PRIVATE SEARCH OVER BIG DATA LEVERAGING DISTRIBUTED FILE SYSTEM AND PARALLEL PROCESSING

by Ayşe Selçuk

Submitted to the Graduate School of Engineering and
Natural Sciences
in partial fulfillment of the requirements for the degree of
Master of Science

Sabancı University

Fall, 2014 - 2015

PRIVATE SEARCH OVER BIG DATA LEVERAGING DISTRIBUTED FILE SYSTEM AND PARALLEL PROCESSING

APPROVED BY:

Prof. Dr. Erkay Savaş (Thesis Supervisor)	.....
Assoc. Prof. Dr. Yücel Saygın	.....
Asst. Prof. Dr. Kağan Kurşungöz	.....

DATE OF APPROVAL: .....

Acknowledgements

I wish to express my gratitude to my supervisor, committee, friends and family as this thesis would not have been possible without valuable support of them.

Especially, I would like to express the inmost appreciation to my thesis supervisor Prof. Dr. Erkay Savaş. Thanks to his endless assistance and support, this thesis existed and completed successfully. He, at all times, has been considerably helpful as both an instructor and a valuable adviser with his patience and creative suggestions. Additionally, I would like to thank to Dr. Cengiz Örencik because he always made significant contributions and encouraged me for this thesis as if he were my co-supervisor. And I am thankful to Mahmoud Alewiwi for his help and contribution to this thesis.

Furthermore, I also would like to thank the member of my thesis committee, Assoc. Prof. Dr. Yücel Saygın and Asst. Prof. Dr. Kağan Kurşungöz. In addition, I am also thankful to TÜRK TELEKOM since I am supported by the MSc. fellowship of TÜRK TELEKOM under Grant Number 3014-07.

Besides, I have highly kind feelings to my FENS 2001 friends (Cryptography and Information Security Lab). I have spent good times with this marvelous friendship. In addition, there is the most important person that I won‘t pass without mentioning. I wish to state my deepest gratitude to M.Burak Demirci who has a special place in my heart.

Last but not least, I want to express my special appreciation and thank to my beloved family as they have always supported and encouraged me. I am always proud of being a part of this family. If I am here now, this is entirely thanks to them and their unconditional trust in me.

PRIVATE SEARCH OVER BIG DATA LEVERAGING DISTRIBUTED FILE SYSTEM AND PARALLEL PROCESSING

Ayşe Selçuk

Computer Science and Engineering,
Master's Thesis, 2015

Thesis Supervisor: Prof. Dr. Erkay Savaş

Abstract

As the new technologies recently became widespread, enormous amount of data started to be generated in very high speeds and stored in untrusted servers. The big data concept covers not only the exceptional size of the datasets, but also high data generation rate and large variety of data types. Although the Big Data provides very tempting benefits, the security issues are still an open problem.

In this thesis, we identify security and privacy problems associated with a certain big data application, namely secure keyword-based search over encrypted cloud data and emphasize the actual challenges and technical difficulties in the big data setting. More specifically, we provide definitions from which privacy requirements can be derived. In addition, we adapt an existing work on privacy-preserving keyword-based search method, which is one of the fundamental operations that can be performed over encrypted data, to the big data setting, in which, not only data is huge but also changing and accumulating very fast. Therefore, in the

big data setting, a secure index that allows search over encrypted data should be constructed and updated very fast in addition to an efficient and effective keyword-based search operation method.

Our proposal is scalable in the sense that it can leverage distributed file systems and parallel programming techniques such as the Hadoop Distributed File System (HDFS) and the MapReduce programming model to work with very large datasets. We also propose a lazy idf-updating method that can efficiently handle the relevancy scores of the documents in dynamically changing and large datasets. We empirically show the efficiency and accuracy of the method through extensive set of experiments on real data.

BÜYÜK VERİ ÜZERİNDE DAĞITIK DOSYA SİSTEMİ VE PARALEL İŞLEME KULLANARAK MAHREMİYET KORUMALI ARAMA

Ayşe Selçuk

Bilgisayar Bilimleri ve Mühendisliği,
Yüksek lisans Tezi, 2015

Tez Danışmanı: Prof. Dr. Erkay Savaş

Özet

Son zamanlarda, yeni teknolojilerin daha yaygın hale gelmesiyle, çok büyük miktarda veri çok hızlı bir şekilde üretilmeye ve güvenilir olmayan sunucularda depolanmaya başlandı. Büyük veri kavramı sadece veri kümesinin olağanüstü boyutunu değil, aynı zamanda yüksek veri oluşum hızını ve verilerin çok çeşitli türlerde olduğunu vurgulamak için kullanılır. Büyük veri, çok cazip avantajlar sağlasa da, güvenlik sorunları hala açık olan bir problemdir.

Bu tezde, belli bir büyük veri uygulaması ile ilişkili güvenlik ve mahremiyet sorunlarını adresliyoruz. Bir diğer deyişle, şifreli bulut verisi üzerinde güvenli kelime-tabanlı arama işleminin büyük veri ortamında zor olduğunu vurgulayıp, bunun önündeki teknik zorlukları belirtiyoruz. Daha özel olarak ise, mahremiyet gereksinimlerinin tam olarak ortaya konabilmesi için gerekli formal tanımları veriyoruz. Ayrıca, sadece devasa değil aynı zamanda değişen ve çok hızlı biriken büyük veri ortamı için, şifreli veriler üzerinde uygulanabilir

temel işlemlerden biri olan mahremiyet korumalı kelime arama işlemi üzerinde varolan bir çalışmayı uyarlıyoruz. Geliştirilen çözümler, büyük veri ortamında, şifreli veriler üzerinde aramaya olanak veren güvenli bir endeks yapısını makul bir hız ile inşa edebilmeli, ayrıca verimli ve etkili bir kelime arama işlemi yöntemi için çok hızlı güncelleyebilmelidir.

Önerdiğimiz çözümlerin, çok büyük veri kümeleri ile çalışacak şekilde ölçeklendirilebilmesi için, Hadoop Dağıtılmış Dosya Sistemi (HDFS) ve MapReduce programlama modeli gibi paralel programlama teknikleri ve dağıtık dosya sistemleri kullanılmaktadır. Dinamik olarak değişen, büyük veri kümesindeki belgelerin ilgili puanlarını verimli işleyebilen bir tembel idf güncelleme yöntemini de öneriyoruz. Gerçek veriler üzerinde gerçekleştirdiğimiz kapsamlı deneyler vasıtasıyla önerdiğimiz yöntemin etkinliğini ve doğruluğunu deneysel olarak gösteriyoruz.

to my beloved family...

Contents

List of Figures

List of Tables

1 Introduction

With the widespread use of the Internet and wireless technologies in recent years, the sheer volume of data being generated keeps increasing exponentially resulting in a sea of information that has no end in sight. Although the Internet is considered as the main source of the data, a considerable amount of data is also generated by other sources such as smart phones, surveillance cameras or aircraft and their increasing use in everyday life. Utilizing these information sources, organizations collect terabytes and even petabytes of new data on a daily bases. However, the collected data is useless unless it is possible to analyze and understand the information within.

The emergence of massive datasets and their incessant expansion and proliferation led to the term, *big data*. Accurate analysis and processing of big data, which bring about new technological challenges as well as concerns in areas such as privacy and ethics, can provide exceptionally invaluable information to users, companies, institutions and in general to public benefit [33]. The information harvested from big data has tremendous importance since it provides benefits such as cost reduction, efficiency improvement, risk reduction, better health care, and better decision making process. The technical challenges and difficulties in effective and efficient analysis of massive amount of collected data call for new processing methods [35, 36], leveraging the emergent parallel processing hardware and software technologies.

Although the tremendous benefits of big data are enthusiastically welcomed, the privacy issues still remain as a major concern. Most of the works in the literature unfortunately prefer to disregard the privacy issues due to efficiency concerns since efficiency and privacy protection are usually regarded as conflicting goals. This is true to a certain extent due to

technical challenges, which however, should not deter the research to reconcile them in a framework, which allows *efficient privacy-preserving process of big data*.

A fundamental operation in a dataset is to find data items containing certain piece of information which is often manifested by a set of keywords in a query, namely *keyword based search*. An important requirement of an effective search method over big data is the capability of *sorting* the matching items according to their relevancy to the keywords in queries. An efficient ranking method is particularly important in the big data setting, since the number of matching data items may also be huge, if not filtered depending on their relevance levels.

In this thesis, we generalize the privacy-preserving search method that is proposed in [27] and apply it in the big data setting. The method in [27], which is sequentially implemented, is only capable of working with small datasets that contain only a few thousand documents. In order to scale the method in [27] to massive datasets, we leverage the Hadoop framework which is based on distributed file systems and parallel programming techniques. Using Hadoop framework, we create our parallel processing environment, which is a multi-node cluster and setup using the Cloudera framework (CDH4) [18]. For ordering the documents based on their relevancy to a keyword search query, we use the well known tf-idf scoring and adapt it to dynamic big data. In addition, we develop our MapReduce implementations of the tf-idf algorithms proposed for Hadoop framework by Li and Guoyon in [24]. Unlike the work in [27], we assume that the dataset is dynamic, which is an essential property of big data. Therefore, we propose a method referred as "Lazy idf Update" which approximates the relevancy scores using the existing information and only updates the inverse document frequency (idf) value of documents when the change rate in the dataset exceeds a threshold. Our analysis demonstrates that the proposed method is an efficient and highly scalable privacy preserving search method which takes advantage of the Hadoop Distributed File System (HDFS) [20] and the MapReduce programming paradigm.

The rest of this thesis is organized as follows. In Chapter 2, we briefly summarize the previous work in the literature on secure search and Hadoop framework in detail. The preliminary background information which is referred and needed throughout the thesis, such as minhash functions [30] and the Hadoop structure, are given in Chapter 3. In this chapter, we introduce the minhash functions known as locality sensitive hash functions (LSH) and

signature structure in an explanatory manner. In addition, we give short information about NoSQL. The details of distributed file systems and the Hadoop framework are given. We formalize the information that we protect in the protocol. We briefly summarize the underlying secure search method of Örencik et al.[27]. We present the crucial steps of the tf-idf scoring algorithms proposed by Li and Guoyong [24]. In Chapter 4, we present the framework of the proposed model. The properties of big data and the new technologies developed to meet the requirements of big data are summarized. The novel idf-updating method for adjusting the tf-idf scoring for dynamically changing dataset is also explained in this chapter. In Chapter 5, we discuss the results of the several experiments we applied on a large dataset using the multi-node cluster. Chapter 6 is devoted for the conclusion and possible future directions.

2 Related Work

The massive size and overwhelming velocity of big data make its processing already a daunting job, even without the security and privacy features. Cloud computing services [7], which allow big data processing by small players that lack computational power and storage capacity, make security and privacy concerns even worse. Data, outsourced to a cloud, must be encrypted for security protection and any operation on the data, such as search, should preserve its privacy.

There are a number of works for search over encrypted cloud data, but most are not suitable for the requirements of big data. Majority of the recent works are based on bilinear pairing [10, 12, 38]. However, computation costs of pairing based solutions are prohibitively high both on the server and on the user side. Therefore, pairing based solutions are generally not practical for big data applications.

Other than the bilinear pairing based methods, there are a number of hashing based solutions. Kuzu et al. [22] proposed a single keyword search method which uses a different technique based on locality sensitive hashes (LSH). In addition, Wang et al. [37] proposed a multi-keyword search scheme, which is secure under the random oracle model. This method uses a hash function to map keywords into a fixed length binary array. Later, an improvement to this work is proposed in [28], which additionally provides strict privacy protection and ranking capability. Cao et al. [9] proposed another multi-keyword search scheme that encodes the searchable database index into two binary matrices and uses inner product similarity during matching. This method is inefficient due to huge matrix operations and also not suitable for ranking according to the relevancy of queries. Recently, Örencik et al. [27] proposed another efficient multi-keyword secure search method with ranking capability, which

is also based on locality sensitive hashes. In this thesis, we adapt the method in [27] to meet the requirements of big data applications by preserving its superior features such as ranking, high accuracy and efficiency.

The requirements of processing big data led the big companies like Microsoft and Amazon to develop new technologies that can store and analyze large amounts of structured or unstructured data in distributed and parallel manner. Some of the most popular examples of these technologies are the Apache Hadoop project [20], Microsoft Azure [8] and Amazon Elastic Compute Cloud web service [16, 34]

In addition, tf-idf scoring which is a well-known scoring method for giving weights to terms of documents is developed for the Hadoop framework by Li and Guoyon [24]. Besides, the company known as RapidMiner developed the RADOOP tool to calculate tf-idf scores using Apache Hadoop [4].

