Similarity based Encrypted Data Search in Cloud Computing

By
Muhammad Umer
NUST201362721MSEECS60013F

Supervisor
Dr. Asad Waqar Malik
NUST-SEECS

A thesis submitted in partial fulfillment of the requirements for the degree of Masters of Science in Information Technology (MS IT)

In
School of Electrical Engineering and Computer Science,
National University of Sciences and Technology (NUST),
Islamabad, Pakistan.

(Feb 2016)

Approval

It is certified that the contents and form of the thesis entitled "**Similarity based Encrypted Data Search in Cloud Computing**" submitted by **Muhammad Umer** have been found satisfactory for the requirement of the degree.

Advisor: **Dr. Asad Waqar Malik**

Signature: ____________________
Date: ____________________

Committee Member 1: **Dr. Tahir Azim**

Signature: ____________________
Date: ____________________

Committee Member 2: **Ms. Hirra Anwar**

Signature: ____________________
Date: ____________________

Committee Member 3: **Mr. Ubaid Ur Rehman**

Signature: ____________________
Date: ____________________

Abstract

Encrypting confidential data before outsourced to cloud storage ensures data security and privacy but encryption hides data information making CSP unable to perform standard lookup queries for searching. This is the problem we addressed in our research to enable searching over outsourced encrypted data. Existing schemes for searching over encrypted data are based on trapdoors and utilizing locally stored indexes which limits searching capabilities to a limited pre-defined trapdoors. Moreover, existing schemes support exact keyword matching. In this thesis we proposed a similarity based searching over outsourced encrypted data while ensuring end-to-end privacy. Unlike existing approaches, our scheme enables subscribers to define their own queries with arbitrary number of keywords. We proposed a private matching algorithm which ensures that cloud computing service performs term matching without revealing anything information about user query and confidential data by utilizing homomorphic encryption. We proposed a novel idea for reducing communication cost overhead and our results shows gain in communication cost reduction over 95%. We have implemented the scheme and our results have demonstrated that search queries with 2 to 10 keywords only cost 0.00001 to 0.00004 $ per 1000 similar requests.

Certificate of Originality

I hereby declare that this submission is my own work and to the best of my knowledge it contains no materials previously published or written by another person, nor material which to a substantial extent has been accepted for the award of any degree or diploma at National University of Sciences & Technology (NUST) School of Electrical Engineering & Computer Science (SEECS) or at any other educational institute, except where due acknowledgement has been made in the thesis. Any contribution made to the research by others, with whom I have worked at NUST SEECS or elsewhere, is explicitly acknowledged in the thesis.

I also declare that the intellectual content of this thesis is the product of my own work, except for the assistance from others in the project's design and conception or in style, presentation and linguistics which has been acknowledged.

Author Name: **Muhammad Umer**

Signature: ________________

Acknowledgment

Up and above everything all glory to **ALMIGHTY ALLAH**. The Beneficent, The most Merciful and Most Compassionate. It's a great blessing from Almighty Allah that gives me the health and strength to do this research work.

I would like to special thank the my supervisor **Dr. Asad Waqar Malik** for his patience, guidance, useful impact and availability throughout my research work.

I would like to express my sincere gratitude to **Dr. Zeeshan Pervez**, **Dr. Tahir Azim** for the continous support of my research, motivation, enthusiasm, and immense knowledge. Their guidance helped me in all the time of research and writing of this thesis. I could not have imagined having a better advisor and mentor for my thesis.

My special thanks to **Mr. Ubaid Ur Rehman** and **Ms. Hirra Anwar** for being my committee members.

Muhammad Umer

Contents

1 INTRODUCTION **1**
1.1 Problem Statement . . . 2
1.2 Thesis Contribution . . . 2
1.3 Thesis Organization . . . 3

2 LITERATURE REVIEW **4**
2.1 Cloud Computing . . . 4
2.2 Encrypted Search . . . 4
2.3 Secret Sharing based Encryption . . . 5
2.4 Homomorphic Encryption . . . 5
2.4.1 Fully Homomorphic Encryption . . . 6
2.4.2 Partial Homomorphic Encryption . . . 6
2.4.3 Pascal Paillier Homomorphic Encryption . . . 6
2.4.3.1 Key Generation . . . 6
2.4.3.2 Encryption . . . 7
2.4.3.3 Decryption . . . 7
2.4.3.4 Homomorphic Operations . . . 7
2.5 Bloom Filters . . . 7
2.5.1 False Positive Rate . . . 9
2.5.2 Sliding Window based Bloom Filter . . . 9
2.6 Trapdoors . . . 10
2.7 Private Matching . . . 10
2.8 Related Work . . . 11
2.8.1 Using Trapdoor Encryption . . . 11
2.8.2 Using Secret Sharing . . . 12
2.8.3 Using Homomorphic Encryption . . . 13

3 Methodology **14**
3.1 Dataset Preparation . . . 14
3.2 Assumptions and Notation . . . 15
3.3 Indexing Data Files . . . 17

3.3.1 Steps 17
3.3.2 Index Entry Size 17
3.3.3 Index Creation Algorithm 18
3.3.4 Keywords Extraction Algorithm 18
3.3.5 SWBF Creation Algorithm 19
3.4 Query Creation 19
3.5 Similarity Based Terms Matching 20
3.6 Communication Cost Reduction 21

4 DESIGN AND ARCHITECTURE 24
4.1 System Design Goal 24
4.2 System Components 25
4.3 System Architecture 26
4.4 Indexing & Data Outsourcing 27
4.5 Searching 27
4.6 Private Matching 28
4.7 Response Extraction 29

5 IMPLEMENTATION 30
5.1 Google Cloud Platform 30
5.2 Indexer Application 31
5.2.1 Sliding Window BF Implementation 31
5.2.2 Encryption 31
5.3 Cloud Storage Server 32
5.4 Cloud Computing Server 32

6 RESULTS AND PERFORMANCE EVALUATION 33
6.1 Experimental Setup 34
6.2 Index Generation and Processing 34
6.3 Searching 36
6.3.1 Cost Estimation 38

7 CONCLUSION AND FUTURE WORK 40
7.1 Conclusion 40
7.2 System Limitations & Future Work 40

Appendix A Performance evaluation: Data Tables 42

List of Abbreviations

Abbreviations	Descriptions
CSP	Cloud Service Provider
PK	Private Key
SK	Secret Key
BF	Bloom Filter
CS	Cloud Storage
TTP	Trusted Third Party
PHR	Personal Health Records
CSS	Cloud Storage Server
CCP	Cloud Computing Server
SWBF	Sliding Window Bloom Filter
HE	Homomorphic Encryption
GAE	Google App Engine

List of Figures

2.1 10-bit BF 8
2.2 Sliding window bloom filter 10

3.1 Compressed data transfer using 64bit paillier key 23

4.1 System components 25
4.2 Architecture diagram 26
4.3 Indexing process workflow 27
4.4 Searching process workflow 28
4.5 Term matching process workflow 28
4.6 Response extraction workflow 29

5.1 Configuration file: config.xml 31

6.1 Index generation time 35
6.2 BF size impact on index generation time 35
6.3 Paillier key size impact on index file size 36
6.4 Search results compression Vs. uncompressed results 37
6.5 Response extraction time 37
6.6 Searching cost with single keyword query 38
6.7 Searching Cost($) with multi-keyword query 39

List of Tables

3.1 Dataset details 14
3.2 Notations used in mathematical and descriptive details 15
3.3 Index BF 20
3.4 Query BF 21
3.5 Addition result 21

5.1 Google cloud platform services 30

6.1 Machine specs used for indexer application 34
6.2 Specs for cloud computing application 34

A.1 Keywords extraction and encryption time 42
A.2 BF size impact on index creation time 42
A.3 Impact of HE key and BF sizes on index file size 42
A.4 Single-keyword query searching cost 43
A.5 Multi-keyword query searching cost 43

Chapter 1

INTRODUCTION

Cloud computing enables computing services over the internet to its subscribers and provides tailored computing resources on-demand. As amount of data is increasing dramatically, new challenges like storage of this big data and its security has arisen. To meet the huge data storage requirements, Cloud Service Providers (CSP) are providing cheap unlimited storage facilities to cloud users that are individuals as well as enterprises. Subscribers of cloud storage can outsource their data on public cloud servers for a longer period of time and without worrying about availability and reliability of the data. It will be the CSPs responsibility regarding data backup, synchronization and sharing of data with relevant stakeholder.

CSP charged its subscribers as per-use policy. It includes physical storage of data and the bandwidth. Thus it is desired to only access most relevant data over the network via searching from a huge amount of data as accessing needless data will consume more bandwidth and hence add more cost to the user. Most of the time data owner does not want to reveal his data to CSP and other users due to sensitive nature of the data. For example a patient wants privacy about his medical record and does not want anyone else to see his reports including CSP.

To achieve data protection, confidentiality and privacy, users place data on cloud after encryption. This solves the confidentiality and privacy problem but searching on encrypted data becomes an issue. User has to download all the data from cloud storage and after decryption on local machine he will perform searching. It is not feasible to download all the outsourced data for searching few data items. Therefore we need a mechanism which enable user to search over the encrypted data without revealing sensitive information and without accessing needless data over the network.

In this thesis we have proposed and implemented similarity based encrypted data search, instead of exact encrypted keywords matching. At

first, the data owner will create index on confidential files which he wants to be stored on cloud after encryption. Confidential data files are encrypted by AES symmetric key while index files with Pascal Paillier [1] homomorphic encryption key. Both encrypted confidential files and encrypted index will be placed on CSS. A subscriber who wants to search will create similar bloom filter as used during index creation and sends that query to CCS. CCS accesses all the index files and perform private matching with user query. Searh results are compressed and encrypted compressed response is send back to user. User will regenerate cloud server matched search terms after decompressing each search result entry and will compute similar score.

Encrypted bloom filters are used to determine similarity score for the user query. Similarity score is determined by using additive homomorphic encryption property by cloud computing server. Cloud server will know nothing about the matched results as addition of two cipher text results another cipher text.

After addition of query and index bloom filter bits, a polynomial will be created over each search result and size of the search entry is compressed to the size of homomorphic encryption (Pascal Paillier) key size. By this communication cost reduction algorithm, we have achieved over 95% reduction in data communication.

