Master's Thesis

Secure Indexes for Keyword Search in Cloud Storage

(クラウド記憶におけるキーワード検索のための安全な索引)

2014 年 8月

Supervisor
Professor Hitoshi Aida
(相田 仁)

Electrical Engineering and Information Systems
Graduate School of Engineering
The University of Tokyo

Wanasit Tanakitrungruang
(タナキットルンアン　ワナシット)

37-126949

Abstract

Our investigation challenge is designing an auxiliary data structure to support keyword lookups on encrypted documents stored in the cloud. We propose an extension of “Color Indexes”, a secure indexing technique introduced by Hore et al. The design utilizes false positive search results to hide sensitive information and allows users to adjust security level in trade-off with the performance by tuning system parameters. We focus mainly on improving practicality of the indexes on real world application, especially, analyzing and conducting experiments on the effects of skewed keyword distribution and document sizes. To handle skewed datasets, we recommend using a second level hash function to prevent information leakage from color statistic and using a sampling based technique for more accurate error prediction. The efficiency of our indexes is demonstrated in experiments on English Wikipedia, Reuter-21578, and Enron Email Corpus.

Table of Contents

1. Introduction

The principle of cloud computing is delegating data storing and processing tasks to some service providers — “on the cloud”. The technology enables organizations to build-up IT capabilities on the fly without investing in infrastructure. At the same time, this concept of handing over important information to the cloud also raises a big privacy and security concern.

From the past, the best practice is to encrypt everything before uploading to the cloud, and decrypt the data only after downloaded into a trusted computer, just before using it. This way, people can ensure confidentiality of their information when it’s stored on remote servers outside.

However, data encryption also prevents cloud services from indexing the data and providing search functionality. Even requiring only a small part of data, users have no choices but download everything, decrypt, and find the data on the client side. When the overall size of data grows larger, it becomes impractical for such a described approach.

Therefore, in addition to the security, the system should provide more efficient ways to search and retrieve encrypted documents from cloud storages. This research and other previous works ([3], [9], [10]) have solved this problem by introducing ‘secure indexes’ or an auxiliary data structure that enables keyword lookups on encrypted documents stored on remote servers while minimizing information leakage.

In this work, we introduce our ‘color indexing’ techniques, as an extension to the work of Hore et al [9]. Our improvement provides users even more flexible control over performance and security level of their queries. The improvement also involves making the indexes more practical in large scale text collections.

All analysis and experiments are based on real world data. The remaining parts of the paper are organized as follow: Section 2 introduces related knowledge; Section 3 explains the advancement of secure keyword search techniques and reviews some previous works; Section 4 and 5 introduce the design and analysis of our secure indexes; Section 6 demonstrates our indexes performance by experiments. Section 5 discusses about experiment results and future improvements; Section 6 provides the conclusion.

1.1 Terms and Notations

Term	Symbol	Meaning
Alice	-	A placeholder name for a user or a trusted client computer.
Bob	-	A placeholder name for a server or cloud service provider.
Document	*d*	An entry or a unit of information being retrieve. In practice, the retrieved entry can be a text-document, a web page, a database record, or a file. However, for simplicity, we define a document as a set of keywords.
Word or Term	*w*	A unit of indexable information. We suppose that a document consists of a set of keywords. $d = \{w_1, w_2, w_3, ..., w_n\}$
Document size	*t*, |*d*|	The number of **unique words** in a document.
Pseudo-random function (PRF)	*f*	A assumedly secure pseudo random function (Section 2.1)
Pseudo-random generator (PRG)	*g*	A assumedly secure pseudo random generator (Section 2.1)
Secret key	*k*	An encryption key. Assumedly, *k* is uniformly randomly chosen from all possible keys *K*; thus, the key is secure and unpredictable by attacker.
Color code of *w*	*code(w)*	A randomly computed set of colors represent a word.
Color index of *d*	*I(d)*	A set of colors represent all indexable words in a document. It is created by combining all color code of each keywords
Number of possible colors	*C*	The number of all possible colors.
Number of colors computed for a word	*S*	The number of color computed for a word or the size of color code ($S = \| code(w) \|$)
Number of colors selected for a word	*S′*	The number of color selected from color code of a word ($S' < S$) used by Here et al. [9]
Number of colors used in a query	*Q*	The number of colors selected color code of a word ($Q < S$) used in a document query.

Term	Symbol	Meaning
Collection Size	N	The number of documents in the collection.
Number of correct documents	N_w	The number of documents contain the query keyword or the true-positive results. (Section 5.1)
Number of documents returned in a query	N_q	The number of documents returned by the query. This number includes both true and false positive results. (Section 5.1)
Number of error documents	N_{err}	The number of documents that match a query just by chance whether the documents contains the query keyword or not* (Section 5.1).
Error rate	R_{err}	The ratio of the error documents to the collection size. (N_{err} / N)
Color Density of an index	D	The ratio of colors in the index to the possible colors. (Section 5.2)
Return Probability of a document	P_{ret}	The probability that the document will be included to the returned results because its index happens to match the query by chance. (Section 5.2)

2. Background

2.1 Pseudo Random Function (PRF)

A pseudo random function (PRF) is a function that products a deterministic but seemingly random output from a given key and input. Given a sequence of input/output pairs of a secure PRF, $(x_1, f(k,x_1))$, $(x_2, f(k,x_2))$, $(x_3, f(k,x_3))$, ..., $(x_n, f(k,x_n))$, if the secret key *k* is unknown, there is no possible algorithm to find another pair $(x_{n+1}, f(k,x_{n+1}))$ with a significant success.

Formally, a PRF is a function $f : K \times D \rightarrow R$, where K is the set of possible keys, D is the set of possible *n* bits input and R is the set of possible *m* bits input. For any key $k \in K$, we define a mapping $f_k : D \rightarrow R$ as $f_k(x) = f(k,x)$. The PRF f is consider secure if f_k behaves indistinguishably from a function randomly selected $D \rightarrow R$ functions given *k* is randomly selected from K.

Similar to other cryptographic primitives, the security of PRF is typically defined by a security game between an *adversary* and an friendly entity, which is normally called the *challenger*.

2.1.1 PRF's Security Game

Setup : First, the challenger choose a bit $b \in \{0,1\}$

- If *b* = 0, the challenger randomly selects a function F from all possible $\{0,1\}^n \rightarrow \{0,1\}^m$ functions
- If *b* = 1, the challenger randomly selects a key $k \in K$ and create F as $f_k(x) = f(k,x)$

Queries : The adversary creates an input *x* of $\{0,1\}^n$ and sends *x* to the challenger. After receive the input, the challenger computes $F(x)$ from *x* and return the result to the adversary. The adversary can repeat the query until he acquire enough information about F

Challenge : The adversary selects bit $b' \in \{0,1\}$ as the answer whether F is a real random function or a PRF

Theorem : f is a (t,ε,q) - pseudorandom function, if for every adversary *Adv* with computation time t using q adaptive queries :

$$| prob[Adv\ outputs\ \ b' = 1\ in\ when\ \ b = 0] - prob[Adv\ outputs\ \ b' = 1\ in\ when\ \ b = 1] | < \varepsilon$$

Intuitively, the PRF is secure when the adversary (after taking t computation time and up to q queries) can distinguish the PRFs or the real random functions with only a negligible chance (less than ε).

2.1.2 Pseudo Random Generator (PRG)

A pseudo random function (PRG) is a deterministic function that, from a given key *k* which sometimes called a *random seed*, products output strings that indistinguishable from the real random strings.

Precisely, a PRG is a function $g : K \to Y$ where K is the set of possible keys and Y is the set of possible $\{0,1\}^n$ strings. g is considered secure if $g(k)$ is indistinguishable from a random $\{0,1\}^n$ string when *k* is randomly selected from K.

A pseudo random generator is an important component of Song et al [19] searchable encryption schema, but it does not directly related to our color indexing. If required, a secure PRG g can be simply built from a secure PRF f as :

$$g(k) = f(1,k) \| f(2,k) \| ... \| f(n,k)$$

2.1.3 Pseudo Random Permutation (PRP or Block cipher)

A pseudo random permutation (PRP) or a block cipher is a deterministic function that cannot be distinguish with a random *permutation*.

Practically, PRP is similar to PRF that under the same number of input bits and output bits ($f : X \times K \to Y$ where $X = Y = \{0,1\}^n$) and the function also needs to be one-to-one. This is because every block cipher requires an inverse function (decryption) that map the random output back to its original input ($f^{(-1)} : Y \times K \to X$). With this limitation, the security of PRP is theoretically considered weaker than PRF.

Despite the different, PRP and PRF can normally be used interchangeably. As will be described later, in our problem, PRF is used as a one-way operation of selecting colors (or numbers) from a finite set of possible colors that usually less than 10,000 ($<< 2^{128}$); hence, the number of output bits will always be fewer than the input's. We do not have to concern about this different and use a normal block cipher as our PRF.

2.2 Indexing & Boolean Retrieval

Information retrieval (which usually called 'search') generally means techniques for finding a set of documents inside a larger collection that matches or satisfies the information need of the user. In this work, we focus only on the keyword search problem. That is, given a set of searching keywords, the system finds the set of documents that include (or exclude) those keywords.

A straightforward technique for checking if a document contains searching keywords is linear scanning. This technique is sometime called *grepping* after *grep* command in UNIX operating system. Obviously, the limitation of this approach is the scanning time depends on the size of the document. The larger the document, the longer time takes for the system to scan the whole content. Moreover, the system also wastes computing resource from re-scanning the whole dataset on each query.

Creating an index or *indexing* is a technique to avoid such described linear scan. In real life, instead of flipping through each page of the book and scanning for the interested topics, we can verify whether the book contains the topics of interests just by lookup the *table of contents*. We can think of a document's index as a book's *table of content*. Thus, during the document update or creation, the system should invest some pre-computation to build a keyword lookup table for each document. This index can reduce the time spending on the document retrievals later.

Keyword indexes can be implemented by various kind of data structures (ex. sorted list, b-tree, or tries) depend on the storage space and performance trade-off. However, for simplicity, all the indexes representation in this work are implemented by *bit array*.

Let W be the set of all indexable keywords or terms. We create an index I for each document as a bit array or boolean vector of size $|W|$ (Figure 2.1). Each boolean item in the index donates existence of a keyword inside the document. We suppose there is an efficient way to lookup the location of the boolean of a given keyword. Let $I[w]$ donates the bit that represent keyword w.

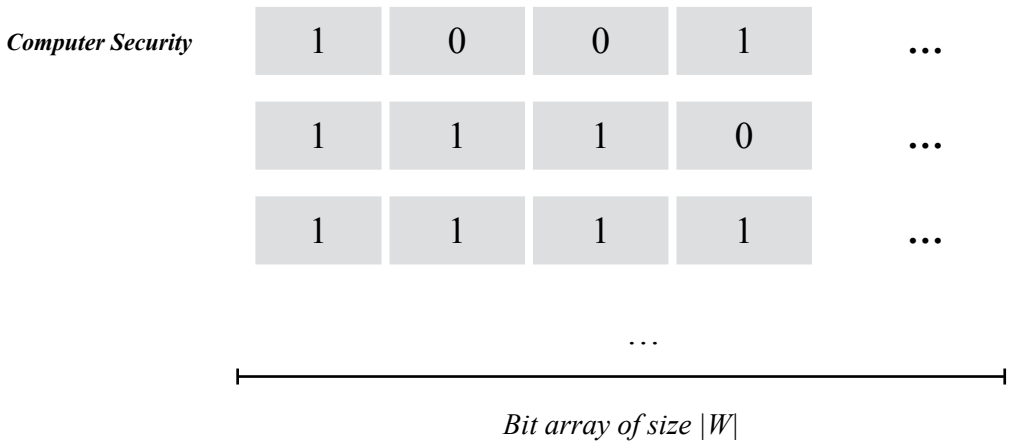

Figure 2.1: Document indexes implemented as bit arrays. Each bit in the array represent existence of a keyword inside a document.

To create an index of a document d, we initialize all bits in side the array as 0's. Then, we extract all the indexable terms $w_1, w_2, w_3, ..., w_n$ from d's content and set the corresponding bits at the locations of each term $I[w_1], I[w_2], I[w_3], ..., I[w_n]$ to 1's.

To check whether a document contains a searching keyword w_q, we simply check the corresponding value of the bit $I[w_q]$. We can also construct more complex lookups using boolean operation (AND, OR, NOT). For example a document that contain keyword X or Y but not Z can be check as "$I[w_x]$ OR $I[w_y]$ AND $I[w_z]$"

In Section 4, both the concept and the structure of our color index is almost similar to the normal keywords index. But instead of indexing the keywords directly, each boolean element in the color index donates existence of a color, which created by applying one-way cryptography technique to the keywords. While providing boolean retrieving features just like the original keyword index, this modification helps concealing the original search keywords from unauthorized eavesdroppers or untrusted cloud.

In addition to indexes, there is another data structure that supports retrieving the list of relevant documents in constant time called *inverted-indexes*. Instead of looking up the index of each document, for each possible keyword, we can pre-compute the list of the documents that contain the keyword called *posting list* (Figure 2).