3 Preliminaries and Background

In this chapter, we provide the necessary background on the concepts, the techniques and algorithms used in the thesis such as locality sensitive hashes, document signatures, NoSQL databases and distributed file system. We also give a set of definitions that capture the privacy requirements of a private keyword search algorithm.

3.1 Signature

The essential aspect of privacy-preserving search is examining the similarity between a query and database elements without leaking the information of search terms in the query. In order to examine the similarity between a query and a database element, a representation called signature is created. Signature represents each document as a small set. While the signature does not provide the exact similarity values, it can be used as a good approximation to accelerate the processing. The signatures include several elements which are generated using the minhash functions. Before defining the minhash function, we first provide basic information regarding the minhash functions in the following subsections.

3.1.1 Matrix Representation of Sets

Constructing small signatures for huge sets is feasible. We primarily visualize a collection of sets as a characteristic matrix. Each column in the characteristic matrix represents a set, which contains certain number of elements, which are shown in the rows. All the elements contained in all sets form what is referred as the *universal set*. If there is a "1" in row r and column c, the set in column c contains the element that corresponds to row r. Otherwise, a "0" in the same location indicates that the set does not contain the corresponding element.

In our context, we can think of a document as a set that contains a certain number of keywords (terms, elements) selected from a dictionary (the universal set). The Table 1 is an

Element	S_1	S_2	S_3	S_4
a	1	0	0	1
b	0	0	1	0
c	0	1	0	1
d	1	0	1	1
e	0	0	1	0

Table 1: Matrix Representation of sets

example of a matrix that represents sets S_1, S_2, S_3, and S_4 which contain elements chosen from the universal set $\{a, b, c, d, e\}$. Here, S_1= $\{a, d\}$, S_2= $\{c\}$, S_3=$\{b, d, e\}$, and S_4= $\{a, c, d\}$.

We use the characteristic matrix for visualization purposes only as storing data in a characteristic matrix is not practical. If data is stored as a characteristic matrix, it will result in a sparse matrix with too many "0" elements. Therefore, the minhashing technique is used to represent data in an efficient and effective manner.

3.1.2 Minhash Function

First, we apply a permutation of the elements in the universal set to compute minhash value of a set. The minhash value of a set represented in a column of the characteristic matrix, is the index number of the first row containing "1" in the permuted order.

Let the permutation P of the universal set $\{a, b, c, d, e\}$ be $\{b, e, a, d, c\}$ as shown by the matrix in Table 2. The permutation P describes a minhash function h which maps sets to rows. Let us calculate the minhash value of set S_1 by using the minhash function h. Firstly, we start with the column of set S_1. Row b has "0", therefore we go ahead to row e that is second element in the permuted order P. However, there is again a "0" in the column of S_1. So, we go ahead to row a, and we come across a "1" in that row. Therefore, the output of the minhash function is $h_P(S_1) = a$ under the permutation P.

To summarize the method in the matrix, we can read off the values of h by scanning from the top value until we come across a "1". Thus, we can say that $h_P(S_2) = c$, $h_P(S_3) = b$ and

Element	S_1	S_2	S_3	S_4
b	0	0	1	0
e	0	0	1	0
a	1	0	0	1
d	1	0	1	1
c	0	1	0	1

Table 2: Matrix Representation of sets permuted

$h_P(S_4) = a$. A formal definition of the minhash function of a set is given in the following.

Definition 3.1.1. ***Minhash[27]:*** *Let Δ be a finite set of elements, P be a permutation on Δ and $P[i]$ be the i^{th} element in the permutation P.* Minhash *of a set $S \subseteq \Delta$ under permutation P is defined as:*

$$h_P(S) = min(\{i \mid 1 \leq i \leq |\Delta| \ \wedge \ P[i] \in S\}) \tag{1}$$

3.1.3 Minhash Signatures

The minhash signatures can be obtained by applying many randomly chosen permutations as given in Definition 3.1.1. The characteristic matrix M, represents a collection of sets. We randomly select λ permutations of the rows of M, which are represented $P_1, P_2, \ldots, P_\lambda$.

The minhash functions determined by these permutations are shown as $h_{P_1}, h_{P_2}, \ldots, h_{P_\lambda}$. Constructing the minhash signature for set S_1 uses the vector $[h_{P_1}(S_1), h_{P_2}(S_1), \ldots, h_{P_\lambda}(S_1)]$. If we represent this list of hash values as a column, we can create a signature matrix from the characteristic matrix M.

In the proposed method, for each signature, λ different random permutations on the set of all possible search terms, are used so the final signature of a set S is defined as:

$$Sig(S) = \{h_{P_1}(S), \ldots, h_{P_\lambda}(S)\}, \tag{2}$$

where h_{P_j} is the *minhash* function under permutation $P_j \ \forall_j, 1 \leq j \leq \lambda$.

The minhash signatures are used as an approximation method for comparing documents with search queries.

Each set is mapped into λ different buckets using different hash functions. The minhash functions provide the property that similar sets are mapped into the same set of buckets with high probability.

3.2 NoSQL

The term NoSQL is generally used for "not only SQL" or "not relational" database management systems. Unlike relational database management systems (RDBMS), NoSQL systems do not utilize SQL to manipulate data. Thus, functionality and query types are limited in a NoSQL system, compared to complex query support in a RDBMS. Still, all NoSQL systems at least support three basic operations: insert, remove, and retrieve. The most important properties of NoSQL data stores can be summarized as follows [11]:

- Only a limited number of functionalities are offered and are scaled over several servers.
- NoSQL systems are scalable. Data can be partitioned over the servers, and since the operations provided in a NoSQL system are rather simple, any server can operate independently from any other.
- Instead of the ACID properties (atomicity, consistency, isolation, durability), a weaker concurrency model is used to optimize the overall performance.
- NoSQL systems utilize RAM to store data (thus sacrifice persistence), and aim to answer simple queries very efficiently (e.g. [26]).
- New attributes can dynamically be added to data records.

In order to meet the scalability and reliability requirements, a new class of NoSQL based data storage technology referred as *Key-Value Store* [6, 15] is developed and widely adopted.

This system utilizes associative arrays to store the key-value pairs on a distributed system. A key-value pair consists of a value and an index key that uniquely identifies that value. A key-value store offers three basic operations: insert, remove, and retrieve. This allows distributing data and query load over many servers independently, thus achieve scalability. Furthermore, none of the key-value stores offer secondary index on data, but only

offer indexing on the primary key. Some popular examples are Amazon's Dynamo [15] and Memcached [26].

3.3 Distributed File Systems

It is not possible to process large amounts of data that are in the order of terabytes by using only a single server, due to their overwhelming storage and computation power requirements. Therefore, the cloud computing services are utilized to meet these requirements. As a result, cost can be decreased and much more computing power available in cloud for storing, accessing, and processing data can be effectively put to use in an affordable way. Most of the cloud computing platforms use Hadoop [20], which is an open-source distributed and parallel computing framework. It provides easy and cost-effective processing solutions for vast amounts of data. Some popular companies like FaceBook, Yahoo, LinkedIn, Twitter, IBM etc. use it to manage and analyze the collected unstructured data. The Hadoop framework is comprised of two main modules which are the Hadoop Distributed File System (HDFS) [32] for storing large amounts of data in a distributed manner and accessing it with high throughput and reliably due to employed redundancy and the MapReduce framework for distributed processing of large-scale data on commodity computers.

3.3.1 Hadoop Distributed File System (HDFS)

The Hadoop Distributed File System (HDFS) is an open source file system that is inspired by the Google's Google File System (GFS) [17]. The HDFS architecture illustrated in Figure 1, runs on distributed clusters to manage massive datasets. HDFS has some important features that other existing distributed file systems do not support. Firstly, it is a highly fault-tolerant system that can work on low-cost hardware. In addition, HDFS enables high throughput access for application data and streaming access for file system data. HDFS is based on a master/slave communication model that is composed of a single master node and multiple data (i.e. slave) nodes. In HDFS, every file is divided into blocks of 64 MB. There exists a unique node called the NameNode that runs on the master node. The master node manages the file system namespace to arrange the mapping between the files and the blocks and

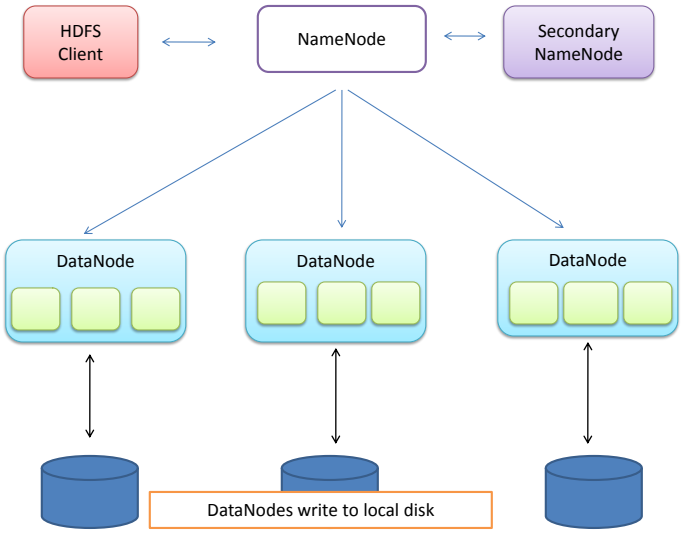

Figure 1: Hadoop distributed files system architecture.

regulates the client access to the files [36].

The NameNode is responsible of some file operations such as file opening, file closing, file deleting and renaming of file names. Also, it provides mapping of blocks to DataNodes. On the other hand, there are a lot of DataNodes which hold blocks for every slave. DataNodes manage to block creation, block deletion and replication by using instructions of NameNode [31, 36].

3.3.2 Hadoop Mapreduce Framework

The Hadoop MapReduce framework is based on the Google's MapReduce algorithm [14]. The MapReduce programming model is derived from the Map and the Reduce functions which are used in functional programming paradigm. Hadoop's MapReduce is the most important implementation of MapReduce Programming model used in real world applications.

The MapReduce programming model, which processes massive amounts of data, provides large-scale computations for large clusters by dividing the tasks into parts that can be processed independently (hence, in parallel). The input data for MapReduce is stored in HDFS. MapReduce utilizes the key-value pairs for distributing the input data to all the nodes in the cluster, in a parallel manner [31]. Firstly, the map function processes all key/value pairs and generates a set of transient key/value pairs. The reduce function, or rather its one

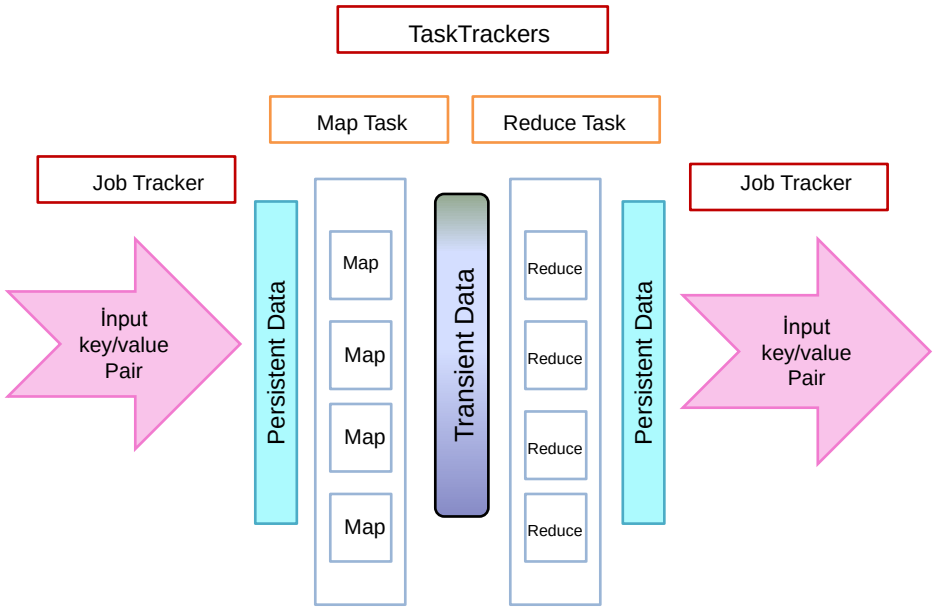

Figure 2: A visualization of the map-reduce process.

instance, processes the transient key-value pairs, with the same key. The Jobtracker instance runs on a single master node to accept the job requests coming from a client and distributes the configurations to the slave nodes. The TaskTracker instance runs on the slave nodes to execute the map and reduce functions on a split data in a parallel manner, using the Java programming language [31]. The map-reduce process is visualized in Figure 2.

