The VLDB Journal (2019) 28:25–46
https://doi.org/10.1007/s00778-018-0517-6

REGULAR PAPER

VBTree: forward secure conjunctive queries over encrypted data for cloud computing

Zhiqiang Wu[1] · **Kenli Li**[1]

Received: 23 December 2017 / Revised: 25 June 2018 / Accepted: 20 July 2018 / Published online: 4 August 2018

Abstract
This paper concerns the fundamental problem of processing conjunctive keyword queries over an outsourced data table on untrusted public clouds in a privacy-preserving manner. The data table can be properly implemented with tree-based searchable symmetric encryption schemes, such as the known Keyword Red–Black tree and the Indistinguishable Bloom-filter Tree in ICDE'17. However, as for these trees, there still exist many limitations to support sub-linear time updates. One of the reasons is that their tree branches are directly exposed to the cloud. To achieve efficient conjunctive queries while supporting dynamic updates, we introduce a novel tree data structure called virtual binary tree (VBTree). Our key design is to organize indexing elements into the VBTree in a top-down fashion, without storing any tree branches and tree nodes. The tree only exists in a logical view, and all of the elements are actually stored in a hash table. To achieve forward privacy, which is discussed by Bost in CCS'16, we also propose a storage mechanism called version control repository (VCR), to record and control versions of keywords and queries. VCR has a smaller client-side storage compared to other forward-private schemes. With our proposed approach, data elements can be quickly searched while the index can be privately updated. The security of the VBTree is formally proved under the IND-CKA2 model. We test our scheme on a real e-mail dataset and a user location dataset. The testing results demonstrate its high efficiency and scalability in both searching and updating processes.

Keywords Privacy preserving · Cloud computing · Searchable symmetric encryption

1 Introduction

1.1 Background and motivation

Cloud computing provides convenient and on-demand network access to a shared pool of configurable computing resources. More and more companies outsource their data and computing services to clouds such as Amazon EC2 and S3 [1], Microsoft Azure [2], and Google App Engine [3] for a lower cost, higher reliability, and better performance. However, data and computing privacy turn out to be the critical roadblock of cloud computing. Clouds are no longer secure due to attacks by hackers, leakages by malicious cloud managers, and unpredictable software security vulnerabilities. To preserve user privacy in cloud computing, researchers propose searchable symmetric encryption (SSE), which allows encryption while still enabling the client to search for keywords. Clouds can then provide computing and searching services on encrypted data without learning the content of files and queries.

In this paper, we study the fundamental problem of processing conjunctive queries and update queries on untrusted public clouds in a privacy-preserving manner. A conjunctive query is a multidimensional condition string, in which each of the conditions can match one or more results. The conjunctive queries and updates are essential features of commonly used databases, such as Mysql and Cassandra. While searching over encrypted data with conjunctive expressions, there are three parties involved—clouds, data owner, and data users. As shown in Fig. 1, with secret keys K, the data owner encrypts her documents $\mathcal{F} = \{d_1, d_2, \ldots, d_n\}$ to encrypted files and then uploads the encrypted files to the cloud. To

✉ Zhiqiang Wu
cxiaodiao@hnu.edu.cn

Kenli Li
lkl@hnu.edu.cn

[1] College of Information Science and Engineering, National Supercomputing Center in Changsha, Hunan University, Hunan 410082, China

Fig. 1 Cloud computing model

quickly match the data, the owner also creates an encrypted index and uploads the index to the cloud. The encrypted files and the index constitute an encrypted data table on the public cloud. The owner and the users share a set of secret keys K, which can encrypt a conjunctive query q to $T_K(q)$. The data user sends $T_K(q)$ to the cloud to quickly search the data table to get a set of file identifiers $DB(q)$ that contains all keywords of q. For example, a user remotely searches her files from the encrypted data table, using a conjunctive SQL query 'select * from files where owner="Tom" and type="pdf"'. We assume the data owner and data users are trusted, but the clouds are not fully trusted. The cloud may attempt to obtain information about the content of files and queries from the users' and the owner's requests while performing any operations. In the scenarios, clouds need to return all results accurately while being prohibited from obtaining any sensitive information.