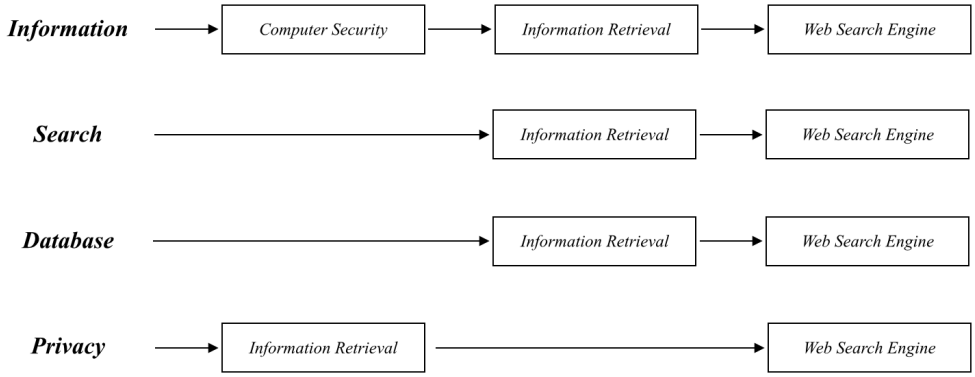

Figure 2: Inverted indexes created from the same examples as in Figure 1

Inverted indexing technique can be used with the basic color indexing (Section 4.1) as a posting list of each color can be created. However, this data structure is incompatible with two-level hashing technique (Section 4.2) since the second level hash distinguishes colors of the same keyword. The design of inverted indexes for color indexing would be introduced in the future works.

2.3 Bloom Filter

Bloom filter [2] is a probabilistic data structure for representing a set of items. It efficiently (O(1) for both running-time and space requirement) supports two operation — adding a new element into the set (1) and testing the existence of a given element in the set with high-probability (2).

Bloom filters are comparable to general implementation of set-type collection using hash table as both data structures provides an efficient way (O(1) in on average) to check whether an element is member of the set. While the traditional hash tables require O(n) space and O(n) worst-case running time in exchange for accurate results, Boom filters allow some errors in other to keeps constant running-time and space consumption.

A Bloom filter's structure consists of *k* independent hash functions $h_1, h_2, h_3, ..., h_k$ and a fixed-size bit array. Initially, all elements in the array are set to 0's. When an item *x* is added into the set, the array's elements at positions determined by hashed values of *x*, $h_1(x), h_2(x), h_3(x), ..., h_k(x)$, will be set to 1's. To determine whether a given element *y* has been added to the Bloom filter, we simply compute *y*'s hashed value and check whether all of the elements at $h_1(y), h_2(y), h_3(y), ..., h_k(y)$ are 1's.

Note that, if *y* has already been added into the Bloom filter, the checking result will always be true (no false-negative error). However, there is a probability that all bits at $h_1(y), h_2(y), h_3(y), ..., h_k(y)$ are set to 1's by some other items; thus, the testing operation may return true even if *y* has not been added to the Bloom filter yet. This false positive probability is depended on the number of hash functions, the number of elements in the set, and the size of bit array.

Let *m* be the size of Boom Filter's bit array. The probability that a particular bit is indexed by a hashed value is 1/*m*. If *n* is the number of the elements in the set, there will be *kn* hashed values calculated. Therefore, the probability that a particular bit inside is 1 is :

$$p = 1 - (1 - 1/m)^{kn}$$

In other for a test result of a value *y* to be true, all bits indexed by $h_1(y), h_2(y), h_3(y), ..., h_k(y)$ must be 1. Since each hashed value $h_i(y)$ is independent, for a long time, many researches, including the original and most followed literatures, calculated the probability for a test result to be true or the false positive rate of Boom filter as :

$$p^k = (1 - (1 - 1/m)^{kn})^k \qquad (2.1)$$

However, an article published by Bose et al. [5] has proved that p^k is, in fact, just a lower bound of the Boom filter's false positive rate. In fact the true error rate $p_{k,n,m}$ can be estimated in range*

$$p^k < p_{k,n,m} \leq p^k(1+O(\frac{k}{p}\sqrt{\frac{\ln m - k\ln p}{m}})) \quad (2.2)$$

In Section 5, all of our designed color indexes are comparable to Boom filters with very large value of *m* and small value of *k*, our calculation follows the traditional calculation in (2.1) as it provides enough accurate estimation.

Eu-Jin Goh [8] adapted cryptography technique to create a secure version of Bloom filter. In their work, instead of using normal hash functions, his Bloom filter adapts a pseudo random function *f* and secret keys $k_1, k_2, k_3, ..., k_k$ (which, in practice, can be derived from a single master key *k*). With this modification, those secret keys are required in order to determine the positions of item *x* in the array as $f(k_1,x), f(k_2,x), f(k_3,x), ...f(k_k,x)$. This way, even when bit array is stored in remote server, there is only the key holder that can compute the locations in the array of an item.

As we will described in Section 5, our color indexes have the structure similar to Eu-Jin Goh's Bloom filters. A color index can be view as a Goh's Bloom filter of bit array size *C* and *S* keys. Each color inside the index is represented by a bit inside the filter. To check the existence of a given keyword, we compute *S* colors of the keyword and check if the corresponding bits are 1's. Our error rate analysis in Section 5 is also based on a statistical model similar to the probabilistic efficiency of Bloom filter.

* The exact error rate involves too complex calculation

3. Related Works

Before introducing previous works and explaining the advancement in secure keyword searches, we begin by describing a naive encryption technique to supports keyword searches, *word-by-word* deterministic encryption, as our starting point.

Note that, throughout this section, we use *Alice* and *Bob* narrative style, which is popular in security related literatures. Alice represents the user or the data owner who desire to store and retrieve documents from the cloud service securely and efficiently. Bob represents the service provider or "*honest but curious people*" on the server side. Without compromising the system or breaking the protocol, Bob tries to learn Alice's sensitive information from any deducible knowledge available to him; and therefore, our security goal is to minimize his chance.

Suppose Alice has a collection of documents $\{d_1, d_2, d_3, ..., d_N\}$, which each document is also a collection of words $d_i = \{w_1, w_2, w_3, ...w_n\}$. To make the encrypted version of the documents, Alice may straightforwardly apply a symmetric encryption (with a secret-key) to each word in each document. An encrypted version of a document would be:

$$E(d) = \{E(w_1), E(w_2), E(w_3), ...E(w_n)\}$$

She sends out the encrypted documents and stored them on Bob's server. Later, when she wants to retrieve documents with a keyword w_q, she simply sends the encrypted keyword $E(w_q)$ to Bob. Bob scans each document for the keyword encrypted keyword $E(w_q)$ and sends the all matched documents back to Alice. Because Bob does not have the secret key, he cannot decrypt neither the query nor the encrypted documents back to the plain text. Alice believes that Bob cannot understand her secret content.

Unfortunately, such a naive technique is not secure in reality. Human languages (ex. English) contain too many patterns and redundancy. There are words that occur numerous times in specific locations (especially prepositions such as 'is', 'are', 'the', and etc). Even when the document is encrypted, each of cipher text of the same word will be the same on every location of the documents ($w_i = w_j$ if $E(w_i) = E(w_j)$). By analyzing the frequencies and locations of those repeated terms, Bob will be able to observe patterns such as 'w_1 *is a* w_2, *the* w_2 *is easy and* w_3'. Given adequate amount of samples, eventually, the content of documents can be decrypted.

The problem can be worst when Bob has already decrypted some of the messages or when he get an access to some of the unencrypted text. For example, he may recognize that the newly added document (the last record in the dataset) is Alice's billing recipe which he also has a copy of it. He can compare the recipe with its encrypted version and use the known words as a context for the further guesses. Once he able to decode any keyword, he can decode the same keyword on every location of every document throughout the whole collection (Ex. he knows that if w_1 = '*contact*', word *w* on other locations where $E(w) = E(w_1)$ must be '*contact*' as well). A leakage of even a small number of documents or keywords could breach the security of the whole system.

3.1 Searchable Encryption Schema

The work of Song et al. [21] is one of the first techniques in cryptography literatures that contributes to solving the problem. Using the described word-by-word encryption together with a PRF and a PRG, they came up with a more secure searchable encryption technique.

Their main idea is using a pseudo random generator *g* (taking Alice's secret key *k* as a random seed) to generate a random sequence $S = S_1, S_2, S_3, \ldots$. A ciphered text is generated by hiding the encrypted words $E(w_i)$ as $C_i = E(w_i) \oplus S_i$. If the sequence *S* is truly random, each C_i will have different random value even if they come from the same word. Thus, this non-deterministic schema is surely more secure. Note that only the key owner can recompute S and use it to recover the original encrypted text ($E(w_i) = C_i \oplus S_i$).

The schema, however, doesn't enable search on the encrypted text. Song et al altered the schema by splitting C_i into two parts. Let *N* be number of bits in C_i and $E(w_i)$ and let *M* (< *N*) be number of bits in S_i. They create the first *M* bits of C_i as the XOR of $E(w_i)[0...M]$ and S_i. Then, they use a pseudo random function *f* with S_i and $E(w_i)[0...M]$ to generate (*N*-*M*)-bit pattern and XOR it with the remaining part:

$$C_i[0...M] = S_i \oplus E(w_i)[0...M]$$

$$C_i[M...N] = f(S_i, E(0...M)) \oplus E(w_i)[M...N]$$

Given an encrypted keyword $E(w_q)$, even without the decryption key, Bob can check if C_i is originally from w_q. First, he XOR $E(w_q)[0...M]$ with $C_i[0...M]$ to obtain S_i'. S_i' is indeed a random value which he doesn't sure if $S_i' = S_i$. He can check the validity of S_i' by compute $f(S_i, E(0...M))$ XOR $E(w_i)[M...N]$ and compare the result with $C_i[M...N]$. If the computed result is equal $C_i[M...N]$ that he knows that $S_i' = S_i$ and C_i matches w_q.

Note that Song et al's schema relies on the *one-way* characteristic of PRF *f* and PRG *g*. **Given $E(w_q)$ Bob can check whether C_i matches w_q . But, if having only C_i, it is impossible for him to guess $E(w_i)$ or w_i.** This one-way characteristic, however, does not stop Bob from indexing locations of $E(w_i)$ once it has been search. After he has learned enough information, he will be able to see relations and patterns. Eventually, he will be able to guess the content like in word-by-word encryption. In other words, the security of the system is leaking overtime when a new $E(w_q)$ is used to retrieve the keyword.

Moreover, Song et al's technique (and word-by-word encryption) requires the server to iterate over the entire content of each document in the collection on every keyword search. Since the technique does not allow indexes to be precomputed.

3.2 Secure Indexes

Over the time, more secure and efficient techniques have been introduced by [3] [9] [10]. Their techniques involve creating ‘secure indexes’. These indexes are somewhat similar to the indexes described in Section 2.2 that they allow the server to check whether the documents consists of a given keyword but in more secure way.

By applying these techniques, Alice creates an encrypted index from her document, then, encrypts the actual content of document with a different cypher. She sends both the index and the document to store on Bob’s server. Later, when she wants to retrieve documents using a keyword, she creates a “*search token*”, which usually an encrypted form of the keyword, and sends it to Bob. By performing some operations using the token on an index, he can determine whether the document contains the keyword encoded in the token. He returns all the documents of which indexes match the query back to Alice. Because Bob is not required (also not allowed) to examine the actual contents, secure indexes are both more secure and more efficient compare to Song et al’s searchable encryption.

Our color indexing schema is also one type of secure indexing techniques. Among previous works in this area, the works of Hore et al [10] and Eu-Jin Goh [9] are most related to our work.

The technique introduced in this paper is an extension to the original color indexes of Hore et al. Their color indexes utilize one-way hashing-like techniques called ‘coloring’ to build secure search indexes. In their system, an index is a set of colors that encode the existence of words but does not disclose their exact identities. The advantage of their design comparing to other previous works is its controllability. The users can balance trade-off between the performance and security level by adjusting three parameters when creating an index — number of possible colors (C), number of colors computed for a keyword (S), and number of colors selected for a keyword (S'). As we will explained in detail later in Section 6.1, our design takes this concept further by allowing the users to adjust the number of colors used for a query (Q) instead of S'. This alteration enables the user to dynamically decide the security level on each query; therefore, provides even more flexible control.

Hore et al. have not described the implementation details of the indexes. We implement the color indexes by adapting the work of Eu-Jin Goh. In his work, Eu-Jin Goh introduce a secure indexing technique based on Bloom filter. As described in Section 2.3, by replacing normal hash functions with one-way PRF, he create a Bloom filter that requires the secret key holder to compute the locations inside bit array of an item. Those locations can be used as a search token that allows the server to look for the query keyword but not something else. Since, our color indexes have the structure similar to Eu-Jin Goh's Bloom filters, our error rate analysis in Section 5.2 uses a statistical model resembling probabilistic efficiency of Bloom filter.

3.3 Homomorphic Encryption

Homomorphic encryptions are special type of encryptions that allow a certain type of operations to be performed on encrypted data. The result of those operations will also be in encrypted form, which can only be decrypted and learned by the key owner. This rare property make homographic encryptions an ideal solution for the future cloud computing.