3.4 Privacy Requirements

In the literature, the privacy of the data analyzed in big data context is usually protected by anonymizing the data [39]. However, anonymization techniques are not sufficient to protect the data privacy. Although a number of searchable encryption and secure keyword search methods are proposed for the cloud data setting [9, 12, 38], none of them is suitable for big data.

A secure search method over big data should provide the following privacy requirements. The definitions are taken from [27].

Definition 3.4.1. ***Query Privacy:*** *A secure search protocol has query privacy, if for all polynomial time adversaries A that, given two different set of search terms F_0 and F_1 and a query Q_b generated from the set F_b, where $b \in_R \{0, 1\}$, the advantage of A in finding b is negligible.*

Intuitively, the query should not leak the information of the corresponding search terms.

Definition 3.4.2. ***Data Privacy :*** *A secure search protocol has data privacy, if the encrypted searchable data does not leak the content (i.e., features) of the documents.*

The search method we adapted [27] satisfies both privacy requirements.

3.5 Relevancy Score

The relevancy score is used to sort the matching results with the query. To calculate the relevancy score, we take advantage of a similarity function. Each matching result of a search query is ranked with the relevancy score by using the similarity function. There are four well-known metrics to find the relevancy score in the information retrieval context [25, 27, 29]:

- Term frequency measures how frequently a term occurs in a document. If the term appears many times in a document, this means that it is more relevant to a query that contains the term. It is formulated as

$$tf = \left(\frac{n}{N}\right), \tag{3}$$

  where n is the number of occurrences of the term in the document, and N is the total number of terms of the document.

- Inverse document frequency measures the importance of a term to distinguish a document that contains it. It means that if a term does not appear frequently in the dataset, but appears in a particular document, then this search term has a higher relevancy score for this document. The inverse document frequency of a search term is computed using the formula

$$idf = log\left(\frac{|\mathcal{D}|}{d}\right), \tag{4}$$

where $|\mathcal{D}|$ is the total number of documents in the dataset $\mathcal{D}$ and d is the number of documents that contain this term.

- Document length (Density) stipulates that if two documents include identical number of terms, the shorter document is more relevant with query.

- Completeness stipulates that more search terms the document has, the higher score the document has.

In the information retrieval context, the results are required to be ordered according to their relevancy with the query. Therefore, the tf-idf scoring [25, 27] generally is used as the scoring metric in the information retrieval applications. In addition, it evaluates the importance of a term within a document for the dataset. The term frequency (tf) and the inverse document (idf) are put together to form the tf-idf score. The tf-idf of a term in a document $\mathcal{D}$ is given by

$$\text{tf-idf} = \text{tf} \times \text{idf}.$$

Additionally, the idf's log function always has the ratio that is exactly larger than or equal to 1. Therefore, the value of idf is larger than or equal to 0. Finally, the tf-idf score is a real number larger than or equal to 0.

Since we use encryption algorithms in our proposal, the relevancy score (rs) should be an integer value; therefore, we multiply a tf-idf score with an appropriately chosen scaling factor and then apply rounding operation to obtain an integer value.

3.6 Secure Search Method

The utilized privacy preserving search method is based on the method by Örencik et al. [27]. In this section, we briefly explain the method for completeness and refer the reader to [27] for the details. The search method is based on the minhashing technique [30]. Each document is represented by minhash signatures, which are constant length sets as explained in Section 3.1. While the signatures hide the information of document features, they still provide a good representation for the underlying features. During the similarity comparison, only the signatures are used and the underlying document feature sets are not revealed to server, assumed to hold

the data and not necessarily being fully trusted (e.g. cloud server). This method cannot provide the exact similarity value, but it can still provide an accurate estimation. The signature of a document is defined in Section 3.1.3.

3.6.1 Index Generation

In [27], the index generation is assumed to be an offline operation initiated by the data owner and creates the secure searchable index that is outsourced to the cloud as data is regarded as static. The searchable index generation process is based on the bucketization technique [19, 21] which is a well known method for data partitioning. The idea of bucketization is utilized in the proposed method. Here, each data object is distributed into several buckets by using minhash functions introduced previously.

In addition, the bucket identifier is used as an identifier for each object in the corresponding bucket. For each minhash function and corresponding output pair, a bucket is created with bucket identifier B_k^j (i.e., j^{th} minhash function produces output k).

Each document identifier is distributed into λ different buckets depending on the λ elements of the document's minhash signature. The number of common buckets between two objects indicates their similarity. Having no common bucket indicates that the documents do not have any common term. In addition to the document identifiers, the corresponding relevancy scores (i.e., tf-idf scores) are also added to the bucket content ($V_{B_k^j}$).

Note that, both the bucket identifiers (B_k^j) and the content vectors ($V_{B_k^j}$) are sensitive information that needs to be encrypted before outsourcing them to the cloud. A secure searchable index $\mathcal{I}$, which is the combination of the encrypted bucket identifiers and the corresponding encrypted content vectors, is generated to allow private search over encrypted data.

The following phases, which are feature extraction, bucket index construction and bucket index encryption [27, 29] are used to generate the secure searchable index $\mathcal{I}$.

1- Feature Extraction: The set of features $F_i = \{f_{i1}, \ldots, f_{iz}\}$ is extracted for every document $D_i \in \mathcal{D}$, where $\mathcal{D}$ is the set of all documents in the dataset. The set of features distinguishes the document. In the scheme, the features are made up of two values, namely $f_{ij} = (w_{ij}, rs_{ij})$. A keyword w_{ij} of the document is the first value. The relevancy score (rs_{ij}) is the second value which is generated by using tf-idf scoring of the keyword w_{ij} for docu-

ment D_i as explained in Section 3.5. Subsequently, we need this relevancy score to rank the matching results utilized in the search method (cf. Section 3.6.3).

2- Bucket Index Construction : To begin with, by selecting λ random permutations on the set of all search terms (Δ), λ *minhash* functions are generated. Then, these *minhash* functions are applied on the first values of the feature sets, namely $F_i^* = \{w_{i1}, \ldots, w_{iz}\}$, of each document as explained in Section 3.1.2. Consequently a signature for each document is generated as

$$Sig(D_i) = \{h_{P_1}(F_i^*), \ldots, h_{P_\lambda}(F_i^*)\}.$$

Note that $\forall j \in \{1, \ldots, \lambda\},\ h_{P_j}(F_i^*) \in F_i^*$. In other words, the outputs of the hash functions are one of the keywords of the input set.

Then, the elements of the signature of the document are used to map feature set of each document to λ buckets. Let $h_{P_j}(F_i^*) = w_k$, then document D_i is added to the bucket that have the identifiers j for the permutation and k for the output (i.e., B_k^j). For example, if B_k^j is a bucket identifier and $V_{B_k^j}$ is the corresponding content vector, then $V_{B_k^j}[id(D_i)] = rs_{ji}$ if and only if $D_i \in B_k^j; V_{B_k^j}[id(D_i)] = 0$, otherwise.

3- Bucket Index Encryption : Due to the privacy requirements, the bucket identifiers and bucket contents should be encrypted. The bucket identifier B_k^j is a sensitive information, since it may disclose a search term in a query. Therefore, bucket identifier must be encrypted. Moreover, the server maps the bucket with the bucket identifier without knowing the decryption keys. Hence, the encryption method should use a deterministic scheme to hide the bucket identifier. HMAC function is one such method that can hide the information in a deterministic way. An HMAC function can be obtained using cryptographic hash functions with a secret key. In the given scheme, since we do not need to decrypt to the encrypted bucket identifier, HMAC functions are used for hashing the bucket identifiers. In addition, we can generate a secret key K_s for encryption and utilize this secret key for the HMAC function. The secret key of the HMAC function should be known only by the data owner and never disclosed to the server. We denote the encrypted bucket identifier as $\pi_{B_k^j} = HMAC_{K_s}(B_k^j)$. The bucket

content vector ($V_{B_k^j}$) has also sensitive information, namely the identifiers of the documents in that bucket and their relevancy scores. The untrusted server should not learn this information; therefore only encrypted versions of these values are outsourced to the server.

The secure index generation method is described in Algorithm 1as given in [27].

Algorithm 1 Index Generation [27]

Require: Δ:set of possible keywords, $\mathcal{D}$: collection of documents, h: λ *minhash* functions, Ψ: security parameter

$K_s = Setup(\Psi)$,
for all $D_i \in \mathcal{D}$ **do**
  $F_i \leftarrow$ extract features of D_i
  $Sig(D_i) = \{h_{P_1}(F_i^*), \ldots, h_{P_\lambda}(F_i^*)\}$
  for $j = 1 \rightarrow \lambda$ **do**
    $B_k^j = Sig(D_i)[j-1]$
    if $B_k^j \notin$ bucket identifier list **then**
      add B_k^j to bucket identifier list
      create $V_{B_k^j}$
    end if
    add rs_{jk} to vector $V_{B_k^j}[id(D_i)]$
  end for
end for
for all $B_k^j \in$ bucket identifier list **do**
  $\pi_{B_k^j} \leftarrow HMAC_{K_s}(B_k^j)$
  $\mathcal{V}_{B_k^j} \leftarrow Enc_{K_s}(V_{B_k^j})$
  add $(\pi_{B_k^j}, \mathcal{V}_{B_k^j})$ to secure index $\mathcal{I}$
end for
return $\mathcal{I}$

3.6.2 Query Generation

The query is generated in the same way as generating secure index entries (Section 3.6.1). Given the set of keywords in a query (to search for documents that contain them with high relevancy scores) (i.e., $F = \{w'_1, \ldots, w'_n\}$), the query signature is generated by using the same minhash functions used in the index generation phase. The elements of the query signature are indeed the identifiers of the buckets that include the documents that contain the queried keywords. The bucket identifiers in the query signature are obtained by the HMAC function using the same secret key used in the index generation. Therefore, the query Q is the set of encrypted bucket identifiers (i.e., $Q = \{\pi_1, \ldots, \pi_\lambda\}$). Independent of the number of queried keywords, the query signature, hence the query itself, has constant length, which is λ.

We utilize the Jaccard distance in order to analyze the difference between two query signatures. Let A and B be two sets, then Jaccard distance is given as follows:

$$J_d(A, B) = 1 - \frac{|A \bigcap B|}{|A \bigcup B|} \tag{5}$$

In Algorithm 2, query generation is formally described as given in [27]:

Algorithm 2 Query Generation [27]

Require: F: feature set of keywords to be queried,
h: λ minhash functions, K_s: encryption key

$Sig(F) = \{h_{P_1}(F[0]), \ldots, h_{P_\lambda}(F[0])\}$
for $j = 1 \rightarrow \lambda$ **do**
  $B_k^j = Sig(F)[j-1]$
  $\pi_{B_k^j} \leftarrow HMAC_{K_s}(B_k^j)$
  $Q[j-1] = \pi_{B_k^j}$
end for
return Q

3.6.3 Secure Search

In the search phase, first, the query Q is accepted by the cloud server. Then, the cloud server finds the requested encrypted content vectors ($\mathcal{V}_{B_k^j}$). These content vectors are related to the

bucket identifiers of the query. After that, the λ encrypted vectors $E_V = \{\mathcal{V}_1, \ldots, \mathcal{V}_\lambda\}$ are sent back to the user by the cloud server. Then, the user receives the encrypted content vectors, decrypts them and ranks the document identifiers using the tf-idf scores. The search method is described in Algorithm 3 as given in [27].