If SSE schemes are not optimized for conjunctive queries, it will degrade to an approximately linear search to perform a query consisting of a high-frequency keyword and a low-frequency one. The query complexity of these schemes is at least $\Omega(\sum_{w\in q}|DB(w)|)$, where q is a u-dimensional conjunctive query $w_1 \wedge w_2 \cdots \wedge w_u$. For example, if the user searches for 'select * from users where name="Tom" and gender="Male"', non-optimized SSE schemes will traverse almost half of the records in the data table.

Apart from creating the index supporting conjunctive queries, the data owner also wants to dynamically add, remove, and modify the index on the cloud. The data owner performs these operations by creating an update trapdoor and sending it to the cloud to update the index. The cloud can accurately update the index, but it doesn't learn the contents of the updated files. For example, the owner wants to update the index, using a SQL query 'insert into files values (owner="Tom", filename="computer.pdf"), (owner="Jerry", filename="paper.pdf")'. The cloud can run this SQL correctly, but it doesn't know how many files Tom has.

Many dynamic SSE schemes leak non-negligible information to the cloud in their updates. The leakage might come from the update tokens that are correlated with historical search queries and historical update queries. In [4], the authors show that even small leakage in updates can be leveraged by an attacker to reveal the user's queries.

1.2 Security model

We adopt the IND-CKA2 (i.e., also called the adaptive IND-CKA) definition, which was proposed by Curtmola et al. in CCS'06 [5], to describe the security level for dynamic SSE schemes. Our target is to build an encrypted index for conjunctive keywords matching and updating, which can achieve IND-CKA2 strong semantic security. IND-CKA2 consists of two parts, IND and CKA2. IND means indistinguishability, under which the adversary cannot distinguish a real view from a simulated view in polynomial time. CKA2 denotes that the adversary can query a random oracle for trapdoors with certain conditions. The queries include search and update queries. A data addition query and a data deletion query are both considered as the update queries. Even if the adversary can choose certain keywords for attacks, the remaining keywords in the index are still hidden from the adversary. If there exists a simulator that can adaptively simulate the adversary's view using only leakage information, it implies that the adversary always cannot infer any remaining keywords in the index in polynomial time, and the index is secure. The IND-CKA2 security model represents the difficulty level in getting the plaintext from the encrypted index.

In the IND-CKA2 model, it is assumed that the adversary cannot distinguish the output of a pseudo-random function used in the scheme from the output of a truly random function, i.e., a random oracle. A random oracle is simply a box that takes a binary string as input and returns a binary string as output. Given a random oracle H and an input x, the only way to evaluate $H(x)$ is to explicitly query x. The internal workings of the box are unknown and inscrutable. No one claims that a random oracle exists, yet we assume that we can use an appropriately designed cryptographic hash function to instantiate the random oracle.

In non-adaptive secure schemes (i.e., also called IND-CKA), the simulator cannot adaptively simulate the adversary's view using the leakage functions, and the simulator can simulate the view only one time after the adversary performs the search and update queries. IND-CKA2 is more secure than IND-CKA. If an encrypted index can achieve IND-CKA2 security, even if the adversary can query the random oracle using a function based on historical queries, the index is still secure. However, in the IND-CKA model, the adversary is prohibited from accessing the random oracle interactively. If a query scheme is secure under the IND-CKA2 model, then it is also secure under the IND-CKA model, but not vice versa.

