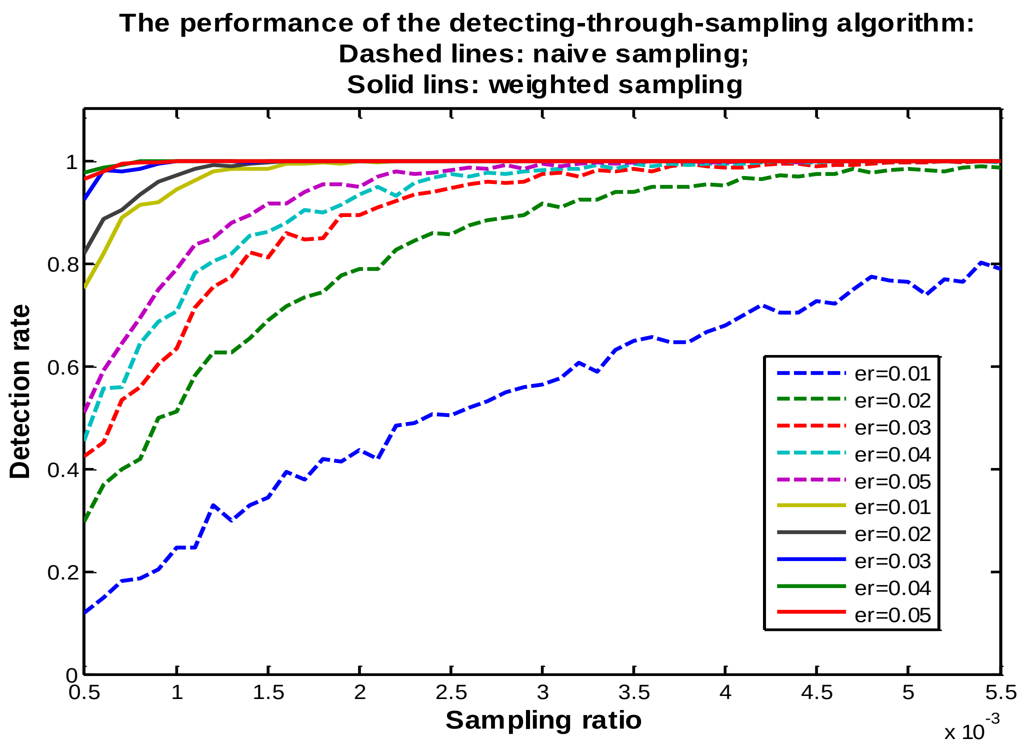


Figure **6-2**: Performance of sampling scheme

With different settings of the predefined error rates, Figure **6-2** demonstrates the distributions of the experimental detection rates over the increasing number of sampling ratios. Overall, we can see that the random sampling approaches, both naive and weighted, perform nicely on the PageRank cheating detection task under the MapReduce environment. All the distributions of the detection rates we got from this experiment are very close to the conceptual distribution as I proposed in the previous chapter.

As defined as the percentage of miscalculations, error rates ranging from 1% to 5% are represented by five different colors as shown in the figure. Dashed lines are corresponding to the naive-based random sampling approach, whereas solid lines are for the more advanced weighted-based random sampling methods. As can be seen from the above figure, both naive and weighted sampling achieves their optimal detection rates given larger percentage of error rate and sampling ratios. Detection rates increase dramatically with lower sampling ratios, in particular when sampling ratios are less than 0.15% in the weighted approach or less than 0.3% in the naive-based method. Then the distribution of the detection rates become relatively stable, with most of the sampling methods achieve about the same performance. In other words, given that 10,000 pages are used in this experiment, when sampling more than 15 of the total 10,000 pages or more than 30 of it, one can get satisfying detection rates of over 90%.

Consistent with my expectations, the empirical results show that weighted random sampling method achieves relatively better detection performance as compared to the alternative naive based approach. Almost all of the weighted experiments reach high detection rates of over 80%, while in contrast, results of the naive based approach are not as desire as the weighted ones under small sampling sizes. Weighted samplings gradually lost its absolute superiority in detection rates as the sampling ratios increases. Except naive samplings under 1% and 2% error rates, all sampling settings tested, both naive and weighted, in this study achieve about the same detection rates when sampling more than 35 pages from the total 10,000 ones.

In addition, there seems to be no significant differences among the weighted approaches under experimental settings of 3%, 4% and 5% error rates. All three sampling methods achieve almost perfect detection rate even with relatively small sampling ratios. In contrast, with less than 0.1% of the sampling ratios, the detection rates of the weighted sampling approaches decreases to only a bit more than 80% when the error rate becomes lower than 3%. Compared with these non-significant differences among weighted sampling experiments, naive-based sampling methods achieve relatively distinct detection rates, especially with one of its highest detection rate less than 80% under the 1% error rate.