Algorithm 3 Secure Search [27]

Require: $\mathcal{I}$: secure index, Q: query

for all $\pi_{B_k^j} \in Q$ **do**
  if $(\pi_{B_k^j}, \mathcal{V}_{B_k^j}) \in \mathcal{I}$ **then**
    add $\mathcal{V}_{B_k^j}$ to E_V
  end if
end for
send E_V to the user

3.7 Document Retrieval

Generally, returning documents that are unrelated to the search query bring about an unnecessary communication burden for the user which the user wants to minimize. Therefore, the user does not basically retrieve all the documents that have at least one common bucket with the query. Instead, only the top t matching results are retrieved by the user. We use the tf-idf score for ranking the matching results since the tf-idf scoring is the standard formula for finding the document-term weights [30]. This method is used especially in the search operation to calculate the relevancy score. The user receives the encrypted vectors $E_V = \{\mathcal{V}_1, \ldots, \mathcal{V}_\lambda\}$, decrypts them to obtain the plaintext vectors $V_i = Dec_{K_s}(\mathcal{V}_i)$. The user, then, sorts the documents by using their scores.

The score of a document D_i (i.e., $score(D_i)$) is obtained as summing the relevancy scores for the buckets which are shared by both the document and the query. The formula is defined as follows:

$$score(D_i) = \sum_{k=1}^{\lambda} V_k[D_i].$$

If the $score(D_i)$ gets higher, it is expected that the relevancy of the corresponding document to the query increases.

After the sorting the scores, the user receives the top t matches from the server. The document retrieval method is described in Algorithm 4 as given in [27].

Algorithm 4 Document Retrieval [27]

USER:

Require: E_V: encrypted vectors, K_s: secret key,
t: limit for number of documents to retrieve

for all $\mathcal{V}_i \in E_V$ **do**
  $V_i \leftarrow Dec_{K_s}(\mathcal{V}_i)$
end for
for $j = 1 \rightarrow |V_i|$ **do**
  $score(j) = \sum_{i=1}^{\lambda} V_i[j]$
end for
sort *score* list
idList $\leftarrow$ identifiers of top t *scores*
send idList to Server

SERVER

Require: idList: requested document identifiers, E_{Doc}: outsourced encrypted documents

for all $id \in$idList **do**
  if $(id, \Omega_{id}) \in E_{Doc}$ **then**
    send (id, Ω_{id}) to user
  end if
end for

USER:

$D_{id} \leftarrow Dec_{K_s}(\Omega_{id})$

3.8 Calculation of TF-IDF in Hadoop Framework using Map-Reduce Functions

Generally, certain software tools are used for calculating tf-idf scores of documents such as the Rapid Miner, which is a popular text mining tool. However, for the big data applications, using this tool is not efficient and useful. Thus, Li and Guoyong [24] proposed an efficient method for calculating tf-idf scores on the Hadoop parallel computation framework. We use their algorithm to calculate tf-idf scores in our test datasets. The Hadoop distributed computing platform is based on partitioning the task and running on partitions concurrently.

If we inspect the formula for the calculation of tf-idf scores, we can see that tf-idf formula is very suitable for distributed computing.

The tf-idf scoring has two part as mentioned them in detail in section 3.5. The first part, the term frequency, is the number of times a term appears in a document therefore, we can compute it in a distributed manner using partitioning the dataset into nodes. This can be done in a very efficient manner in the MapReduce framework. After the term frequencies are computed, inverse document frequency (the second part of tf-idf) can be calculated using the number of documents containing a specific term, which is known after the first phase. Moreover, since the number of documents that contain a term are now known and fixed in static dataset scenario, we can compute tf-idf scores in parallel. The authors [24] design three MapReduce processes to this end, which are described below.

3.8.1 The number of words in a document

In the first phase of the computation, the aim is to find the number of occurrences of the term in a document. One can explain this process in more detail in terms of mapper and reducer functions. The mapper function produces key-value pairs as follows

$$<< term\#documentName >, 1 >,$$

where the key is a combination of a term (word) and the document that contains the term. These pairs are used as intermediate values which will be processed by the reducer function. Later, the number of occurrences of the term in document is calculated directly in the reducer part by combining the pairs with the same key value. Finally, the results of the reducer should be written to the intermediate file (temporaryFile1). The file contains key-value pairs as

$$<< term\#documentName >, n >,$$

where n is the number of occurrences of *term* in the document *documentName*. The map and reduce functions are formally described in Algorithm 5.

Algorithm 5 calculate the number of occurrences of the term in document

MAP FUNCTION 1

Input: $< documentLineNumber, contents >$
Output: $<< term\#documentName >, 1 >$

REDUCE FUNCTION 1

Input: $<< term\#documentName >, 1 >$
Output: $<< term\#documentName >, n >$

3.8.2 The total number of words of each document

In the second phase, the map function uses $temporaryfile1$ to compute the total number of term (word) in each document. Reorganizing of the $< key, value >$ pairs is needed in this process. The pair $< documentName, < term = n >>$ is generated as key-value pairs in the mapper function. Then, the total number of terms of each document is calculated in the reducer function. The output of the reducer should be written to an intermediate file ($temporaryFile2$) since its content will be processed in the reducer function. The output of the reducer is

$$<< term\#documentName >, < n/N >>,$$

where n is the number of occurrences of *term* in the document *documentName*, and N is the total number of terms of the document *documentName*. The map and reducer functions are described in Algorithm 6.

3.8.3 Calculation of TF-IDF in Hadoop Framework

In the last stage, the $< key, value >$ pairs are reorganized in the mapper function. The mapper generates key-value pairs as

$$< term, < documentName\#n/N >>$$

Algorithm 6 calculate the number of occurrences of the term in document

MAP FUNCTION 2

Input: $<< term\#documentName >, n >$
Output: $< documentName, < term = n >>$

REDUCE FUNCTION 2

Input: $< documentName, < term = n >>$
Output: $<< term\#documentName >, < n/N >>$

Then, the reducer, using the term as the key value, can compute the tf-idf score of the term using the formula

$$tf\text{-}idf = \frac{n}{N} \cdot \log\left(\frac{|\mathcal{D}|}{d}\right),$$

where $\mathcal{D}$ is the total number of documents in the dataset and d is the number of documents that contain the term. The last MapReduce phase is described in Algorithm 7.

Algorithm 7 calculate the number of occurrences of the term in document

MAP FUNCTION 3

Input: $<< term\#documentName >, < n/N >>$
Output: $< term, < documentName\#n/N >>$

REDUCE FUNCTION 3

Input: $< term, < documentName\#n/N >>$
Output: $<< term\#documentName >, < (d/D), (n/N), tf - idf >>$

3.9 Datasets

We need a dataset to test the proposed system and show its effectiveness, efficiency and scalability. We use both real and synthetic datasets in our analysis. The real dataset used in the experiments is only a small part of the Enron Corpus data, which is a large database of over 517,000 emails generated by 158 employees of the Enron Corporation [2]. The synthetic dataset is generated to test the efficiency of the proposed system.

4 Problem Definition

The system requirements for secure and privacy-preserving keyword search and document access scheme over encrypted cloud data are specified in the previous chapter. In this chapter, we give detailed information about the protocol design of the proposed scheme. Mainly, we provide more comprehensive descriptions for the steps of the algorithm of the protocols developed for the Hadoop framework. These algorithms are implemented in the real environment and tested by using real and synthetic datasets. As the real dataset, Enron mails are preferred since the size of the documents in the Enron dataset is substantially large for our test efforts.

In this thesis, we consider privacy-preserving keyword search over encrypted cloud data for the database outsourcing scenario as illustrated in Figure 4.1. In the system, we assume that there are three entities, namely the data owner, the server and users.

4.1 Requirements of the Proposed Scheme

1. **Data Owner** is the actual entity that is responsible for the establishment of the database. The data owner collects and/or generates the information in the database. The owner does not have sufficient resources or is unwilling to store the whole database. So, the owner outsources the data to an untrusted, semi-honest server (trusted but curious). The data owner encrypts the sensitive documents to be outsourced and generates a searchable index using the features of these sensitive documents.

2. **Server** is a professional entity (e.g., cloud server) that offers information services to authorized users. It is often required that the server should be oblivious to content of

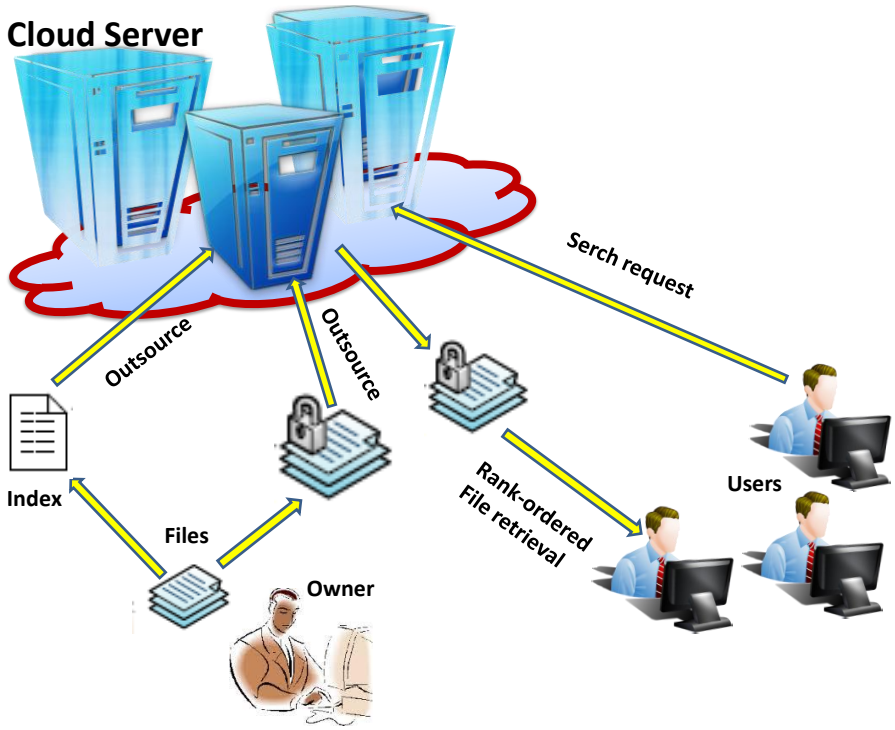

Figure 3: Architecture of the search over encrypted cloud data

the database it maintains, the search terms in queries and documents retrieved. Additionally, the cloud server should not learn anything other than that the data owner allows to leak.

3. **Users** are the members in a group who are entitled to access (part of) the information of the database. Users may send queries consisting of multiple keywords and receive the documents associated with these queried keywords. Finally, user decrypts the retrieved documents using the decryption key.

4.1.1 Cloudera CDH

Cloudera Inc. provides Cloudera CDH which is the most popular distribution of Apache Hadoop. It is an open-source technology which includes some different projects such as Apache Hive, Apache Avro, Apache HBase, etc. Also, it offers Hadoop platform by combining these all projects.

In this thesis, we used the Cloudera CDH 4.7.0 version by installing the Cloudera Manager. Thanks to Cloudera Manager, configuration of Hadoop is very easy. In Figures 4, 5

and 6, we give screen shots of the Cloudera Manager Admin Console. Using this console showed in Figure 4, we can perform actions for different configurations such as adding new service or role, adding and deleting hosts etc. In Figure 5 and 6, we show the screen shots of the Hue console used to manage the jobTracker and file server. We upload the data to HDFS by using this console, and we can add map-reduce files that are project codes to run on our data. In addition, we can check the output files after having run our map-reduce functions.

In addition, Hadoop Framework has three types to be able to create the cluster, which are Standalone mode (single node cluster), Pseudo distributed mode (single node cluster) and Fully-distributed mode (multi-node cluster). In this thesis, we run our project in the Fully-Distributed Mode Multi-Node Hadoop cluster.