Besides the IND-CKA2 security, forward privacy is also necessary for dynamic SSE schemes. If a dynamic SSE scheme doesn't achieve forward privacy, for example, a work of Kamara et al. [6], the scheme can be easily broken by an attack, called file injection attack proposed in [4]. According to their design, an adversary can break the index of non-

forward-private schemes by injecting files into the index. Since the injected files contain keywords created by the adversary, the adversary can easily infer the queried keywords by adaptively injecting files and performing a binary search to see whether these keywords are contained in a file. To thwart this attack, dynamic SSE schemes require their forward privacy to be guaranteed.

A forward secure dynamic SSE scheme is an SSE scheme whose update procedure leaks nothing about the correlation between the updated elements and historical queries (search and update queries). The update leakages of these schemes contain the operation name, the file identifiers, and the number of updated keywords, at most. It is not allowed leaking any information correlated with previously queried keywords in updates.

1.3 Limitation of prior art

A dynamic SSE scheme requires at least two features to become secure and expressive. One is achieving adaptive and forward security, and the other is supporting efficient conjunctive queries. To the best of our knowledge, there are few SSE works achieving these two simultaneously, and even fewer works can achieve forward privacy.

The dynamic SSE scheme Sophos proposed by Bost in CCS'16 [7] achieves forward privacy and leaks fewer than most of the other ones. However, Sophos achieves its forward privacy at the cost of large client-side storage of $O(m \log_2 n)$ both for the data owner and for the data users, where m is the number of distinct keywords, and n is the number of files. In the case of the index being frequently searched but infrequently updated, the user's local storage of Sophos is as much as the data owner's. Sophos's construction requires RSA-based expensive asymmetric encryption, which is much less efficient than schemes based on pseudo-random functions. The later forward-private scheme FFSSE proposed by Liu et al. [8] improves the performance, but FFSSE faces the same problem as Sophos: The data user's storage is large. Sophos and FFSSE are single-keyword schemes.

The tree-based data structures are suitable for handling efficient conjunctive queries, such as the known KRB proposed by Kamara et al. [6] and IBTree introduced by Li et al. in ICDE'17 [9]. The conjunctive query complexity of them is sub-linear in the average case. However, it is difficult to achieve sub-linear update complexity in KRB and IBTree. One of the reasons is due to their leaked branch privacy. There are two limitations to a branch-leaked tree. First, the cloud learns the sizes of every tree node. Although they can pad all tree nodes or parts of tree nodes to the same size to avoid leaking the number of real keywords in each node, this makes it difficult to update the tree, not to mention achieving forward privacy. The padding also brings other unavoidable performance penalties, such as unscalable index size of $O(mn)$ in KRB. Second, the cloud still learns all locations of the updated entries in each tree node. To avoid the potential leakage, KRB rebuilds all tree nodes involved in every node update. Compared to hash-table-based SSE schemes, Bloom-filter-based ones, for example, Z-IDX [10], PBTree [11], and IBTree, have significant drawbacks: lacking scalability in dynamically modifying the Bloom filter and changing the Bloom filter size.

1.4 Proposed approach

To achieve efficient conjunctive query processing, we propose a novel tree data structure called virtual binary tree (VBTree). Unlike KRB and IBTree, we create the tree fully stored in a hash table, for efficient updates and better scalability. Informally, a virtual binary tree is an encrypted full binary tree stored in a hash table without leaking their tree branch privacy to the cloud. In the VBTree, to traverse children items of a tree item, we use a binary-path-based function to search for its left child item and its right child item. We store all tree items without storing tree branches and tree nodes. The tree structure is implicit in the encrypted tree items. The tree only exists in a logical view. The size of each data nodes is completely hidden from the cloud. The cloud learns that there is a tree, but it learns nothing about where the tree nodes are. If the data user performs a conjunctive query, the cloud only learns paths from the tree root to all of the result leaves (i.e., access pattern). Items that have not been visited are still hidden from the cloud.

VBTree achieves IND-CKA2 security. All keywords in VBTree are encrypted by a pseudo-random function and a random oracle, which enables the simulator to adaptively simulate the adversary's queries. We formally prove that our scheme is IND-CKA2 secure.