1.1 Problem Statement

Encryption hides the data information and searching cannot be performed as standard lookup queries no longer be evaluated. Subscribers have to download entire data from cloud storage in order to perform searching. Existing schemes are based on trapdoors and using exact keyword matching which restricts users to perform queries under certain boundaries. Moreover, communication cost is very high as these schemes return all the data back to searching application without any compression.

1.2 Thesis Contribution

Our research work contributed in searching over encrypted data in an untrusted cloud environment. The summary of our contributions is given as follows:

- Similarity based search for encrypted data to ensure that CSP cannot learn any information from the search query. Also, CSP is unable to identify the terms matching result that satisfy the selection criteria.

Moreover, CSP cannot relate search queries submitted by multiple users searching for same data

- End-to-end privacy-aware data search, which ensures that only authorized subscriber can search the outsourced data
- A search service that enables users to create arbitrary queries without relying on pre-computed trapdoors
- Communication cost reduction over 95% by compressing index-query matching search results.

1.3 Thesis Organization

The rest of the thesis is organized as follows:
Chapter 2 discusses about the preliminaries and research carried out so far in privacy aware encrypted data search in cloud computing. Chapter 3 describes our research methodology and its objectives. Chapter 4 has contents for the design and architecture of our proposed system. Chapter 5 includes detailed implementation while chapter 6 describes the results along with detailed discussions. Chapter 6 concludes our thesis with a conclusion and future work direction.

Chapter 2

LITERATURE REVIEW

In this chapter, preliminaries and literature review is presented related to research already carried out regarding privacy aware encrypted search in cloud environment. In this context, focus is existing techniques of searching over encrypted data and how they affect utility of cloud storage.

2.1 Cloud Computing

Cloud computing referred as on-demand and scalable computing platform. It provides online virtualized computing resources as services, which can be subscribed on pay-as-you-use model. Subscribers of cloud services can simply employ the computing resources of a public cloud, managed by CSP. Thus cloud computing is enabling the subscribers from keeping and building in-house expensive an IT staff and data. CSP provides an abstraction of unlimited processing power and storage facility, which can be subscribed as a pay-as-you-use subscription model. Now, subscribers don't need to buy expensive licences upfront instead they will have to pay as they will utilize the services.

2.2 Encrypted Search

With the exponential growth of data, CSPs are providing cheap and virtually unlimited storage facilities. Subscribers often outsource confidential data after encryption to the CSP. Now, on encrypted data it is not possible to perform normal daily life lookup queries to retrieve their desired results. In order to search, subscribers have to download complete data and after decrypting locally they can lookup desired documents. Certainly, downloading entire data is not feasible, also it consumes bandwidth and adds

more cost to the subscriber's budget. So, encrypt search is about enabling and performing search over encrypted outsourced data residing in the cloud environment without compromising data confidentiality and ensuring end to end privacy.

2.3 Secret Sharing based Encryption

In the secret sharing based security, no single entity holds a complete key. A key is distributed to a number of partipants in such a way that no one can use his part of the key to find out the secret. All the participants must teamed up in order to unlock the secret.

2.4 Homomorphic Encryption

Using Homomorphic Encryption one can perform mathematical operations over cipher text without decrypting them first. These mathematical operations have usual effect on plaintext. Given two plaintexts $m1$ and $m2$ and homomorphic encryption function $\mathcal{E}$, one can calculate $\mathcal{E}(m1 + m2)$ as $\mathcal{E}(m1) * \mathcal{E}(m2)$ without knowing $m1$ and $m2$. As mathematical operations are performed on an encrypted text so outcome of the mathematical operation will also be encrypted by default and hence cannot be determined without decryption. When decryption is applied on the outcome of the homomorphic mathematical operation, the result is equal to as by plain text. i.e.,

$$operation(plaintext) == decrypt(operation(encrypt(plaintext))) \tag{2.1}$$

Homomorphic encryption supports both additive and multiplicative properties over cipher texts but currently no scheme exists to support both operations. Some schemes are additive homomorphic for example Pascal Paillier and some are multiplicative for example RSA.

A homomorphic encryption is said to be an additive homomorphic if unlimited additions can be performed over its ciphered texts. mathematically, it's function $\mathcal{E}_h$ holds following property:

$$\mathcal{E}_h(m1) * \mathcal{E}_h(m2) = \mathcal{E}_h(m1 + m2) \tag{2.2}$$

Multiplicative homomorphic encryption scheme will have following property:

$$\mathcal{E}_h(m1) * \mathcal{E}_h(m2) = \mathcal{E}_h(m1 * m2) \tag{2.3}$$

2.4.1 Fully Homomorphic Encryption

A homomorphic encryption scheme holding both additive and multiplicative properties is said to be fully homomorphic. Given $\mathcal{E}(m1)$ and $\mathcal{E}(m2)$, one can compute $\mathcal{E}(m1) * \mathcal{E}(m2)$ and $\mathcal{E}(m1) + \mathcal{E}(m2)$. Unfortunately, no such practical scheme exists so far which both supports unlimited addition and multiplications. A scheme devised by Boneh, Goh and Nissim (BGN) [2] was the first to allow both additions and multiplications. However, this scheme does allow unlimited additive operations like Pascal Paillier but it only allows one multiplication. In 2009, Craig Gentry [3] proposed a fully homomorphic scheme but it is so far impractical to use due to very slow performance.

2.4.2 Partial Homomorphic Encryption

Partial homomorphic cryptosystems which were called just as homomorphic before fully homomorphic were discovered. In partial homomorphic schemes one can perform one operation either addition or multiplication. Few of the schemes that are partial homomorphic are Pascal Paillier, RSA and ElGamal.

2.4.3 Pascal Paillier Homomorphic Encryption

Pascal Paillier is an additive homomorphic encryption scheme proposed by Pascal Paillier. It is a probabilistic asymmetric cartographic system in which public key is used for mathematical operations. As Pascal Paillier is additive homomorphic cryptosystem, given only the public-key and the encryption of $m1$ and $m2$, one can compute encryption of $m1 + m2$ as below:

$$\mathcal{E}_h(m1 + m2) = \mathcal{E}_h(m1) * \mathcal{E}_h(m2) \tag{2.4}$$

2.4.3.1 Key Generation

Let p and q are randomly generated two large prime numbers which are independent of each other such that $\gcd(pq, (p-1)(q-1)) = 1$. This property holds to check if both primes are of equal length. Public key PK and private key SK are computed as below:

1. Compute $n = pq$ and $\lambda = \operatorname{lcm}(p-1, q-1)$
2. Select random integer g where $g \in \mathbb{Z}^*_{n^2}$
3. Compute μ such as, $\mu = (L(g^{\lambda} \bmod n^2))^{-1} \bmod n$, where function L is defined as $L(u) = \frac{u-1}{n}$

- The public key PK is (n, g)
- The private key SK is (λ, μ)

2.4.3.2 Encryption

To encrypt a message m where $m \in \mathbb{Z}_n$, generate a random number r where $r \in \mathbb{Z}_n^*$ and define an encryption function $\mathcal{E}_h(m)$ such that

$$\mathcal{E}_h : \mathbb{Z}_n \, x \, \mathbb{Z}_n^* \mapsto \mathbb{Z}_{n^2}^* \tag{2.5}$$

Then ciphered text c can be computed by following function:

$$c = \mathcal{E}_h(m, r) = g^m \cdot r^n \bmod n^2 \tag{2.6}$$

2.4.3.3 Decryption

Let c be the cipher-text to decrypt, plaintext m can be computed as

$$m = L(c^\lambda \bmod n^2) \cdot \mu \bmod n \tag{2.7}$$

2.4.3.4 Homomorphic Operations

Pascal Paillier is additively homomorphic and following properties can be described:

- Arithmetic addition on ciphered texts $c1 = \mathcal{E}_h(m_1, r_1)$ and $c2 = \mathcal{E}_h(m_2, r_2)$ can be computed as:

$$D((c1 \cdot c2) \bmod n^2) = m_1 + m_2 \bmod n. \tag{2.8}$$

- Paillier cipher-text $c = \mathcal{E}_h(m_1, r_1)$ can be multiplied with a constant plaintext $m2$ as:

$$D(c^{m_2} \bmod n^2) = m_1 \cdot m_2 \bmod n \tag{2.9}$$

2.5 Bloom Filters

Bloom Filter is a probabilistic data structure to efficiently evaluate membership queries. Its basic operations include adding an element in the filter and then querying its membership.

Bloom Filters (BF) consist of a bit space in the form of 0 and 1. When BF is created, its all bits are initialized to zero. When we insert an element into a BF, corresponding bit locations are set to 1. For each element we insert, it is first passed through a certain number of hash functions. Each hash function returns a bit location. That bit in the BF has been set to 1.

While working with BF, we don't need to store actual string in the index. So these are efficient in space. Below is an example of adding a string element in a bloom filter:

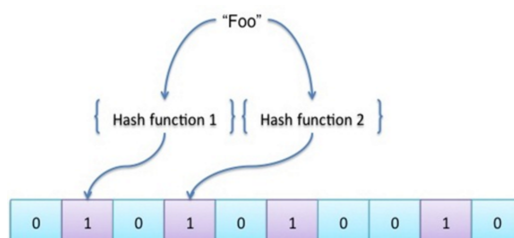

Figure 2.1: 10-bit BF

Foo is first passed to hash function *Hash Function 1* which returns index 1 and corresponding bit in BF is set. Hash function *Hash Function 2* returns index value 03 and bit 4 of BF is set to 1. There are two other bits appearing as 1, which depicts that a word is already added. Now, in order to query if *Foo* exists in the BF or not, we will apply *Hash Function 1* on *Foo* and hash function will return BF bit number and if its value is 1 which means *Foo* is present as per *Hash Function 1*. Similarly we will repeat same operation for *Hash Function 2* and if both hash functions returned a bit value 1 of BF then we will able to say that *Foo* may be present in our BF.

Due to quick lookup, bloom filters are being used in daily life industry applications. In Google Chrome web browser bloom filters are being used to detect malicious URLs. A bloom filter is downloaded as part of browser set and any URL is first checked against this local bloom filter and user is warned about the site. Bitcoins utilizes bloom filters to speed up wallet synchronization process.

Bloom filters have following interesting useful properties:

- Not storing actual data in the structure thus efficient in terms of space
- Very low main memory required for processing large set of data
- No False Negative rate

Although they have 0% false negative rate but they can end up with high false positive rates if not used carefully. In this thesis, BF are used for:

- Indexing user data file space
- Searching in index file
- Calculating similarity score

2.5.1 False Positive Rate

Bloom filters underlying space is constant and all initializes to 0. Upon adding elements in bloom filters corresponding bits are set to 1. A stage comes when majority of the bits are set as 1. At this stage when a membership query will come, there is a high probability that the hash function h will corresponds to a 1 bit in bloom filter. Now let that bit was set by hash function h for some other element then we end up with a false positive.