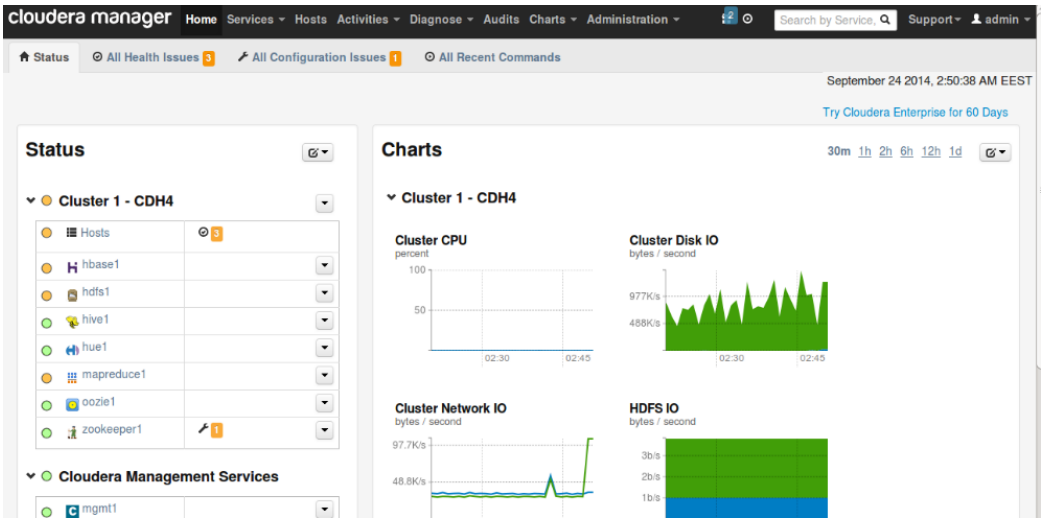

Figure 4: Screen shot of Cloudera Manager Admin Consol

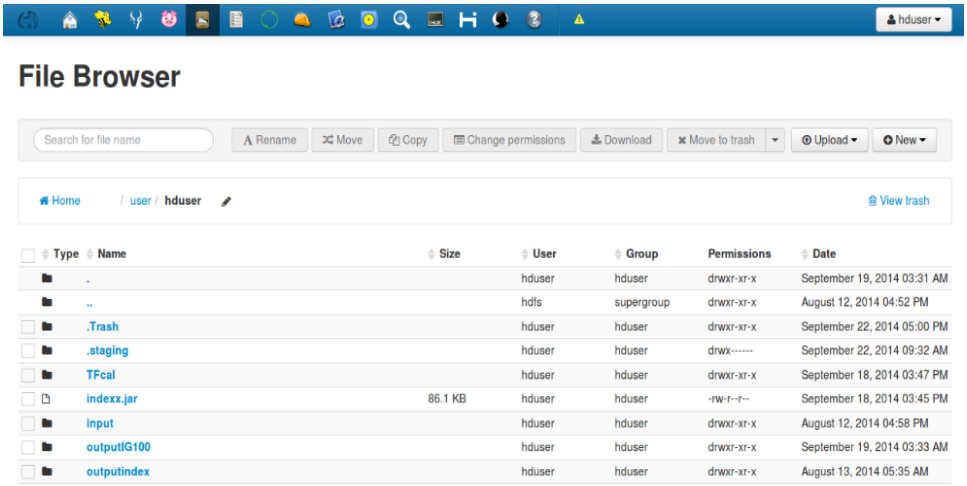

Figure 5: Screen shot of Hue which is in File Browser Tab

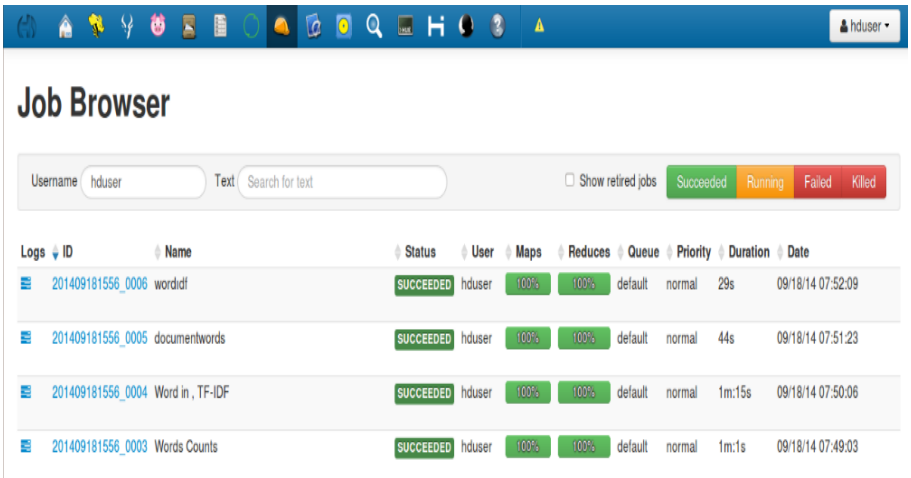

Figure 6: Screen shot of Hue which is in Job Browser Tab

4.1.2 Using Hadoop Commands

In this section, we define fundamental and required commands related to execution of the Cloudera Hadoop Framework [13].

- Starting the cluster is performed in two phases. Firstly, we begin with starting the HDFS daemons using the command: "./start-dfs.sh "
  - The NameNode daemon is started on the master node.
  - The DataNode daemons are started on all slaves (here: master and slave).
- Then, we start the MapReduce daemons using this command: "./start-yarn.sh "
  - The JobTracker is started on the master,
  - The DataNode daemons are started on all slaves (here: master and slave).
- After starting, we check following Java processes that should run on master using the command: "jps".
  - $Secondarynamenode$
  - $NodeManager$
  - $NameNode$
  - $DataNode$
  - $ResourceManager$
  - jps
- to return the list of a directory with direct children we use the command: "ls".
  - hdfs dfs $-ls$ hdfs: //localhost:9000/user/hduser/
- to copy source paths to stdout we use the command: "cat".
  - hdfs dfs $-cat$ hdfs: //localhost:9000/ user/hduser/output/part-r-00000
- to delete files specified recursively we use the command: "rm".

- hdfs dfs $-rm\ -R$ hdfs: //localhost:9000/user/hduser/output

- to copy single src, or multiple srcs from local file system to the destination filesystem we use the command: “copyFromLocal”

  - hdfs dfs $-copyFromLocal$ file: ///home/hduser/Desktop/file.txt
    hdfs: //localhost:9000/user/hduser

4.2 Challenges

As the name implies, the concept of big data implies a massive dynamic dataset that contains a great variety of data types. There are several dimensions in big data that makes management a very challenging issue. The primary aspects of big data is best defined by its volume (amount of data), velocity (data change rate) and variety (range of data types) [23].

Unfortunately, standard off-the-shelf data mining and database management tools cannot capture or process these massive, unstructured datasets within a acceptable time period [33]. This led to the development of new technologies adapted for the requirements of big data. In order to meet the scalability and reliability requirements, a new class of NoSQL based data storage technology referred as *Key-Value Store* [15] is developed and widely adopted.

This system utilizes associative arrays to store the key-value pairs on a distributed system. A key-value pair consists of a value and an index key that uniquely identifies that value. This allows distributing data and query load over many servers independently, thus achieve scalability. Furthermore, none of the Key-Value Stores offer secondary index on data, but only offer indexing on the primary key. We also adapt the *key-value store* approach in the proposed method, where the details are explained in Section 3.3. While the bucket identifiers are used as the key, the corresponding encrypted documents identifiers and scores are stored as the value.

4.3 Preprocessing Operations Before Computing Tf-idf

In this section, we explain the preprocessing steps in detail and software tools used in the procedure. We need the preprocessing steps to obtain more efficient computation of tf-idf scores with the Hadoop framework. The preprocessing steps are illustrated in Figure 7.

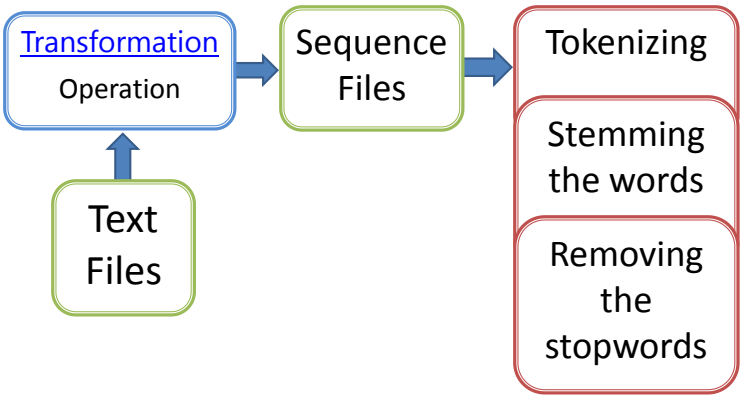

Figure 7: Preprocessing Operations Before Computing Tf-idf

4.3.1 SequenceFile

The Hadoop framework does not offer great performance for a dataset with files which are smaller than the typical HDFS Block size. If Hadoop held huge amounts of small files, it would cause the memory overhead for the NameNode.

We use Enron dataset which has relatively short emails as the input files. Therefore, to solve the overhead problem with the small files we utilize SequenceFile format, which is a flat file consisting of binary key/value pairs. As an original contribution to the MapReduce algorithms for tf-idf computation in [24], we use SequenceFile format for efficient computation of tf-idf values of small files. Thus, we easily overcome the problems such as the storage overhead and time-consuming processes thanks to SequenceFile format used for the input files in Hadoop. SequenceFile is a binary storage format that consists of binary key/value pairs. If there are excessively many small files in text format, SequenceFile is used as a con-

tainer to store them. The advantage of SequenceFile is that the small files can be compressed and will still be splittable unlike text format files. On the other hand, files in normal text format, which are compressed, cannot be splittable to assign them to different nodes. Therefore, we prefer SequenceFile for input files before calculating tf-idf score.

4.3.2 Some Filters using Lucene in Hadoop

Documents should be processed using some filtering techniques such as tokenizing, stemming, removing stop words, (typical steps applied by RapidMiner, which is a widely used machine learning tool to compute tf-idf values [1] before running the tf-idf computation algorithm. For filtering, we prefer Apache Lucene tool, which is a widespread open source information retrieval software library [3]. The Lucene has the most important API to be able to perform stemming operation and remove stop words which are ”a”, ”an”, ”and”, ”are”, ”their”, ”the”, etc. Therefore, we developed a preprocessing program by utilizing the Apache Lucene, which returns documents that contain only key words or search terms.

4.4 Protocol of the Proposed Scheme for Hadoop Framework

In the previous system [29], it is possible to process datasets with relatively small number of documents. Since we work with big data, which is huge both in number of data items and their sizes, and rapidly changing, we need to use a distributed computation environment that can cope with the associated challenges. Therefore, we exploit the Hadoop framework which is based on the distributed file systems and parallel programming techniques such as the Hadoop Distributed File System (HDFS) and the MapReduce programming. The HDFS architecture runs on distributed clusters to manage massive datasets. The HDFS is based on a master / slave communication model that is composed of a single master node and multiple data (i.e., slave) nodes. The MapReduce programming model, used to processes massive data, provides large-scale computations for large clusters by dividing the data into independent splits. The input data for MapReduce is stored in the HDFS. The MapReduce programming model utilizes the key-value pairs for distributing the input data to all the nodes in the cluster, in a parallel manner. In this scheme, unlike the previous work in [29], we address the

bucket identifier as key in key-value pairs and we use three computers for Hadoop clusters which have a master and two slave machines. Moreover, we also propose a lazy idf-updating method that can efficiently maintain the tf-idf scores in a large and dynamically changing dataset. With the lazy idf-updating method, very close estimates on the real tf-idf scores can be calculated in a very efficient manner. Therefore, the proposal is suitable in the Big Data setting. In the subsequent sections, we give the Index Generation algorithm for Hadoop and explain the lazy idf-updating method in more detail.

4.4.1 Index Generation with Hadoop

In the Hadoop framework, we use Map-Reduce functions for calculating a searchable index item for each document. Here, as input we use document name as the key and the contents of the documents are the value for the map function. In the map function, minhash signatures of documents are computed. Since Minhash functions used in document signatures are created randomly and used by all cluster nodes, they are kept in the distributed cache for easy access by the cluster nodes. The key for the output of the map function is the bucket identifier (bucketID) and the value is $< documentName, score >$ pairs. Then, reordering is applied before the reduce function. Finally, in the Reduce Function, we use the output of the map function as the input. Lastly, we use the bucketID as the key for the output of the reduce function and all $< documentName, score >$ pairs in the corresponding bucket. The MapReduce phase of the Index Generation operation is defined in Algorithm 8.

4.4.2 Secure Search with Hadoop

Firstly, the user generates the bucketIDs of a query using the query generation algorithm given in Algorithm 2. Then, the bucketIDs are sent to the server to be matched with the buckets which contain the related documents. In order to perform the search operation in Hadoop, bucketIDs are placed in the distributed cache that distributes and copies the files among the nodes since all nodes should use the same bucketIDs for each query. In addition, before the searching operation, the data owner should create all bucketIDs for every documents and outsources them to the server, which is explained in Section 3.6.3.