VBTree also achieves forward privacy in updates. We introduce a novel storage structure called version control repository (VCR), which contains a local repository (LR) and a cloud repository (CR). Keywords and queries are labeled with versions. If a keyword has been queried by the data users, the data owner creates a new version number for this keyword in the next updates. The version control repository makes the update procedures leak nothing about user's historical queries.

The client-side storage of LR can be divided into two classes, the storage for the data users, which is mainly used in searching, and the storage for the data owner, which is mainly used in updates. In the cases of the index being frequently searched but infrequently updated, the client-side storage of the data user is small, as the data user can use a default keyword version to search. The default value don't need to be stored on a disk.

We design the tree and VCR to be as concise as possible provided that we can achieve the goals. Our search and update

algorithms involve only pseudo-random computations, without any symmetric encryption and decryption (except for the encrypted files). This makes the searches and the updates extremely efficient.

Like PBTree in VLDB'14 [11] and IBTree, VBTree still supports multidimensional range queries, for example, a SQL 'select * from users where age $\leq$ 20 and salary $\geq$ 10,000.' One can convert a numeric range-query scheme to a Boolean keyword-query scheme like the work in PBTree. For example, the number 6 is converted into {'0110', '011 $*$ ', '01 $**$', '0 $***$ ', '$****$'}. We call these strings as keywords. Given a set of numbers, we consider these numbers as a set of files. These files, as well as all keywords, will be put into the index with different file identifiers. A range query can be considered as a disjunctive keyword query. For example, the range [0, 8] is converted into {'0 $***$ ', '1000'}, where the string '0***' means the range [0, 7]. Our algorithm $\mathcal{VSA}$-3 in Sect. 4 has shown how to run a CNF query. The query is '0 $***$ ' $\vee$ '1000'. Then, the number 6 will be matched by the range query [0, 8]. In this paper, we focus only on conjunctive keyword queries.

The leakage of the VBTree scheme mainly consists of the search and access patterns. We allow for the leakage of the access pattern of all subqueries, which is also leaked in KRB, PBTree, and IBTree, in return for all work of conjunctive queries that can be fully done on the cloud. In our design, file identifiers don't need to be sent back to the client for decryption. One can combine VBTree with other techniques, for example, the blind storage in S&P'14 [12], to reduce the access pattern leakage.

Table 1 shows the comparison of current tree-based SSE schemes and typical forward-private schemes.

1.5 Key contributions

Our scheme is the first dynamic-SSE-based conjunctive query scheme that achieves adaptive and forward security, efficient conjunctive query processing, scalable index size, and sub-linear updating time with small leakage. The key novelty of this paper exists in five aspects. First, VBTree helps us to achieve efficient conjunctive queries, structure indistinguishability, and dynamical updating with small leakage. Second, the version control repository (VCR) can help non-forward-private dynamic SSE schemes to achieve forward privacy. Third, our VCR has a smaller client-side storage size compared to existing forward-private works. Fourth, we propose a novel approach to instantiate a VBTree online with small leakage. Fifth, our schemes and proposed algorithms are scalable and can be easily deployed to a distributed network.

Table 1 Comparison of current tree-based SSE schemes and typical forward-private schemes