RSA [18] is a basic example of multiplicatively homomorphism. In fact, it was also this famous public cryptography that introduced the concept of Homomorphism to cryptography researchers for the first time. The paper that mentioned about Homomorphism property of RSA was published by its inventor soon after he realized the potential of this property [19].

Given two RSA cipher messages*, $E(m_1) = m_1^{\,e} \bmod n$ and $E(m_2) = m_2^{\,e} \bmod n$, it is obvious that, without decrypting the messages, users can compute the encrypted multiplication result $E(m_1 m_2)$ simply by multiplying those messages together

$$\begin{aligned} E(m_1)E(m_2) &= (m_1^{\,e} \bmod n)(m_2^{\,e} \bmod n) \\ &= (m_1 m_2)^e \bmod n \\ &= E(m_1 m_2) \end{aligned}$$

Additive homomorphism is more difficult to achieve. Pailler cryptosystem [14] is an example of additive homomorphic encryption (discovered around 10 years later). Given two cipher messages in the system* , $E(m_1) = g^{m_1} r_1^{\,n} \bmod n^2$ and $E(m_2) = g^{m_2} r_2^{\,n} \bmod n^2$, this time, users can compute the encrypted additive result by multiplying the messages

$$\begin{aligned} E(m_1)E(m_2) &= (g^{m_1} r_1^{\,n} \bmod n^2)(g^{m_2} r_2^{\,n} \bmod n^2) \\ &= g^{m_1+m_2}(r_1 r_2)^n \bmod n^2 \\ &= E(m_1 m_2) \end{aligned}$$

* In RSA cryptosystem, a public key is (*e, m*) tuple that pair with a private key (*d, m*). To encrypt a message *m*, the sender compute a cipher $c = m^e \bmod n$ which can be decrypted by the receiver as $m = c^d \bmod n$

** In Pailler cryptosystem, a public key is (g, *n*) tuple that pair. To encrypt a message *m*, the sender compute a cipher $c = g^m r^n \bmod n^2$ where *r* can be any number. For simplicity, we omit the detail of decryption part.

For a long time, cryptographer had discovered only encryption schemas provide either multiplicative or additional but not both of them*. The world first *fully-homorphic encryption,* an encryption schema that both multiplicative and additive, just has been discovered recently encryption schema by Craig Gentry [8].

Homomorphic cryptosystems are still impractical in our opinion. There are many researches on using additive homographic encryption in private database queries. However, most researches focus on quantitative equal or range queries of tabular data (ex. relational databases) rather than full-text search or pattern matching ([4], [15]). The research, which most related to text search, [25] still requires special hardware tuning to achieve high performance query.

Thus, although we believe that Homomorphic encryption is the future of private documents retrieval and many other tasks in cloud computing, we do not adapt the Homomorphic cryptography in our work and focus on traditional secure indexes.

* Note that any computation can be broken down to adding and multiplying bits; thus, having both multiplicative and additive homomorphism means an ability to compute any arbitrary function on the data

4. Design

The main idea behind the design of our secure indexes is using a PRF (with a secret key) to create one-way function of selecting *S* unique colors (from a large set of *C* colors) for word each word. We call the set of colors selected for word *w* as *w*'s *color code* or $code(w)$.

Let document *d* consists of keywords, $w_1, w_2, w_3, ...w_n$, we create *d*'s *color-index* or $I(d)$ by combining the colors of all its keywords :

$$I(d) = code(w_1) \bigcup code(w_2) \bigcup code(w_3) \bigcup ... \bigcup code(w_n)$$

Given a color index $I(d)$ and a color code of lookup keyword $code(w_q)$, we can test whether a keyword w_q exists inside the document by checking if $I(d)$ contains all colors of $code(w_q)$.

Since all color code of *d*'s keywords have been added to $I(d)$ when the indexes was built. If w_q is indeed one of *d*'s keywords, $I(d)$ must contains all colors of $code(w_q)$. The opposite may not be true, however. There is a chance that a combination of color code from other words creates the set of colors that similar to $code(w_q)$. Thus, the testing result can be positive even w_q is not one of *d*'s keywords. Therefore, we can conclude the behavior of color indexes as :

1. If w_q is a keyword of d, $code(w_q)$ must be subset of $I(d)$
2. If $code(w_q)$ is subset of $I(d)$, there is a probability that is w_q a keyword of d

4.1 Basic Color Index

Alice prepares *S* one-way hash functions $h_1, h_2, h_3, ..., h_S$. For each document she wants to store on the cloud, she extracts the document's keywords and computes color code for each keyword *w* as:

$$code(w) = \{h_1(w), h_2(w), h_3(w), ...h_S(w)\}$$

The color selection result must be exactly *S* unique colors. If some hash functions result in a duplicate color, we resolve the color collision by re-hashing util the colors become unique.

According to [2], instead of having *S* hash functions, Alice can prepare only two hash function. And according to [9], those two hash function can, again, be replaced by a PRF with two secret keys (which can also be derived from a single master key). Each hash function h_i is computed as :

$$\begin{aligned} h_i(x) &= h_a(x) + ih_b(x) \\ &= f(x, k_1) + if(x, k_2) \end{aligned}$$

From the color codes, Alice creates an index $I(d) = code(w_1) \bigcup code(w_2) \bigcup ... \bigcup code(w_n)$ for each document *d*. The actual content of the document will be encrypted using a different encryption technique and a different secret key. She then saves both documents and their indexes on the server. *

Note that the index building process is one-way operation. If the pseudo random function is truly secure, it will be difficult for Bob to guess which words a color represent. Therefore, Bob cannot guess the original content easily just by observing the colors of $I(d)$.

In spite of being secure, $I(d)$ enables the server to do a keyword lookup on the encrypted collection. Later, when she want to retrieve documents with a keyword w_q, Alice computes $code(w_q)$ and sends the colors to Bob. Bob iterates through each index $I(d)$ and check if $I(d)$ contains all the seeking colors. He then returns all the matched documents (or the list of the document IDs) back to Alice.

As described earlier, if w_q is one of *d*'s keywords, $I(d)$ must contain all colors of $code(w_q)$ and *d* will be included in the returned results. But, there is a chance that $I(d)$ contains $code(w_q)$ even if w_q is not *d*'s keywords. Thus, a number of wrong documents will be included in the returned results. Alice has to finalize the search on her computer by decrypting the returned documents and filtering out the errors.

Those false positive results are indeed decreasing the performance of the system. Alice wastes her network bandwidth to the unwanted documents, and need to spend her computing resource to refine the final results. For this reason, in this paper, we measure the efficiency of our encryption schema by the proportion between the actual number of seeking documents and the number of documents returned by the query.

However, at the same time, the false positive result also provides a crucial benefit for Alice as it helps hiding her access pattern. Intuitively, the more documents returned by the query, the harder for Bob to distinguish which documents Alice is actually looking for. **Therefore, we can think of the false positive errors as security feature that trade off with the system performance.** And, one important goal of color indexes design is to provide users a flexible control over this trade off by adjusting number of total colors (*C*) and colors per keyword (*S*).

4.1.1 Query colors adjustment

When Alice want to queries using keyword w_q, instead of sending all *S* colors in $code(w_q)$, she can randomly picks *Q* ($\leq S$) colors from $code(w_q)$ and sends those colors to Bob. Bob follows the same described procedure and returns all of the documents of which color indexes contains all of those *Q* colors.

* Ideally, the documents can be stored in a different server separated from the indexes. For example, Alice queries Bob's server for document IDs or their locations where she can download the content later.

The number of errors in returned results depends on *Q*. The more colors Alice includes into the query, the more difficult for indexes of unrelated documents to match all of them. Alice can freely control the number of false positive documents of her query by changing value of *Q*. When she wants her queries to be more secure, she can use small value of *Q*. Conversely, she can increase value of *Q* when she wants more accurate queries.

4.1.2 Comparison to Hore et al's color indexes

The color indexes was originally introduced in the work of Hore et al [10]. We directly inherit the concept of using random colors to represent keywords from their work. However, the schema introduced in Section 4.1.1 is different from the original.

When Alice creates an index *I(d)* for each document *d*, instead of including all colors in *code(w)*, the original technique suggests including only randomly selected $S' \leq S$ colors. During the query, Alice recomputes and sends *code(w)* to the server. The server returns all the documents of which color indexes match *code(w)* by at least by S' colors.

Like other parameters, S' value can be used to control the error rate. Decreasing S' makes the system more secure while increasing S' results in more accurate queries. However, note that this schema does not allow adjusting query colors (*Q*) as suggested in Section 4.1.1. Since the indexes were built with random subsets of *code(w)*, to ensure that the query match all possible subsets, all colors in *code(w)* are needed to be included in the query. **Thus, we cannot have S' and *Q* in the same system.**

Because the value of *Q* can be selected per query as opposed to S' that need to be decided before creating the indexes and can not be changed later, we believe that the schema in Section 4.1.1 is more suitable for the real application.

4.1.3 Basic Color-Index's limitation

Color indexes rely on the one-way property of a PRF. If the PRF is truly secure, color encoding will be one-way process; there is no efficient algorithm that able to transform the colors back to their original word. Thus, having only a color index alone, it is difficult for attackers to reconstruct the content or the meaning of the document.

However, this basic color indexing has a several limitations that allows Bob to take advantage of some open knowledge for deducing the keywords or the information inside the documents.

First, Bob can compare color indexes of two documents and recognize the similarities between their contents. For example, if Bob found that two color indexes $I(d_a)$ and $I(d_b)$ have more than 90% of colors in common. Even Bob does not know the meaning of the colors nor the content of d_a and d_b. He can tell from the colors that d_a's and d_b's content are somewhat similar, and perhaps they are in the same category or related to each other.

Second, he can compare the frequency distribution of the keywords and colors. Ideally the occurrence of keywords should be uniformly distributed and the number of documents contains each keyword should be around the same. However, in practice, the distribution of each keyword is visibly skew

depend on the type or category of the documents. Therefore, comparing keywords frequency with colors frequency, Bob may be able to guess keywords that some colors represent.

For example, Bob know that a frequently used term like '*the*' appears in almost every document while a very rare term like '*hardihood*' (an old-fashion term for '*boldness*' or '*daring*') appears just only in 39 pages of 4.4 millions Wikipedia articles. He also observed that 'red' is the most popular colors, which included in almost every indexes. From these two information, he could guess that 'red' may be the color that represent word 'the'.

4.2 Two-level hashed Color Index

Alice prepares *S* random hash functions $h_1, h_2, h_3, ..., h_S$ from a pseudo random function and secret keys as for basic color indexing. She then prepares another one-way PRF f_2 and sends this function to Bob.

Before creating an index for document *d*, she also prepares a unique ID, $docID_d$, for the document This can be randomly generated or if the document already have a unique ID field or attribute, she can use it as the $docID_d$ as well.

During the index creation, she extracts the document's keywords and computes the colors of each keyword as $\{h_1(w), h_2(w), h_3(w), ..h_S(w)\}$ like creating a basic color index. But this time, she uses the function f_2 with $docID_d$ to transform the colors one more time :

$$code'(w) = \{f_2(docID_d, h_1(w)), f_2(docID_d, h_2(w)), f_2(docID_d, h_3(w)), ..., f_2(docID_d, h_n(w))\}$$

The color index will be created by combining the colors of transformed color codes of each keyword $code'(w)$ together :

$$I(d) = code'(w_1) \cup code'(w_2) \cup code'(w_3) \cup ... \cup code'(w_n)$$

Alice then saves the index, $I(d)$, along with the document's ID, $docID_d$, on Bob's server. We call the color indexing technique from Section 4.1 using $code(w) = \{h_1(w), h_2(w), ..h_S(w)\}$ as *one-level hashed color-indexing.* And we call the color indexing technique in this section using transformed color code $code'(w) = \{f_2(docID_d, h_1(w)), f_2(docID_d, h_2(w)), ..., f_2(docID_d, h_n(w))\}$ as *two-level hashed color indexing*.

As the name suggested, the each color of $code'(w)$ is created by applying one-way hash function twice. The second level hashing make the color code of the same word becomes different on each document.

When she wants to query documents with a keyword w_q, Alice computes the first level color code of w_q ($code(w_q)$) and sends it to Bob. Having both indexes associate with their IDs, Bob can iterates through each ($I(d)$, $docID_d$) pair, computes the second level color code $code'(w_q)$ from $code(w_q)$

and $docID_d$. He checks if $I(d)$ contains all $code'(w_q)$'s colors. He then returns all the matched documents (or the list of the document IDs) back to Alice.

4.2.1 Two-level hashed color indexing limitation

Certainly, the two level hashed color index doesn't provide perfect protection against attackers. Since both document IDs and indexes are stored on Bob's server. He can transform two-level hashed indexes back to one-level hashed form by trying all possible colors. Precisely, for every possible color *c,* he can check whether $I(d)$ has $f_2(docID_d, c)$ or not. With certain amount of computation, this brute-force technique will eventually reconstruct the original indexes.

Let *D* is the density of a two-level hashed color index calculated as number of colors in the index divide by number of possible colors. Since Alice has to compute f_2 only for colors in the index while Bob has to compute the function for all possible color space. He needs to spend $1/D$ times more computation than Alice. The system can also be designed to discourage the attack by using a *slow hash function* (Ex. *PBKDF-2*, *bcrypt*, or *scrypt*) as f_2. This technique is widely used in password-based key derivation systems.