In the map function, each cluster node retrieves the bucketIDs of the query from the

Algorithm 8 Index Generation Algorithm

MAP FUNCTION

Distributed Cache: Minhash Functions

Input: $< documentName, contents >$

Output: $< bucketID, < documentName, score >>$

REDUCE FUNCTION

Input: $< bucketID, < documentName, score >>$

Output: $< bucketID, (< documentName_1, score1 >, ..., < documentName_n, score_n >) >$

distributed cache and the output file of the index generation phase, which has bucketID as the key, and all $< documentName, score >$ pairs as the value for the input file of the map function. For the matching buckets, the map function returns the bucketID as the key and a document in the corresponding bucket and its score as the value. It performs the same operation for every document in the bucket. The reduce function gets the reordered outputs of the map function as the input file. Then, it merges pairs of the value that has the same bucketID. Lastly, the output file of the reduce function contains the potentially relevant documents. Note that the key value of the reduce function *bucketID* is the query bucket ids. The MapReduce phase of the secure search operation is described in Algorithm 9.

4.4.3 Insertion operation with Hadoop

The insertion operation is used to add a new data file to the index file. Therefore, the bucketIDs of the new file should be generated by the data owner. Map function 1 in Algorithm 10 is used to generate the bucketIDs of the new file. The map function in Algorithm 10 uses document name as the input key and the contents of the documents as the input value. Then, the map function returns an output that includes bucketID as the key and $< documentName, score >$ pair as the value.

Simultaneously, map function 2 gets the input file that is the output of the index generation operation. In addition, map function 2 scans all buckets and produces $< bucketID, <$

Algorithm 9 Secure Search Algorithm

MAP FUNCTION

Distributed Cache: BucketIDs for search query

Input: $< bucketID, (< documentName_1, score1 >, ..., < documentName_n, score_n >) >$

Output: $< bucketID, < documentName, score >>$

REDUCE FUNCTION

Input: $< bucketID, < documentName, score >>$

Output: $< bucketID, (< documentName, score_1 >, ..., < documentName, score_n >) >$

documentName, *score* >> as the key-value pair, which reflects the index file before the insertion operation. The aim is to merge all $< documentName, score >$ pairs with the same *bucketID*, generated by map function 1 and map function 2. Before Reduce function, the outputs of both map functions are reordered automatically.

In the reduce function, $< bucketID, < documentName, score >>$ pairs that are the outputs of the map functions are used as the input file. As the output file, bucketID as the key and the chain of $< documentName_i, score_i >$ as the value, which are merged according to the key, are generated. The MapReduce phase of the insertion operation is described in Algorithm 10.

4.4.4 Deletion operation with Hadoop

The deletion operation is implemented to remove documents from the data set and thus to update the index. Initially, we find the bucketIDs of the documents that will be deleted. In the deletion operation, we have two map functions and also two reduce functions. Map function 1 is used to generate to the bucketIDs of the deleted documents. It is similar to map function 1 of the insertion operation. We again use the distributed cache to share the Minhash functions among the nodes in map function 1. For the input file, $< documentName, contents >$ pairs are used in the map function 1. The *documentName* is used as the key and the contents of the documents are used as the value. And as the output file, $< bucketID, documentName >$ are generated as key-value pairs. Then, these outputs are reordered according to a common key

Algorithm 10 Insertion Operation Algorithm

MAP FUNCTION 1

Distributed Cache: Minhash Functions

Input: $< newdocumentName, contents >$

Output: $< bucketID, < newdocumentName, score >>$

MAP FUNCTION 2

Input: $< bucketID, (< documentName_1, score1 >, ..., < documentName_n, score_n >) >$

Output: $< bucketID, < documentName, score >>$

REDUCE FUNCTION

Input: $< bucketID, < documentName, score >$

Output: $< bucketID, (documentName_1, score1 >, ..., < documentName_{n+1}, score_{n+1}) >$

automatically before the reduce function 1. And it generates the chain of the documentName as value for each bucketID in reduce function 1, which are basically the deleted documents in the corresponding bucket.

Reduce function 1 places its output in the distributed cache, from which map function 2 retrieves the documents in the corresponding buckets. All cluster nodes access the distributed cache to remove the indexes of the deleted files. In addition, as the input file, map function 2 takes the output of the index generation operation. BucketID is used as the key and the chain of the $< documentName, score >$ pair are used as the value in map function 2. Simply speaking, map function 2 returns the documents that are not deleted for each bucket.

Before the reduce function, reordering is made automatically over undeleted documents and finally, all undeleted documents are merged according to the common key, which is the output file of the reduce function 2. The MapReduce phases of the deletion operation are described in Algorithm 11, where x is the number of deleted documents.

Algorithm 11 Deletion Operation Algorithm

MAP FUNCTION 1

Distributed Cache: Minhash Functions

Input: $< documentName, contents >$

Output: $< bucketID, documentName >$

REDUCE FUNCTION 1

Input: $< bucketID, documentName >$

Output: $< bucketID, (< documentName_1, ..., documentName_x >) >$

MAP FUNCTION 2

Distributed Cache: $< bucketID, < documentName_1, ..., documentName_x >>$

Input: $< bucketID, (< documentName_1, score1 >, ..., < documentName_n, score_n >) >$

Output: $< bucketID, undeletedDocumentName >$

REDUCE FUNCTION 2

Input: $< bucketID, undeletedDocumentName >$

Output: $< bucketID, (undeletedDocumentName_1, ..., undeletedDocumentName_{n-x}) >$

4.5 Lazy idf Update

When the dataset is changed, by adding new files, deleting or updating existing files, the index should also be updated. Even a small change to the data set, such as adding or deleting a single document, can result in the idf value of a search term. Therefore, potentially we may have to recalculate the index for every document containing the term whose idf is changed. We formalize the discussion in the following.

Let a new document $\mathcal{D}$ contain k previously indexed terms. If this new document $\mathcal{D}$ is added to the dataset, the scores of all the documents that contain any of those k terms should be updated since their tf-idf scores change. However, dynamically applying this change for

each single data item, added or removed from the dataset, is not feasible. Hence, we propose a *lazy idf-updating* method which aims to maintain the scores of existing documents as they are and only set a new score for the newly added items. Moreover, calculating the idf of each term of a newly added data item is still a costly operation that requires scanning the whole dataset. In order to reduce the cost of scoring, we propose keeping the idf values of the terms separately. As new data elements are added, the idf values slightly change and the stored idf values will not be exactly correct. However, they still provide accurate estimates since the size of the existing dataset is much larger than the size of the data elements added. In a timely bases[1] (e.g., every 20 minutes), the whole dataset is scanned and all the idf values are updated with the exact results.

Due to the privacy requirements, the server cannot see the actual documents, but only stores the encrypted versions. It is not possible to calculate, neither the term frequencies, nor the inverse document frequencies from the encrypted data, therefore a trusted proxy should be used for updating the tf-idf scores. Each new data item is first indexed and encrypted by the proxy and then uploaded to the server. Similarly, the idf value updating operation is also done by the proxy. Therefore, the idf values that are separately stored are only kept in the trusted proxy. Since the idf value updating operation is performed by the proxy, the cloud server will be up and running during this period and the search operation can be done using the existing relevancy scores.

We assume that the size of the dataset will be very large, hence the effect of the additional items on the idf values will be very limited. Note that, the term frequency (tf) part of the tf-idf score is calculated using only the document itself. Therefore, the change in the dataset does not affect the tf values of the existing items. With this *lazy idf-updating* method, very close estimates on the real tf-idf scores can be calculated in a very efficient way, hence it is suitable in the big data setting. The actual comparative results using a large, real dataset is provided in Section 5.3.

[1]The update frequency can be set according to the change rate of the dataset.

5 Experimental Evaluation

In this section, we extensively analyze and demonstrate efficiency and effectiveness of the proposed method. The entire system is implemented by Java language using 64-bit Ubuntu 12.04 LTS operating system. In order to observe the benefits of distributed file systems, a multi-node Hadoop cluster is configured. The interface of Cloudera CDH4 with a three node (i.e., computer) cluster is utilized in the experiments. Two of the computers have an Intel Xeon CPU E5-1650 @ 3.5 GHz processor with 12 cores, 15.6 GB of main memory and the other computer has an Intel i7 @ 3.07 GHz processor with 8 cores and 15.7 GB of main memory.

In our experiments, we used the Enron dataset [2], which is a real dataset that contains approximately 517.000 email documents. For encrypting bucket contents, the AES algorithm is used in the CTR mode.

To evaluate the accuracy of the search scheme we use two metrics, namely precision and recall rates. Success of a search scheme can best be analyzed using these metrics.

Let $R(F)$ be used to show the set of items, which are retrieved for a query with feature set F. Furthermore, let $R^*(F)$ be a subset of $R(F)$, whose elements contain all the features in F. Lastly, let $\mathcal{D}(F)$ denote the set of items that contain all the features in F.

Therefore, we have $R^*(F) \subseteq R(F)$ and $R^*(F) \subseteq \mathcal{D}(F)$; and precision ($prec(F)$), recall ($rec(F)$), average precision ($aprec(F)$) and average recall ($arec(F)$) for a set $\mathcal{F} = \{F_1, \ldots, F_n\}$ can be given as follows: [27]

$$prec(F) = \frac{R^*(F)}{R(F)}, \quad aprec(\mathcal{F}) = \sum_{i=1}^{n} \frac{prec(F_i)}{n} \tag{6}$$

$$rec(F) = \frac{R^*(F)}{\mathcal{D}(F)}, \quad arec(\mathcal{F}) = \sum_{i=1}^{n} \frac{rec(F_i)}{n} \tag{7}$$

5.1 Performance of the Method

In this section, we present the experiment results, where we measure the time spent for generating the secure searchable index and applying search operation.

The index generation time for 517.000 documents, is given in Figure 8 for different values of λ. The experiments demonstrate that the index generation, which is the most time consuming part of the method, can be done in only a few minutes. And the system can index about 2750 documents per second for $\lambda = 100$. Note that this operation is performed only after the change in the dataset, due to the documents being added, exceeds a threshold. Moreover, since this operation is done by a trusted proxy, the cloud server can still continue to serve the incoming search requests, using the existing index.