False positive rate can be decreased by increasing underlying bloom filter space and more number of hash functions. Mitzenmacher and Upfal have given a mathematical formula to approximate false positive rate dependent on underlying bloom filter space. Let, we have m bloom filter bits, k number of hash functions and n inserted elements then false positive rate will be:

$$\approx \left(1 - e^{-kn/m}\right)^k \tag{2.10}$$

2.5.2 Sliding Window based Bloom Filter

Sliding Window based BF are special type of bloom filters which we are going to use in this research. In SW based BF a window size is defined and based on that window size a keyword is mapped to a BF. For example, lets we have a word *Cloud* and *window size = 2*. Now *Cloud* will be sliced into *CL*, *Lo*, *ou*, *ud* and will be mapped on a bloom filter as shown in figure 2.2. With a larger word, by using SWBF, we have better similarity score.

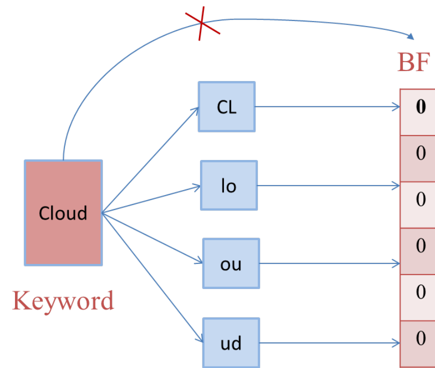

Figure 2.2: Sliding window bloom filter

2.6 Trapdoors

Trapdoor is a one way function which is used for secure indexing of confidential data. Trapdoors are based on secret key sk and it is very difficult to reverse trapdoor function without specific information. Although, trapdoors are easy to utilize but they limit the searching capabilities as each query has to be mapped to already existing trapdoor. Let f is a one way function, ω_i is a keyword with given secret key sk, trapdoor can be computed as below:

$$T\omega_i = f(sk, \omega_i) \tag{2.11}$$

Trapdoors have following drawbacks:

- Trapdoors are subjected to any modification in the original data. They need to be recomputed and redistributed on each update in the confidential outsourced data.
- Trapdoors need to be distributed to the subscribers to allow searching in the encrypted data
- Searching is limited to the available trapdoors

2.7 Private Matching

Private Matching is a protocol for intersecting user data and data set of service provider without accessing each others data. So, this does not reveal any sensitive information to the service provider and it let user to check if his anonymous data intersects with the datasets of the service provider.

Homomorphic encryption schemes help in private matching as mathematical operations are performed on encrypted text and the outcome of a mathematical operation is also encrypted so CSP cannot learn about searched query, matched terms and patterns while subscriber cannot learn about data items present on the cloud storage.

2.8 Related Work

Traditionally, confidential information is protected by means of access control mechanism. This mechanism may work if confidential information is present on a local trusted server. But, this assumption will fail if confidential information is outsourced to public servers that is some untrusted third party or CSP. Though CSP provides a way to encrypt data by their own which can secure data from outside world but it still has two drawbacks. One, after encryption, searching will not be possible and 2nd, CSP having the security keys can access confidential data.

In this section, some solutions to above problem are discussed and they can be categorized as:

- Trapdoor based Schemes
- Secret sharing based Schemes
- Homomorphic Encryption based Schemes

In the rest of this section, the existing solutions will be categorized as one of the above categories and will be discussed in more detail.

2.8.1 Using Trapdoor Encryption

Song et al. [4] song was the first to propose a symmetric key based searchable scheme for encrypted data. Each keyword in the document was encrypted independently using trapdoors. Later on various schemes were proposed based on this to search encrypted index instead of original data itself. Goh proposed bloom filter based encrypted search. Trapdoors are generated against all the keywords in a file and added to a bloom filter. In this way for each file a single bloom filter is created using trapdoors and stored in the CSS. To search, user computes trapdoor for the keyword and sends to cloud server. The cloud server will check the trapdoor in the bloom filter and in case it exists cloud return the file identifier. Above both schemes were doing an exact keyword matching and relying on trapdoors. Boneh. et al [5]

presented first public key based searchable scheme which enables authorized users having private key to search in the index. These schemes were using trapdoors based indexing, having exact keyword matching and not utilizing cloud services efficiently.

Li et al. [6] proposed a scheme on encrypted Personal Health Records (PHR) by using Hierarchical Predicate Encryption (HPE). They introduced a Trusted Third Party (TTP) for the distribution of trapdoors. Authorized users obtained trapdoors from TTP and then submit it to CSP for evaluation. This scheme can greatly affect the cloud storage utility as only predefined trapdoors search can be possible and users cannot model their own queries. Saibal et al. [7] proposed an encrypted search scheme using BF. They also used soundex coding [8] for each word to search similar words which are pronounced similar. They created single bloom filter for each document thus having a high volume of false positive rate. Also, They created bloom filter based index after encryption of user documents which means while query modelling we can't randomize encryption process which can be a security loophole. Mehmet et al. [9] proposed a similarity based encrypt search scheme. Scheme was using Locality Sensitive Hashing (LSH) for the similarity score calculation. This scheme was using Trapdoors and hence leaking private matching information upon which statistical attacks can be possible. To avoid this, multi-server were introduced for the separation of leaked information.

Waters et al. [10] extended work of Song et al. and proposed a similar technique to secure audit logs. Audit logs contain sensitive information about series of events, actions and actors who are responsible for triggering particular event or performing an action. Therefore encryption is secured for its confidentiality and only when a searching is required, a trusted third party issues a trapdoor for a specific keyword search.

2.8.2 Using Secret Sharing

A typical usage of secret sharing can be of Private Information Retrieval (PIR). Using PIR users can query a database without revealing what data is queried. In PIR, data is replicated among non-interacting several servers with no communication link. Now, a user can create a query to get a part of data from each server without revealing complete query to a single server.

Lin and Candan [11] used a single server for PIR and it can be treated as a privacy compromise and also a single point of failure. The basic idea is that, user query asks more data than is required so that server could not figure it out what actually is being searched. In order to avoid statistical learning and replay attacks, retrieved results are shuffled and storage location gets

changed.

2.8.3 Using Homomorphic Encryption

Zeeshan et al.[12] proposed an inverted index based encrypted data search scheme. They used homomorphic encryption to ensure end to end privacy. Only authorized users can query and each user will have its proxy encryption key in order to transform index. As they were using inverted index so their index files actually contained data. They used TTP for the ranking of searched results and query modelling.

Chapter 3

Methodology

This chapter discusses about the details of the research methodology used for the realization of our proposed system. It includes details of the algorithms and assumptions our system will be based on.

The research is carried out in two perspective one is indexing the data files and second is, searching in the index files placed on CSS. Below are our research objectives:

- Data files collection and dataset preparation
- Indexing data files
- Similarity based terms matching
- Communication cost reduction

3.1 Dataset Preparation

Dataset collection and preparation is the basic part in our research. In order to carry out research analysis and to test our proposed system we prepared dataset of around 150 pdf files. These all books are a collection of IT self help literature. As data files were in PDF format so firstly we extracted text from these files by using our own *Java* code. *Java* code uses *PDFBOX* [13] library to extract text from PDF files.

After text extraction we get a set of text files with ASCII data. These data files are further divided into 06 groups of 5mb, 10mb, 20mb, 40mb, 60mb and 100mb. Further details of the dataset are given below in the Table 3.1

Table 3.1: Dataset details

Dataset Group	Files Size (mb)	Keywords
5mb	5	5,000
10mb	10	10,000
20mb	20	17,000
40mb	40	42,000
60mb	60	70,000
100mb	100	129,000

For keywords extraction keyword size is set as 8, that is any candidate keyword with length < 8 is not accepted.

3.2 Assumptions and Notation

Our proposed system focuses on privacy-aware similarity based searching in cloud averment. We intentionally ignored the details of data sharing between cloud system and calling applications. Security details on data sharing in cloud storage systems can be read here [14]. We also ignored, key exchange mechanism among data owner, CSP and end user.
Table 3.2 illustrates the notations that we used in order to explain basic concepts of our proposed system.

Table 3.2: Notations used in mathematical and descriptive details

Notation	Description
$\mathcal{F}$	Confidential file that needs to be outsourced
$\mathcal{K}\omega$	List of keywords in a data file $\mathcal{F}$
$f\omega$	Frequency of a keyword $k\omega$
$\mathcal{L}$	Allowed keyword length for extraction of list of keywords from data
$\mathcal{I}$	Index file having encrypted bloom filter for each keyword, frequency and number of 1ns as index entries
$\mathcal{E}_h$, $\mathcal{D}_h$	Homomorphic encryption and decryption functions
$\mathcal{E}_S$, $\mathcal{D}_S$	Symmetric encryption and decryption functions
σ_{pk} , σ_{sk}	Homomorphic encryption public and private keys
$\mathcal{K}_S$	Symmetric encryption and decryption key. It is used to encrypt index file name
$\mathcal{S}_q$	Similarity score of a query
$\mathcal{R}_q$	Compressed search result entry
$\mathcal{T}_b$, $\mathcal{O}_b$	Number of 2s and 1ns in a resultant term
$\mathcal{BF}_1, \mathcal{BF}_2, \dots, \mathcal{BF}_n$	Variables of a polynomial $\mathcal{P}$ where n is the length of BF
$3^1, 3^2, \dots, 3^{n-1}$	Coefficients of a polynomial $\mathcal{P}$ where n is the length of BF