4.3 Other possible improvements

The second hashing in two-level hashed color index helps preventing the attackers from recognize the keywords by looking that the distribution of colors, however, the schema has not consider the effect of having document of different sizes.*

As we will see from the data in Section 6, most collections consist of documents with various number of keywords. The number of colors inside an index depends on the number of keywords in the source document. The more number of keywords needed to be encoded into the index, the more colors were likely to be included. Knowing this fact, attackers can distinguish very large (or very small) documents in the collection by simply looking for high-density indexes.

The variety of documents sizes also affects query performance. Those high-density indexes of large documents are likely to match color queries and be included as false positive results. Thus, having a lot of very large documents (the heavy-tailed distributions in Section 6.1) degrade our system performance.

The problem of document size diversity is still an open problem in our research. In this section, we introduce two potential ideas, which are still impractical however.

* Throughout this paper, we follow the notation described in Section 1.1. The document size means the number of **unique words** rather than the actual space requirement. Thus, in our context, 'large' documents means documents with a lot of unique terms and 'small' document means documents with fewer unique terms.

4.3.1 Including random keywords

When building indexes, Alice can add some random colors into the low-density indexes to make all the indexes having around the same number of colors. Adding random colors into the indexes is practically equal to adding random keywords to the documents. Now, Bob cannot distinguish size of documents from the number of colors in their indexes.

Note that we can only add random colors to the indexes, but we cannot remove the original colors. Doing so, the resulted indexes will not include all necessary colors and the query can miss target documents (Section 4.1.2).

The obvious downside of this technique is it increase overall color density of color indexes. The number of colors on small documents' indexes need to be raised to the same level as the largest documents'. As described earlier, high-density indexes are more likely to match unrelated queries. Thus, this approach has a significant cost to system performance.

Another concern about practicality of this approach is the number of color (equal to the largest document) need to be decided from the beginning before building indexes. And the number cannot be increased (or decreased) later. Thus, once the indexes were built, it will be difficult to add a new document of size larger than the previous largest one to the collection.

4.2.1 Split large documents into multiple blocks

Another possible solution is dividing the large documents into several blocks around the same size and create an index for each block separately. By carefully choosing the size of those blocks we can control the color density in each index.

However, splitting the content of the document could conflict with propose of keyword search. For example, users may query documents using a name or title as the keyword, the returned results will include only the headers or the first part of documents while the information that users needed may be on the following parts which do not contain the title.

Furthermore, splitting a document into multiple parts by the number of uniques terms is not a straightforward task like splitting the document by actual size. Even the actual document size (number of bytes) and the number of keywords are highly correlated, the occurrence of each keyword may not be uniformly distributed throughout the document.

Thus, in order to break the content of large documents into equally keyword-ed blocks, some other information retrieval or natural language processing techniques are needed to applied. We leave this consideration to the future work.

5. Analysis

As mentioned in Section 4, an important advantage of color indexing technique is it allows users to adjust system security (in trade-off with performance) by taking advantage of the false positive query results. The false positive error rate is related to three configurable parameters — the number of colors (*C*), the number of colors per keywords (*S*), and the number of colors using in document queries (*Q*).

In this section, we analyze the effect of each parameters and provide some basic calculation for estimating the value of each parameters during the indexes design.

5.1 Retrieval Efficiency

We define efficiency of a query for a keyword w_q on indexes of N documents as the ratio between the number of the actual documents containing w_q (N_w) and the number of the results returned by the query (N_q). This value indicates the overhead of the document retrieval as the percentage of unused documents that will be transferred and filtered by the clients.

$$Efficiency = N_w / N_q$$

The set of returned documents is the union of two sets. The first one is the set of the documents contain *w* (*A*) and the second one is the set of documents which of their color indexes happen to match the query by chance (*B*). Hence, *Nq* is the size of A u B.

$$N_q = |A \cup B| = |A| + |B| - |A \cap B|$$

Let $N_{err} = |B|$ be the number of the document in the second group. Then, we have:

$$N_q = N_w + N_{err} - N_{err}(N_w / N)$$

and

$$Efficiency = N_w / (N_w + N_{err} - N_{err}(N_w / N)) \qquad (5.1)$$

N_{err} is the system property that can be adjust by choosing *C, S* and *Q* (explained further in following sections). However, as we can observed from equation (5.1), the efficiency also depend on the number of actual matches (N_w) itself. In other words, **different keywords queries on the same system setting yield different efficiencies**.

For example, if a user queries for '*world*' (807,955 referred articles in Wikipedia), ten thousands unrelated documents overhead would be reasonable. In contrast, if the user queries for '*tokyo*' (46,810 referred articles), he would not be satisfy if with the same error rate.

Ideally, the design of our color indexes enables the system to keep the efficiency unchanged and predictable by adjusting value of Q for different keywords (Section 4.1.1). When query for a keyword with lower N_w (ex. '*tokyo*'), the system can automatically increase Q to reduce the N_{err}. At the same, the system can use smaller Q when the larger error rate is acceptable (ex. '*world*'). In practice, however, such a describe scenario could be possible only if the system is able to estimate N_w before selecting Q and submitting the query. The techniques for estimating the number of keyword matches are beyond our scope.

While the value of Q can be decide during the query, C and S need to be decided before the building the indexes and cannot be changed afterword.

Straightforwardly, the system efficiency can be increased simply by increasing C to reduce the collision of colors. This adjustment is equivalent to expanding the size of hash table or Bloom filter's bit array. Note that, increase in the number of possible colors also means increase the storage requirement for storing an index as bit array. Thus, in practice, when most of the indexes have very low color density, we would consider adopting other sparse data structure (ex. a list of colors) to save the storage cost.

Increasing in S normally results in higher color density (more colors inside an index). Thus, it increases space requirement for indexes in sparse formats. It also increases the error rate when query indexes with the same of value of Q. But, since the maximum value of Q is limited by S ($Q \leq S$), the increase in S enables the indexes to serve more accurate queries.

5.1 Index Density and Error Rate

For comparing the error rate later on collections of different size, let define error rate R_{err} as number of errors divided by collection size (N_{err} / N).

During indexes building, we randomly compute select S unique colors (from C possible colors) for each word. Thus, the probability that $code(w)$ contains a given color is equal to S / C. We repeat this selection for all words in the document. Suppose the document has t unique terms, the probability that the document's color index contains a given color is:

$$D(C,S,t) = 1-((C-S)/C)^t \tag{5.2}$$

We call D the density of the color index or the ratio of colors in the index and possible colors, which depends on C, S of the system and t of the document. The expected number of colors in the document's index can be calculated as the expected value from C binomial experiment with probability equal to D — $D \times C$. Note that color density of the basic color index (Section 4.1) and two level hashed index (Section 4.2) of the same document will be the same.

The color density value in equation (5.2) resembles the probability that a certain bit in Bloom filter is 1's in equation (2.1). In fact, our color index implementation is comparable to a Bloom filter with bit array size C and S hash functions, especially the secure Bloom filters of Goh et al. [9] that also adopts a pseudo random function and a secret key to create one-way computation. In spite of that, the different of two equation comes from the fact that color indexing selects exactly S unique colors for each word instead of using S repeatable hashes.

When we retrieve documents with Q ($\leq S$) colors, a document will match the query of if its color index contains all the Q seeking colors. Similar to the Bloom filter false positive error, we can further calculate the probability that a document will match the query and be returned as a result by :

$$\begin{aligned} P_{ret}(C,S,Q,t) &= D(C,S,t)^Q \\ P_{ret}(C,S,Q,t) &= (1-((C-S)/C)^t)^Q \end{aligned} \tag{5.3}$$

The expected number of random documents returned by a query of C, S, and Q on the collection can be calculated by summarizing all the return probability of document size (t_i) of each document in the collection.

$$\begin{aligned} R_{err} &= \sum_{i=1}^{N} P_{ret}(C,S,Q,t)/N \\ R_{err} &= \sum_{i=1}^{N} (1-((C-S)/C)^t)^Q/N \end{aligned} \tag{5.4}$$

The calculation (5.4) in could be costly when there are too many documents in the collection. To estimate the value of D_{err}, it is desirable to apply the statistic mean of document size to the equations (5.3) and (5.4). Let $\overline{t}$ is the average number of terms per documents.

$$R_{err} = (1-((C-S)/C)^{\overline{t}})^Q \tag{5.5}$$

In fact, this estimation is frequently used in the previous literatures. Unfortunately, in practice, the result of equation (5.5) is invalid. The distribution of document size t in actual text documents collection can be very skewed, as oppose to a normal or uniform distribution that we cannot use statistic mean or median as a reliable point representation. Thus, we need some other forms of calculation for estimated the error rate.

5.2 Error Estimation for Skewed Distributions

To correctly predict the error of color indexes, the shape of document size distribution need to be considered. In this section, we introduce two possible technique to make more accurate prediction and avoid running experiments on the large dataset on equation (5.4).

5.2.1 Fitting probability distribution

Let suppose that we know the underline probability distribution of document size in the collection. Let the probability that a document in the collection has t unique keywords be *P(t)*. The expected error rate can be calculated as an expectation of weighted random variable:

$$\begin{aligned} R_{err} &= \sum_{i=1}^{N} P_{ret}(C,S,Q,t) \times P(t) \\ R_{err} &= \sum_{i=1}^{N} (1-((C-S)/C)^{t})^{Q} \times P(t) \end{aligned} \tag{5.5}$$

As we will see in Section 6, most collections have a long-tailed document size distribution. As it has happened in information retrieval and natural language researches, let suppose that the underline distribution follows *power-law [6]*.

$$p(t) = Kt^{-\alpha}$$

Where K and α are some positive constants. As we will show in Section 6, this assumption is actually true when t is larger than some value.

$$R_{err} = \sum_{i=1}^{N} (1-((C-S)/C)^{t})^{Q} Kt^{\alpha} \tag{5.6}$$

The estimation in the equation (5.6) is still not very enlightening or useful. As we already know, the error rate can be generally reduced by increasing *C, Q* and reducing *S*. However, to calculate the exact error rate for the setting, we still need to measure the distribution parameters K and α, which is also an expensive computation.

5.2.2 Distribution sampling technique

Instead of consider the size every single document as in equation 5.5, we introduce a more practical sampling technique that can be used to estimate the error rate of any given *C, S* and *Q* configuration by the following

1. Randomly select *n* number sample documents $d'_1, d'_2, d'_3, ..., d'_n$ from the collection. We then measure the size (number of unique keywords) of each sample $t'_1, t'_2, t'_3, ..., t'_n$. The information of sample sizes should be small enough to be saved on the client side.

2. To estimate R_{err} of a certain *C, S, Q* setting, use the equation (5.4) to calculate error rate of the query in the sample documents R'_{err} and multiply the result by N / n.

$$R'_{err}(C,S,Q) = \sum_{i=1}^{n} P_{ret}(C,S,Q,t'_i) / n$$

$$R'_{err}(C,S,Q) = \sum_{i=1}^{n} (1 - ((C-S)/C)^{t'_i})^{Q} / n \quad (5.7)$$

The intuition behind the estimation in equation (5.7) is we represent the distribution of *N* document collection with n sampling points. Each of those *n* random documents roughly represents *N/n* documents of the same size.

6. Experiment

In this section, we show color indexes' efficiency on real text collections. There are three different text collections used in our experiment, *English Wikipedia* (Wikipedia) (Section 6.1), *Reuters-21578* (Section 6.4.1), and *Enron Emails* (Section 6.4.2). However, we mainly focus on Wikipedia as duo to its size that significantly larger than the two others.

We show the flaw of one-level hashed indexes using Wikipedia dataset. We also demonstrate more secure indexes' using two-level hashing technique (Section 6.2). On the next part, we show efficiency of different color indexes' configurations on Wikipedia. The experiment result is also compared our naive prediction and sampling-based estimation (Section 6.3).

In the last part, we compare the measure statistics of other two datasets and compare them with Wikipedia. We describe how characteristics of different text collections can effect the efficiency of color indexes (Section 6.4).

6.1 Wikipedia Dataset

We use 2 October 2013 version of xml dump English Wikipedia (downloaded on 5 November 2013 [24]). The dump file in xml is uploaded to Hadoop cluster [1] and extracted only text content with a library provided at [23]. We then run another Hadoop job to filtered-out all '*redirect*' and '*disambiguation*' pages and selected only 'articles' pages into our collection.

For preserving unprocessed natures and the simplicity of the experiment, we simply extract indexable terms and keywords by breaking text content on every non-word characters*. The segmented keywords are converted the words into lowercase form and filtered out again using MySQL's stop words list [13].

Important statistics that directly related to our indexes' performance are keywords frequency and number of unique terms in each document. We measure those two statistics on the dataset by counting number of documents each keyword appears in (Keywords Distribution — Figure 6.1, 6.2 and Table 6.1) as well as number of keywords appear in each document (Document Size Distribution — Figure 6.3, 6.4 and Table 6.2).

From Figure 6.1 and 6.2, the keywords distribution seem to be *double exponential*. The most frequently used words, such as 'references', 'links', and 'externals', have more that 500,000 refer documents each while most of the keywords (around 90% out from 9.6 millions unique term) have only less than 10 refer documents. This skewness directly effect the color distribution in Section 6.2.

