Future Generation Computer Systems 30 (2014) 179–190

Contents lists available at ScienceDirect

Future Generation Computer Systems

journal homepage: www.elsevier.com/locate/fgcs

Efficient multi-keyword ranked query over encrypted data in cloud computing

Ruixuan Li [a,*], Zhiyong Xu [b], Wanshang Kang [a], Kin Choong Yow [c], Cheng-Zhong Xu [c,d]

[a] School of Computer Science and Technology, Huazhong University of Science and Technology, Wuhan, Hubei 430074, China
[b] Department of Mathematics and Computer Science, Suffolk University, Boston, MA 02114, USA
[c] Shenzhen Institute of Advanced Technology, Chinese Academy of Science, Shenzhen, Guangdong 518055, China
[d] Department of Electrical and Computer Engineering, Wayne State University, Detroit, MI 48202, USA

HIGHLIGHTS

- Design a novel storage and encryption algorithm to manage the keyword dictionary.
- Greatly reduce both the dictionary reconstruction overhead and the file index re-encryption time as new keywords and files are added.
- Design a novel trapdoor generation algorithm.
- Take the keyword access frequencies into account when the system generates the ranked list of the returning results.

ARTICLE INFO

Article history:
Received 31 December 2012
Received in revised form
24 June 2013
Accepted 28 June 2013
Available online 17 July 2013

Keywords:
Cloud computing
Multi-keyword query
Ranked query
Top-*k* query
Data encryption
Privacy preserving

ABSTRACT

Cloud computing infrastructure is a promising new technology and greatly accelerates the development of large scale data storage, processing and distribution. However, security and privacy become major concerns when data owners outsource their private data onto public cloud servers that are not within their trusted management domains. To avoid information leakage, sensitive data have to be encrypted before uploading onto the cloud servers, which makes it a big challenge to support efficient keyword-based queries and rank the matching results on the encrypted data. Most current works only consider single keyword queries without appropriate ranking schemes. In the current multi-keyword ranked search approach, the keyword dictionary is static and cannot be extended easily when the number of keywords increases. Furthermore, it does not take the user behavior and keyword access frequency into account. For the query matching result which contains a large number of documents, the out-of-order ranking problem may occur. This makes it hard for the data consumer to find the subset that is most likely satisfying its requirements. In this paper, we propose a flexible multi-keyword query scheme, called MKQE to address the aforementioned drawbacks. MKQE greatly reduces the maintenance overhead during the keyword dictionary expansion. It takes keyword weights and user access history into consideration when generating the query result. Therefore, the documents that have higher access frequencies and that match closer to the users' access history get higher rankings in the matching result set. Our experiments show that MKQE presents superior performance over the current solutions.

1. Introduction

Cloud computing is getting more and more attention from both academic and industry communities as it becomes a major deployment platform of distributed applications, especially for large-scale data management systems. End users can outsource their personal data onto public clouds, and then access the data at anytime and anywhere. In the cloud environment, the resources allocated for each application can be scaled up and down according to the fluctuating demand. It adopts a pay-per-use resource sharing model, which allows a user to pay only for the number of service units it consumes. Cloud computing infrastructure provides a flexible and economic strategy for data management and resource sharing. It can reduce hardware, software costs and system maintenance overheads. It can also offer a convenient communication channel to share resources across data owners and data consumers. With the popularity of cloud services, such as Amazon Web Services,[1]

* Corresponding author.
E-mail address: rxli@hust.edu.cn (R. Li).

[1] Amazon web services, http://aws.amazon.com.

0167-739X/$ – see front matter © 2013 Elsevier B.V. All rights reserved.
http://dx.doi.org/10.1016/j.future.2013.06.029

Microsoft Azure,[2] Apple iCloud,[3] Google AppEngine,[4] more and more companies are planning to move their data onto the cloud.

Despite of its advantages, the cloud computing infrastructure faces very challenging tasks, especially on data privacy, security and reliability issues. The fact is that private data are now placed on public clouds which are out of their trusted domains in cloud computing. Data owners do not have direct control over their sensitive data and are increasingly worrying about possible data loss and/or illegal use of their private data. Usually, cloud servers are considered as curious and untrusted entities. Data owners will hesitate to adopt cloud technologies if there are risks of data exposure to a third party or even the cloud service provider itself. Therefore, providing sufficient security and privacy protections on sensitive data is extremely important, especially for those applications dealing with health, financial and government data.

Some approaches have been proposed to evaluate cloud computing security and introduce a "trusted third party" to assure security characteristics within a cloud environment, such as [1,2]. To prevent information disclosure, the mainstream solution is to encrypt private data before uploading it onto the cloud server. On one hand, this approach ensures that the data are not visible to external users and cloud administrators. On the other hand, there are severe processing limitations on encrypted data. For example, standard plain text based searching algorithms are not applicable any more. To perform a keyword-based query, the entire data set has to be decrypted even if the matching result set is very small. It poses unbearable query latency and incurs unacceptable computational overhead.

To solve this issue, current solutions use the following strategy to provide keyword-based searching capabilities on encrypted data. First, a set of keywords are defined. An index vector is calculated for each file. It maintains the information of which keywords this file contains. After constructing the index vectors, an index file that combines all the index vectors is generated. The index file has to be encrypted as well. Second, both the encrypted data and index files are uploaded onto the data center servers in the cloud. Now, the data are ready to accept queries from the data consumers. The cloud servers can then support cipher text based queries as follows. A data consumer submits a keyword-based query, and the encrypted keywords are sent to the cloud server. The cloud server conducts a search on the encrypted index and returns a list of most relevant files. The user makes the decision that which files are needed and retrieves them from the server. After receiving encrypted files, the user decrypts the files with the associated key. This approach can guarantee the data security and preserve the data privacy. During the whole process, no plain text data or keywords are visible to the cloud servers.

Although substantial research works, such as [3–5], have been done to study keyword-based queries on encrypted data, many of them only address single keyword queries. Others use disjunctive or conjunctive searches for multi-keyword queries which have great limitations in flexibility and performance. Furthermore, few of them offer the ranking algorithm for matching results. MRSE [6] is the first and latest work to define such a multi-keyword ranked query problem, and proposes a viable solution to address it. In MRSE, all keywords are stored in a dictionary and a certain keyword can always be identified by its location in the dictionary. MRSE has two randomly generated invertible matrices for data and file index encryption operations. It uses the inner product of two vectors to build the trapdoor for secure keyword queries. It also applies an internal ranking algorithm to determine the top k files to be returned to the data consumer.

However, this approach suffers from three major drawbacks. First, it uses a static dictionary. If new keywords to be added, the dictionary has to be rebuilt completely which leads to substantial computational overhead. Second, an out-of-order problem occurs if using its trapdoor generation algorithm. Such a problem brings the result that the files with more matching keywords are likely excluded from the top k positions in the matching set. This means that the data consumer may not be able to find the most relevant files they want. Lastly, MRSE does not consider the effects of keyword weight and access frequencies. Therefore, the files that contain frequent keywords might not be included in the top k locations in the returning result at all.

In this paper, we design a new strategy called MKQE to address the aforementioned issues. In MKQE, we assume that the amount of data continues to increase from time to time. Accordingly, the keyword dictionary has to be expanded periodically. We propose a new dictionary construction paradigm, introduce a new trapdoor generation algorithm to reduce the query latencies, and take the keyword access frequencies into consideration to generate better matching result sets. In summary, we make the following contributions.

- We introduce partitioned matrices in the system design. The keyword dictionary can be expanded dynamically without touching the contents in the original dictionary. We design the novel storage and encryption algorithm to manage the keyword dictionary. MKQE greatly reduces both the dictionary reconstruction overhead and the file index re-encryption time as new keywords and files are added.
- We design a novel trapdoor generation algorithm. It can effectively reduce the impacts of dummy keywords on the ranking scores. With this new strategy, the out-of-order problem in the matching result set is solved.
- We take the keyword access frequencies into account when the system generates the ranked list of the returning results. Besides, we add the weights of the keywords in the index file. The files which contain more frequently accessed keywords will have higher weights in the query. The files with higher weights will have higher probabilities to appear in the first k locations of the matching result set. Hence, the data consumers have better chances to retrieve the desired files easily.

The rest of the paper is organized as follows. Section 2 defines the problem. Section 3 discusses current research works including MRSE and its drawbacks. Section 4 presents the system overview and the technical details of the proposed MKQE solution. Section 5 discusses the correctness and privacy-preserving analysis. Section 6 describes the experimental configurations and performance evaluations. Section 7 introduces the related works. Finally, Section 8 concludes the paper and gives the future work.

2. Problem definition

We aim to design a new approach to improve the performance for multi-keyword ranked queries on encrypted data in public cloud servers. In this section, we will introduce the notations and define the problem.

2.1. Notations

- F: the set of original files, assuming there are m files. F is denoted as $F = (F_1, F_2, F_3 \ldots F_m)$.
- C: the set of encrypted files, corresponding to the files in F. C is denoted as $C = (C_1, C_2, C_3 \ldots C_m)$.

[2] Microsoft azure, http://www.windowsazure.com.
[3] Apple icloud, https://www.icloud.com/.
[4] Google appengine, https://appengine.google.com/.