$\mathcal{F}$ represents owner data file which needs to be outsourced to CSP. $\mathcal{I}$ is the searchable index file. Each index entry in $\mathcal{I}$ will consists of encrypted BF bits $\mathcal{BF}_1, \mathcal{BF}_2, \dots, \mathcal{BF}_n$, term frequency and number of ones in a BF. $\mathcal{E}_h$ and $\mathcal{D}_h$ are the homomorphic encryption functions used to encrypt BF bits and performing mathematical operation on homomorphically encrypted data. σ_{pk} and σ_{sk} are public and private keys of additive homomorphic encryption cryptosystem. $\mathcal{K}_S$ is the symmetric key for symmetric encryption function $\mathcal{E}_S$. It is used for the encryption and decryption of data file and index file names. $\mathcal{BF}_1, \mathcal{BF}_2, \dots, \mathcal{BF}_n$ are bits of encrypted BF. They also represents variables for the polynomial $\mathcal{P}$ where $\mathcal{BF}_i \in \langle 0, 1, 2\rangle$. $\mathcal{P}$ represents compressed output term after matching with an index bloom filter and user search query. $3^1, 3^2, \dots, 3^{n-1}$ are the coefficients for polynomial $\mathcal{P}$. Standard form of polynomial $\mathcal{P}$ is:

$$\mathcal{P} = 3^0 \cdot \mathcal{BF}_1 + 3^1 \cdot \mathcal{BF}_2 + 3^2 \cdot \mathcal{BF}_3 + \dots + 3^{n-1} \cdot \mathcal{BF}_n \tag{3.1}$$

3.3 Indexing Data Files

Indexing on data helps to make efficient lookup especially in relational databases. But in this research, indexing helps to perform similarity based search. Index on a file has been created in such a way that oblivious query can be performed and similarity score can be determined.
Each index entry $\mathcal{I}_i$ in an index file $\mathcal{I}$ have following structure:

$$\mathcal{I}_i = \mathcal{BF}_1, \mathcal{BF}_2, \dots, \mathcal{BF}_n, f\omega, \mathcal{O}_b \tag{3.2}$$

3.3.1 Steps

Indexing is applied on each data file one by one after reading from a specified path. Only text files are currently considered for indexing. Below are steps for indexing data files: $\mathcal{S}$

1. **Read a file** from a specified path. All the files in a specified directory will be read recursively. Hidden and invalid files are ignored.
2. **Extract all keywords** from a file and determine frequency for each keyword. Only unique words with keyword length criteria are considered. Stop-words are not considered and they will be ignored. All the keywords fulfilling the criteria are converted to their lexical meaning by applying stemming. Porter stemming algorithm [15] is being used for stemming provided by Apache Lucene [16].
3. **Create SWBF** for each keyword and note number of ones in SWBF. Ones in a SWBF will be those indexes where hash functions have set 1 value from 0.
4. **Encrypt each bit of a SWBF** by Pascal Paillier public key.
5. **Write each SWBF as an index entry** with frequency and number of 1 bits as noted in point 3.

3.3.2 Index Entry Size

As per Eq.3.2, each index entry size will be dependent on size of BF $\mathcal{L}$ and key size of Pascal Paillier encryption. Following equation can be used to find out the size (in bytes) of an index entry:

$$SizeOf(\mathcal{I}_i) = SizeOf(PascalPaillierKey) * \mathcal{L} + 04 + 04 \tag{3.3}$$

By using above, we can find out size of an index file $\mathcal{I}$ as:

$$\mathit{SizeOf}(\mathcal{I}) = \mathit{SizeOf}(\mathcal{I}_i) * \mathcal{K}\omega \tag{3.4}$$

3.4 Query Creation

Query creation process is similar to as index creation. A user will enter his searching keyword in plain text. First, it will be stemmed using Porter stemmer and then SWBF will be created. All the bits of SWBF is encrypted by the public key σ_{pk} of Pascal Paillier homomorphic encryption key.
Query creation processing took place on end user's machine. After encryption is done, query is sent to cloud for terms matching.

3.4.1 Index Creation Algorithm

Pseudo code for index creation process is given below. It illustrates the steps performed for index files:

Algorithm 3.1: Index Creation

Input: A collection of text files $C = \langle \mathcal{F}_1, \mathcal{F}_2, \ldots, \mathcal{F}_n \rangle$
Output: Index files $I = \langle \mathcal{I}_1, \mathcal{I}_2, \ldots, \mathcal{I}_n \rangle$ for each text file in C

1 $\forall \mathcal{F}_i \in \langle \mathcal{F}_1, \mathcal{F}_2, \ldots, \mathcal{F}_n \rangle$
2 **while** $\mathcal{F}_i \in C$ **do**
3 $\quad \mathcal{K}\omega \leftarrow \mathit{extractAllKeywords}(\mathcal{F}_i)$
4 $\quad \forall k\omega_i \in \mathcal{K}\omega$
5 $\quad$ **while** $k\omega_i \in \mathcal{K}\omega$ **do**
6 $\quad\quad \mathit{SBF} \leftarrow \mathit{createSWBF}(k\omega_i)$
7 $\quad\quad f_i \leftarrow \mathit{getKeywordFrequency}(k\omega_i)$
8 $\quad\quad \mathcal{O}_b \leftarrow \mathit{getOnes}(\mathcal{BF}_i)$
9 $\quad\quad \forall bf_i \in \mathit{SBF}$
10 $\quad\quad$ **while** $bf_i \in \mathit{SBF}$ **do**
11 $\quad\quad\quad \mathcal{BF}_i \leftarrow \mathcal{E}_h(\sigma_{pk}, bf_i)$
12 $\quad\quad\quad \mathit{IndexEntry} \leftarrow \mathit{IndexEntry}, \mathcal{BF}_i$
13 $\quad\quad\quad bf_i \leftarrow \mathit{getNextBit}(\mathit{SBF})$
14 $\quad\quad \mathit{IndexEntry} \leftarrow \mathit{IndexEntry}, f_i, \mathcal{O}_b$
15 $\quad\quad \mathcal{I}_i \leftarrow \mathit{writeToIndexFile}(\mathit{IndexEntry})$
16 $\quad\quad k\omega_i \leftarrow \mathit{getNextKeyword}(\mathcal{K})$
17 $\quad \mathcal{F}_i \leftarrow \mathit{getNextFile}(C)$
18 **return** $\mathcal{I}_i$

3.4.2 Keywords Extraction Algorithm

Algorithm used to extract list of keywords $\mathcal{K}\omega$ from a file $\mathcal{F}_i$ is given below:

Algorithm 3.2: Keyword Extraction

```
   Input: A file F_i, Maximum keyword length L
   Output: Set of keywords Kω and frequency fω
 1 len ← numberOfLines(F_i)
 2 Kω ← null
 3 fω ← 0
 4 for i ← 1 to len do
 5     line ← getFileLine(kω_i, i)
 6     tokens T ← tokenize(line)
 7     for ∀t_i ∈ T do
 8         if t_i >= L and t_i ∉ Kω then
 9             add (Kω, t_i)
10             fω_{t_i} = 1
11         else if t_i ∈ Kω then
12             fω_{t_i} = fω_{t_i} + 1
13 return Kω, fω
```

3.4.3 SWBF Creation Algorithm

Algorithm for sliding window bloom filter creation is given below:

Algorithm 3.3: SWBF Creation

```
   Input: A keyword kω_i and sliding window size sws
   Output: Sliding Window Bloom Fitler SWBF
 1 len ← sizeOf(kω_i)
 2 for i ← 1 to len do
 3     slide ← substring(kω_i i, i + sws)
 4     map(SWBF, slide)
 5     if i = len − sws then
 6         exit
 7 return SWBF
```

3.5 Query Creation

Query creation process is similar to as index creation. A user will enter his searching keyword in plain text. First, it will be stemmed using Porter

stemmer and then SWBF will be created. All the bits of SWBF is encrypted by the public key σ_{pk} of Pascal Paillier homomorphic encryption key.
Query creation processing took place on end user's machine. After encryption is done, query is sent to cloud for terms matching.

3.6 Similarity Based Terms Matching

After the owner outsourced encrypted data files and index files to CSP, he allows end users to perform searching by giving necessary keys for decryption of end results returned by CSP.
On receiving a request, CCS will access all the files stored and will match all index entries with the incoming query. Here, a notable point is that only corresponding bits of index and query bloom filters are matched.
Matching algorithm for indexed BF and queried BF is different than traditional matching algorithms which are based on standard logical operators. In order to preserve end-to-end privacy, index files have encrypted bits of a bloom filter so as the query. So, as they are encrypted, cloud sever cannot match them by using equal logical operator because on runtime CCS cannot determine whether an encrypt BF bit $\mathcal{BF}_i$ is 0 or it is 1. Also, by using random number r during encryption, encrypting same number $\mathcal{E}_h(n = 0)$ twice will result in two different cipher-texts but decryption of both will result as 0. This property will randomize all the encrypted bits and there is no way to determine which bit is 0 and which is 1.
HE provides a way to perform mathematical operations over ciphered text. As, in our case, bloom filter bits are encrypted with Pascal Paillier key so we can add them and can multiply them with a constant. The idea to determine similarity is that, we will add up index and query bloom filter corresponding bits. The output of a bits addition will result in either 0,1 or 2 Table 5.1. The output result will also be a cipher text by the fact that adding two cipher text homomorphically generates a cipher text.
After addition the $2s$ result in the resultant output represents the matched bits and $1ns$ represents unmatched. Let $\mathcal{T}_b$ represents number of $2s$ and $\mathcal{O}_b$ represents number of $1ns$ in the query BF, then similarity $\mathcal{S}_q$ is calculated as below:

$$\mathcal{S}_q = \frac{\mathcal{T}_b}{\mathcal{O}_b} \tag{3.5}$$

$$Percentage(\mathcal{S}_q) = (\frac{\mathcal{T}_b}{\mathcal{O}_b}) * 100\% \tag{3.6}$$

In order to clarify above let us consider an example. Assume, we are using BF size as 8. Let index BF is:

Table 3.3: Index BF

1	0	1	1	0	1	0	1

and query bloom filter is:

Table 3.4: Query BF

0	1	1	0	1	1	0	1

For the sake of simplicity we are using plaintext BF otherwise they are encrypted in real-time environment. After adding above two 3.3 and 3.3:

Table 3.5: Addition result

1	1	2	1	1	2	0	2

In the output we have : $\mathcal{T}_b = 3$ and $\mathcal{O}_b = 5$.
By using Eq.3.5, similarity is calculated as: $\mathcal{S}_q = \frac{3}{5} = 0.6 = 60\%$