* Regular expression "\W"

Figure 6.1: Histogram of keywords distribution of **1000 randomly selected terms** in Wikipedia sorted by frequency. Each point in the figure represents a keyword and y-axis represents the number of documents containing the keyword.

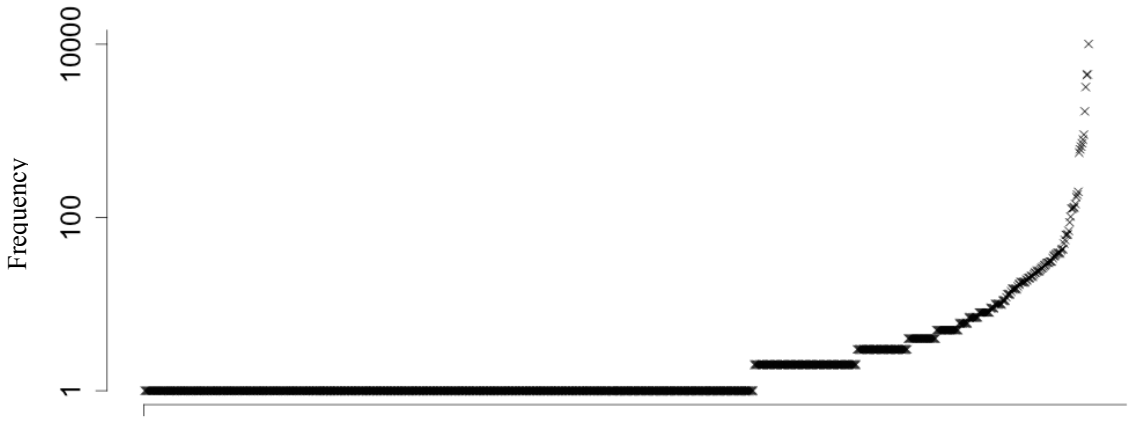

Figure 6.2: **Log-scaled** histogram of keywords distribution of **1000 randomly selected terms** in Wikipedia sorted by frequency. Each point in the figure represents a keyword and y-axis represents the number of documents containing the keyword.

Minimum	1st Quartile	Median	Mean	3rd Quartile	Maximum
1	1	1	50.3	3	176,700

Table 6.1: Statistic summary of overall Wikipedia's keywords distribution (9,605,342 unique terms).

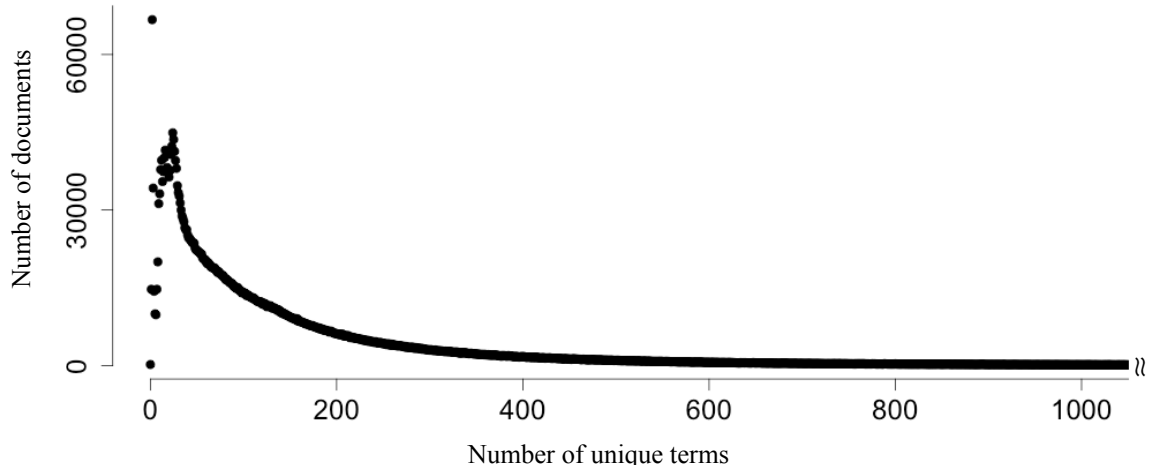

Figure 6.3: Histogram unique terms count on Wikipedia in **normal scale**. The x-axis represents the document size or the number of unique terms in the document and y-axis represent the number of documents of different size ranges.

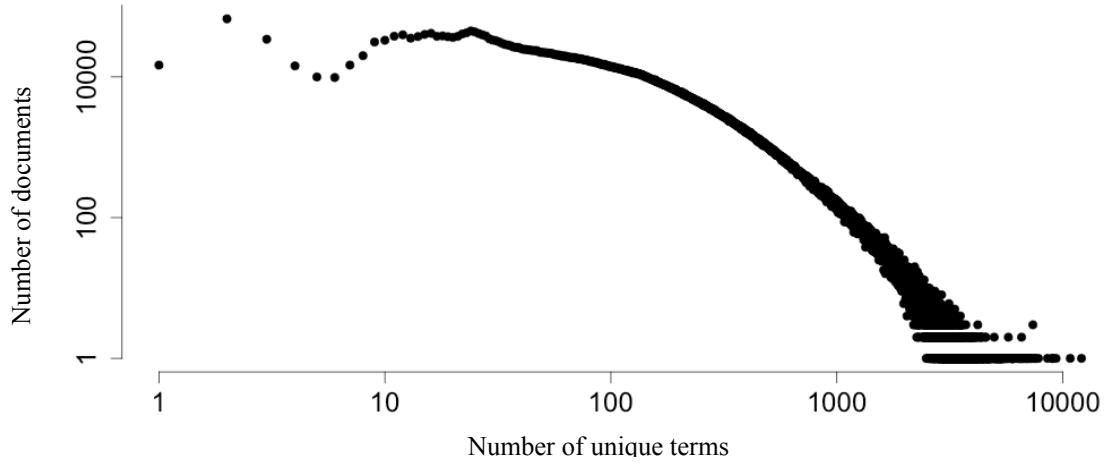

Figure 6.4: Histogram of unique terms count on Wikipedia in **log-log scale**. The x-axis represents document size or the number of unique terms in the document (in log scale) and y-axis represent the number of documents at different sizes (in log scale).

Minimum	1st Quartile	Median	Mean	3rd Quartile	Maximum
0	33	86	159.5	191	12,100

Table 6.2: Statistic summary of overall Wikipedia's document size distribution (4,442,789 documents).

Figure 6.3 and 6.4, Wikipedia has unexpectedly large number of empty articles (the left part of the graph). For simplicity, we keep using all of the articles without filter as we suppose it may happen in other large document collections (Enron Emails also has a large number of empty documents).

It is clearly visible that Wikipedia's document size does not follow well-known normal distribution nor uniformly distributed. The distribution seem to follow a some kind of heavy-tailed distributions. We predict that the document size follows a *power law* distribution ($P(x) = Kx^{-\alpha}$) from its almost straight curve in log-log scaled histogram (when the documents size > 100).

We test the power law hypothesis by a *goodness-of-fit* test described in [6] (using R package available in [16]). From the result, when $X_{\min}$ around 400, Wikipedia's document size fits power law distribution with *p*-value around 0.8 (highly fit). This confirms that, for the documents of more than 400 unique terms (around 20% of the collections), we can apply the power law's distribution to estimate the number of documents. However, we still not be able to find distribution that fit the most part of the distribution.

6.2 Color Distribution

Our first experiment on Wikipedia demonstrate a flaw in color distribution of one-level hashed color indexes or Hore et al.'s original indexing schema described in Section 4.1.3.

In this experiment, we created two sets of one-level hashed indexes for Wikipedia using by two different encryption keys on *C*=8192, *S*=4 setting (Section 4.1). On each set of indexes, we counted the number of occurrence of each color. The distribution of colors on each indexes are shown in Figure 6.5 and 6.6.

As we could expected, the color distribution on two different set of color indexes (using two different keys) are seemingly different. However, comparing statistic summary of the two distributions (Table 6.3), the underline statistic values of both distribution are actually the same. This is because, even using different keys, the colors of both indexes sets are generated from the same keywords distribution.

We further demonstrate this relation by comparing sorted color distribution of both indexes sets in Figure 6.7 and 6.8. Two seemingly similar distributions in the figures are created by ranking the colors in each distribution based on their frequencies. In other words, after sorting each entry by its value in y-axis, the distribution in Figure 6.5 and 6.6 are actually the same.

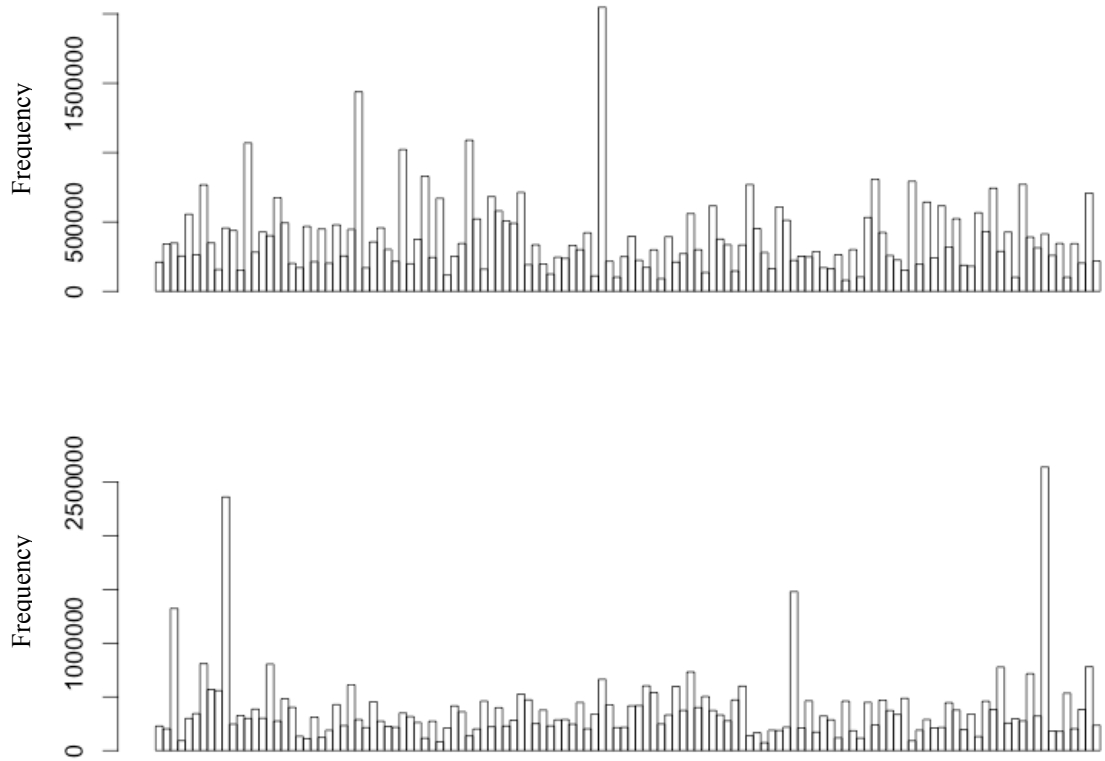

Figure 6.5 (top) histogram of color distribution on *C*=8192, *S*=4 one-level hashed color indexes using key derived from string ‘test’ and Figure 6.6 (bottom) histogram of color distribution on the same color indexes setting using key derived from string ‘test2’. (Duo to the difficulty in visualize all 8192 colors, we show 128 randomly sampled colors in the histograms)

	“test”	**“test2”**
Minimum	52862	61731
Median	285944	285730
Mean	379384	378975
Maximum	4077273	4074589
SD	358245	359255

Table 6.3 Statistic summary of color distribution on *C*=8192, *S*=4 on level based color indexes of secret keys derived from strings ‘test’ and ‘test2’.