- W: keyword dictionary, assuming we have n keywords. W is denoted as $W = (W_1, W_2, W_3 \ldots W_n)$.
- F_{id}: the index set of every file, which is denoted as $F_{id} = (F_{idx1}, F_{idx2}, F_{idx3} \ldots F_{idxm})$.
- $index_x$: the keyword set of each F_{indx}, assuming it contains keywords $index_{kw1,kw2\ldots kwk}$.
- $weight_x$: the weight set for keywords in index $index_x$, $weight_x = (w_1, w_2, w_3 \ldots w_k)$.
- p: the index vectors for F_{idx}. p is denoted as $p = (p_1, p_2, p_3 \ldots p_n)$.
- I: the encrypted index vectors for p. I is denoted as $I = (I_1, I_2, I_3 \ldots I_n)$.
- W_q: a plain text query, assuming it contains k keywords, and can be represented as $W_{kw1,kw2\ldots kwk}$.
- $weight_q$: the weight set for keywords in query W_q, $weight_q = w_{q1}, w_{q2} \ldots w_{qk}$.
- q: for a query W_q, the corresponding query vector.
- T: the trapdoor for a query W_q, which is based on q.
- $\bar{R}$: the list of files in the returned matching result set. It is a sorted list, the order of the files is determined by the scores.

2.2. Problem description

Fig. 1 shows the basic framework of secure multi-keyword ranked query in the cloud computing environment. There are three kinds of roles in a cloud, data owner (DO), data consumer (DC) and cloud service provider (CSP). The DO identifies the files containing sensitive data. Those files are then encrypted using a standard symmetric algorithm such as DES [7] or AES [8]. It also specifies the set of keywords to form a keyword dictionary to be used for queries. In our discussions, we assume that the keyword dictionary is dynamic. The DO may add keywords later depending on the changes in the set of sensitive files and applicable keywords. For each file, an index vector is generated based on which keywords are contained inside. Those index vectors are also encrypted and combined together to generate an encrypted index file. Both the encrypted files and the encrypted index file must be uploaded onto the cloud servers. To facilitate secure queries, the DO needs to define a secret key. The secret key contains two invertible matrices and a bit vector. All the elements in the secret key are randomly generated (The details will be presented in the following section). The secret key is kept on the DO only. After the above processes finish, all the sensitive files stored on the cloud servers in encrypted formats. Only the DC can decrypt them. There is no information leakage to the CSP or a third party.

In general, the DCs are able to conduct secure multi-keyword ranked queries on those encrypted files. A query is executed as follows. Firstly, a DC sends a set of searching keywords to the DO. Secondly, the DO builds the trapdoor **T** based on the set of keywords using the secret key. It sends **T** to the DC who initiates the query request. Finally, the DC sends **T** to the CSP who stores the encrypted files. A matching process based on **T** is conducted, and a set of encrypted files are identified. All the operations on the CSP are executed on the encrypted data using the secure trapdoor **T** only. There is no plain text information exposure on the CSP. Because the number of files which contains one or more keywords specified in the **T** could be very large, it results in considerable overhead to return all the results to the DC. In order to select an adequate set of files, a ranking algorithm is applied on these files based on the relevance scoring. Typically, the DC only has to retrieve the top k most relevant files. After receiving the results, it sends a request to the DO for the decryption keys, and then decrypts these files.

In this paper, we employ attribute based encryption (ABE) [9,10] technology to manage the keys for the files. ABE is a key management and authorization technology used in untrusted server.

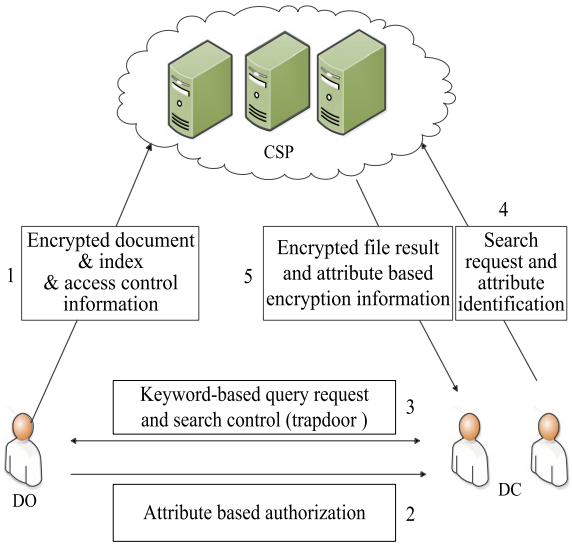

Fig. 1. The basic framework of secure multi-keyword ranked query in the cloud.

All the operations are done in encryption environment. Through this way, the DO does not need to management the file encryption keys and transport them to the DC. It reduces the overhead of the DO. In this paper, we assume that the DO is the authorization center in his trusted domain. The DO distributes attributes to the DC. The attributes are the only authorization certificates in the cloud computing environments. When uploading files and indexes to CSP, the DO encrypts the index with some attributes which relate to the access control information. When the DC gets the trapdoor from the DO, he will send the trapdoor and part of his attribute information to CSP. Before querying, the CSP computes the authorization first. If the file is allowed, query request is allowed. Finally, the CSP returns the top-k results to the DC, and the DC decrypts the files using his attributes.

2.3. Threat model

Normally, in the cloud environment, the CSP is considered as "honest but curious" [11]. In other words, the CSP is expected to execute the procedure or algorithm faithfully (without adding or subtracting any operations from the command set received from the DCs and DOs). However, it is curious and eager to know the keyword information and the contents in the data files stored on its servers. Here, we adopt two threat models defined in [3]. The first model is "Known Ciphertext Model". This model assumes that the CSP can only see the encrypted files and indexes. The second one is "Known Background Model". Here, the CSP may intentionally collect the statistical information of queries (trapdoors). Based on that, the CSP may be able to calculate which file contains a certain keyword.

The privacy preserving requirement in the cloud environment is defined as follows. The CSP cannot obtain the information of which file contains which keywords directly from the indexes. The CSP does not know how many keywords are searched in a specific query as well. For a certain query, the CSP can only calculate the scores of the files in the result set using a given ranking algorithm, but know nothing about the matching keywords. For two different queries which have the same set of keywords, the generated trapdoors must be different, and the scores must not be the same as well. Furthermore, in our paper, we consider the keyword dictionary can be expanded dynamically. Thus, we assume that the CSP might know how many new keywords are added. However, it does not pose a security threat since the CSP has no ideas of how many original keywords we have, and what is the actual plain text information of the keywords newly added.

2.4. Our goals

In this paper, we aim to design a new algorithm to achieve the following goals.

Multi-keyword query. It can support efficient multi-keyword based query with low keyword, trapdoor and encryption overheads. It should be able to rank the query result as well.

Dynamic keyword dictionary size. It supports dynamic keyword dictionary size. An efficient algorithm should be designed. Only minor changes are necessary when the keyword dictionary expands.

In-order ranking. The ranking algorithm should be effective. The system makes the in-order result set which preserve the most relevant files appear in the top k locations with a high probability.

Keyword weight and access frequency consideration. It considers the keyword weight and access frequency information for query results. A file containing a more popular keyword should be ranked higher in the matching result set that a file containing a less popular keyword with higher probability.

3. Existing solutions

MRSE is the first and latest work to address multi-keyword ranked queries on encrypted data in the cloud. In this section, we first describe MRSE [6] architecture, and discuss its drawbacks. Then, we introduce query techniques which taking keyword weights and query frequencies into consideration based on plain text files.

3.1. MRSE algorithm

In MRSE, initially, before any operation happens, the DO has to build a dictionary containing a set of keywords. Assume **n** keywords are identified, MRSE has to sort these keywords. In another word, the positions of the keywords in the dictionary are fixed. In order to provide the privacy preserving property, MRSE adds **u** dummy keywords. Those dummy keywords do not represent any meaningful keywords and they are used to improve the encryption strength only. In addition, an extra random bit is placed at the end. After that, the DO has to generate two invertible matrices $\mathbf{M_1}$ and $\mathbf{M_2}$ with the sizes $(n+u+1)\times(n+u+1)$ and a vector **S** of the size $(n+u+1)$. The DO uses them as a secret key.

For each file to be encrypted and stored in the cloud, an index vector p with the same size as S is created. This vector is built as follows. If the file has the keyword W_i $(1 \le i \le n)$, the ith element $p[i]$ is set to 1. Otherwise, $p[i]$ is set to 0. Dummy keywords are represented by the elements located between positions $n+1$ and $n+u$. The values in these locations are set as $p[i] = \epsilon(n+1 \le i \le n+u)$. ϵ follows the uniform distribution $M(u'-c, u'+c)$ (u' and c are the two parameters, and are described below). The $(n+u+1)$th dimension is always set to the constant 1.