3.7 Communication Cost Reduction

The result generated after adding index and query BF will be encrypted and have to be communicated back to client application in order to decrypt and find out similarity score as in section.3.5. A straightforward approach can be that we send each term of the resultant output and on receiving at client side we decrypt and find out the similarity score. This approach is simple and easy but it has bandwidth overhead as we have to send back all the matched search entries to calling application in the original form. This a lot of data communication between CCS and end user will not only increase the response time but will also increase cost to data owner in terms of budget as more bandwidth means more billing cost as per cloud service oriented architecture.
Let we are using 4bytes(32bit) of BF and 64bit Pascal Paillier key. Suppose owner data file have 1000 keywords. Data owner will create index file and after encryption, index file will be placed on CSS. Now by using Eq.3.3, the size of an index entry is:

$$SizeOf(\mathcal{I}_i) = 32 * 64 = 2048 bits \div 8 = 256 bytes$$

And similarly, the total size of the index file:

$SizeOf(\mathcal{I}) = 256 * 1000 = 256000 bytes/1024 \approx 250 kb$

So, we have to transfer $250Kb$ of data with above configuration. If owner has more index files then returned data will increase as well.
Here, to reduce this communication cost overhead, we devised a method by which we are able to reduce communication cost over $\approx 95\%$.
After addition operation between index BF and query BF as explained in3.5 , we will obtain a resultant output having sum of corresponding bits. Here, we define a polynomial $\mathcal{P}$ as in Eq.3.1. The sum of polynomial $\mathcal{P}$ is the resultant compressed term which we return back to client application:

$$\mathcal{R}_q = \sum_{i=1}^{n} 3^{i-1} \cdot \mathcal{BF}_i \tag{3.7}$$

In the above equation, $\mathcal{BF}_i$ are the Paillier sum of index and query corresponding BF bits and 3^{i-1} are the constants. From Eq.3.7 we get a single compressed term $\mathcal{R}_q$ and its size will be the size of Pascal Paillier key size. That is no matter how big the size of BF is, we will always return same sized compressed search entries which on receiving at client side will be decompressed to compute original Paillier addition result.
Now, lets consider similar scenario which we assumed above for the layman approach. As, after compression we will be sending single entry with size $64bit$:

$SizeOf(\mathcal{I}_i) = 64 bits \div 8 = 8 bytes$

And the total size of the index file:

$SizeOf(\mathcal{I}) = 8 * 1000 = 8000 bytes \div 1024 \approx 8 kb$

Here, we have to transfer $8kb$ of data after compression. If owner increases the size of the BF to reduce false positive rate then communication cost will stay same for the same file having same Paillier key.

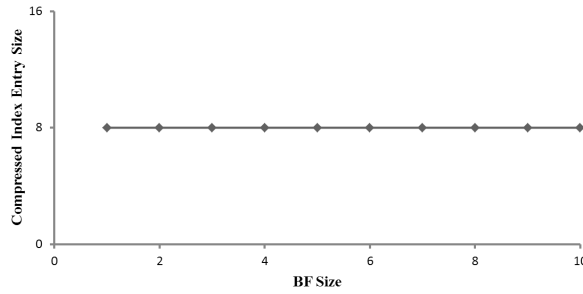

Figure 3.1: Compressed data transfer using 64bit paillier key

Total gain G_{ccr} in communication cost reduction:

$$G_{ccr} = \frac{242}{250} * 100 = 96.8\%$$

On receiving $\mathcal{R}_q$ at client side, it will be decrypted and then decompressed. Via decompression Pascal addition result is regenerated and similarity score is determined. Algorithm for decompression is given below:

Algorithm 3.4: Index entry decompression

Input: A compressed index entry $\mathcal{R}_q$

Output: Matched result $\mathcal{M}_q$ as in Table:A.5

1 $i \leftarrow 0$
2 $b \leftarrow 3$
3 **while** $\mathcal{R}_q > 0$ **do**
4     $i \leftarrow \log_b \mathcal{R}_q$
5     $\mathcal{R}_q \leftarrow \mathcal{R}_q - b^i$
6     $\mathcal{M}_q[i] + +$
7 **return** $\mathcal{M}_q$

Chapter 4

DESIGN AND ARCHITECTURE

This chapter discusses about the detailed architecture of our proposed system, highlights its components and explains information flow. In the rest of the sections of this chapter following major parts of proposed similarity based encrypted search is discussed in detail:

- Proposed system components
- System Architecture
- Information flow in system components

Entities involved in the proposed system are data owner, cloud service provider and end user. Data owner is an entity who wants his confidential data to be stored on cloud storage with the capabilities of privacy-aware searching. Cloud service provider is hosting public cloud storage services for its subscribers on pay-as-you-use model. End user is an entity who will perform searching on the encrypted data stored on the cloud server. End user can submit search queries to CSP and CSP can evaluate user queries and returns results back to user. Meanwhile during query evaluation, CSP cannot learn anything about the query, stored data and matching results.

4.1 System Design Goal

Privacy of the outsourced data can be ensured by applying appropriate encryption before outsourcing to an untrusted service provider. Although, Encryption limits the accessibility of the data and service provider cannot

evaluate search queries on encrypted data. The primary goal of this thesis is to enable user to perform similarity based search on encrypted data without revealing any information about the outsourced data and search query to the service provider and hence service provider should not be able to learn about the outcome of the search queries.

4.2 System Components

In this section, we will give a high level overview of the components our proposed system have and details of each component is discussed. Figure 4.1 shows the components of our developed system. These components are User Application, Searching Application, CCS and CSS.

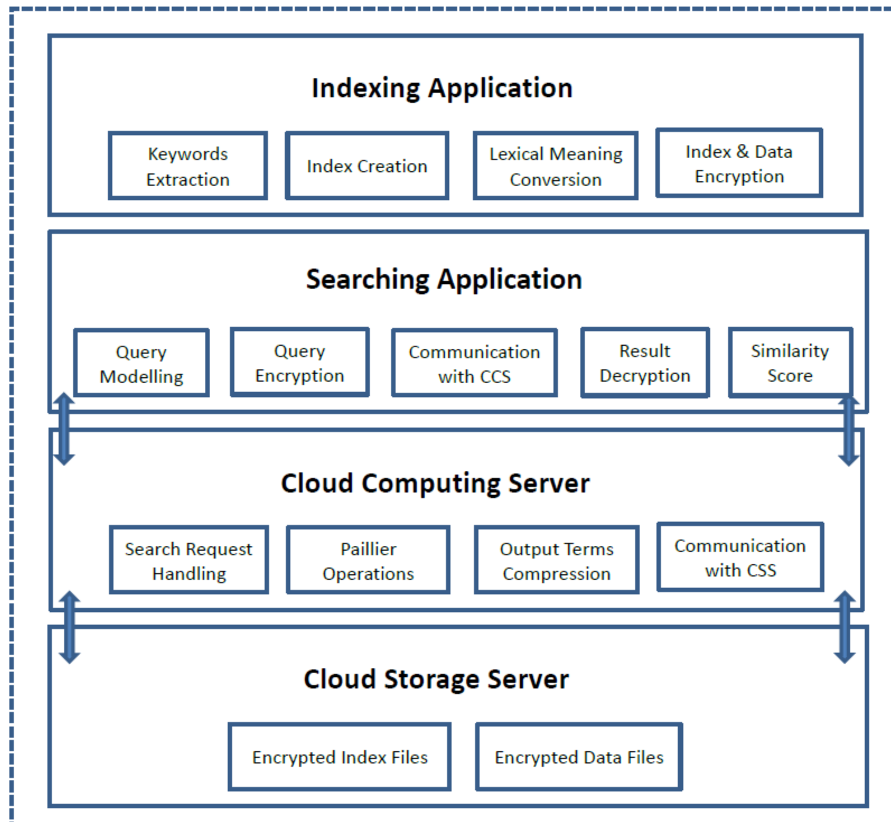

Figure 4.1: System components

Indexing and Searching applications are hosted on user end and considered as trusted components while CCS and CSS are hosted and maintained by a third party and considered as non-trusted components. Communication between

Searching Application and CCS should be considered as non-trusted as all requests routed via open internet.
Indexer Application has no communication with any of the other components. Data owner use this application and generates searchable index files for his confidential data. It is responsible for index creation and cryptographic keys generation. Searching Application communicates directly to CCS. Authorized end users will have access to this application. This application accepts a query with single or multiple keywords from user, creates encrypted BF, sends it to CCS, retrieves encrypted and compressed results. Further, this application decrypts the results, decompresses them and finds out the similarity score. There is a web based application residing on CCS. This application handles user query request, performs Paillier addition operation on encrypted index and user encrypted query and after compressing each term returns the output to Searching Application. Only CCS directly communicates to CSS and accesses index files for each search request.

4.3 System Architecture

In this section we described a high level architecture diagram of our proposed system. Figure 4.2 shows the position of all components and communication among them. Our proposed scheme starts when a data owner wants to outsource a confidential data file while having searching capability.

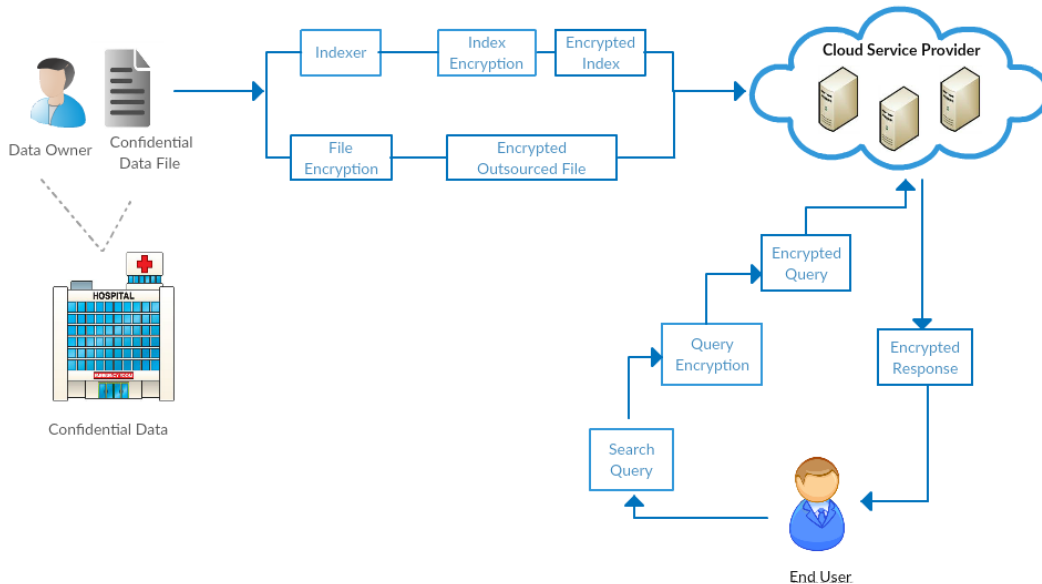

Figure 4.2: Architecture diagram

4.4 Indexing & Data Outsourcing

Data owner will provide a confidential file to Indexing Application. The Indexing Application extracts list of keywords $\mathcal{K}\omega$ from the file and generates index files. *Indexer Application* on its completion outputs encrypted index files $\mathcal{I}_1$, $\mathcal{I}_2$,..., $\mathcal{I}_n$ and public and secret keys σ_{pk} , σ_{sk} and $\mathcal{K}_S$. These keys are used by Searching Application and CCS. Encrypted index files will be outsourced to CSP. Complete workflow for indexing process is show in Figure 4.3