Scheme	Security	Query type	Forward private	Index size	Search time	Update time	Owner storage	User storage
KRB [6]	IND-CKA2	Conj.	x	$O(mn)$	$O(x \log n)$	$O(m \log n)$	$O(m \log n)$	$O(1)$
PBTree [11,13]	IND-CKA	Range	x	$O(n \log n)$	$O(r \log n)$	$O(\log n)$	$O(n \log n)$	$O(1)$
KBB [14]	COA	Multi-k.	x	$O(mn)$	$O(m^2 + xm \log n)$	$O(m^2 \log n)$	$O(mn)$	$O(m^2)$
IBTree [9]	IND-CKA2	Conj.	(Static)	$O(cn \log n)$	$O(x \log n)$	–	–	$O(1)$
Sophos [7]	IND-CKA2	Single-k.	$\surd$	$O(N)$	$O(t_a r_u)$	$O(t_a)$	$O(m \log n)$	$O(m \log n)$
FFSSE [8]	IND-CKA2	Single-k.	$\surd$	$O(N)$	$O(r_u)$	$O(1)$	$O(m \log n)$	$O(m \log n)$
VBTree (static)	IND-CKA2	Conj.	(Static)	$O(N \log n)$	$O(x \log n)$	–	–	$O(1)$
VBTree (dynamic)	IND-CKA2	Conj.	$\surd$	$O(N \log n)$	$O(r_q x \log n)$	$O(L)$	$O(m \log n)$	$O(1) \sim O(m_u)$

n denotes the number of files, with m the number of keywords that can be queried, with N the number of keyword/document pairs, with r_u the number of update times of the keyword, with t_a the time cost for a RSA-based asymmetric encryption operation, with r the number of matched documents, and with c the maximum number of keywords of all documents. For a conjunctive query q, $x = \min_{w \in q} |DB(w)|$, r_q denotes the number of update times of the keywords of q after the last search of these keywords, m_u denotes the number of newly updated keywords, and L is the height of the tree. The search time of PBTree is evaluated by range queries, with Sophos and FFSSE single-keyword queries, and others are conjunctive queries. Owner storage denotes the storage size of the data owner. This storage information might be downloaded from the cloud for decryption. User storage denotes the storage size of the data user. COA denotes that the scheme is secure against Ciphertext-Only Attacks

2 Related work

There are a series of SSE-based privacy-preserving keyword query schemes such as the ones proposed by Song et al. in [15], by Chang et al. in ACNS'05 [16], and SSE-1 in CCS'06 [5]. In ICDCS'15 [17], the authors proposed a scheme for privacy-preserving string matching. However, all of them only achieve non-adaptive security. Curtmola et al. introduced the notion of adaptive security for SSE and proposed the adaptively secure scheme SSE-2. The adaptively secure schemes of [6,18–20] and SSE-2 have an unscalable index size of $O(mn)$. The construction proposed by Chase et al. in ASIACRYPT'10 [18] supports two-dimensional arrays and graphs. Kurosawa and Ohtaki [19] designed the first UC-secure SSE scheme, but their construction needs a linear search time. Cash et al. showed a Boolean query scheme OXT protocol based on modular exponentiation in CRYPTO'13 [21], and they provided both non-adaptively secure and adaptively secure constructions. The PBTree data structure in VLDB'14 [11] achieves fast numeric range query with non-adaptive security. PBTree can be easily extended to support range conjunctive queries. In S&P'14 [22], Pappas et al. showed how to build a Boolean encrypted-search solution, which makes use of adaptively secure Bloom filters. Ishai et al. proposed a two-server SSE scheme in [23] that hides the access pattern and supports various complex queries including ranges and substrings. The work of Kamara et al. [24] introduced a scheme to process SQL queries over encrypted data for a relational database, but the work doesn't support conjunctive queries. Li and Liu proposed the first adaptively secure SSE scheme supporting conjunctive numeric range and keyword queries in ICDE'17 [9]. They also proposed the indistinguishable Bloom filter (IBF), which helps them to build a compact and adaptively secure index.