For a multi-keyword query q, a query vector with the size $(n+u+1)$ is constructed. The values of the first n locations are determined using the same method for the file index generation. For the locations between $n+1$ and $n+u$, MRSE uses the following strategy to set the values. First, v locations are chosen randomly, and the values of these locations are set to 1. Then, the values of all the remaining locations are set to 0. Finally, the system picks two numbers t and r. The value of the last dimension $(n+u+1)$ is set to t, and the values of all the other locations are multiplied by r. After that, a query vector is created. Normally, v is set to $u/2$. The sum of all the dummy keywords follows the normal distribution $N(\mu, \sigma^2)$, $u' = \mu/v$ and $c = \sqrt{\frac{3}{v}}\sigma$. The DO applies the same strategy to encrypt the file index p and the query vector q. Two vectors $\overrightarrow{p_1}$ and $\overrightarrow{p_2}$ are constructed for p, and two vectors $\overrightarrow{q_1}$ and $\overrightarrow{q_2}$ are constructed for q. Each vector has $(n+u+1)$ locations. They follow the rule that, for any number j, if $S[j] = 1$,

$$p[j] = \overrightarrow{p_1}[j] + \overrightarrow{p_2}[j],\ q[j] = \overrightarrow{q_1}[j] = \overrightarrow{q_2}[j]. \tag{1}$$

Otherwise, if $S[j] = 0$,

$$p[j] = \overrightarrow{p_1}[j] = \overrightarrow{p_2}[j],\ q[j] = \overrightarrow{q_1}[j] + \overrightarrow{q_2}[j]. \tag{2}$$

Then the system uses the invertible matrices M_1 and M_2 to compute the encrypted index for the file and the trapdoor for the query. The formulas are as follows.

$$encrypt(p) = \{M_1^T\overrightarrow{p_1}, M_2^T\overrightarrow{p_2}\} \tag{3}$$

$$encrypt(q) = \{M_1^{-1}\overrightarrow{q_1}, M_2^{-1}\overrightarrow{q_2}\}. \tag{4}$$

During the query q execution, the DC sends the trapdoor to the CSP. The CSP uses "inner product similarity" [12] to compute the scores. The scores measure the coordinate matching between the indexes and the query. Finally, the first k results with the highest scores are returned to the DC.

The formula to calculate the score is as follows.

$$\begin{aligned} score &= encrypt(p) \times encrypt(q) \\ &= \{M_1^T\overrightarrow{p_1}, M_2^T\overrightarrow{p_2}\} \times \{M_1^{-1}\overrightarrow{q_1}, M_2^{-1}\overrightarrow{q_2}\} \\ &= M_1^T\overrightarrow{p_1} \times M_1^{-1}\overrightarrow{q_1} + M_2^T\overrightarrow{p_2} \times M_2^{-1}\overrightarrow{q_2} \\ &= \overrightarrow{p_1}^T M_1 \times M_1^{-1}\overrightarrow{q_1} + \overrightarrow{p_2}^T M_2 \times M_2^{-1}\overrightarrow{q_2} \\ &= \overrightarrow{p_1}^T\overrightarrow{q_1} + \overrightarrow{p_2}^T\overrightarrow{q_2} \\ &= p^T q. \end{aligned} \tag{5}$$

From the above description, we can see that MRSE provides a working solution for the problem of the multi-keyword query on the encrypted data. It also considers the ranking issue to select an appropriate subset if the searching result set is large. Clearly, without the effects of the dummy keywords, the more common keywords in the query vector and the file index, the higher the score a file will get.

3.2. Drawbacks of MRSE

MRSE is an effective mechanism which can partially solve the multi-keyword ranked query problem. However, it has the following drawbacks that affect its efficiency.

First, the keyword dictionary is static and should be created at the beginning. MRSE does not provide a viable solution for dynamically expanding the set of keywords. In case new keywords are added, the keyword dictionary has to be reconstructed. The original set of index vectors cannot be used for the queries using the newly constructed trapdoors. Thus, the index vectors of all the files have to be recalculated from scratch. Such an approach incurs unacceptable overhead.

Second, in MRSE, the values of dummy keywords follow the normal distribution $N(\mu, \sigma^2)$. In our experiments, we found that such an approach has great impact on the file score. For any query, the scores that the matching files receive would vary widely. A file with more matching keywords may have a much lower score than another file which has fewer matching keywords. It results in severe out-of-order placement issue, which means that in MRSE, it has a high probability that highly relevant files may be excluded from the top k locations in the result set, especially when the number of matching keywords is small and the variance σ is relatively big.

Third, MRSE does not take the access frequencies of the keywords into account. The ranking algorithm assumes that all the keywords are the same when calculating the scores. In other words, the system considers a file $f1$ to have a higher score than another

file $f2$ if it has more matching keywords even if the set of keywords matching in $f2$ are more popular than the keywords matching in $f1$. However, in reality, keywords have vastly different popularity and popular keywords are always more preferable than less-popular ones from the data consumer's point of view. Thus, the keyword access frequencies should be considered to better serve the DCs.

3.3. Weighted keyword query

When generating the query result, taking the keyword weights into account is important, especially for multi-keyword query scenarios. Typically, if the number of files containing certain keywords is large, selecting an appropriate subset which best matching the DC's requirement becomes important. Therefore, the query results should be ranked. We have to place the files in the result set in such an order that the higher a file is located, the better chances it matches the DC's expectation. Certain metrics have to be used to generate this order and reflect the DC's need.

Weighted query problem has been well studied in the past research works. Solutions such as the term-weighting [13] or similarity space [14] query are being proposed. However, all these works only consider the queries on the plain text files. However, in our scenario, both the files and the keyword indexes stored on the cloud servers are encrypted. Thus, the CSP cannot apply those techniques to conduct the queries directly. New algorithms have to be developed. In this paper, a practical and novel keyword query framework on encrypted data is developed.

4. System design

We propose MKQE, a new solution to address the above issues. MKQE includes a set of novel strategies to provide effective and efficient mechanisms on the issue of the multi-keyword ranked query on encrypted data. In this section, we present the details of our solution.

4.1. Overview

In MKQE, we adopt "inner product similarity" to quantitatively evaluate the coordinate matching like MRSE. MKQE also defines an index vector for each file based on the keywords it contains. Two invertible matrices and a bit vector are also used for the index vector encryption and the trapdoor generation. In MKQE, when a multi-keyword query comes, a query vector based on the set of requesting keywords is constructed. However, our approach differs from MRSE in the way when new keywords have to be added into the dictionary. In MKQE, only minimal overhead is introduced in this scenario. We also modify the trapdoor generation mechanism to improve the in-order ranking performance in the matching file set. Furthermore, we take the keyword access frequency distribution into consideration when creating the ranked list. Such a strategy can better satisfying the DC's request and reflect the real world situations.

4.2. MKQE framework

The MKQE system consists of the following components:

- *Setup*: based on the sensitive data, the DO determines the keyword dictionary size **n**, the number of dummy keywords **u**, and then sets the parameter $\mathbf{d} = 1 + u + n$.
- *Keygen(d)*: the DO generates a secret key **SK** k1, two invertible matrices M_1 and M_2 with the dimension $d \times d$, and a d-bit vector S.

	Security locations	Dictionary keyword locations
1	U	N

Fig. 2. The index and trapdoor structure in MKQE.

- *Extended-Keygen*($k1, z$): if z new keywords are added in the dictionary, the DO generates a new **SK** k2, two invertible matrices M_1' and M_2' with the dimension $d+z \times d+z$, and a new $(d+z)$-bit vector S'.
- *Build-Index*(F, SK): for each file, the DO determines the set of keywords F_{idx}, and builds p for it. Then it encrypts the index vector with an SK (either $k1$ or $k2$). After that, all the encrypted indexes are added to I. All the files are encrypted with DES or AES, and added to C. Finally, upload I and C onto the CSP.
- *Trapdoor*($\mathbf{W_q}$, SK): The DC sends a multi-keyword ranked query W_q to the DO. The DO generates an index vector q and calculates the trapdoor T using an SK and sends it back to the DC.
- *Query*($\mathbf{T}, k, I$): The query is sent to the CSP. The CSP runs the query on I and returns the most relevant top k scored files back to the DC.

4.3. Detailed design

4.3.1. Vector structure

As we described in Section 3, in MRSE, a file index vector contains three parts, and the order of these parts is fixed. The first n locations are used for the real keywords, followed by u locations for the dummy keywords, and the last dimension is the constant 1. This structure is not suitable in MKQE because the number of keywords changes from time to time. When there are new keywords introduced, we have to increase the number of locations for index vectors as well. If we add them in the end, the locations representing new keywords are not adjacent to the locations representing the original keywords. When executing a query, such a structure causes difficulty in checking the results of matrix operations.

To solve this issue, in MKQE, we reorganize the structure of the file index vectors. As shown in Fig. 2, the secure locations are now placed at the beginning of the vector, followed by the n locations used for the real keywords. With this approach, for any position j in the vector, we have the following relation: if a file has the real keyword W_j, then in the corresponding index vector, $p[1 + u + j] = 1$. Otherwise, it is 0. For query vectors, the same structure is applied.

4.3.2. Extended key generation

As we discussed in previous sections, a secret key can be used to encrypt file index vectors and query vectors. However, in MRSE, the size of the matrices and the vector in the secret key is determined by the keyword dictionary size n. If the dictionary is expanded, the value of n has to change. Thus, the current secret key cannot be used for file and index encryptions any more. In other words, the DO has to generate a new secret key, and the system has to encrypt all the file index vectors with the newly generated key again. As more and more data are accumulated in the cloud, we suspect that adding more query keywords is not a rare operation. Clearly, MRSE is not suitable for such a scenario, it incurs severe computational overhead. To resolve this issue, in MKQE, we introduce a new approach using partitioned matrix operations to reduce the computational overhead.

In MKQE, when there are new keywords added in the dictionary, we do not change the original secret key. Instead, we only add a new secret key to support queries for the newly added keywords. With this approach, we can keep the original secret key untouched, and avoid the expensive keyword dictionary reconstruction process. The new algorithm is described as follows.