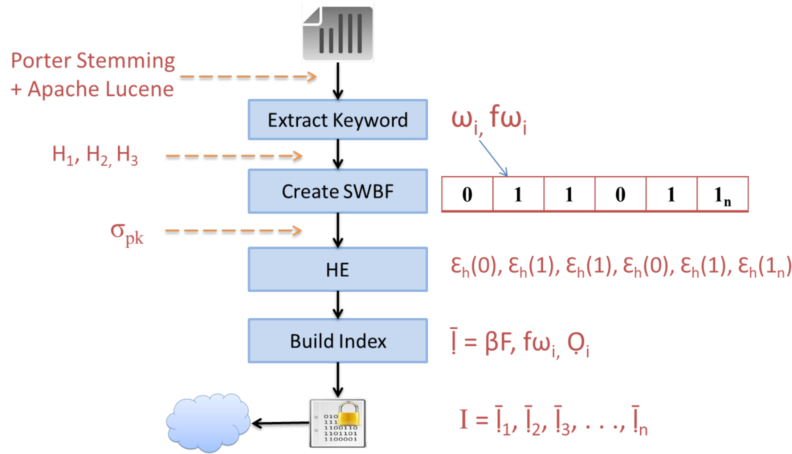

Figure 4.3: Indexing process workflow

4.5 Searching

Searching Application will have σ_{pk} , σ_{sk} and $\mathcal{K}_S$ and starts when a user performs a search query. It accepts a query and creates an encrypted BF similar to that created for indexing by Indexing Application. It will encrypt the query with σ_{pk} and forward it to CCS. CCS being a non-trusted third party only have access to σ_{pk}. By using σ_{pk} it performs addition between index files and user query by adding corresponding bits of index and query BF that is $\mathcal{BF}_{index} i * \mathcal{BF}_{query} i$ where $i = 1 \ldots \mathcal{L}$. After addition for each term, it creates a polynomial $\mathcal{P}$ by using Equation 3.1 adds all terms of the $\mathcal{P}$ to obtain $\mathcal{R}_q$ as in Equation3.7. It repeats this process for each term and returns the output back to Searching Application. Workflow for searching process is shown in figure 4.4.

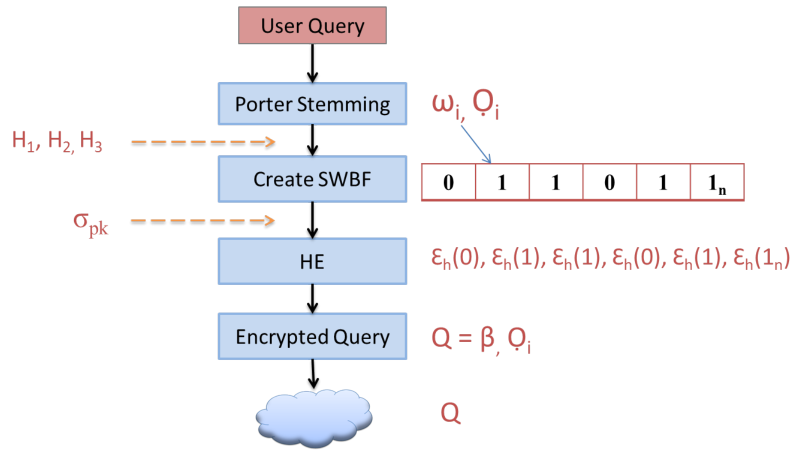

Figure 4.4: Searching process workflow

4.6 Private Matching

Computation is deligated to CSP and cloud computing service evaulates term matching without knowing and revealing anything about user query, index entries and matched search results by using Paillier public key σ_{pk}. Cloud computing service accesses all the index files stored on the cloud storage and performs homomorphic additions with user query. After homomorphic adddditions, each search result is compressed and encrypted response is sent back to user application. The workflow for private term matching process is show in figure 4.5

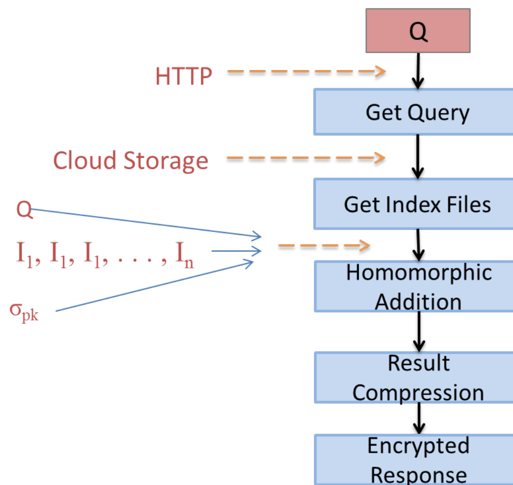

Figure 4.5: Term matching process workflow

4.7 Response Extraction

Each search call made by *Search Application* will be a blocking call and control will not return until CCS responds with $\mathcal{R}_q$ entries. On receiving oblivious response from CCS, *Searching Application* will split the output by comma separation and gets list of $\mathcal{R}_q$. Searching Application will iterate overall $\mathcal{R}_q$ terms and decrypt them one by one using σ_{sk}. Once decryption is done, it applies decompression algorithm to find out original matching result. Via decompression it recreates matching result and computes similarity score using Equation3.5. Workflow for searching process is shown in figure 4.6.

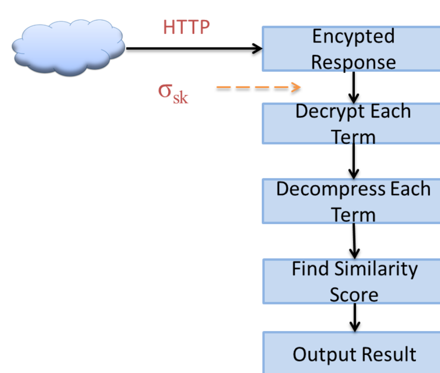

Figure 4.6: Response extraction workflow

Chapter 5

IMPLEMENTATION

This chapter discusses about the implementation details which we used to implement our proposed system in order to demonstrate and test its viability. We choose Google^{TM} as CSP and used its cloud services for the implementation and testing purposes. All system components are developed in *Java* programming language and can be executed on any operating system.

5.1 Google Cloud Platform

Google platform [17] is a set of tools and enhanced cloud based services upon which simple to complex applications can be created and deployed. It provides computing services like Google app engine [18], storage services like blobstore [19] and BigData processing services like BigQuery. As a sum, Google Cloud Platform has following services for its developers:

Table 5.1: Google cloud platform services

Compute	Storage	Big Data
App Engine	Cloud Storage	BigQuery
Compute Engine	Cloud Datastore	Dataflow
Container Engine	Blobstore	Dataproc

Google supports Java and Python a programming languages for the application development. Developers create application on local machine using well known IDE, Eclipse, and deploy them on Google server by using Google provided plugin.

5.2 Indexer Application

For keyword extraction we used Apache Lucene [16] library. Lucene is an enrich text search library with built-in stemming algorithms. *Indexer Application* is a standard *Java 8.0* desktop based application, on start it reads an *XML* file *config.xml* to get the user preferred configurations. A user can mention the length of the keyword to be extracted, the number of hash-functions to be used for BF and cryptographic encryption algorithms specific keys sizes. Below is the structure of the *config.xml* is given in figure 5.1:

```
1  <?xml version="1.0"?>
2  <config>
3      <stopwords>
4          <wordsize>8</wordsize> <!-- Minimum keyword length  -->
5          <word>a</word> <!-- A stop word  -->
6          <word>about</word> <!-- A stop word  -->
7      </stopwords>
8      <bloomfilter>
9          <slidesize>2</slidesize> <!-- Sliding window size  -->
10         <BFsize>3</BFsize> <!-- Bloom Filter Size in bits  -->
11         <hashfunctions>2</hashfunctions> <!-- # of hash functions used for indexing  -->
12     </bloomfilter>
13     <encryption>
14         <paillierKeyLen>64</paillierKeyLen> <!-- Pascal Paillier key length in bits. Should be in power
           of 2  -->
15         <AESkey>0hRU/QuyGv7HHzbJxzrhXA==</AESkey> <!-- AES Key for filename encryption  -->
16     </encryption>
17 </config>
```

Figure 5.1: Configuration file: config.xml

5.2.1 Sliding Window BF Implementation

All keywords are mapped on a separate BF and then after encryption they get index in a file. We used open source, well known fast hash functions to realize BF implementation. These hash functions are:

1. Murmur3 Hashing [20]
2. Jenkings Hashing [21]
3. Zero Allocation Hashing [22]

5.2.2 Encryption

We used open source Pascal Paillier implementation [23]. We optimized the code by pre-calculating certain variable values used for encryption and decryption.*Indexer application* will read encryption related parameters from the *config.xml* and each bit of the BF is encrypted and indexed.

We encrypt file name of the data file with AES key and set the same name for the corresponding index file. In this way we don't have to place encrypted data file name in each index entry.

5.3 Cloud Storage Server

Google's *Blobstore* is used for storing encrypted index files. *Datastore* can be used as an alternative but with the increase of number of files Datastore will become expensive in terms of budget and difficult to manage. While Blobstore is rather scalable and easy to manage.

5.4 Cloud Computing Server

We deployed our cloud computing application on *Google App Engine* by using frontend *F4* instance class. A *servlet* is made available for request handler. Searching Application sends request to this servlet and includes encrypted query as a *HTTP Get* request. CCS server when receives the request, gets the encrypted query and access all the files stored in the *Blobstore*. All the index entries in the index files are added with the query and compressed by using polynomial $\mathcal{P}$.