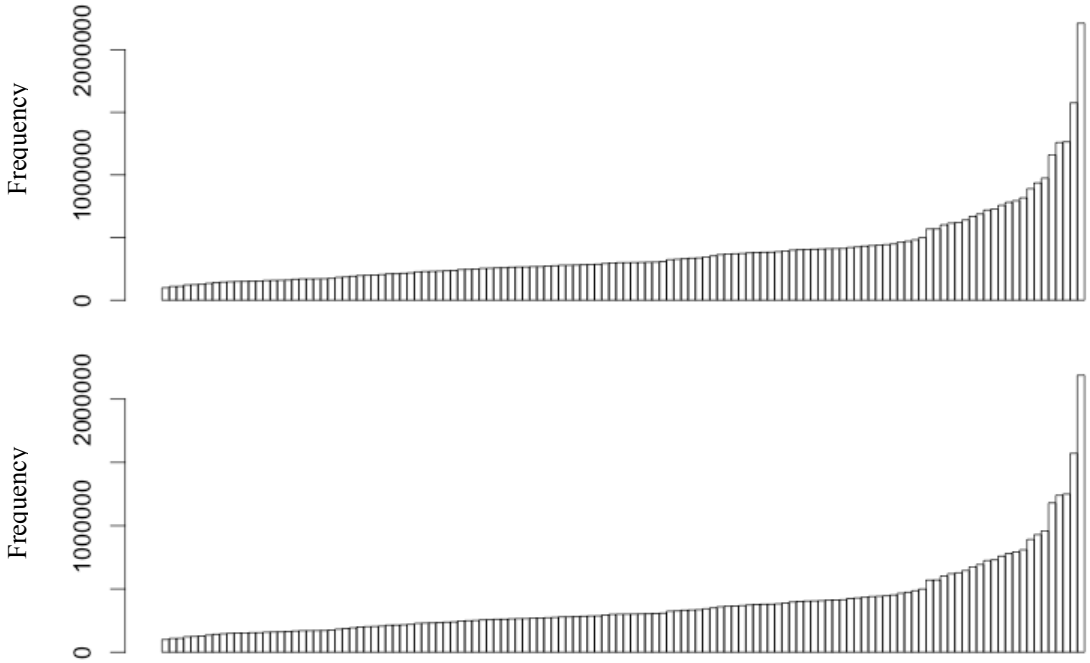

Figure 6.7 (top) histogram of sorted color distribution on *C*=8192, *S*=4 one-level hashed color indexes using key derived from string 'test' and Figure 6.8 (bottom) histogram of sorted color distribution on the same color indexes setting using key derived from string 'test2'. (Duo to the difficulty in visualize all 8192 colors, we show 128 randomly sampled colors in the histograms)

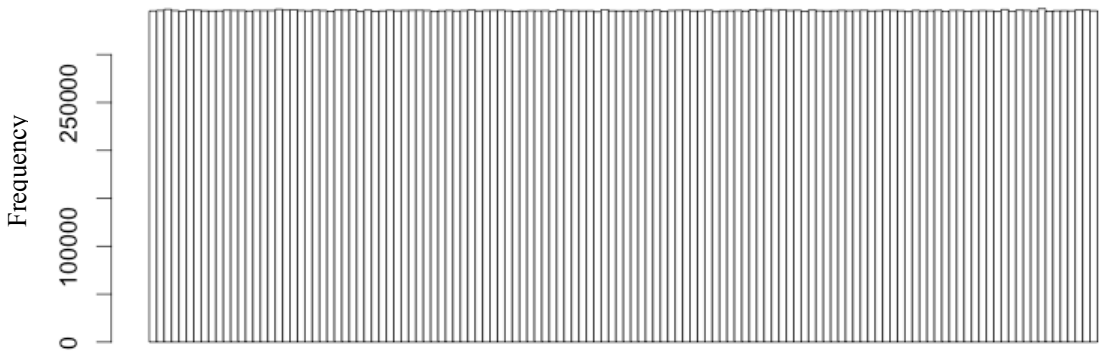

Figure 6.9: Histogram of color distribution on *C*=8192, *S*=4 two-level hashed color indexes using key derived from string 'test'. (Duo to the difficulty in visualize all 8192 colors, 128 entries shown in the graph were randomly sampled from the 8192 colors.)

This experiment result demonstrate that one-level hashing technique (and Hore et al's original design) cannot hide the underline statistic of skewed keywords distribution. Knowing the keyword distribution, attackers are able to guess that the frequently used colors must represent frequently used keywords.

Our improved color indexing removes the influence of keyword frequencies from the indexes by using second-level hashing (Section 4.2). This second-level hash makes the final colors used to represent the same keyword on each index become different. We show the color distribution on two-level hashed indexes of the same configuration in Figure 6.9.

6.3 Error Rate

As described in Section 2.2, when the distribution is skew, simply using the average document size to estimate the error rate will result in an inaccurate prediction. Our goal of the next experiment is to show the different between the estimated error rate and the real error rate of color indexing.

First, the error rate (R_{err}) was estimated by equation (5.5) using Wikipedia's average document size $\bar{t}$ = 159.5. We estimate R_{err} for color indexes settings using different number of colors (C = 1024, 2048, 4096 and 8192) and color code size (S = 4, 6 and 8). On each configuration, we estimated error rate of query the minimum (Q=1) and maximum (Q=S) accuracy setting. Those estimation are shown in Table 6.4.

For the real error rates, we measured the errors by querying random keywords that do not exist in the dataset. Because those non-existing words queries should return empty results in non-encrypted collection; therefore, the number of returned documents in this experiment is entirely due to the false positive error (N_{err}).

We created indexes of the defined configuration, queried them with five different random non-existing keywords, average the number of returned documents, and divide the result with the collection size to get expected R_{err} on the real data (using Q=1 and Q=S). The results of those queries are shown in Table 6.5.

	S=4		*S=6*		*S=8*	
	Q=1	*Q=S*	*Q=1*	*Q=S*	*Q=1*	*Q=S*
C=1024	**46.4%**	**4.6%**	**60.8%**	**5.1%**	**71.4%**	**6.7%**
C=2048	**26.8%**	**0.5%**	**37.4%**	**0.3%**	**46.4%**	**0.2%**
C=4096	**14.4%**	**0.04%**	**20.8%**	**0.008%**	**26.8%**	**0.003%**
C=8192	**7.5%**	**0.003%**	**11.0%**	**0.0001%**	**14.4%**	**0%**

Table 6.4: Estimated error rate (R_{err}) from equation (5.5) using $\bar{t}$ =159.5

	S=4		*S=6*		*S=8*	
	Q=1	*Q=S*	*Q=1*	*Q=S*	*Q=1*	*Q=S*
C=1024	**35.0%**	**9.8%**	**44.1%**	**12.2%**	**50.8%**	**14.8%**
C=2048	**22.0%**	**3.3%**	**29.2%**	**4.1%**	**35.0%**	**5.1%**
C=4096	**12.8%**	**0.8%**	**17.7%**	**1.0%**	**22.0%**	**1.3%**
C=8192	**7.0%**	**0.1%**	**10.0%**	**0.2%**	**12.8%**	**0.2%**

Table 6.5: Average error rate (R_{err}) measured from five random keyword queries on Wikipedia.

The experiment result confirmed that using normal estimation with average document size lead to inaccurate estimations. As shown in Table 3 and 4, the estimation always overestimate the error rate when Q=1 while underestimate the result when Q=S. The underestimation also increases with value of C and S. For example, the calculated error from C=8192, S=8, Q=8 is less than 0.000001, however, real error rate from the experiment is around 0.018 (or 80,000 false articles from 4.4 million Wikipedia articles).

6.3.1 Sampling Error Rate

A sampling technique for better estimation is introduced in Section 5.3. In the next part of experiment, we compare estimation in Section 5.2.2 and the real error rate in the last section.

We randomly randomly select 1000 Wikipedia articles as our samples and measure their sizes. The histogram of those document sizes is shown in Figure 6.10. Then, we use those sampled sizes to estimate the error rate using equation (5.7). The estimation results are shown in Table 6.6.

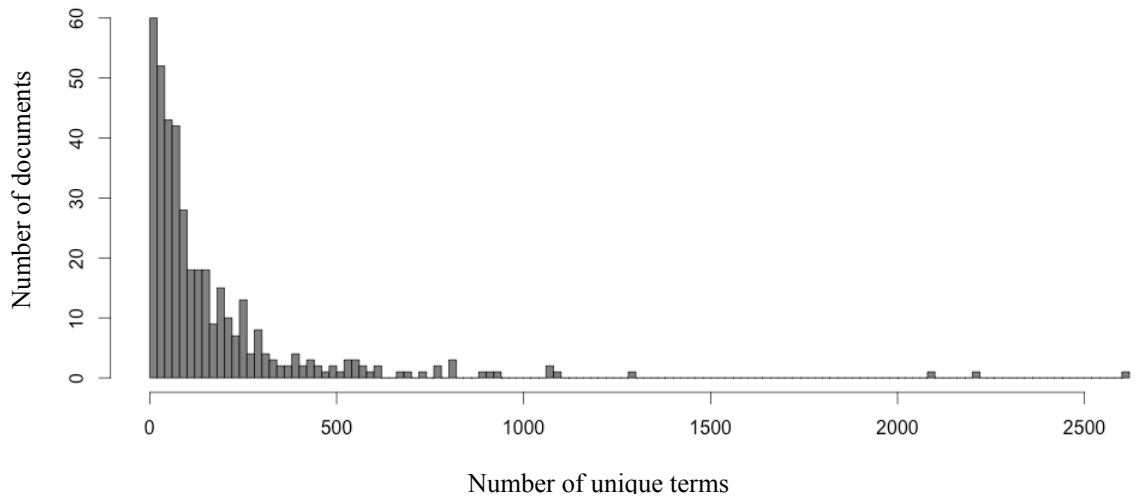

Figure 6.10. Histogram of 1000 randomly selected document sizes from Wikipedia.

	S=4		*S=6*		*S=8*	
	Q=1	*Q=S*	*Q=1*	*Q=S*	*Q=1*	*Q=S*
C=1024	**35.4%**	**10.6%**	**44.3%**	**13.0%**	**50.9%**	**15.5%**
C=2048	**22.5%**	**3.8%**	**29.6%**	**4.8%**	**35.4%**	**5.9%**
C=4096	**13.2%**	**1.1%**	**18.2%**	**1.3%**	**22.5%**	**1.7%**
C=8192	**7.3%**	**0.2%**	**10.4%**	**0.3%**	**13.2%**	**0.4%**

Table 6.6: Estimated error rate (R'_{err}) using equation (5.7) on randomly selected documents from Wikipedia

Comparing the results in Table 6.6 with Table 6.4 and 6.5, the sampling technique yields acceptable prediction on most settings. The sampling prediction does not underestimate error rate when querying with $Q > 1$ unlike the naive prediction.

6.4 Efficiencies on other different datasets

Other than Wikipedia, which is our main experimental dataset because of its size, we also analyze and estimate the efficiency of color indexes in another two smaller datasets.

6.4.1 Reuters-21578

Reuters-21578 (Reuters) is a collection of documents published in Reuters newswire in 1987 (Reuters-22173). The data set was originally collected by Reuters Ltd. and Carnegie Group, Inc and made available in 1990. Since then, there are a few cleanups by researchers, including 595 duplicates removal, resulting in 21578 documents dataset called Reuters-21578. [20]

We downloaded the dataset from UCI website [17] (28.0MB uncompressed). The collection consists of 22 files which each contains around 1000 documents in a format similar to hypertext markup. We extracted only the news contents inside <BODY> tags using a regular expression*. We then replaced new-lines characters with white spaces, removed the empty documents, and made the final cleaned data is a single 19043 line text file of size 16 MB. (This is the format that can be easily processed by Hadoop [1])

Similar to the Wikipedia dataset, we start analyzing the efficiency of color indexing on Reuter collection by measuring keyword and document size distribution. (Figure 6.11-6.14 and Table 6.7-6.8)

Comparing to the other two, Reuters is the smallest in terms of dataset size and number of the documents**. However, the quality of each document is the highest among the three (the length of each article is closer to the average and the keywords are more evenly distributed). Owning to the fact that news articles must be written by professional writers and carefully checked and cleaned before published.

Among the three, the document size of Reuters is the closest to the normal distribution (Figure 1). All the news articles are less than 500 unique terms and most of the article sizes are close to the mean and median (20~80). But, the distribution is still considered skewed.

* Regular expression "<BODY>(.+?)</BODY>"

** The number of documents in Reuters, Enron, and Wikipedia are 19043, 472368, and 4442789 documents respectively. The dataset size, Reuters has 16 MB, Enron has 511.6 MB, and Wikipedia has more than 40 GB

Figure 6.11: Keywords distribution of 200 randomly selected terms from Reuters-21578 collection in **normal scale** sorted by frequency. Each point in the figure represents a keyword and y-axis represents the number of documents containing the keyword.

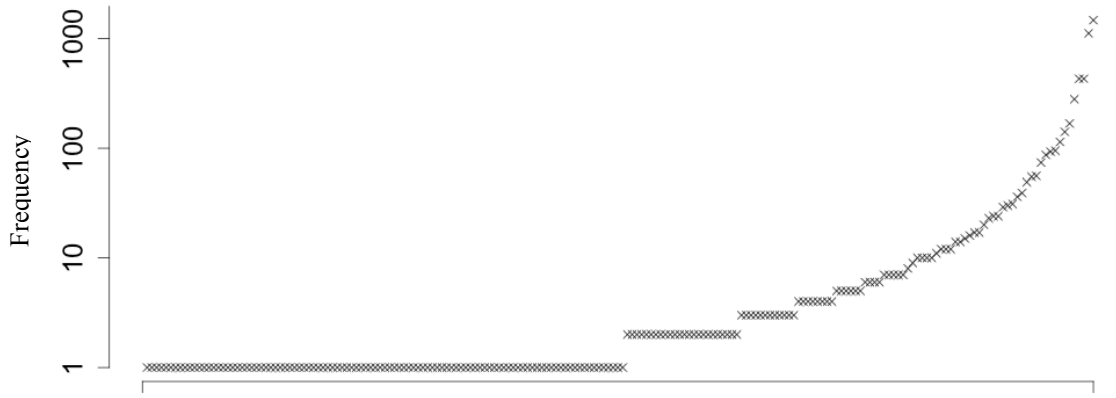

Figure 6.12: Keywords distribution of 200 randomly selected terms from Reuters-21578 collection in **log-scale** sorted by frequency. Each point in the figure represents a keyword and y-axis represents the number of documents containing the keyword.