Chapter 7

Conclusion and Future Direction

This study was aimed to develop result verification schemes that can detect the security threats exist due to MapReduce’s geographically distributed computation resource. In this research, we have presented two lightweight results verification schemes for detecting the cheating behaviors occurred on both lazy and malicious MapReduce workers. Proposed methods were validated on typical information retrieval applications, including the word frequency count, inverted index and the page rank calculation problem. This was done in consideration of the practical value of this study, given that MapReduce is heavily adopted by major search engine companies, such as Google. We have demonstrated that through theoretical derivations watermark injection was proved to work perfectly protecting the service integrity involved in the word count and inverted index tasks. In the same way, the theoretical validity of the random sampling method was also confirmed by its desired detection performance on page rank calculation. In order to empirically evaluate the new schemes, simulations were conducted for both methods using Hadoop's MapReduce implementation. Consistent with the previous theoretical derivations, experimental evaluations again achieved satisfying results. Both watermark injection and random sampling methods showed their effectiveness on ensuring the computational service integrity under the MapReduce environment.

Like any other researches in this field, this study also has a few limitations, centering on the lack of generalizability. First, as we have introduced in the scope section, the two schemes introduced in this study are proposed to detect service integrity violations happened in the data-intensive text processing tasks. Although this work can be extended to some other MapReduce applications, the watermark injection and random sampling schemes may not be very well generalize to other numerical computation applications, such as the ones for data mining, machine learning, and statistical analysis etc. Second, only lazy and malicious attacks have been considered in this study. Although they do cover a large proportion of the possible security risks, still there are threats in purpose from those strategic attackers. For instance, attackers who want to increase the frequencies of their proposed keywords. In that case, our watermark injection method would not work as well as it did while detecting the lazy and malicious attacks. Watermarked words that one injected are usually nonexistent words, so it would be very unlikely for the rational strategic attackers to manipulate the word count for those watermarked words. Therefore, with small watermark percentages, it would be hard to detect this kind of strategic cheating behavior using the watermark verification scheme as we proposed. Another trivial limitation for this study is that the detection method of random sampling in PageRank calculation requires client's input with link structures. Because only in this way can the result verification process know specific inlinks of each sampled node. Clients have to parse out the link structures themselves before they can send them to MapReduce. So this to some extent increases client's load, even though the random sampling also avoids possible replications afterwards and thus save a lot of cost for the clients. In summary, developing a generic solution for MapReduce applications remains an open question and it can be a possible direction for my future research.

Except the few limitations, this research also has several strengths, including its innovations, satisfying detection performance, and low cost and complexity. To date, there has been very limited research work on MapReduce security issues, especially on the aspect of result verification. This thesis introduced innovative mechanisms for detecting cheating services under the MapReduce environment based on watermark injection and random sampling methods. Results of these new detection schemes removed the redundant computation process as required in the replication-based methods [10]. Therefore, it significantly reduced the cost of verification overhead. I believe that the research reported is an important step towards trusted computational services and will assist in directing avenues for future research. In practical, the results of this research will hopefully further promote the secure adoption towards trusted MapReduce services and therefore help to bring profits to MapReduce service providers with increasing number of potential clients.

References

[1] W. Du and M. T. Goodrich. Searching for High-Value Rare Events with Uncheatable Grid Computing. In *Proc. 3rd Applied Cryptography and Network Security Conference (ACNS)*, pages 122-137, 2005

[2] J. Dean and S. Ghemawat. MapReduce: Simplified Data Processing on Large Clusters. *Proceeding of 6th Symposium on Operating System Design and Implementation (OSDI)*, pages 137-150, 2004

[3] J. Dean and S. Ghemawat. MapReduce: A Flexible Data Processing Tool. *Commun. ACM,* 53(1): 72-77, 2010

[4] L. Leydesdorff and L. Vaughan. Co-occurrence Matrices and Their Application in Information Science: Extending ACA to the Web Environment. *Journal of the American Society for Information Science and Technology*, In print.

[5] J. Lin. Scalable Language Processing Algorithms for the Masses: A Case Study in Computing Word Co-occurrence matrices with MapReduce. In *Proceedings of the 2008 Conference on Empirical Methods in Natural Language Processing (EMNLP 2008)*, pages 419-428, Honolulu, Hawaii, 2008

[6] S. Brin, L. Page. The Anatomy of a large-scale Hypertextual Web Search Engine. *In WWW7: Proceedings of the seventh international conference on World Wide Web 7*, pages 107-117, Amsterdam, The Netherlands, The Netherlands, 1998. Elsevier Science Publishers B.V.

[7] L. Pages, S. Brin, R. Motwani, and T. Winograd. The pagerank citation ranking: Bringing order to the web. Technical report, Stanford Digital Library Technologies Project, 1998

[8] C. A. Hansen. Opimizing Hadoop for the cluster. A assignment in the course INF-3203-Adcanced Distributed Systems offered at Univeristy of Troms.