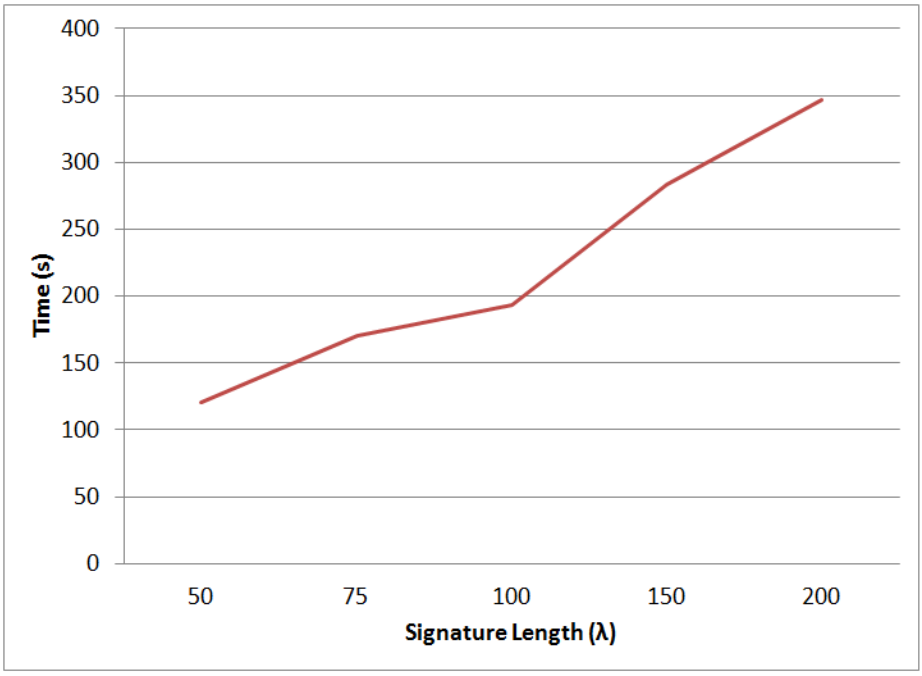

Figure 8: Index Generation Time as λ change

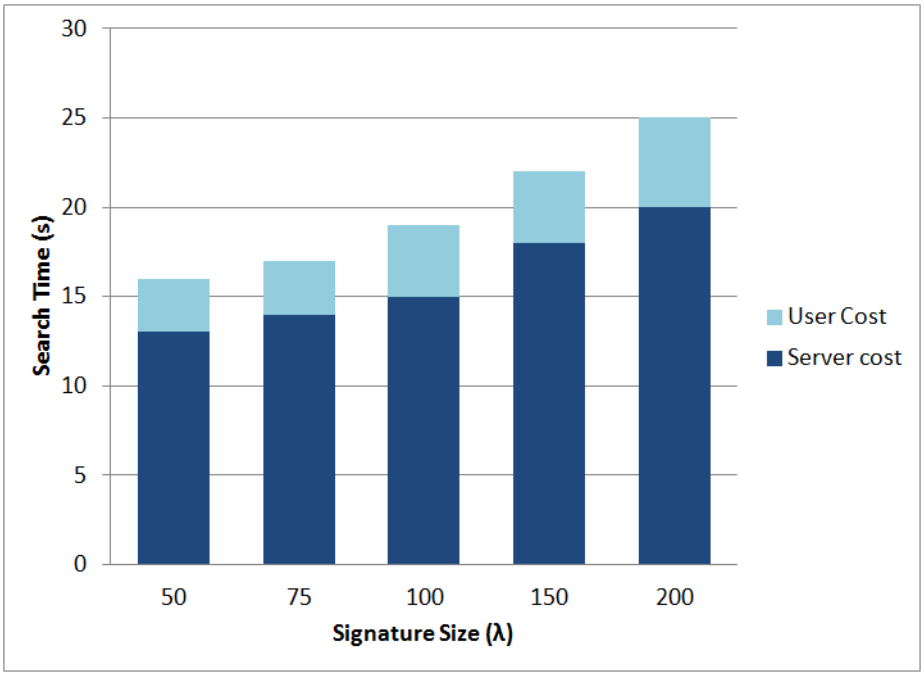

Figure 9: Search Time

The search operation has two major parts. Firstly, the server fetches the content vectors of the queried buckets and sends them to the user. Then, the user (or trusted proxy) decrypts those vectors and sorts the document identifiers according to the corresponding relevancy scores. Unfortunately, due to the distributed setting of the Hadoop file system, finding the queried buckets requires a search over all the buckets. Figure 9 demonstrates the average search time required both for the server and user sides, in the dataset of 517.000 documents.

5.2 Accuracy of the Method

In the information retrieval community, two of the most common metrics for measuring the accuracy of a method are precision and recall. The metrics compare the expected and the actual results of the evaluated system. Precision measures the ratio of correctly found matches over the total number of returned matches. Recall, on the other hand, measures the ratio of correctly found matches over the total number of the correct matches. Both precision and recall are real values between 0 and 1, where the higher the value the better the accuracy is.

In the case of a single term search, the proposed method guarantees all the matches that have non-zero relevancy scores, contain the searched term. Hence, retrieving all the items

with non-zero scores satisfies perfect precision and recall. In the case of multiple keyword searches, the matches with non-zero scores definitely contain at least one of the queried keywords but it may or may not contain all. We test the accuracy of the method for multi-term queries with $\lambda = 100$ (i.e., signature length) using the precision and recall metrics. The average precision and recall rates for a set of 20 queries with 2 and 3 keywords are given in Figure 10 and 11, respectively. The retrieval ratio in the figures represents the ratio of the documents with nonzero scores that are considered as a relevant match with the query. The figures show that while the precision decreases as retrieval ratio increases (i.e., more documents are considered as match), the recall increases. The retrieval ratio can be selected by the user according to the requirements of the application. The figures also show that the increase in the number of keywords decreases the precision, but increases recall. The main reason of this is that, as the number of queried terms increases only very few documents contain all the queried terms which have a positive effect on recall. However, this also increases the documents with nonzero scores (i.e., contain at least one of the queried terms) which have a negative effect on precision.

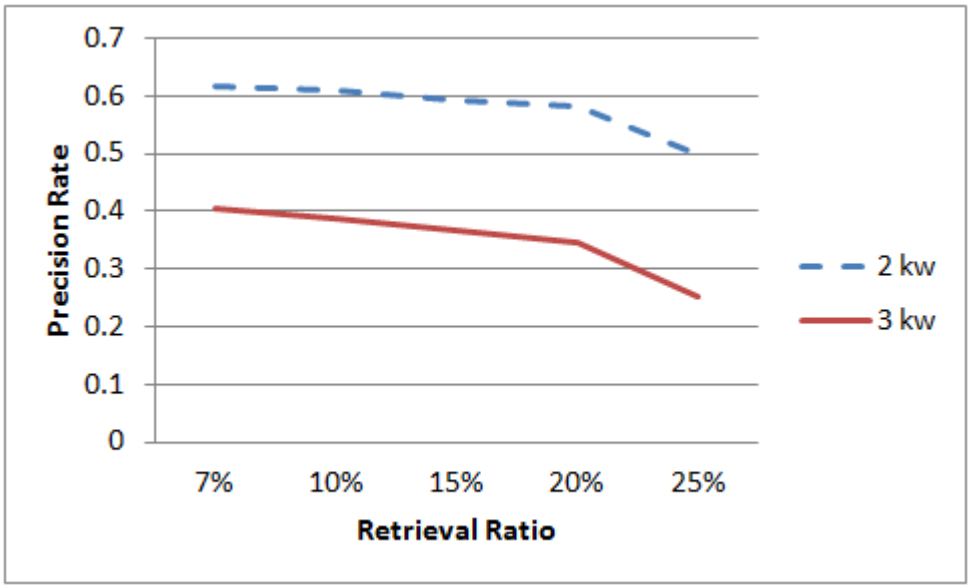

Figure 10: Average Precision Rate, $\lambda = 100$

Although the precision and recall metrics are very commonly used and very suitable for several problems such as conjunctive search and relational database search over structured

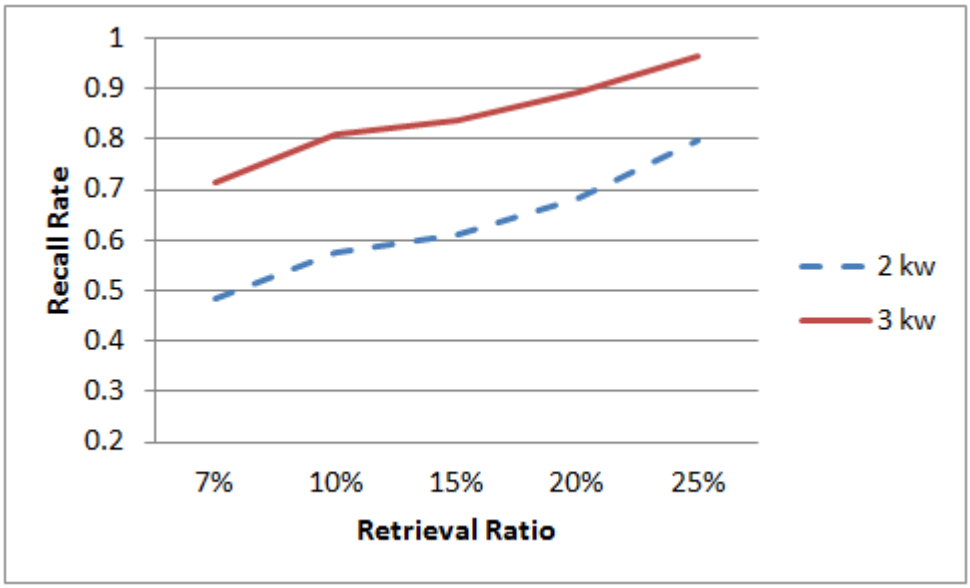

Figure 11: Average Recall Rate, $\lambda = 100$

data, they may not be very accurate for multi-keyword search over unstructured data. The main difference between search over structured and unstructured data is that, in the case of structured data, each field has an equal importance and the corresponding results should satisfy all the queried features. However, in the case of search over unstructured data, some of the queried features may be significantly more important than the others. For example, let a query have three features and a document contain only two of those features but with very high tf-idf scores. The precision and recall metrics will consider this document as a false match since it does not contain all the queried features, but in the case of big data application where the data is unstructured, we claim that, this document is very relevant with the given query and should be considered as a match.

It is important to note that, precision and recall metrics cannot consider the importance of the queried features in the compared document hence, may not perfectly measure the success rate of a search method over data. Therefore, we also compare the output of the method with the ground truth. For calculating ground truth, the documents with top 50 scores in the dataset are considered as the actual match results, where the complete tf-idf scores of the documents are used without any encryption. These actual results are then compared with the results evaluated by the system. The average precision and recall rates in comparison with the ground truth, for a set of 20 queries with 2 and 3 keywords and $\lambda = 100$, are given in

Figure 12 and 13, respectively. The figures show that, the actual accuracy of the method is quite promising when the tf-idf scores are considered in calculating the actual results, instead of the conjunctive (i.e., contain all terms) case.

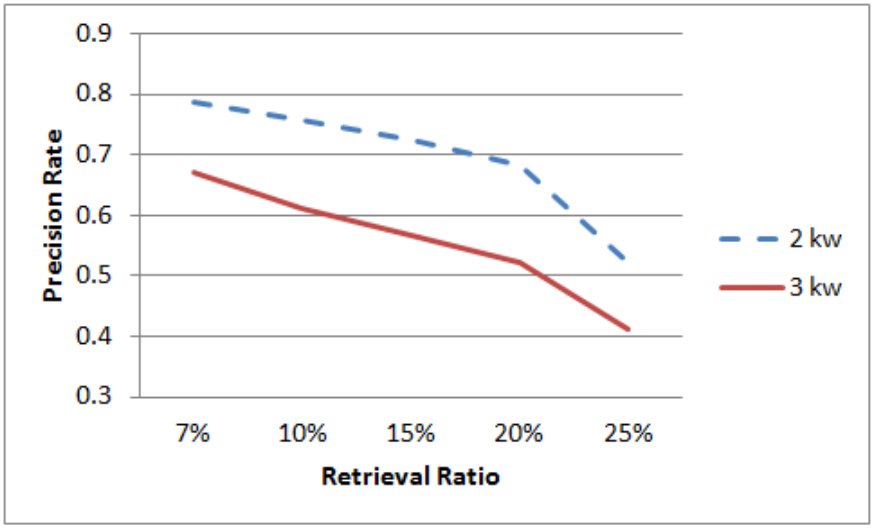

Figure 12: Average Precision Rate using Ground Truth, $\lambda = 100$

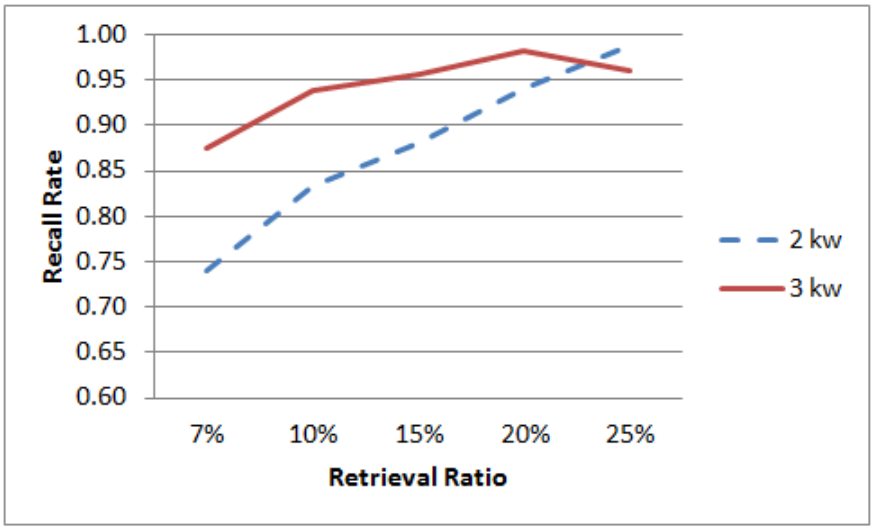

Figure 13: Average Recall Rate using Ground Truth, $\lambda = 100$

We also measure the effect of λ on accuracy. Figures 14 and 15 show that increase in λ has a positive effect on both precision and recall. However, increase in λ also linearly increases search and index generation times as shown in Section 5.1 and improvement in accuracy is very limited. Hence, an optimum value for λ should be set according to the properties of the

dataset used, which is set as 100 in our case.