Chapter 6

RESULTS AND PERFORMANCE EVALUATION

In cloud environment when the confidential data is outsourced after encryption then end user can no longer perform searching and gets desired file from the set of encrypted files. This is mainly due to the fact that standard lookup queries cannot be evaluated on cipher-text by using logical comparison. Although, CSP provides built-in encryption for the outsourced data but it does not ensure end to end privacy as CSP can learn what a user is searching and also as security keys are managed by CSP so it can also learn about the outsourced data.

Certain efforts have been made in order to provide searching capabilities over the encrypted data. Most of the techniques rely on cryptographic trapdoors to enable searching over encrypted data. However, these techniques are not suitable for searching in cloud environment due to the drawbacks of trapdoors.

This chapter aims at highlighting the results and performance of our proposed system. Proposed system is evaluated on two levels. One is indexing and second is searching. Indexing part is evaluated based on execution time of index creation, size of the resultant index file and varialbes effecting overall indexing mechanism like Pascal Paillier key size. Evaluation of searching part inlcudes query evaluation time on cloud side by having certain number of index files and response extraction. Moreover, results of our novel approach for similarity score calculation and communication cost reduction are also presented.

Let's start evaluation by describing the details of our testing envirement.

6.1 Experimental Setup

Dataset aquizition is basic requirment for testing our system. We collected and prepared dataset as describe in Table 3.1. For *Indexer Application*, we used a standard laptop machine with following specs 6.1:

Table 6.1: Machine specs used for indexer application

Spec Name	Spec Value
Machine Type	ThinkPad 430
OS	Windows 7 64-bit
CPU	Core(TM) i5-332M 2.6 GH
RAM	8-GB

Same machine is used for *Searching Application*. For *Cloud Computing Applicatin*, we deploy it on *GAE* and used standard *B4* frontend instance class of *GAE* having following specs Table 6.2:

Table 6.2: Specs for cloud computing application

Spec Name	Spec Value
Cloud Computing Server	GAE
Instance Class	B4
CPU	2.4 GH
RAM	512 MB
Cost	$.20/hour per instance

6.2 Index Generation and Processing

Technical details of index creation process is discussed in section 3.3. Index creation is a compute intensive task as it has to perform keywords extration until phsycial index file creation on the disk. Figure 6.1 shows execution time for the index creation. Figure fig: Index-Generation-Time is based on Table 3.1 with 64bit Pascal key and 3Kb BF size.

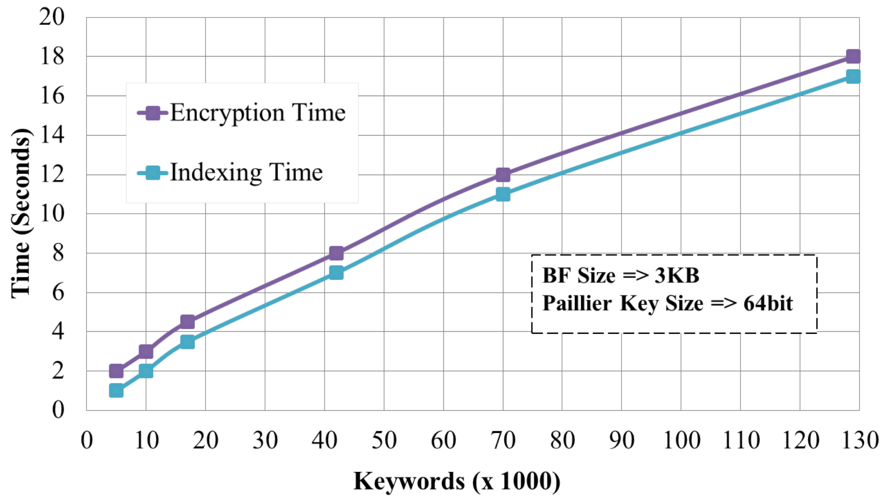

Figure 6.1: Index generation time

Paillier key size and BF size have a huge impact on index file size. As with the increase of BF size, number of indexes will get increased and hence size of an index entry will be increased which will result increase in overall index file size. With the larger BF size, more index entry needs to be processed so execution time will get increased as shown in figure 6.2

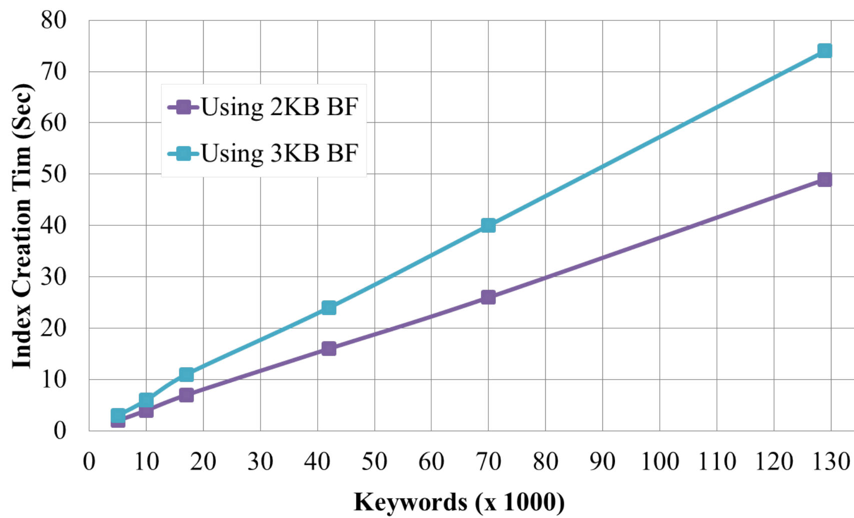

Figure 6.2: BF size impact on index generation time

Security can be strengthened with a larger Pascal key size. However, with the increase of Pascal Key size index file size will also get increased. Following figure 6.3 shows the result of increasing Pascal key size over same data:

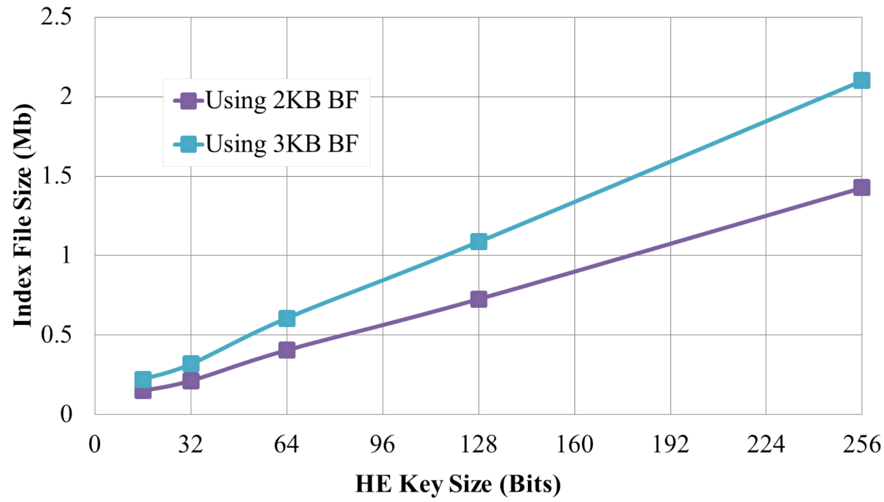

Figure 6.3: Paillier key size impact on index file size

6.3 Searching

Searching process is evaluated in terms of data returned by a query, response time, CPU cycles used to process a search request and the cost($) CSP will charge for search queries. As query creation is a similar process as index creation so we will have similar results as those for indexing process. For private term matching and data returned, our implementation has demonstrated that by using over compression algorithm data returned by cloud computing serive is 96% less and remains constant despite increasing size of BF as shown in figure 6.4.

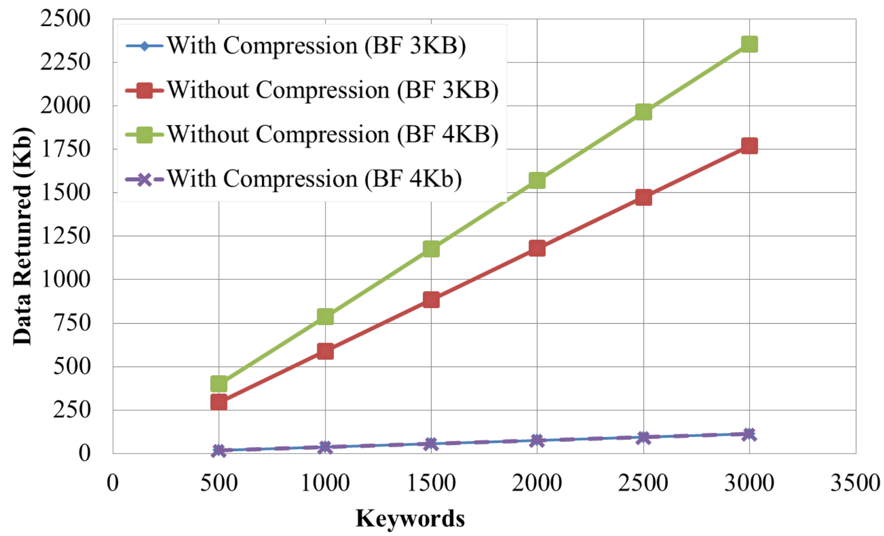

Figure 6.4: Search results compression Vs. uncompressed results

We evaluated our application's response extraction time and results show that its time increases linearly as number of search term increases as demonstrated in figure 6.5.

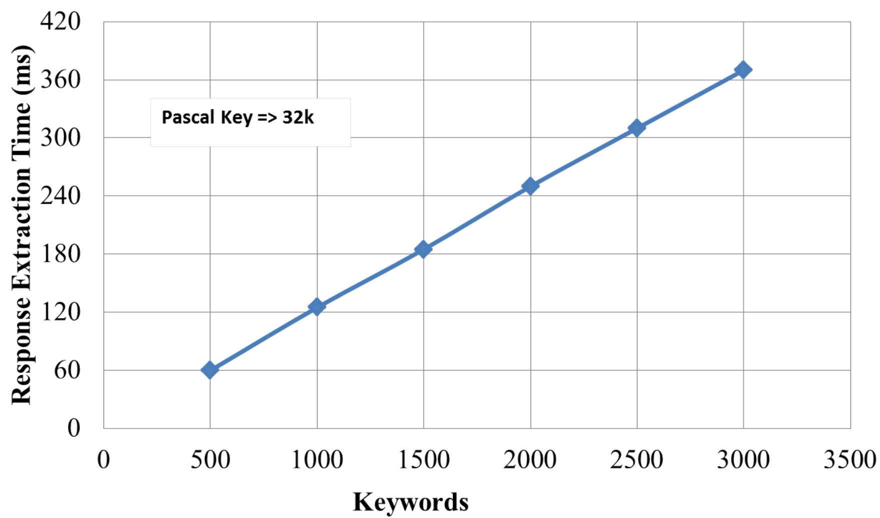

Figure 6.5: Response extraction time

6.3.1 Cost Estimation

We have evaluated cost($) that CSP will charge to data owner for search queries. Our results as in figure 6.6 shown that single keyword searching query with having 500-3500 index entries will cost only 0.000002 to .00002 $ per 1000 similar queries. We obtained this cost from Google App Engine logs for each request. In each log entry *ms*, *cpu_ms* and *cpm_usd* depicts response time, number of clock cycles used by CPU to process the request and cost($) incured for 1000 similar requests.