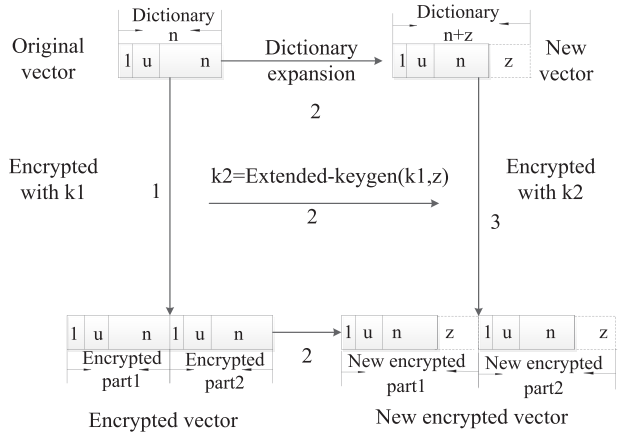

Fig. 3. Keyword dictionary expansion operations.

Assume there are z new keywords to be added,

- Generates two invertible matrices M_z and M'_z with the dimension $z \times z$. The system also generates a z-bit vector S_z.
- The new matrices M'_1, M'_2 and the secret key S' are generated with the formulas:

$$M'_1 = \begin{pmatrix} M_1 & 0 \\ 0 & M_z \end{pmatrix} \qquad M'_2 = \begin{pmatrix} M_2 & 0 \\ 0 & M'_z \end{pmatrix} \tag{6}$$

$$S' = (S, S_z). \tag{7}$$

In MKQE, we use the matrix transpose and inverse operations to encrypt file index vectors and query vectors. According to the block matrix theorem, we have the following formulas.

$$M'^T_1 = \begin{pmatrix} M^T_1 & 0 \\ 0 & M^T_z \end{pmatrix} \qquad M'^T_2 = \begin{pmatrix} M^T_2 & 0 \\ 0 & M'^T_z \end{pmatrix} \tag{8}$$

$$M'^{-1}_1 = \begin{pmatrix} M^{-1}_1 & 0 \\ 0 & M^{-1}_z \end{pmatrix} \qquad M'^{-1}_2 = \begin{pmatrix} M^{-1}_2 & 0 \\ 0 & M'^{-1}_z \end{pmatrix}. \tag{9}$$

Fig. 3 shows the encryption procedure as new keywords are added. As we can observe, the first $1 + u + n$ locations are unchanged.

4.3.3. Matrix processing

As described above, when the keyword dictionary is extended, block matrices are used to generate the new secret keys. As shown in the formulas, plenty of elements with 0 value appear in the matrices. These elements are not really used during the query keyword encryption/decryption processes. Thus, we do not have to store those elements. To save space, we define our matrix key storage structure and algorithm for dictionary expansion operations as follows.

We first define a data structure called *Node*, and use it to record the information for a certain inverted matrix either M'_1 or M'_2 during the keyword expansion process. A node has the following elements. The first element **start_row** records the id of the starting row (since it is a square matrix, it also represents the starting column). The second one **dimension** keeps the size (the number of rows) information of a certain inverted matrix. The third one **matrix** stores the actual block matrix M'_1 or M'_2. The last element **next** points to the next node in the list. All the nodes in the expansion operations form two linked lists. Each time when a keyword expansion operation happens, two nodes are generated and added to the lists separately.

```
struct Node {
        int start_row;
        int dimension;
        Matrix matrix;
        struct Node next;
}
```

We use two linked lists to store two inverse matrix.

At the beginning, the system has two inverted matrices M_1 and M_2 with the dictionary size n.

Two linked lists are initialized by M_1 and M_2 separately. When a dictionary expansion operation occurs, two inverse matrix have to be expansion, new node objects representing newly generated inverted matrices have to be added to the list. We develop the *Matrix-Storage* algorithm to store the inverse matrix. Two parameters used in the algorithm are the newly generated matrix and the head of the list. We take M'_1 as an example for explanation. In *Matrix-Storage* algorithm, Lines 4 to 7 show how to store M_1 in a node and initialize the values of the elements. Lines 11–22 show the process that a new node contains M_z is added to the list and linked with the previous nodes.

```
MATRIX-STORAGE(MATRIX, HEAD)
 1  dimension ← matrix.getDimension()
 2  if head is null
 3  {
 4      start_row ← 0
 5      node =new Node(start_row, dimension, matrix)
 6      node.next = null
 7      head = node
 8  }
 9  else
10  {
11      pre_pointer = head
12      tmp = head.next
13      while tmp is not null
14       do
15          {
16              pre_pointer = tmp
17              tmp = tmp.next
18          }
19       start_row = tmp.start_row + tmp.dimension
20       node =new Node(start_row, dimension, matrix)
21       pre_pointer.next = node
22       node.next = null
23  }
```

With this approach, MKQE can not only save the space requirement of keyword dictionary, but also can reduce the system encryption time using our modified encryption algorithm introduced in the next subsection.

4.3.4. Index building

A major advantage of MKQE is that we provide the easy keyword expansion capability with low overheads. In MKQE, we assume that the set of keywords could change from time to time. In such a scenario, only minor changes are needed for the original set of files. For a certain file, MKQE extends the corresponding index vector by adding z elements in the end. The values for these newly added elements are determined as follows. If it contains a newly added keyword j ($1 \leq j \leq z$), then in its corresponding index vector, $p[1+u+n+j]$ is set to 1, otherwise it is 0. Obviously, the first $1 + u + n$ locations are unchanged.

For a new coming file, an index vector with the dimension $1 + u + n + z$ has to be built. The values of the dimensions are determined by the fact that if it has the corresponding keyword or not. The detailed strategy is the same as we described in Section 3.

In MKQE, new inverse matrices can be stored using *Encryption-Alg* algorithm. After that, we build the index vector p using the keywords in the index. The process is the same as MRSE. Besides that, in MKQE, we construct two extra vectors $\overrightarrow{p_1}$ and $\overrightarrow{p_2}$ using formulas (1) and (2). Plain text vector($\overrightarrow{p_1}$ or $\overrightarrow{p_2}$) are used as the parameter vector to be encrypted in the *Encryption-Alg* algorithm. The other parameter *head* in the algorithm is the head of the linked list which stores all the inverse matrices information as described in Section 4.3.3. Furthermore, $\overrightarrow{p_1}$ and $\overrightarrow{p_2}$ are to be encrypted separately by inverse matrices M_1 and matrix M_2.

Finally, we concatenate the two encryption vectors as a single one and use it as the encryption index.

```
ENCRYPTION-ALG(T[] VECTOR, NODE HEAD)
 1  length ← length[vector]
 2  T en_vector[length]
 3  offset ← 0
 4  Node tmp ← head
 5  while tmp is not null
 6      do {
 7          matrix ← head.matrix
 8          T array[][] ← matrix.getArray()
 9          for i ← 0 to matrix.dimension
10              do a ← offset + i
11                  en_vector[a] ← 0
12                  for j ← 0 to matrix.dimension
13                      do
14                          {
15                          b ← offset + j
16                          tmp ← vector[b] * array[i][j]
17                          en_vector[a]+ = tmp
18                          }
19                  }
20          base+ = matrix.dimension
21          tmp ← tmp.next
22          }
23  return en_vector
```

We employ matrix multiplication to encrypt the index, and the time complexity of matrix multiplication is $O(n^2)$. We use two linked lists, which store the two new inverse matrix separately, to analog the expanded key. It greatly reduces the time encryption complexity. For example, the original secret key contains two inverse matrix, and the expanded secret key contains two inverse matrix. The encryption needs $4x^2$ operations in the new key, while only $2x^2$ operation needed by using linked analog list in the same situation. It will be more obvious with the extended increased frequently. Our experiments will prove this in Section 6.

4.3.5. Weighted index

In keyword-based query applications including web engine, database search etc., keywords have vastly different popularity and thus have different access frequencies. However, in MRSE, the different access features of keywords are not well considered. Furthermore, data consumers also have their own preference on the keyword searches. There are many research work [13–15] have been done on term-weighting (or keyword weight) query on plain text files. Luhn [16] first suggests that the automatic text retrieval systems should be designed based a comparison of content identifiers attached on both the stored texts and the user' information queries [14]. Apparently, to provide a meaningful query results back to the data consumer, taking the keyword weights into account is unavoidable.

As we can see from the above, the weighted keyword queries on plain text files have been well investigated. However, for the encrypted files, such a problem is not solved yet. An effective solution to offer the weighted query capability on encrypted files and preserve security and privacy concerns is in great need. In MKQE, we incorporate the keyword weights in the index vector when generating the dictionary. We also develop novel algorithm to utilize this information to generate the query result. To the best of our knowledge, MKQE is the first to address this issue over encrypted data in cloud computing.

In MKQE, when the DO builds the index for a file, assume that the keyword set is $word\_set = \{word_a, word_b \ldots word_k\}$. For each keyword $word_x$ in the set and each document d containing $word_x$, we consider the frequency of $word_x$ is d. The weight of a keyword $word_x$ is defined in Formula (10).

$$w_x = 1 + \log_e f_{d,x}. \tag{10}$$

Before building the index vector, the weight of each keyword is computed at first. If the keyword $word_x$ is in location x in the dictionary, we set $p[1 + u + n + x]$ to w_x instead of 1 and the value of dummy keywords and the constant dimension are determined with the same rule as MRSE.