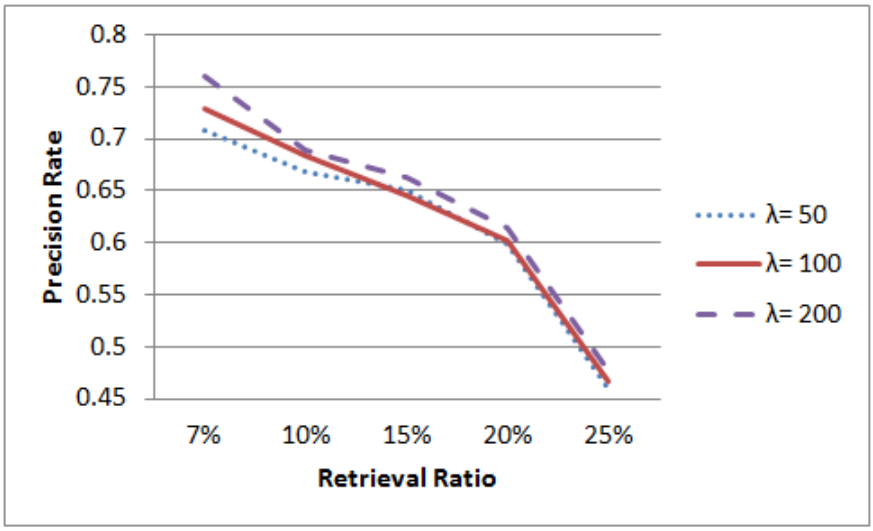

Figure 14: Average Precision Rate for different λ

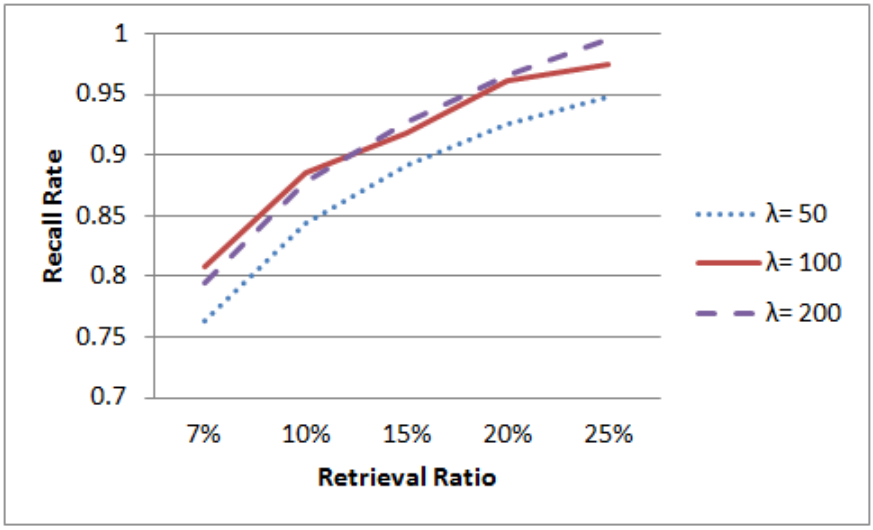

Figure 15: Average Recall Rate for different λ

5.3 Dataset Update

In Section 4.5, we propose a lazy update scheme that does not update the idf values of the existing scores at each data set update, but uses the existing scores as an approximation. In this subsection, we provide the change rate of the idf due to updating in the dataset. We calculate the average idf value of a dataset of size 400.000 documents while adding new

set of documents of size 10.000. As Table 3 indicates, the effect of adding new documents is very low especially if the dataset size is large hence, the lazy update does not reduce accuracy.

# documents	400000	410000	420000	430000
avg idf	2.26678	2.26716	2.26713	2.26717

Table 3: Average IDF values

Inserting index entries for newly added documents first requires calculating the corresponding signatures by a trusted proxy and than updating the encrypted bucket content vectors accordingly. We performed experiments for the updating times for bulk insertions of 1000, 5000 and 10000 documents and the entire updating operations are calculated i $52,5$, $59,5$ and 67 seconds, respectively. This shows that the updating operation should better be done for large sets of documents, which are also suitable in the big data setting.

6 Conclusion and Future Work

Cloud computing technologies became more and more popular nowadays. Naturally, many organizations prefer to outsource their data to the cloud services such as Amazon Elastic Compute Cloud [16] or Google Cloud Platform [5] to take advantage of their high storage capacity and computation power and lower IT costs. However, as these companies such as Amazon or Google have access to the search and access patterns of their customers, the privacy and security concerns will not be mitigated even if the outsourced data is encrypted. Furthermore, encryption by itself, hinders the most basic operations over the outsourced data such as keyword-based search to access the relevant documents without downloading the entire database. While secure search operation over encrypted data is generally possible thanks to searchable encryption techniques, speed and accuracy remain always a problem, especially in big data applications.

In this work, we addressed the problem of applying an existing privacy-preserving search method for big data. We utilized the private search method by Örencik et al. [27] as the underlying technique and adapted it for the Hadoop distributed file system and the MapReduce programming model. We implemented the entire system and tested it in a three-node Hadoop cluster with the Enron email data set and demonstrate the effectiveness and scalability of the system. We also proposed a *lazy idf update* method that can be used for dynamically changing large data sets and provided experimental results using a large real data set.

In the light of the promising results, we believe this method will increase the applicability of privacy preserving search over big data.

References

[1] Rapidminer. https://rapidminer.com/, last accessed on January 2015, 2011.

[2] Enron email dataset. http://www.cs.cmu.edu/enron, last accessed on January 2015, 2012.

[3] Apache Lucene. https://lucene.apache.org, last accessed on January 2015, 2013.

[4] RADOOP Big Data Analytics. http://www.radoop.eu, last accessed on January 2015, 2014.

[5] Google Cloud Platform. https://cloud.google.com, last accessed on January 2015, Sept 2008.

[6] Divyakant Agrawal, Sudipto Das, and Amr El Abbadi. Big Data and cloud computing: current state and future opportunities. In *Proceedings of the 14th International Conference on Extending Database Technology*, pages 530–533. ACM, 2011.

[7] Amazon Web Services. What is cloud computing. http://aws.amazon.com/what-is-cloud-computing, last accessed on January 2015, 2013.

[8] Microsoft Azure. https://azure.microsoft.com/tr-tr/, last accessed on January 2015, 2014.

[9] Ning Cao, Cong Wang, Ming Li, Kui Ren, and Wenjing Lou. Privacy-preserving multi-keyword ranked search over encrypted cloud data. *Parallel and Distributed Systems, IEEE Transactions on*, 25(1):222–233, 2014.

[10] David Cash, Stanislaw Jarecki, Charanjit Jutla, Hugo Krawczyk, Marcel-Cătălin Roşu, and Michael Steiner. Highly-scalable searchable symmetric encryption with support for boolean queries. In *Advances in Cryptology, CRYPTO 2013*, volume 8042 of *Lecture Notes in Computer Science*, pages 353–373. 2013.

[11] Rick Cattell. Scalable SQL and NoSQL data stores. *SIGMOD Rec.*, 39(4):12–27, May 2011.

[12] Zhenhua Chen, Chunying Wu, Daoshun Wang, and Shundong Li. Conjunctive keywords searchable encryption with efficient pairing, constant ciphertext and short trapdoor. In *Intelligence and Security Informatics*, pages 176–189. Springer, 2012.

[13] Hadoop shell commands. http://hadoop.apache.org/docs/current/hadoop-project-dist/hadoop-common/FileSystemShell.html, last accessed on January 2015, 2013.

[14] Jeffrey Dean and Sanjay Ghemawat. Mapreduce: Simplified data processing on large clusters. *Commun. ACM*, 51(1):107–113, January 2008.

[15] Giuseppe DeCandia, Deniz Hastorun, Madan Jampani, Gunavardhan Kakulapati, Avinash Lakshman, Alex Pilchin, Swaminathan Sivasubramanian, Peter Vosshall, and Werner Vogels. Dynamo: Amazon's highly available key-value store. In *Proceedings of Twenty-first ACM SIGOPS Symposium on Operating Systems Principles*, SOSP '07, pages 205–220, New York, NY, USA, 2007. ACM.

[16] Amazon Elastic Compute Cloud. http://aws.amazon.com/ec2, last accessed on January 2015, 2014.

[17] Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung. The google file system. In *ACM SIGOPS Operating Systems Review*, volume 37, pages 29–43. ACM, 2003.

[18] Cloudera Manager Installation Guide. Cloudera manager. http://www.cloudera.com, last accessed on January 2015.

[19] Hakan Hacigümüş, Bala Iyer, Chen Li, and Sharad Mehrotra. Executing SQL over encrypted data in the database-service-provider model. In *Proceedings of the 2002 ACM SIGMOD international conference on Management of data*, SIGMOD '02, pages 216–227. ACM, 2002.

[20] The Apache Hadoop Project. http://hadoop.apache.org/core, last accessed on January 2015, 2009.

[21] Bijit Hore, Sharad Mehrotra, Mustafa Canim, and Murat Kantarcioglu. Secure multidimensional range queries over outsourced data. *The VLDB Journal*, 21(3):333–358, June 2012.

[22] Mehmet Kuzu, Mohammad Saiful Islam, and Murat Kantarcioglu. Efficient similarity search over encrypted data. In *Data Engineering (ICDE), 2012 IEEE 28th International Conference on*, pages 1156–1167. IEEE, 2012.

[23] Doug Laney. 3d data management: Controlling data volume, velocity and variety. *META Group Research Note*, 6, 2001.

[24] Bin Li and Yuan Guoyong. Improvement of tf-idf algorithm based on hadoop framework. 2012.

[25] Christopher D Manning, Prabhakar Raghavan, and Hinrich Schütze. *Introduction to information retrieval*, volume 1. Cambridge university press Cambridge, 2008.

[26] Memcached. http://memcached.org, last accessed on January, 2015, 2013.

[27] Cengiz Orencik, Murat Kantarcioglu, and Erkay Savas. A practical and secure multi-keyword search method over encrypted cloud data. In *CLOUD 2013*, pages 390–398. IEEE, 2013.

[28] Cengiz Orencik and Erkay Savas. An efficient privacy-preserving multi-keyword search over encrypted cloud data with ranking. *Distributed and Parallel Databases*, 32(1):119–160, 2014.

[29] Cengiz Orencik, Ayse Selcuk, Murat Kantarcioglu, and Erkay Savas. Multi-keyword search over encrypted data with scoring and search pattern obfuscation. International Journal of Information Security (under review), 2014.

[30] Anand Rajaraman and Jeffrey David Ullman. *Mining of massive datasets*. Cambridge University Press, 2011.

[31] B. Thirumala Rao and LSS Reddy. Survey on improved scheduling in hadoop mapreduce in cloud environments. *International Journal of Computer Applications*, 34, 2011.

[32] Konstantin Shvachko, Hairong Kuang, Sanjay Radia, and Robert Chansler. The hadoop distributed file system. In *Mass Storage Systems and Technologies (MSST), 2010 IEEE 26th Symposium on*, pages 1–10. IEEE, 2010.

[33] Chris Snijders, Uwe Matzat, and Ulf-Dietrich Reips. Big data: Big gaps of knowledge in the field of internet science. *International Journal of Internet Science*, 7(1):1–5, 2012.

[34] Luis M. Vaquero, Luis Rodero-Merino, Juan Caceres, and Maik Linder. A break in the clouds: Towards a cloud definition. *Future Generation Computer Systems*, 29(3):739–750, 2008.

[35] Lizhe Wang, Marcel Kunze, Jie Tao, and Gregor von Laszewski. Towards building a cloud for scientific applications. *Advances in Engineering Software*, 42(9):714–722, 2011.

[36] Lizhe Wang, Jie Tao, Rajiv Ranjan, Holger Marten, Achim Streit, Jingying Chen, and Dan Chen. G-hadoop: Mapreduce across distributed data centers for data-intensive computing. *Future Generation Computer Systems*, 29(3):739–750, 2013.

[37] Peishun Wang, Huaxiong Wang, and Josef Pieprzyk. An efficient scheme of common secure indices for conjunctive keyword-based retrieval on encrypted data. In *Information Security Applications*, Lecture Notes in Computer Science, pages 145–159. Springer, 2009.

[38] Bo Zhang and Fangguo Zhang. An efficient public key encryption with conjunctive-subset keywords search. *Journal of Network and Computer Applications*, 34(1):262–267, January 2011.

[39] Xuyun Zhang, Laurence T. Yang, Chang Liu, and Jinjun Chen. A scalable two-phase top-down specialization approach for data anonymization using mapreduce on cloud. *IEEE Trans. Parallel Distrib. Syst.*, 25(2):363–373, February 2014.