An important aspect of SSE is whether the scheme is dynamic or not. In dynamic SSE schemes, the client can update the document collection after starting to search on the index. There are many dynamic SSE-based privacy-preserving keyword query schemes such as Z-IDX by Goh [10], the work by Liesdonk et al. [20], DSSE by Kamara et al. in CCS'12 [25], the work by Cash et al. in NDSS'14 [26], KRB [6], Sophos [7], DMRS (KBB) [14], and DIEX [27]. DMRS is COA-secure, which guarantees that the scheme is secure against ciphertext-only attacks. Z-IDX is non-adaptively secure. The others are adaptively secure. COA-security is much weaker than IND-CKA security and IND-CKA2 security since the adversary of COA cannot choose keywords for attacks. DSSE in [25] leaks the intersecting information of all update queries. The schemes in [20,25,26], and Sophos don't optimize for conjunctive keyword queries. Their search time is linear in the maximum sizes of the subqueries. DIEX achieves only worst-case sublinear disjunctive queries. DMRS supports multikeyword ranked search with an unscalable index size of $O(mn)$ and index construction time of $O(m^2n)$. In DMRS, a tree data structure called KBB was introduced. The advantage of KBB is that its search and access patterns are small.

In [28], Wang et al. introduced a fuzzy keyword SSE scheme, which supports misspelled-keyword queries. Later, Fu et al. [29] also presented a fuzzy keyword SSE scheme. They use Locality-Sensitive Hashing (LSH) to preprocess all keywords. However, their search efficiency is linear in the total number of files.

The Oblivious RAM (ORAM) schemes proposed by Stefanov et al. in NDSS'14 [30], by Garg et al. [31], and by Bost et al. [32] can privately perform dynamic additions and deletions without leaking the search and access patterns, but they need multiple rounds to interact between the client and the server per query. The best round-efficient ORAM scheme in [31] still requires two rounds to perform a query with a big communication overhead. The current ORAM schemes are summarized by Chang et al. in work VLDB'16 [33]. They don't meet the requirement of conjunctive queries, because of extremely large communication overhead. To thwart inference attacks, Islam et al. proposed a mitigation scheme in [34] to partially hide the access pattern.

The CryptDB system proposed by Popa et al. [35] makes use of property-preserving encryption schemes such as deterministic encryption and order-preserving encryption [36] to support keyword and numeric range conjunctive queries. However, property-preserving encryption schemes leak too much statistical information to the cloud and have been proved to be insecure in CCS'15 [37].

Other approaches for encrypted search include secure multiparty computation [38,39] and fully- homomorphic encryption by Dijk et al. [40].

3 Static VBTree construction

To search over an encrypted data table with conjunctive queries, the data owner and the user mainly use following two steps: Setup and Search. First, the owner encodes all multi-dimensional data as keywords, which are organized into a tree data structure as an index. The index is submitted to the cloud together with the encrypted documents. Second, the data user creates a search trapdoor and sends it to the cloud. The cloud issues a search query on the index of the data table and sends back all matched results to the user.

3.1 Indexing element encoding

To support indexing multidimensional data records, we consider all types of index elements as keywords. The multidimensional data records are encoded as one-dimensional data by concatenating the keyword with an attribute string.

Given a keyword w and a corresponding attribute 'attr', the keyword w is encoded into a string 'attr:w'. This can avoid duplicated keywords appearing in one record. The attribute name can be viewed as the column name of the data table or other meta strings. For example, we encode a date '2018-1-1' into a keyword 'D:2018-01-01', where 'D' is the attribute name of the date. Therefore, a multidimensional data record is converted into an unstructured data file consisting of prefix string keywords.

3.2 VBTree construction

Our goal is to index all unstructured data files to create an index for efficient conjunctive queries. The indexing keywords can be put into tree-based data structures, such as KRB [6] and IBTree [9], for fast query processing. However, they expose all tree branches to the cloud. The major difficulty in updating a branch-leaked tree is that the cloud should dynamically manage the tree in every node update while it is prohibited from learning any information about any tree nodes (e.g., the number of real keywords in a tree node). To deal with this paradox, we now propose a novel tree data structure called virtual binary tree (VBTree). Here, we present the static VBTree first, as the dynamic VBTree is also based on the static one. We will discuss the branch-leaked trees in the next section.