In [14], Zobel et al. pointed out that the ranked query should list the most closely matched documents in the decreasing order. It considers the keyword weight characteristic during the query execution. In his paper, the weight of index keywords and query keywords can be computed by several formula separately. They identified the combinations of them to compare which similarity measure gives good effectiveness. At last they found that with weight consideration the match is more effective to user's request. So we consider the weight computing in cloud computing over encrypted data during building the index is reasonable. Although many formula listed in [14] cannot be used directly in our paper, we give the modified formula which is also similarity as described in [13].

4.3.6. Trapdoor generation

In MKQE, when a multi-keyword query q comes, a query vector is created using the strategy discussed in the previous section. Again, v number of these dummy locations are set to 1 and all the remaining locations are set to 0. We encrypt the query vector using the *Matrix-Storage* algorithm as described above. The query vector and the inverse of a matrix are used as two parameters of the algorithm. MKQE generates a score to determine the location of a file in the matching result set. For a file with an index p_i, its score is calculated as follows.

$$p_i \bullet q = r\left(x_i + \sum \epsilon_i^{(v)}\right) + t_i \tag{11}$$

where x_i represents the number of keywords appearing in both p_i and q. $\epsilon_i^{(v)}$ represents the v dummy keywords. Typically, v is set to $u/2$. t_i is the constant dimension. According to the previous description, the sum of the selected dummy keywords follows the normal distribution $N(\mu, \sigma^2)$. Thus, the value of the sum is in the range $[v * (u' - c), v * (u' + c)]$, where $u' = \mu/v$ and $c = \sqrt{\frac{3}{v}}\sigma$. The difference between the maximal and minimal values is $2 \times v \times \sqrt{\frac{3}{v}} \times \sigma$. In our experiments, we found that such a large variance can significantly affect the ranking of the results, and the files which contain popular keywords could have very low scores and cannot be placed in the top k locations in the result set. We define such a problem as an out-of-order problem.

To improve the in-order ranking result and maintain the privacy preserving property, we change the structure of query vectors. For all the real keywords in the dictionary, the values in the

corresponding locations are multiplied by a random number r. For all the locations representing dummy keywords, they are multiplied by another number $r2$. $r2$ is calculated as follows.

$$r2 = \frac{Random(0, r)}{(v * MAX(abs(u - c), abs(u + c)))}. \quad (12)$$

After q is created, we use the Formulas 1, 2 and 4 to split and encrypt q. Finally, the trapdoor T is generated. Now, for an index p_i and the query q, the matching score is computed as follows.

$$p_i \bullet q = I_i \bullet T = r \times x_i + r2 \times \sum \epsilon_i^v + t_i. \quad (13)$$

The values of $r2 \times \sum \epsilon_i^v$ is within $[-r, r]$. Here, x_i is a common keyword in p_i and q. Since the sum of the chosen dummy keywords is within $[-r, r]$, it has much smaller impacts on the score calculation. The out-of-order problem is solved.

To further improve the result set accuracy, the keyword weights have to be considered. Substantial works [17,18] have been conducted on weighted keyword queries. In MKQE, we also consider that the keyword access frequencies could be used. Since the keyword weight has been used to generate the index for each file, when the DC executing the query, it helps us to determine the ranking of the files in the result. We conduct several experiments and found that if no keyword weight considered, a file containing a smaller number of popular keywords may get a much lower score compared to another file which has more unpopular keywords. Hence, this file might be excluded from the top k results and cannot be retrieved by the DC.

To resolve this problem, we make the following changes in MKQE. For a particular DC, when executing a multi-keyword query, each keyword is assigned with a weight. The weight is determined by the historical statistical information depending on the past history on this DC as well as the accumulated keyword access frequencies in the whole system. Thus, for a multi-keyword query q which contains the keyword set $(W_{kwa}, W_{kwb}, \ldots, W_{kwk})$, MKQE sets the corresponding weight $(w_{qa}, w_{qb}, \ldots, w_{qk})$ for them. Based on this principle, we modify the matching score calculation formula as follows.

$$p_i \bullet q = r \times \sum (w_j \times w_{qj}) + r2 \times \sum \epsilon_i^v + t_i \quad (14)$$

where w_j and w_{qj} are the weight of common keyword W_{kwj} which correspond to the p_i and q. With this improvement, the result matching set can better satisfy the DC's preference by returning the most relevant files.

4.3.7. Query

The DC sends the trapdoor T and parts of his attributes to the CSP. With the information, the CSP firstly determines which files can be accessed by the DC, and then computes the matching score of each authorized file in the encrypted index set I. After that, the CSP sorts the results based on the scores, and only returns the top k files in the resulting set to the DC. In our trapdoor algorithm, the score is calculated using the Formula (13). When the keyword weight is considered, the values of the locations in the query vector are very likely determined by their corresponding weights. In this scenario, the scores is calculated using the Formula (14).

5. Correctness and privacy-preserving analysis

In this section, we briefly discuss the correctness and privacy preserving features in our formulas.

5.1. In-order ranking

We define a variable b to denote the sum of the dummy keyword values in p. Apparently, it is within $[-r, r]$. Let us assume that there are two files i and j. p_i matches $n1$ keywords with a score1, p_j matches $n1 + 1$ keywords with a score2. If score1 $\geq$ score2, we define it as an out-of-order error. It happens only when the inequation $b_i + n1 * r \geq b_j + (n1+1) * r$ holds. Such a situation is possible if $u' - c < 0$ and $u' + c > 0$. However, in MKQE, we avoid this by guarantee that $u' - c > 0$ (by changing the mean and variance values in the normal distribution). Because b_i and b_j are both within $(0, r]$, it is impossible that $b_i \geq b_j + 1$. When the variance is set less than $r/2$, the value of b must be within $(-r/2, r/2)$. In our experiments, we found that even if the variance is set to r, the probability of the out-of-order errors is extremely low. For example, if we set b_i is within $[0.5r, r]$ and b_j is within $[-r, -0.5r]$, according to the Formula (12) and the normal distribution function, the probability of the out-of-order errors is $(1 - F(v * c/2)) * F(-v * c/2)$. If we set v to 7 and use the normal distribution $N(0, 1)$, the probability is only 0.0121%.

5.2. Privacy

For any two separate queries, the MKQE algorithm always choose different set of dummy keywords. Hence, the probability of two $\sum \epsilon^v$ having the same value is less than $1/2^{(v)}$. Since for each query, $r2$ is randomly selected, the CSP cannot reproduce the same query and extract the same information. However, we might have a problem when the value of random$(0, r)$ is set too small in two queries, the CSP can construct a pseudo-query if it has some background information about the two trapdoors. This is because, in this scenario, the variance effects of $r2$ become too small. Thus, in order to avoid this situation, each time when we choose the variable random $(0, r)$, a relatively large value should be used. Although the number of keywords increased in the newly generated dictionary may be known by the CSP, it does not result in a real data privacy problem since the CSP does not know what are these keywords.

5.3. Key management

In our system model, we employ ABE technology to manage the file keys. The DO uses the access control information to encrypt the file keys, and then stores the encrypted keys in the cloud. The access control information associates to the attributes of the users. The ABE scheme is selectively secure under the composite 3-party Diffie–Hellman assumption and the bilinear Diffie–Hellman assumption. The CSP can judge whether the users can access the file keys with parts of their private keys that are the attribute information, but cannot decrypt the ciphertext to get the file keys. In this way, the security of the key management scheme is ensured.

6. Performance evaluation

We conduct extensive simulation experiments to evaluate MKQE and compare its performance with MRSE. Our testbed is a server with a Xeon Processor 2.40 GHz *16 core. It has Ubuntu 12.04 as the OS, and Java JRE 7.0 and J2EE 7.0 SDK are installed. The total number of the simulation code is 5000 lines written in java language. The data type of the elements in invertible matrices is double, and these elements are randomly generated using jema.jar package. In our experiments, we chose a real world dataset, Enron Email Dataset,[5] as the underlying dataset, and randomly select various numbers of emails from it.

[5] W.W. Cohen, Enron email dataset, http://www.cs.cmu.edu/~enron/.

Table 1
Secret key generation overhead (s), starting from 1000 keywords.

	1000	2000	4000	6000	8000
MRSE	3.8	33.9	320	1013	2540
MKQE	3.8	4.1	36.1	38.1	40.7

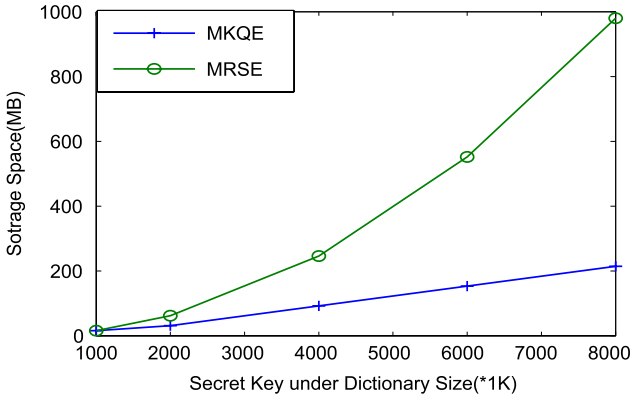

Fig. 4. The storage comparison under the same dictionary size.