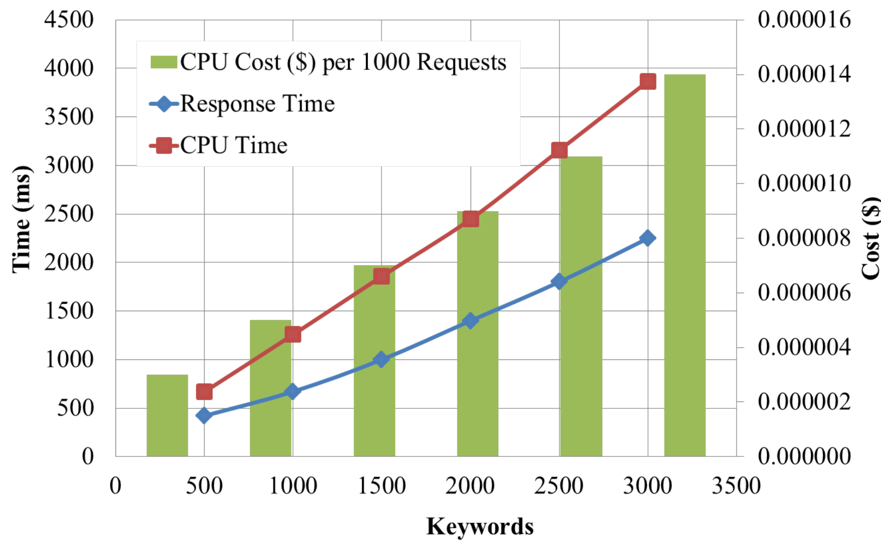

Figure 6.6: Searching cost with single keyword query

We also evaluated perform cost($) analysis with multi-keyword query. And we noticed that number of keywords in a query has huge impact on response time. With the increase of search criteria cost($) increase exponentially as shown in fgiure 6.7. With the increase of each keyword, more search results are created and hence more CPU cycles are required to process them which in turn increase response time and cost ($).

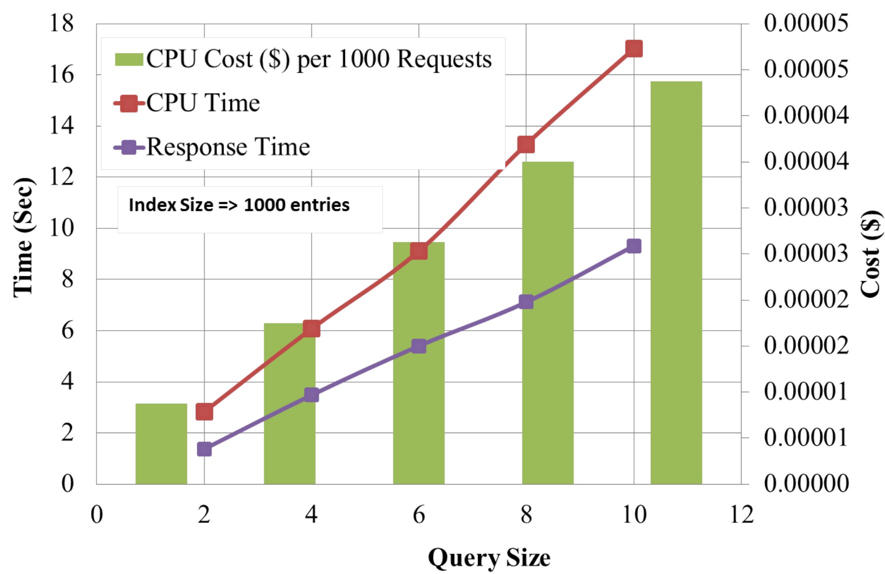

Figure 6.7: Searching Cost($) with multi-keyword query

Chapter 7

CONCLUSION AND FUTURE WORK

This chapter discusses conclusion of the thesis and describes the future work that can be done to extend this research.

7.1 Conclusion

In this thesis, we proposed and implemented a privacy-aware similarity based searching scheme over encrypted data residing in an untrusted cloud domain. Our scheme ensures privacy of user query and by using HE for index files, CSP cannot learn anything about index files, search query and cannot create any pattern even similar queries are performed by different users. Moreover, unlike other techniques, our proposed system performs similarity based searching so that user can get most relevant files even if he does not know the exact keyword. By using our technique data owner will just need to create index files once and all authorized users can query from that index. Our system is not using any trapdoor so end users are not limited to the already defined trapdoors, thus end user can create any arbitrary query of his own choice and can perform searching. We have reduced communicated cost between CCS and searching application over 95% by compressing search result entries.

7.2 System Limitations & Future Work

Current implementation of our proposed system does not support enriched range queries. We have intensions to incorporate range queries with the current implementation. We have demonstrated and tested our system with

comparatively smaller set of data. Hence, with the larger amount of data like petabytes, *Indexer Application* might take significant time from minutes to hours to generate encrypted index. Thus, *Hadoop* framework can be utilized for indexing purpose also *Parallel Computing* can be used to reduce indexing process time.

In the current implementation, *Searching Application* have a lot of work processing load. As this will be a client side and may not be on an optimized machine, we can introduce a TTP which can share from Searching Application.

Appendix A

Performance evaluation: Data Tables

Graphs visualized in chapter 6 are based on following data tables.

Table A.1: Keywords extraction and encryption time

File Size	Keywords (x1000)	Indexing Time	Encryption Time
5	5	1	2
10	10	2	3
20	17	3.5	4.5
40	42	7	8
60	70	11	12
100	129	17	18

Table A.2: BF size impact on index creation time

File Size	Keywords (x1000)	Using 2KB BF (sec)	Using 3KB BF (sec)
5	5	2	3
10	10	4	6
20	17	7	11
40	42	16	24
60	70	26	40
100	129	49	74

Table A.3: Impact of HE key and BF sizes on index file size

Paillier Key Size (bits)	Using 2KB BF (MB)	Using 3KB BF (MB)
16	.15	.222
32	.214	.319
64	.406	.607
128	.727	1.087
256	1.427	2.1

Table A.4: Single-keyword query searching cost

Index Entries	CPU Time (ms)	Response Time (ms)	Cost($) per 1000 requests
500	668	421	0.000003
1000	1257	670	0.000005
1500	1858	1000	0.000007
2000	2447	1400	0.000009
2500	3157	1800	0.000011
2500	3868	2250	0.000014

Table A.5: Multi-keyword query searching cost

Selection Criteria	CPU Time (sec)	Response Time (ms)	Cost($) per 1000 requests
2	2.827	1.372	0.000003
4	6.076	3.484	0.000005
6	9.115	5.406	0.000007
8	13.291	7.119	0.000009
10	17.018	9.315	0.000011

Bibliography

[1] P. Paillier, "Public key cryptosystems based on composite degree residuosity classes," in *17th international conference on theory and application of cryptographic techniques*, p. 223238, Springer, 1999.

[2] K. N. Dan Boneh, Eu-Jin Goh, "Evaluating 2-dnf formulas on ciphertexts," in *Theory of Cryptography*, pp. 325–341, Springer, 1999.

[3] C. Gentry, "Fully Homomorphic Encryption Using Ideal Lattices." https://www.cs.cmu.edu/ odonnell/hits09/gentry-homomorphic-encryption.pdf, 2009.

[4] "Practical techniques for searches on encrypted data." `ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=848445`, 2000.

[5] "Public Key Encryption with Keyword Search." `http://link.springer.com/chapter/10.1007%2F978-3-540-24676-3_30`, 2004.

[6] "Achieving secure, scalable, and Fine-grained data access control in cloud computing." `ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=5961719`, 2011.

[7] "Efficient Search on Encrypted Data Using Bloom Filter." `ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=6828170`, 2014.

[8] NARA and R. Daitch, "Soundex Coding System." http://www.jewishgen.org/InfoFiles/soundex.html, 2007.

[9] "Efficient Similarity Search over Encrypted Data." `ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=6228164`, 2012.

[10] S. J. Golle P. and W. B., "Secure conjunctive keyword search over encrypted data," in *Applied Cryptography and Network Security Conference*, Springer, 2004.

[11] P. Lin and K. S. Candan, "Ensuring privacy of tree structured data and queries from untrusted data stores," in *Information Systems Security Journal*, 2004.

[12] "Privacy-aware searching with oblivious term matching for cloud storage." http://link.springer.com/article/10.1007/s11227-012-0829-z, 2013.

[13] Apache, "Apache PDFBox." https://pdfbox.apache.org/.

[14] R. K. L. Yu S, Wang C, "Achieving secure, scalable, and fine-grained data access control in cloud computing," in *29th conference on information communications, INFOCOM 10, Piscataway, NJ, USA, New York*, p. 534542, IEEE, 2010.

[15] M. Porter, "The Porter Stemming Algorithm." http://tartarus.org/martin/PorterStemmer/.

[16] Apache, "Apache Lucene." https://lucene.apache.org/core/.

[17] Google, "Google Cloud Platform." https://cloud.google.com/.

[18] Google, "Google App Engine." https://appengine.google.com/.

[19] Google, "Google blobstore." https://cloud.google.com/appengine/docs/java/blobstore/.

[20] Apache, "Murmur3 Hashing." https://svn.apache.org/repos/asf/mahout/trunk/math/src/main/java/org/apache/mahout/math/MurmurHash.java.

[21] GitHub, "Jenkins Hashing." https://github.com/vkandy/jenkins-hash-java/blob/master/src/JenkinsHash.java.

[22] GitHub, "XXH Hashing." https://github.com/Cyan4973/xxHash.

[23] K. Liu, "Pascal P Implementation." http://www.csee.umbc.edu/~kunliu1/research/Paillier.html.