Before doing so, we introduce three concepts relating to this data structure. We will make use of them throughout this work.

Full Binary Tree A full binary tree is a binary tree with $2^L - 1$ tree nodes and 2^{L-1} leaves, where L (root $L = 1$) is the height of the tree.

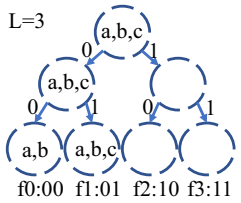
Path(v) Given a non-terminal tree node, a tree branch to its left child is encoded as a character '0,' and a tree branch to its right child is encoded as a character '1.' Given a tree node v, $Path(v)$ denotes a string concatenated with all tree branches beginning from the tree root to the current node v. In the special case, $Path(v)$ of the root of the full binary tree is an empty string.

Nodes(i) Let $i \in [0, 2^{L-1} - 1]$, and $leaf_i$ denote the ith leaf in the tree. $Nodes(i)$ denotes a set of all traversed nodes beginning from the root to $leaf_i$.

Definition 3.1 (*VBTree*) Let F be a keyed pseudo-random function and H_1 be a random oracle. A virtual binary tree (VBTree) is an encrypted full binary tree stored in a hash table such that:

1. Each tree node contains zero or more different encrypted keywords.
2. The hash table only stores encrypted keywords, without storing any tree nodes and tree branches.

Fig. 2 VBTree example



3. To index a keyword w of a file identifier i ($i \in [0, 2^{L-1} - 1]$), the keyword is inserted into each tree node of $Nodes(i)$. The inserted items are a set of encrypted key-value pairs $\{(H_1(Path(v)||F_K(w)), 1)\}_{v \in Nodes(i)}$, where K is a set of keys of the data owner.

Let $WD(w, i) = \{(H_1(Path(v)||F_K(w)), 1)\}_{v \in Nodes(i)}$.

Figure 2 shows an example of a virtual binary tree of height $L = 3$. There are four leaves in the tree. $Path(leaf_1)$ denotes string '01.' $Nodes(1)$ denotes a set of tree nodes $\{root, node_0, node_{01}\}$. The values of keyword 'c' of file identifier '1' in the hash table are a set of key-value pairs, i.e., $WD(\text{'}c\text{'}, 1) = \{(H_1(Path(v)||F_K(\text{'}c\text{'})), 1)\}_{v \in Nodes(1)} = \{(H_1(''||F_K(\text{'}c\text{'})), 1), (H_1(\text{'0'}||F_K(\text{'}c\text{'})), 1), (H_1('01'||F_K(\text{'}c\text{'})), 1)\}$. All key-value pairs in this VBTree are $WD(\text{'}a\text{'}, 0) \bigcup WD(\text{'}b\text{'}, 0) \bigcup WD(\text{'}a\text{'}, 1) \bigcup WD(\text{'}b\text{'}, 1) \bigcup WD(\text{'}c\text{'}, 1)$. The tree branches and tree nodes are implicit in these encrypted key-value pairs. The cloud doesn't know where the tree nodes are.

Given a tree node v, $W(v)$ denotes a set of keywords of v. For any non-terminal node v, if v_l is its left child node and v_r is its right child, we can prove that the keyword set of v are $W(v) = W(v_l) \cup W(v_r)$.

A virtual file is a real file or an empty file. Each $leaf_i$ in a VBTree is mapped into a virtual file with a unique identifier i. The binary string $Path(leaf_i)$ can be converted into the number i as the unique file identifier. If the VBTree has stored n real files from $leaf_0$ to $leaf_{n-1}$, the root node will contain m distinct keywords, which is the number of keywords that can be queried, and all of the leaves will contain N keywords, where N is the number of appearances of keywords in all files (a file is considered as a collection of distinct keywords). The value N is also called the database size or the number of (w, id) pairs in Sophos of [7] and other SSE works.