6.1. The overhead of keyword dictionary expansion

A superior feature of MKQE over previous solutions is that it can naturally extend the keyword dictionary set at the minimal cost. In this set of experiments, we first compare the time consumption to generate new secret keys of the proposed Extended-Keygen algorithm with the MRSE algorithm when new keywords are introduced in the dictionary. Then, we compare the storage consumption performance.

The time consumption during the keyword dictionary expansion includes two parts: the time of generating a bit vector and two inverse random matrices, the time to compute the transport and inverse of two matrices. As we can observe from Table 1, MKQE is much more efficient than MRSE. Initially, when the original dictionary size is 1000, MRSE and MKQE use the same algorithm to generate the secret keys, thus their performances are the same. However, as the number of keywords in the dictionary increases to 2000, the overhead for new secret key generations in MRSE is increased by almost 9 times, while in MKQE, the difference is minimal. Thus, the time consumption in MRSE is about 825% higher than MKQE. Furthermore, the performance gap becomes even wider as more and more keywords are added. When the number of keywords increases from 6000 to 8000, the overhead in MRSE is 2540s, which is about 62 times higher than MKQE. Apparently, MKQE achieves much better performance than MRSE because it reuses the original set of indexes during the keyword expansion. The number of elements it needs to produce in the matrices is much smaller than in MRSE. In MRSE, all the elements have to be regenerated.

We also compare the storage consumption to update the keyword dictionary and other data structures in our scheme with MRSE as well. The result is shown in Fig. 4. From the result, we observe that MKQE consumes much less space. When the size of the dictionary is 2000, MKQE uses 31.0 MB to store the secret keys, while the MRSE requires 61.9 MB storage space. MKQE only needs 50% of the storage space required by MRSE. As the size of the dictionary increases, MKQE saves even more storage spaces than MRSE. For example, when the dictionary has 8000 keywords, MRSE needs 918 MB to store the whole set of secret keys, while our scheme only consumes 214.1 MB, which is 23.3% of MRSE. The reason is that in MKQE, we employ partitioned matrices and a large quantity of unused elements are not stored. With the help of the linked matrix list, MKQE can ensure that the space consumption grows linearly as the dictionary expands.

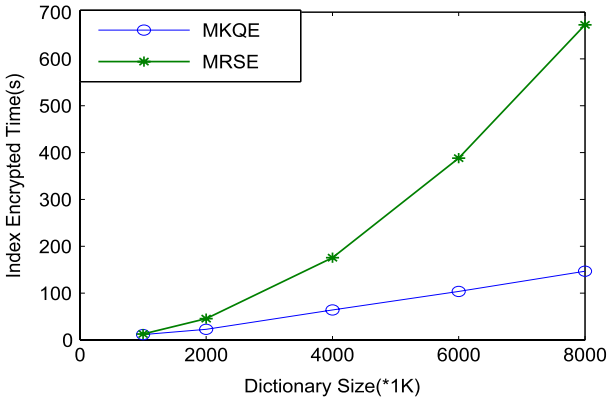

Fig. 5. Index encryption time comparison with 2000 file indexes encrypted.

Table 2
Time consumption comparison of weighted and nonweighted indexes (s).

	1000	2000	4000	6000	8000
Weighted	11.5	22.5	63.9	103.4	146.8
NoWeighted	11.3	22.4	63.1	102.2	144.1

6.2. Encryption time of index and trapdoor

To further investigate the performance of MKQE, we also analyze the time consumption for various operations. In this set of experiments, we evaluate the index construction time, the weight index construction time, the index update time and the trapdoor construction time. For all the experiments here, we use 2000 files.

Fig. 5 compares the time consumption to generate the encrypted file indexes with different sizes of the keyword dictionary. As shown in the result, MKQE takes less time to generate the indexes in all scenarios.

In the figure, we can see that when the number of keywords in the dictionary is 2000, MKQE spends 22.4 s to encrypt all the indexes, while MRSE takes 45.4 s. MKQE only exhausts 49.3% of time MRSE takes. As the dictionary becomes larger, MKQE still outperforms MSRE. In case the dictionary has 8000 keywords, MRSE takes 672.3 s to encrypt all the indexes, and MKQE only spends 144.1 s. It is about 78.6% less time than MRSE spends. This performance gain comes from the fact that in MKQE, we use block matrix to generate new secret key, and use linked list to store it and encrypt the index. It reduces the computation greatly. As shown in the figure the larger the size of the keyword dictionary, the more performance gain we can achieve.

As described in Section 4, MKQE can add the keyword weight features into the index construction process. To evaluate the performance impact, we compare the time consumption to generate the weighted indexes with the original nonweighted indexes scenarios. The result is shown in Table 2. Similarly, in these experiments, 2000 data files are chosen under the same keyword dictionary. From the result we can see that the additional overhead is minimal with the keyword weights being considered. Even in case the size of the dictionary is 8000, generating the weighted indexes only needs 2.7 s more compared with nonweighted indexes generation. This is because, in weighted index construction, we only change the values of the matrix elements, the basic matrix operations are identical as the construction process for the nonweighted indexes.

Fig. 6 shows the results of another time consumption metric when the dictionary is extended. In this set of experiments, we compare the time spending on the index update operation in MKQE with the time to re-encrypt the complete set of the file indexes in MRSE when the dictionary is expanded. Again, 2000 file indexes need to be generated. From the result we can observe that when

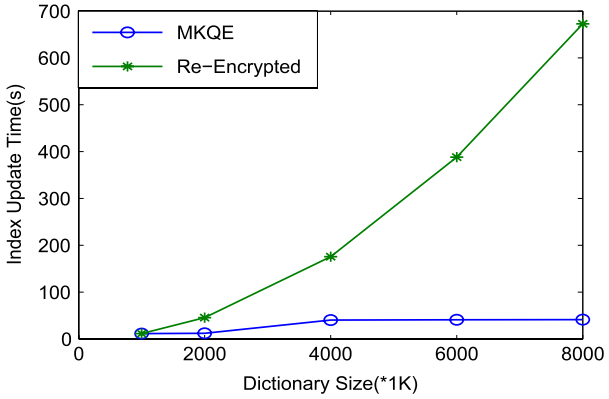

Fig. 6. Update time comparison with index re-encryption.

Table 3
Time consumption comparison on trapdoor generation (ms).

	1000	2000	4000	6000	8000
MRSE	27	45	97	192	313
MKQE	27	42	90	153	238

the keyword number in the dictionary increases from 1000 to 2000, MKQE only spends 12.0 s to update the original file index set, while re-encrypt the whole index set needs 45.4 s in MRSE. The index update operation in MKQE takes only 26.4% time of the index re-encryption operation in MRSE. As the dictionary becomes larger, MKQE achieves even better performance. When the dictionary is expanded from 6000 to 8000, it takes 41.2 s to update the original set of file indexes, while in MRSE, index re-encryption approach needs 672.7 s to do so. It is 16.3 times higher than index update mechanism adopted in MKQE scheme. This tremendous time savings comes from that MKQE uses the block matrix design and such a strategy guarantees that we can reuse the original set of indexes without any modifications when the dictionary is expanded, the DO only needs to encrypt the portion of indexes associated with the newly added keywords. This greatly reduces the index construction time.

Each time when a multi-keyword query is executed, a trapdoor has to be constructed on the DO. Thus, the trapdoor generation is a critical operation, and its performance has great influence on the system overall performance. Table 3 compares the trapdoor generation time consumption of MRSE and MKQE. From the result, we can see that it shows similar behaviors as the index encryption operation. In all scenarios, MKQE outperforms MRSE, and the performance gap becomes larger as the size of keyword dictionary increases. We believe this is because of the same reason as we mentioned above.

Finally, we also conduct the experiments to compare the query execution time between MRSE and MKQE, and we found that these two algorithms achieve comparable performance. It proves that although MKQE provides enhanced multi-keyword ranked query support for encrypted files, it does not adds extra overhead for query operations. Thus, we do not present the results here. However, from all the discussions above, we can draw the conclusions that MKQE has less time consumption than MRSE in all major encryption operations.

One advantage of our seamless expanding strategy is that we can use the newly generated secret key to search the original set of files. There is no need to restart the encrypt index generation operations for the original set of files. This feature provides great flexibility, especially for the distributed applications which cannot have service interruptions at any time.

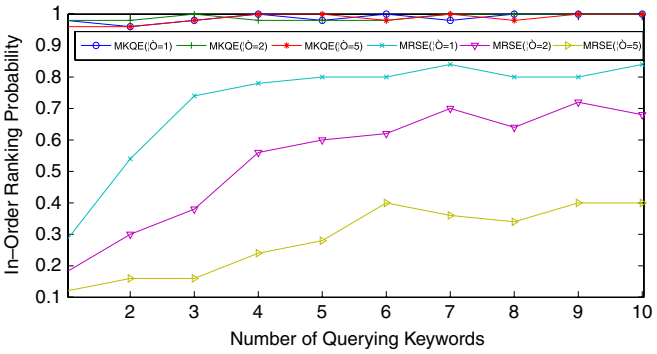

Fig. 7. The in-order ranking probability with different σ.