Minimum	1st Quartile	Median	Mean	3rd Quartile	Maximum
1	1	2	27.74	7	19040

Table 6.7: The summary of Reuters-21578's keywords distribution.

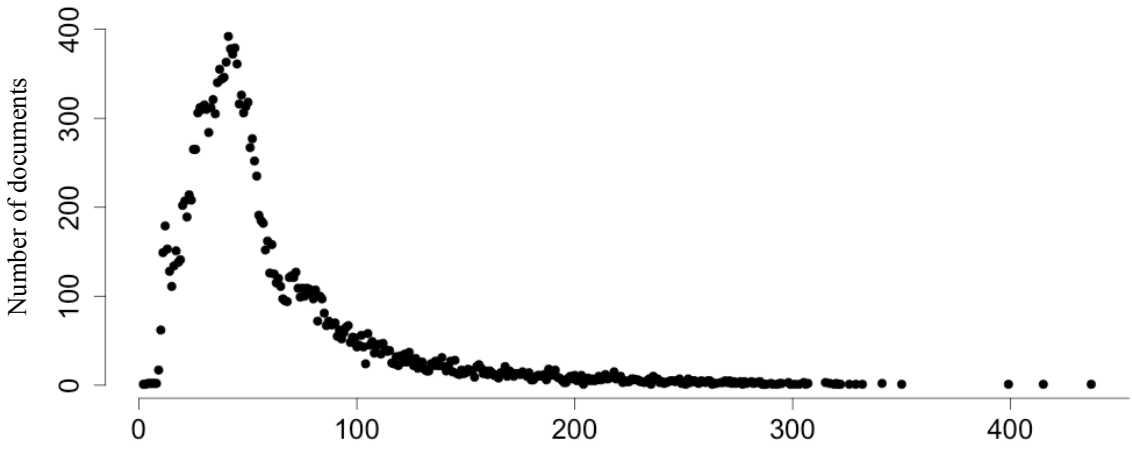

Figure 6.13: Histogram of unique terms count on Reuters-21578 collection in **normal scale**. The x-axis represents the document size or the number of unique terms in the document and y-axis represent the number of documents of different size ranges.

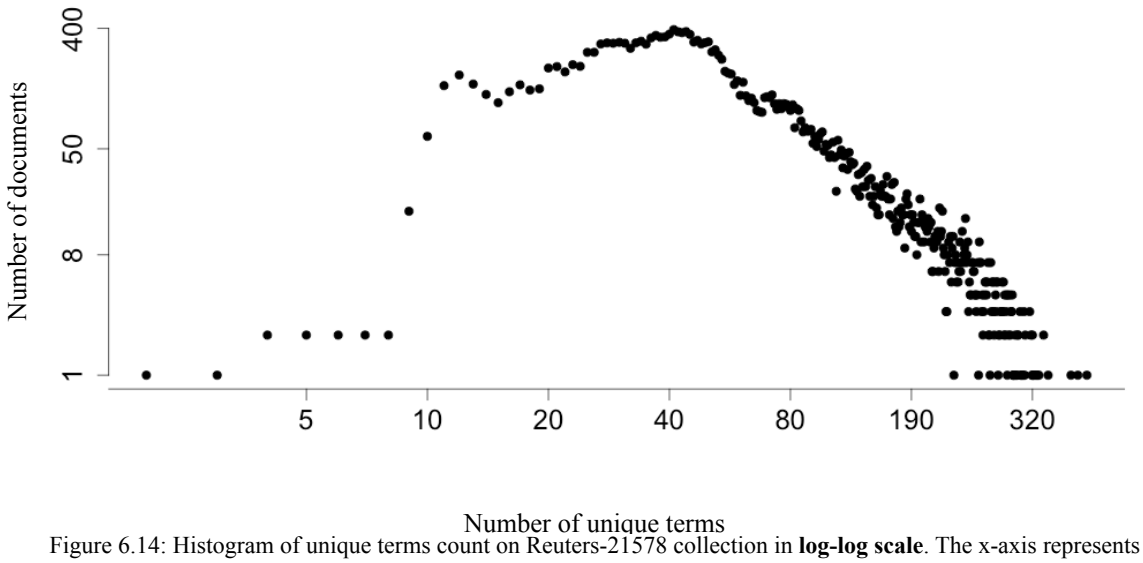

Figure 6.14: Histogram of unique terms count on Reuters-21578 collection in **log-log scale**. The x-axis represents document size or the number of unique terms in the document (in log scale) and y-axis represent the number of documents at different sizes (in log scale).

Minimum	1st Quartile	Median	Mean	3rd Quartile	Maximum
2	32	46	60.03	72	437

Table 6.8: The summary of Reuters-21578's unique terms count distribution.

	S=4		*S=6*		*S=8*	
	Q=1	*Q=S*	*Q=1*	*Q=S*	*Q=1*	*Q=S*
C=1024	**20.9%**	**0.19%**	**29.7%**	**0.07%**	**37.6%**	**0.04%**
C=2048	**11.07%**	**0.02%**	**16.1%**	**0.002%**	**20.9%**	**0.0004%**
C=4096	**5.7%**	**0.001%**	**8.4%**	**0%**	**11.1%**	**0%**
C=8192	**2.9%**	**0%**	**4.3%**	**0%**	**5.7%**	**0%**

Table 6.9: Estimated error rate (R_{err}) from equation (5.5) using $\bar{t}$ = 60.3

	S=4		*S=6*		*S=8*	
	Q=1	*Q=S*	*Q=1*	*Q=S*	*Q=1*	*Q=S*
C=1024	**19.8%**	**0.81%**	**27.6%**	**0.88%**	**34.3%**	**1.2%**
C=2048	**10.7%**	**0.1%**	**15.5%**	**0.07%**	**19.8%**	**0.7%**
C=4096	**5.6%**	**0.009%**	**8.2%**	**0.003%**	**10.7%**	**0.002%**
C=8192	**2.9%**	**0.0007%**	**4.3%**	**0%**	**5.6%**	**0%**

Table 6.10: The expected error rate using equation (5.4) on Reuter-21578 dataset.

We compare the naively estimated error rate and the real error rate on Reuter collection as we did with Wikipedia (Section 6.3). We estimated error rate by equation (5.5) using Reuter average document size $\bar{t}$ = 60.3. The estimation results are shown in Table 6.9. Unlike other two datasets, Reuter has very small number of document that makes the calculation of expected error rate from the whole dataset in equation (5.4) feasible. The calculation results are shown in Table 6.10.

Similar to Wikipedia, the naive estimation trend to underestimated the error rate when using $Q > 1$. This is another claim on invalidity of (5.5). Even Reuter has most of the document size close to the mean, using only the average number is still not enough for representing the distribution.

In Reuter the setting that provide the highest query performance is *C*=8192, *S*=8 setting. While the highest performance setting for Wikipedia is *C*=8192, *S*=1. In fact, when *C* is large enough, increasing *S* will enable in higher performance queries. The optimal setting depends on the dataset which we are currently able to measure it only by experiments.

6.4.2 Enron Emails

Enron Emails Corpus (Enron) is a collection of real-life email messages. The original corpus contains 619,446 messages belonging to 158 employees of Enron corporation. We use August 21, 2009 downloaded from [7].

The data was organized into folders. After decompression, the emails are prepared by by filtering out all header information and reply messages. We extracted only text content from the emails using a regular expression*. We also transformed each message content into a single-line format by replacing new-line characters with spaces. The cleaned data is a single file of 472,368 lines (511.6 MB).

We measure keyword and document size distribution of the dataset as in Wikipedia and Reuter. (Figure 6.15-6.18 and Table 6.11-6.12)

From Figure 6.15-6.16, we observe that the distribution of Enron's keywords is similar to the Reuter's and Wikipedia's keywords, which is seem to be double exponential distribution. In fact, at this point, it can be concluded that, when designing any searchable encryption system, we should expect to handle extremely skewed distribution.

The document size of Enron has a long-tail distribution (Figure 6.17, 6.18) similar to Wikipedia (Figure 6.3, 6.4). However, if we compare both distributions in detail, we will find that more than 70% of Enron's documents are at the head part of distribution with less than 100 unique keywords (the median is 27). While the remanding part is distributed into the tail which has length around Wikipedia's. This makes the tail of Enron's distribution very flat.

The distribution property could be explained as the nature of email messages. Most of exchanging emails are shorten and quick language. Moreover, we found that many emails have been forwarded with no or almost empty additional content**. At the same time, there are small number emails that were written formally to describe the subjects in detail. Those emails have the size (number of keywords) comparable to the Wikipedia articles.

* Regular expression "(?:.+?:.+?(?:\r)?\n)+(?:\r)?\n((?:.|(:\r)?\n)*?)(?=-----|$)"

** In case of forwarded messages and discussion threads, we consider only the content of latest message as our indexible text.

Figure 6.15: Keywords distribution of Enron Emails collection using 200 randomly selected terms in **normal scale** sorted by frequency. Each point in the figure represents a keyword and y-axis represents the number of documents containing the keyword.

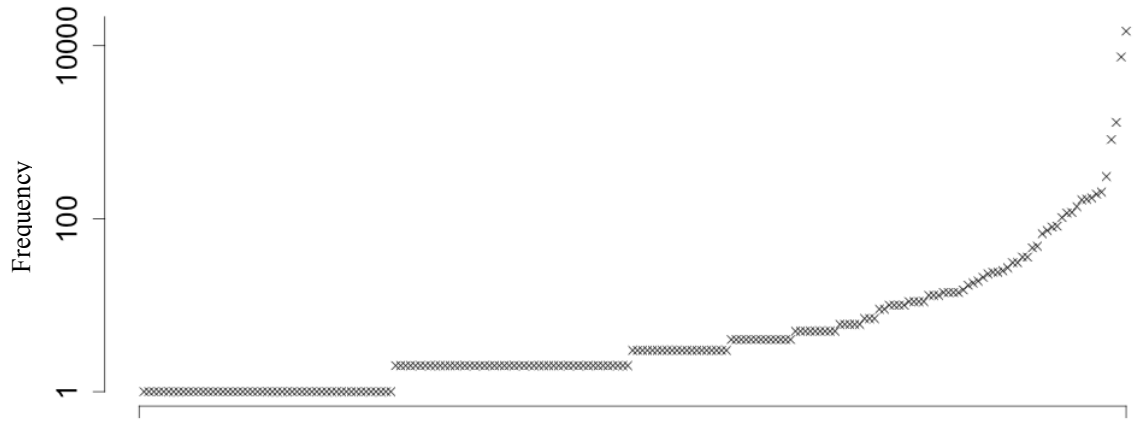

Figure 6.16: Keywords distribution of Enron Emails collection using 200 randomly selected terms in **log-scale** sorted by frequency. Each point in the figure represents a keyword and y-axis represents the number of documents containing the keyword.

Minimum	1st Quartile	Median	Mean	3rd Quartile	Maximum
1	1	3	77.92	7	240100

Table 6.11: The summary of Enron Emails' keywords distribution.

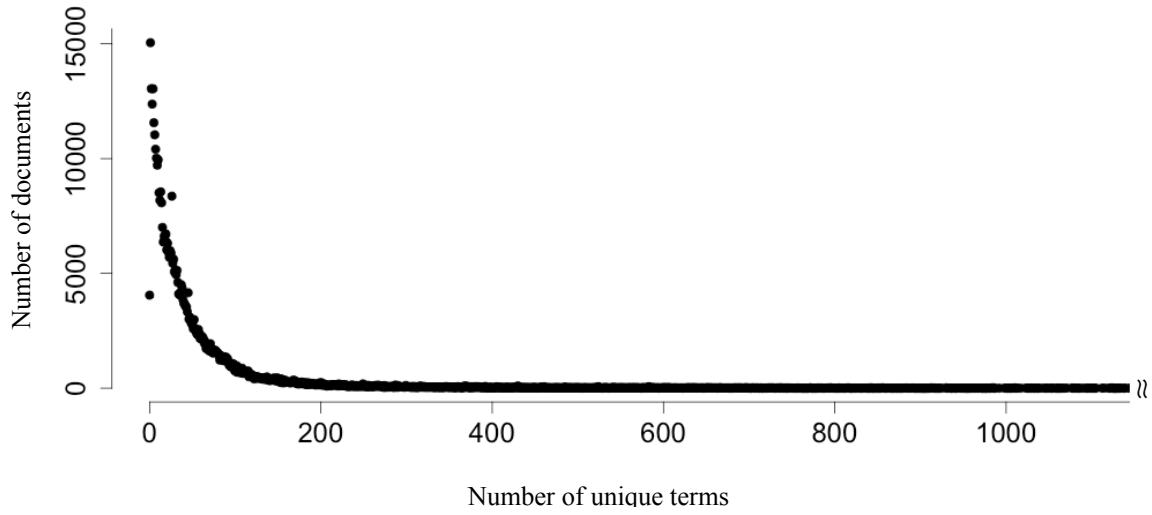

Figure 6.17: Histogram of unique terms count on Reuters-21578 collection in **normal scale**. The x-axis represents the document size or the number of unique terms in the document (between 0 to 1100) and y-axis represent the number of documents of different size ranges.