[9] C. Germain-Renaud and D. Monnier-Ragaigne. Grid result checking. In *CF'05*: Proceedings *of the 2nd conference on Computing frontiers*, pages 87-96, New York, NY, USA, 2005. ACM Press.

[10] W. Wei, J. Du, T. Yu, and X. Gu. SecureMR: A service integrity assurance framework for MapReduce. Annual Computer Security Applications Conference (ACSAC), pp.73-82, 2009

[11] I. Roy, S. T. V. Setty, A. Kilzer, V. Shmatikov, and E. Witchel. Airavat: security and privacy for MapReduce. In *Proceeding of 7th USENIX Symposium on Network Systems Design and Implementation (NSDI '10)*, pages 297-312.

[12] W. Du, J. Jia, M. Mangal, and M. Murugesan. Uncheatable grid computing. In *ICDCS '04: Proceedings of the 24th International Conference on Distributed Computing System (ICDCS'04).* Washington, DC, USA: IEEE Computer Society, 2004, pp.4-11.

[13] S. Zhao, V. Lo, C. Gauthier Dickey. Result verification and trust based scheduling in peer-to-peer grids. In *P2P '05: Proceedings of the Fifth IEEE International Conference on Peer-to-Peer Computing.* Washington, DC, USA: IEEE Computer Society, 2005, pp.31-38

[14] P. Golle and I. Mironov. Uncheatable distributed computations. *2001: Proceedings of the 2001 Conference on Topic in Cryptoloty.* London, UK: Springer-Verlag, 2001, pp. 425-440.

[15] H. Karloff, S. Suri, and S. Vassilvitskii. A model of Computation for MapReduce. In *Symposium on Discrete Algorithms (SODA)*, 2010.

[16] Luis F. G. Sarmenta. Satotage-tolerance mechanisms for volunteer computing systems. In *CCGrid*, 2001.

[17] P. Golle and S. Stubblebine. Secure distributed computing in a commercial environment. In P. Syverson, editor, *Proceedings of Financial Crypto 2001*, volumne 2339 of *Lecture Notes* in *Computer Science*, pages 289-304, Spring-Verlag, 2001.

[18] P. Mell and T. Grance. The NIST definition of cloud computing. National Institute of Standards and Technology, 2009

[19] L. Gillam. Cloud computing: principles, systems and applications. 1st Edition, Springer, 2010. ISBN 1849962405

[20] D. Szajda, B. Lawson, and J. Owen. Toward an optimal redundancy strategy for distributed computations. In *Cluster Computing, 2005, pages, 1-11, IEEE International*, Sept. 2005.

[21] D. Szajda, B. Lawson, and J.Owen. Hardening functions for large-scale distributed computations. In Proceedings *of the 2003 IEEE Symposium on Security and Privacy, pages 216-224*, Berkeley, CA, May 2003

[22] F. Monrose, P. Wyckoff, and A. Rubin. Distributed execution with remote audit. In *Proceeding of the 1999 ISOC Network and Distributed System Security Symposium*, pages 103-113, 1999.

[23] B. Barney and L. Livermore. Introduction to parallel computing. Springer, 2003. ISBN0201648652

[24] Aaron Kimball, Sierra Michels-Slettvet, Christophe Bisciglia, et al.. Lecture 1 - Introduction to Distributed Computing & Systems Background, Introduction to Problem Solving on Large Scale Clusters. Spring 2007. Available for download at: http://code.google.com/edu/submissions/uwspr2007_clustercourse/listing.html

[25] Preetam Ghosh, Kalyan Basu and Sajal Das, A Game Theory based Pricing Strategy to support Single/Multi-Class Job Allocation Schemes for Bandwidth Constrained Distributed Systems. *IEEE Transactions on Parallel and Distributed Systems, 2007*, Volume 18, Issue 3, pp. 289-306

[26] I. Foster, Y. Zhao, I. Raicu, S. Lu. Cloud computing and grid computing 360-degree compared. In *Grid Computing Environment Workshop, 2008. GCE'08*, pages: 1-10.

[27] M. Armbrust, A. Fox, R. Griffith, A. D. Joseph, R. H. Katz, A. Konwinski, G. Lee, D. A. Patterson, A. Rabkin, I. Stoica, M. Zaharia. 2009. Above the clouds: a Berkeley view of cloud computing. Technical Report. University of California at Berkeley.

[28] K. Shvachko, H. Kuang, S. Radia, R. Chansler. The Hadoop distributed file system. In *Proceeding of MSST 2010*, May 2010.

[29] M. A. Vouk. Cloud computing issues, research and implementations. *In 30th International Conference on Information Technology Interfaces (ITI2008)*. Cavatat/Dubrovnik, Croatia, June 2008, 31-40.

[30] A. Jamakovic, S. Uhlig. On the Relationships between topological measures in real-world networks.