6.3. In-order result evaluation

We also compare the in-order ranking performance in the returned result set of MKQE with MRSE. In this experiment, we set the number of file indexes is 1000, the number of keywords in the dictionary is 1000, and the number of keywords in a query varies from 1 to 10. The sum of the dummy keywords follows the normal distribution with the mean $u = 0$. We choose three different variance σ: 1, 2 and 5. For each query, the top 50 ranked files are returned. As we mentioned above, for two files i and j, if I_i matches less keywords and gets a lower score than I_j, we call it is an in-order ranking result. Otherwise, an out-of-order error occurs. Fig. 7 shows the result.

As we can perceive from the results, for MRSE, the larger the variance σ, the lower the probability of the in-order ranking performance. Furthermore, when the number of keywords in the query becomes smaller, the probability turns to be even lower. For example, when there is only 1 keyword and the variance is 5, the in-order ranking probability is less than 5% in MRSE. Even for the queries with 10 keywords, the in-order probability are only 82%, 60% and 40% for the variance 1, 2 and 5, respectively. Such a result is not very satisfactory, the DC has a pretty high chance to miss the files they really needs in the top k locations. While in MKQE algorithm, no matter how many keywords are included in a query, and no matter what is the variance σ, the in-order probability is always above 95%. Especially, when the number of keywords is large ($\geq$6), MKQE can correctly identify almost all the matching files with the in-order probability close to 100%. In another word, the DC has very high probability to retrieve all the files it really needs.

As we discussed in Section 4, The reason MKQE achieves such a high in-order ranking performance than MRSE is because in MRSE, σ has great impacts on the range the sum of the dummy keyword values locates. A large variance always results in a lot of out-of order errors. In MKQE, we carefully choose two parameters r and $r2$ to narrow the range, thus to avoid out-of-order ranking to occur as much as possible, and achieve better in-order ranking performance.

6.4. Keyword access frequency analysis

Many research works [17,18] have studied the multi-keyword query problem and found out that information retrieval applications often have power-law constraints (also known as Zipf's Law and long tails), and the access frequency (weight) always follows the heavy tail distributions. Typically, a DC prefers to see the files which have high weight keywords. MKQE consider this factor when generating the query result.

In this set of experiments, we compare the query results in MRSE and MKQE when taking keyword weights into accounts. 1000 queries are executed, and each query has 4 keywords. The total number of keywords in the system is 1000. The weights of

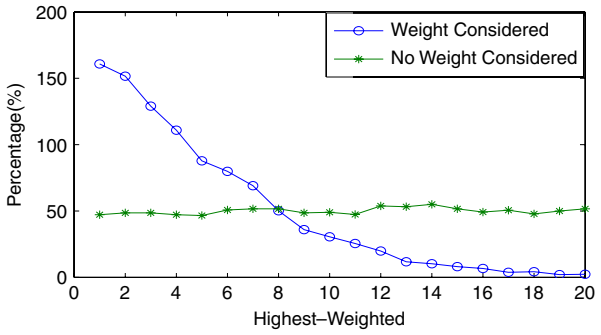

Fig. 8. Percentage of files containing highest-weighted keywords in the top 10 locations.

keywords follow the heavy tail distributions, and we count the percentage of the 20 highest-weighted keywords appeared in the top k positions in the matching result set. Here, we set $k = 10$, which means the first 10 files with the highest scores will be returned for each query. Fig. 8 shows the result. X axis represents the keywords ordered by their weights. The first one has the highest weight overall. Y axis shows the percentage of the returned files in which those keywords appears. As we can see from the figure, in MRSE, no matter how popular a keyword is, only 5% of the returned files have one of these keywords. While in MKQE, a keyword with a larger weight has much higher probability to appear in the result set. Clearly, due to the lack of the weight consideration, the data consumers using MRSE has less chances to get the files they really need. MKQE solve this problem by giving more preference on the highest-weighted keywords. We found that the probability of a file contains certain keywords also follows the heavy-tail distribution.

Although we use the heavy tail distribution to determine the keyword weights in this set of experiments, as we can see from the Formulas (13) and (14), MKQE works for any distributions where the weights of the keywords are uneven.

7. Related works

Cloud computing provides an efficient way for end users to access data anywhere and anytime. However, security and privacy concerns force data owners to encrypt sensitive data before uploading onto the cloud. Thus, supporting keyword-based searching capability on encrypted data is critical. The standard mechanism is to encrypt keyword indexes of stored files and upload them onto the cloud servers as well. The search is conducted with a secure key generated trapdoor on the data owner. Kamara et al. [19] propose a possible architecture for a cryptographic storage service for cloud storage. When preparing data to be stored in the cloud, the data owner creates indexes and encrypts the data with a symmetric encryption scheme (e.g., AES) under a unique key. It then encrypts the indexes using a searchable encryption scheme and encrypts the unique key with an attribute-based encryption scheme under an appropriate policy. Finally, it encodes the encrypted data and indexes in such a way that the data verifier can later verify their integrity using a proof of storage. The similar strategy used in many other research projects as well. In [20], Song et al. apply a deterministic encryption algorithm to encrypt the keywords, and use stream ciphers to post-encrypt keywords for security. Goh et al. [21] adopt the bloom-filter technology to test if a keyword is included in an index, which can tolerate low probability of mistaken identification to speed up the queries. Other researches [3,22] have made improvements on this issue by using xor operations between encrypted index vectors and query vectors. In [4], Wang defines a novel infrastructure on secure ranked query, which is based on the file length and term frequency. Liu et al. [23] employ an asymmetric setting to construct searchable encryptions, in which users encrypt their indexes with a public key, and authorized users have a private key to conduct searches. Similar ideas are proposed in [24,25,5] as well. However, all these solutions focus on the single keyword query problem only.

Many research works have been conducted on multi-keyword queries to enrich search functionalities. Park et al. [26] proposes a multi-keyword search scheme on encrypted data. It is based on the bilinear map, which was used in the single keyword query [23]. This paper defines two schemes for multi-keyword queries. Every index contains the same number of keyword fields, and each field is filled with a keyword. The keyword is encrypted with the public key. The trapdoor constructed with the privacy key must have the same number of keyword fields under a specified order. Research works in [27–31] adopt conjunctive searches over encrypted keywords. They also require that the indexes of the returned files must match all the query keywords. Recent works [32–34] propose a more general approach. They employ bilinear maps to encrypt the keywords and uses Lagrangian Coefficients to find the matching keywords. If one or more keywords are matched in an index, the corresponding file is returned. It can support both conjunctive and disjunctive searches. However, a disjunctive keyword search could result in undifferentiated results, and it is hard to design an effective ranked algorithm on the result. Thus, for a multi-keyword query, all the files which contain a subset of the searching keywords are returned. The large number of files makes it hard to find the ones the DC really needs.

None of the above algorithms can support multi-keyword ranked search. This question was first raised in [6]. As more enterprises and private data owners are migrating their data onto the cloud environment, it is increasingly important to provide an efficient solution. MKQE addresses this issue with a novel algorithm. It can support effective multi-keyword ranked queries without the drawbacks in [6].

8. Conclusions and future work

In this paper, we aim to provide a viable solution for multi-keyword ranked query problems over encrypted data in the cloud environment. We first define the problem, analyze the existing solutions, and design a novel algorithm called MKQE to address the issues. MKQE uses a partitioned matrices approach. When the amount of encrypted data increases and more keywords need to be introduced, the searching infrastructure can be naturally expanded with the minimal overhead. We also design a new trapdoor generation algorithm, which can solve the out-of-order problem in the returned result set without losing the data security and privacy property. Furthermore, the weights of the keywords are taken into consideration in the ranking algorithm when generating the query result. The DC has high probability to retrieve the files they really need. The simulation experiments confirm that our approach can achieve better performance with a satisfactory security level. In the future, we will explore new approaches to further enhance multi-keyword query capabilities. We are designing new algorithms to provide extra functionalities such as semantic query and fuzzy keyword query. We are also working on applying new storage techniques such as Solid State Disk (SSD) to boost the query performance.

Acknowledgments

This research is partially supported by National Natural Science Foundation of China under grants 61173170 and 60873225, Innovation Fund of Huazhong University of Science and Technology under grants 2013QN120, 2012TS052 and 2012TS053.