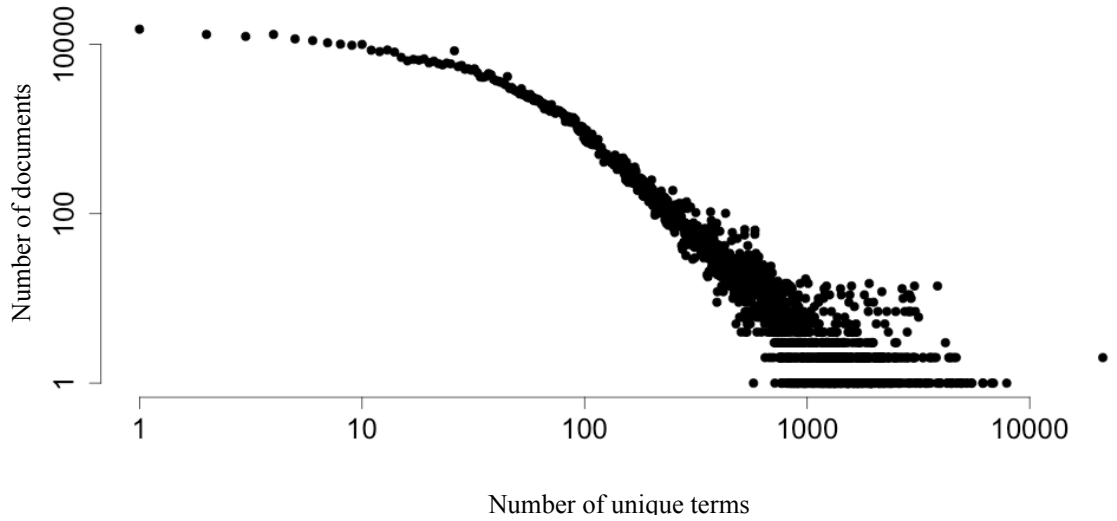

Figure 6.18: Histogram of unique terms count on Reuters-21578 collection in **log-log scale**. The x-axis represents document size or the number of unique terms in the document (in log scale) and y-axis represent the number of documents at different sizes (in log scale).

Minimum	1st Quartile	Median	Mean	3rd Quartile	Maximum
0	10	27	58.27	61	21300

Table 6.12: The summary of Reuters-21578's unique terms count distribution.

Color indexes' errors of different configuration on Enron are measure as same as on Wikipedia. Estimated error rates by equation (5.5) using the average email size $\bar{t}$ =58.27 are shown in Table 6.13. The real error rates are measured by five different non-existing words queries are shown in Table 6.14. And the error rates estimated via sampling technique (Section 5.2.2) are shown in Table 6.15.

	S=4		*S=6*		*S=8*	
	Q=1	*Q=S*	*Q=1*	*Q=S*	*Q=1*	*Q=S*
C=1024	**20.4%**	**0.2%**	**29.0%**	**0.06%**	**36.7%**	**0.03%**
C=2048	**10.8%**	**0.01%**	**15.7%**	**0.002%**	**20.4%**	**0.0003%**
C=4096	**5.5%**	**0.0009%**	**8.2%**	**0%**	**10.8%**	**0%**
C=8192	**2.8%**	**0%**	**4.2%**	**0%**	**5.5%**	**0%**

Table 6.13: Estimated error rate (R_{err}) from equation (5.5) using $\bar{t}$ = 60.3

	S=4		*S=6*		*S=8*	
	Q=1	*Q=S*	*Q=1*	*Q=S*	*Q=1*	*Q=S*
C=1024	**16.0%**	**2.1%**	**21.5%**	**2.5%**	**26.3%**	**3.0%**
C=2048	**9.1%**	**0.65%**	**12.7%**	**0.78%**	**16.0%**	**1.0%**
C=4096	**5.0%**	**0.19%**	**7.1%**	**0.22%**	**9.1%**	**0.27%**
C=8192	**2.63%**	**0.05%**	**3.8%**	**0.05%**	**5.0%**	**0.07%**

Table 6.14: Average error rate (R_{err}) measured from five random keyword queries on Enron Emails Corpus

	S=4		*S=6*		*S=8*	
	Q=1	*Q=S*	*Q=1*	*Q=S*	*Q=1*	*Q=S*
C=1024	**15.3%**	**2.31%**	**20.5%**	**2.81%**	**25.1%**	**3.27%**
C=2048	**8.79%**	**0.69%**	**12.2%**	**0.92%**	**15.3%**	**1.24%**
C=4096	**4.81%**	**0.12%**	**6.88%**	**0.13%**	**8.79%**	**0.17%**
C=8192	**2.53%**	**0.013%**	**3.70%**	**0.007%**	**4.81%**	**0.006%**

Table 6.15: Estimated error rate (R'_{err}) using equation (5.7) on randomly selected documents from Enron Emails Corpus

7. Discussion

As measured from the datasets in our experiments, the distribution of both number of keywords and document size from the real text can be very skewed. This conflicts with the assumption of previous secure indexes designs.

Every collection in our experiments has the same skewed distribution, which seems to be double exponential. This result confirms our believe that, in practice, the frequency of some colors, which represent frequently used keywords, will be visibly higher than the others. Attackers can take advantage of this flaw to guess the original keywords of some colors (Section 4.1.3 and 6.2). Thus, using our proposed two-level hashed indexes (Section 4.2), which can prevent this attack, is preferred for any kind of text collections.

Distribution of document size (or number of keywords per documents) from different kinds of text collections are different. However, we can conclude from our experiment that most of them are highly skewed and have a long-tailed shape. Moreover, those shapes seem to be unfit with any well-known distribution. Thus, in practice, correctly estimating the error rate and tuning index's parameters to achieve expected performance can be challenging and still need to be done by experiments.

Using average document size to estimate the error rate (equation 5.5), as suggested in some previous works, is not recommended. Especially when using $Q > 1$ queries on heavy-tailed distributed collections (ex. Wikipedia and Enron) the prediction will underestimate the real error rate by significant degrees.

In order to estimate query performance correctly, it is inevitable to consider the skewed shape of document size distribution. For small datasets (ex. Router) we can summarize the probability for hitting queries of every single document (equation 5.4). However, for the larger collections, using a sampling technique (Equation 5.7) is more preferable.

The heavy-tailed distribution in the datasets also effect overall efficiency of color indexes queries. As described in section 4.4, color indexes suffer from large documents at the tail of distributions. Those documents have very high-density indexes that could match almost every query color. Handling those extremely large documents in collections is still an open problem in secure indexes design.

In this experiments, we test color index's settings up to *C=8192, S=8* due to our schedule constrain. We are aware that, in practice, it may be favorable to use even more colors. For example, in our Wikipedia experiment, the error rate in highest accuracy setting is 0.1% *(C=8192, S=4, Q=4)*. In large collections like Wikipedia, that small percentage means around 4,400 unrelated documents that will be returned to users.

Bit array representation of 8192 colors use 1 kilobyte space. For 4.4 million documents of Wikipedia, we need to prepare 4.5 gigabytes storage for storing the indexes. If the density of color indexes are low, which is very likely in the when the median document size is smaller than the average like in our datasets, using sparse vector representation would help saving the storage cost and enable us to use even higher number of colors.

8. Conclusion

In this work, we introduce ‘color indexes’ a secure indexing technique that enables secure keyword lookup on encrypted data. The design utilizes false positive errors of color matching as a mechanism to hide real information; thus, increase system security in exchange for query performance. Users can control the number of those false positive errors by adjusting system parameters (*C*, *S*, and *Q*).

Our color indexes deviate from Hore et al’s original technique as it introduce query parameter *Q* that allows users to adjust security level per query. This provides users more flexibility comparing to the previous design which each parameter needed to be configured before building the indexes. We also add the second level hash function that prevents information leakages from colors distribution of the real data.

The previous works have not considered the effect of content statistic. But our experiments have shown that the skewness of data, especially the long-tailed distributions of document size, has a significant effect on the error rate of estimation. It makes naive error estimation using average document size becomes inaccurate. To makes more accurate prediction by considering the distribution’s shape while avoiding large computation, a sampling technique has been introduced.

There are still other issues of color indexes that have not been addressed in this work. One of the most important issues is how to handle exceptionally large documents at the tail of distribution. Applying information retrieval technique to work with those large documents is still an open problem to the future work.

List of Publication

[1] Tanakitrungruang, W., H. Aida, “Secure Indexes for Keyword Search in Cloud Storage”, The 13th Forum on Information Technology (FIT 2014), September 2014. (To be presented)

References

[1] “Apache Hadoop”. http://hadoop.apache.org/ (Retrieved on 2013-10-30).

[2] Bloom, B. “Space/time trade-offs in hash coding with allowable errors”. *Communications of the ACM,* 13(7):422–426, July 1970.

[3] Boneh, D., G. D. Crescenzo, R. Ostrovsky, and G. Persiano. “Searchable public key encryption”. *Eurocrypt 2004*, LNCS 3027, pp. 506-522*,* 2004.

[4] Boneh, D., C. Gentry, S. Halevi, F. Wang, D. J. Wu. “Private Database Queries Using Somewhat Homomorphic Encryption”. *Lecture Notes in Computer Science* Volume 7954, pp 102-118, 2013.

[5] Bose, P., H. Guo, E. Kranakis, A. Maheshwari, P. Morin, J. Morrison, M Smid and Y. Tang. “On the false-positive rate of Bloom filters”. *Information Processing Letters* Volume 108, Issue 4,, Pages 210–213, 31 October 2008.

[6] Clauset, A, CR Shalizi, MEJ Newman, “Power-law distributions in empirical data”. *SIAM review*, 2009.

[7] “Enron Email Dataset”. http://www.cs.cmu.edu/~./enron, (Retrieved on 2014-07-10).

[8] Gentry C. “A Fully Homomorphic Encryption Scheme (Ph.D thesis)” http://crypto.stanford.edu/craig/ (Retrieved on 2013-07-01).

[9] Goh, Eu-Jin. “Secure Indexes”. *IACR Cryptology ePrint Archive*, 2003.

[10] Hore, B., E.C. Chang, M. H. Diallo, S. Mehrotra. “Indexing Encrypted Documents for Supporting Efficient Keyword Search.” *Secure Data Management Lecture Notes in Computer Science* Volume 7482, pp 93-110, 2012.

[11] Kirsch, A., M. Mitzenmacher. “Less Hashing, Same Performance: Building a Better Bloom Filter”. *Lecture Notes in Computer Science* Volume 4168, pp 456-467, 2004.

[12] Klimt, B., and Y. Yang. “Introducing the enron corpus”. *In First Conference on Email and Anti-Spam (CEAS)* 2004.

[13] “MySQL — Full-Text Stopwords”. http://dev.mysql.com/doc/refman/5.1/en/fulltext-stopwords.html (Retrieved on 2013-11-2).

[14] Paillier, P., “Public-key cryptosystems based on composite degree residuosity classes”, *Advances in cryptology—EUROCRYPT'99 ,* 1999.

[15] Park, J.H., “Efficient Hidden Vector Encryption for Conjunctive Queries on Encrypted Data”, *IEEE Transactions on Knowledge and Data Engineering* Vol.23, No.10, October 2011.

[16] “poweRlaw: Fitting heavy tailed distributions: the poweRlaw package” http://cran.r-project.org/web/packages/poweRlaw/index.html (Retrieved on 2014-3-2).

[17] “Reuters-21578 Text Categorization Collection Data Set”. https://archive.ics.uci.edu/ml/datasets/Reuters-21578+Text+Categorization+Collection, (Retrieved on 2014-07-10).

[18] Rivest, R. L., A. Shamir, and L. Adleman. “A method for obtaining digital signatures and public key cryptosystems”. *Communications of the ACM*, 21(2) :120-126, 1978.

[19] Rivest, R. L., L. Adleman, M. L. Dertouzos. “On data banks and privacy homomorphisms”. *in Foundation of Secure Computations, Academic Press* 1978.

[20] Silva, C., and B. Ribeiro. “Inductive Inference for Large Scale Text Classification: Kernel Approaches and Techniques — Appendix A. REUTERS-21578”. *Springer Science & Business Media,* Nov 11, 2009.

[21] Song, D. X., D. Wagner, and A. Perrig. “Practical techniques for searches on encrypted data”. *IEEE Symposium on Security and Privacy,* pages 44–55, 2000.

[22] Tebaa M., S. El Hajji and A. El Ghazi, “Homomorphic Encryption Applied to the Cloud omputing Security”, *Proceedings of the World Congress on Engineering,* Vol I, London, U.K. July 4 - 6, 2013.

[23] “Working with Wikipedia”. http://lintool.github.io/Cloud9/docs/content/wikipedia.html (Retrieved on 2013-11-10).

[24] “Wikipedia, English Wikipedia dumps in SQL and XML” http://en.wikipedia.org/wiki/Wikipedia:Database_download (Retrieved on 2013-11-2).

[25] Yasuda, T. Shimoyama, J. Kogure, K. Yokoyama, T. Koshiba, “Secure pattern matching using somewhat homomorphic encryption”, *Proceedings of the 2013 ACM workshop on Cloud computing security workshop* Pages 65-76 , 2013.