References

[1] D. Zissis, D. Lekkas, Addressing cloud computing security issues, Future Gener. Comput. Syst. 28 (3) (2012) 583–592.
[2] Identity-based data storage in cloud computing, Future Gener. Comput. Syst. 29 (3) (2013) 673–681.
[3] Y.-C. Chang, M. Mitzenmacher, Privacy preserving keyword searches on remote encrypted data, in: J. Ioannidis, A. Keromytis, M. Yung (Eds.), Applied Cryptography and Network Security, in: Lecture Notes in Computer Science, vol. 3531, Springer Berlin, Heidelberg, 2005, pp. 391–421.
[4] C. Wang, N. Cao, J. Li, K. Ren, W. Lou, Secure ranked keyword search over encrypted cloud data, in: The 30th International Conference on Distributed Computing Systems, ICDCS'10, 2010, pp. 253–262.
[5] J. Li, Q. Wang, C. Wang, N. Cao, K. Ren, W. Lou, Fuzzy keyword search over encrypted data in cloud computing, in: IEEE Conference on Computer Communications, INFOCOM'10, 2010, pp. 1–5.
[6] N. Cao, C. Wang, M. Li, K. Ren, W. Lou, Privacy-preserving multi-keyword ranked search over encrypted cloud data, in: IEEE Conference on Computer Communications, INFOCOM'11, 2011, pp. 829–837.
[7] National Bureau of Standards, Data Encryption Standard, U.S. Department of Commerce, Washington, DC, USA, Jan. 1977.
[8] H. Dobbertin, V. Rijmen, A. Sowa, Advanced Encryption Standard—AES: 4th International Conference, AES 2004, Bonn, Germany, May 10–12, in: Lecture Notes in Computer Science, 2004.
[9] V. Goyal, O. Pandey, A. Sahai, B. Waters, Attribute-based encryption for fine-grained access control of encrypted data, in: Proceedings of the 13th ACM Conference on Computer and Communications Security, CCS'06, ACM, New York, NY, USA, 2006, pp. 89–98.
[10] R. Ostrovsky, A. Sahai, B. Waters, Attribute-based encryption with non-monotonic access structures, in: Proceedings of the 14th ACM Conference on Computer and Communications Security, CCS'07, 2007, pp. 195–203.
[11] L. Kissner, Privacy-preserving distributed information sharing, Ph.D. Thesis, University of Carnegie Mellon, 2006.
[12] I.H. Witten, A. Moffat, T.C. Bell, Managing Gigabytes: Compressing and Indexing Documents and Images, second ed., Morgan Kaufmann, San Francisco, CA, 1999.
[13] G. Salton, C. Buckley, Term-weighting approaches in automatic text retrieval, in: Information Processing and Management, 1988, pp. 513–523.
[14] J. Zobel, A. Moffat, Exploring the similarity space, Sigir Forum 32 (1998) 18–34.
[15] R. Kumar, S. Vassilvitskii, Generalized distances between rankings, in: WWW, 2010, pp. 571–580.
[16] H.P. Luhn, A statistical approach to mechanized encoding and searching of literary information, IBM J. Res. Dev. 1 (4) (1957) 309–317.
[17] X.Z. Jie, J. Yu, J. Doyle, Heavy tails, generalized coding, and optimal web layout, in: IEEE INFOCOM 2001, IEEE Press, 2001, pp. 1617–1626.
[18] S. Chaudhuri, K. Church, A.C. König, L. Sui, Heavy-tailed distributions and multi-keyword queries, in: The 30th Annual International ACM SIGIR Conference on Research and Development in Information Retrieval, SIGIR'07, ACM, New York, NY, USA, 2007, pp. 663–670.
[19] S. Kamara, K. Lauter, Cryptographic cloud storage, in: R. Sion, R. Curtmola, S. Dietrich, A. Kiayias, J. Miret, K. Sako, F. Seb (Eds.), Financial Cryptography and Data Security, in: Lecture Notes in Computer Science, vol. 6054, Springer Berlin, Heidelberg, 2010, pp. 136–149.
[20] D.X. Song, D. Wagner, A. Perrig, Practical techniques for searches on encrypted data, in: Security and Privacy, 2000. S P 2000. Proceedings. 2000 IEEE Symposium on, 2000, pp. 44–55.
[21] E.-J. Goh, Secure indexes, Cryptology ePrint Archive, Report 2003/216, 2003.
[22] R. Curtmola, J. Garay, S. Kamara, R. Ostrovsky, Searchable symmetric encryption: improved definitions and efficient constructions, in: Proceedings of the 13th ACM Conference on Computer and Communications Security, CCS '06, ACM, New York, NY, USA, 2006, pp. 79–88.
[23] Q. Liu, G. Wang, J. Wu, An efficient privacy preserving keyword search scheme in cloud computing, in: International Conference on Computational Science and Engineering, CSE'09, vol. 2, 2009, pp. 715–720.
[24] D. Boneh, G. Di Crescenzo, R. Ostrovsky, G. Persiano, Public key encryption with keyword search, in: C. Cachin, J. Camenisch (Eds.), Advances in Cryptology - EUROCRYPT'04, in: Lecture Notes in Computer Science, vol. 3027, Springer Berlin, Heidelberg, 2004, pp. 506–522.
[25] M. Abdalla, M. Bellare, D. Catalano, E. Kiltz, T. Kohno, T. Lange, J. Malone-Lee, G. Neven, P. Paillier, H. Shi, Searchable encryption revisited: consistency properties, relation to anonymous ibe, and extensions, in: V. Shoup (Ed.), Advances in Cryptology, CRYPTO'05, in: Lecture Notes in Computer Science, vol. 3621, Springer Berlin, Heidelberg, 2005, pp. 205–222.
[26] D. Park, K. Kim, P. Lee, Public key encryption with conjunctive field keyword search, in: C. Lim, M. Yung (Eds.), Information Security Applications, in: Lecture Notes in Computer Science, vol. 3325, Springer Berlin, Heidelberg, 2005, pp. 73–86.
[27] P. Golle, J. Staddon, B. Waters, Secure conjunctive keyword search over encrypted data, in: M. Jakobsson, M. Yung, J. Zhou (Eds.), Applied Cryptography and Network Security, in: Lecture Notes in Computer Science, vol. 3089, Springer Berlin, Heidelberg, 2004, pp. 31–45.
[28] L. Ballard, S. Kamara, F. Monrose, Achieving efficient conjunctive keyword searches over encrypted data, in: S. Qing, W. Mao, J. Lapez, G. Wang (Eds.), Information and Communications Security, in: Lecture Notes in Computer Science, vol. 3783, Springer Berlin, Heidelberg, 2005, pp. 414–426.
[29] D. Boneh, B. Waters, Conjunctive, subset, and range queries on encrypted data, in: S. Vadhan (Ed.), Theory of Cryptography, in: Lecture Notes in Computer Science, vol. 4392, Springer Berlin, Heidelberg, 2007, pp. 535–554.
[30] R. Brinkman, Searching in encrypted data, Ph.D. Thesis, University of Twente, 2007.
[31] Y. Hwang, P. Lee, Public key encryption with conjunctive keyword search and its extension to a multi-user system, in: T. Takagi, T. Okamoto, E. Okamoto, T. Okamoto (Eds.), Pairing-Based Cryptography, Pairing 2007, in: Lecture Notes in Computer Science, vol. 4575, Springer Berlin, Heidelberg, 2007, pp. 2–22.
[32] J. Katz, A. Sahai, B. Waters, Predicate encryption supporting disjunctions, polynomial equations, and inner products, in: The Theory and Applications of Cryptographic Techniques 27th Annual International Conference on Advances in Cryptology, EUROCRYPT '08, 2008, pp. 146–162.
[33] A. Lewko, T. Okamoto, A. Sahai, K. Takashima, B. Waters, Fully secure functional encryption: attribute-based encryption and (hierarchical) inner product encryption, in: H. Gilbert (Ed.), Advances in Cryptology, EUROCRYPT 2010, in: Lecture Notes in Computer Science, vol. 6110, Springer Berlin, Heidelberg, 2010, pp. 62–91.
[34] E. Shen, E. Shi, B. Waters, Predicate privacy in encryption systems, in: O. Reingold (Ed.), Theory of Cryptography, in: Lecture Notes in Computer Science, vol. 5444, Springer Berlin, Heidelberg, 2009, pp. 457–473.

Ruixuan Li received the B.S., M.S. and Ph.D. in Computer Science from the Huazhong University of Science and Technology, China in 1997, 2000 and 2004 respectively. He was a Visiting Researcher in the Department of Electrical and Computer Engineering at the University of Toronto in 2009–2010. He is currently a full Professor in the School of Computer Science and Technology at Huazhong University of Science and Technology. His research interests include cloud computing, big data management, and distributed system security. He is a member of IEEE and ACM.

Zhiyong Xu received the B.S. and M.S. in Computer Science from the Huazhong University of Science and Technology, China in 1994 and 1997 respectively, and Ph.D. degrees in Computer Engineering from the University of Cincinnati in 2003. He is currently an Associate Professor in the Department of Mathematics and Computer Science at Suffolk University. His research interests include cloud computing, peer-to-peer computing, and parallel and distributed systems. He is a member of IEEE.

Wansheng Kang received the B.S. degree in Computer Science and Engineering from Shandong University, China in 2010. He is currently a master candidate in the School of Computer Science and Technology at the Huazhong University of Science and Technology. His research interests include cloud computing, information retrieval, and distributed system security.

Yow Kin Choong obtained his B.Eng. with 1st Class Honor from the National University of Singapore in 1993, and his Ph.D. from Cambridge University, UK in 1998. He joined the Nanyang Technological University, Singapore as a faculty member in May 1998. He is a researcher in Shenzhen Institute of Advanced Technology, Chinese Academy of Science since 2012. His research interests include cloud computing, system security, wireless communications and computational intelligence.

Cheng-Zhong Xu received the B.S. and M.S. in Computer Science from Nanjing University, China in 1986 and 1989 respectively, and Ph.D. degrees in Computer Engineering from the University of Hong Kong in 1993. He is now a Professor of Electrical and Computer Engineering at Wayne State University and the Director of the Cloud Computing Center in Shenzhen Institute of Advanced Technology, Chinese Academy of Science. Professor Xu's main research interests include networked computing systems, reliability, availability, power efficiency, and security. He is an editor of IEEE Trans. on Parallel and Distributed Systems (TPDS) and Journal of Parallel and Distributed Computing (JPDC).