In Fig. 2, there are $n = 2$ inserted files but there are four virtual files in total, as the virtual files f2 and f3 are empty files. Each leaf is mapped to a virtual file. The file f0 of path '00' contains keywords $\{a, b\}$. The file f1 of path '01' contains keywords $\{a, b, c\}$, and then their parent tree node of path '0' contains keywords $\{a, b\} \cup \{a, b, c\}$. In the demo tree, we have $N = 5$ and $m = 3$. The dotted lines denote that the cloud learns nothing about how many items are in each tree node and even learns nothing where the tree nodes are.

The static construction of a VBTree can use the following steps in detail. Given a keyword w and a numeric file identifier i (the file i contains the keyword w), the data

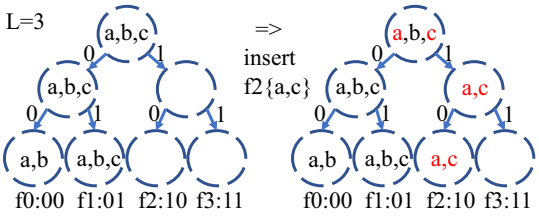


Fig. 3 VBTree local construction example

owner inserts encrypted items from the root to the $leaf_i$. As the file identifier i is unique, it can be converted into a binary path. Then, the data owner inserts L items into the VBTree along the path. Suppose all traversed nodes are $\{v_1, v_2, \ldots, v_L\}$, then the inserted items are $WD(w, i) = \{(H_1(Path(v_1)||F_K(w)), 1), (H_1(Path(v_2)||F_K(w)), 1), \ldots, (H_1(Path(v_L)||F_K(w)), 1)\}$. The insertion of a keyword w of a file with identifier i can be denoted as $\mathcal{T} \leftarrow \mathcal{T} \bigcup WD(w, i)$, where $\mathcal{T}$ is the hash table of VBTree. The insertion of an existing value is ignored. The data owner repeats this process until all of the keywords as well as all of the files are inserted into the VBTree. Next, the data owner inserts some random values into the VBTree and sends the tree to the cloud as an index. Finally, the data owner tells the cloud how many files are stored on the cloud. To create a highly balanced tree, n files are sequentially inserted into the VBTree from the file identifier '0' to the file identifier '$n-1$'. A VBTree of height L can be used to accommodate 2^{L-1} files at most.

Figure 3 shows how to construct a VBTree on the local client. To insert a file f2 with two keywords $\{a, c\}$, the data owner converts f2 to path '10' and inserts the encrypted items from the root to the leaf of path '10.' The inserted items are $WD(\text{'}a\text{'}, 2) \bigcup WD(\text{'}c\text{'}, 2)$.

Note that the cloud, the data owner and the data users don't need additional space to store tree branches and tree nodes, and the VBTree only exists in a logical view. Without search queries of the user, the cloud learns nothing about the VBTree even the total number of files, only when the owner tells the cloud how many files are stored in the tree. For the sake of easily maintaining the tree and the encrypted files, we enable the cloud to learn the total number of files.

We now analyze the index size and the construction time of VBTree. In the best case, if all of the files are same (except for the file identifiers), the index size is $O(2N)$. In the worst case, if all of the keywords in all documents are different, the index size is $O(LN)$. Therefore, the construction time of VBTree is $O(LN)$, and the index size of VBTree is $O(\beta N) \approx O(N \log n)$, where $\beta \in [2, L]$.

3.3 Trapdoor computation

To search over the encrypted index, the data user creates a trapdoor and sends it to the cloud. The trapdoor is a set of encrypted conditions constructed from the query. Given a u-dimensional conjunctive keyword query $q = w_1 \wedge w_2 \wedge \cdots \wedge w_u$, the trapdoor is $T(q) = \{F_K(w_1), F_K(w_2), \ldots, F_K(w_u)\}$. For example, to search for a file 'paper.pdf' created on 2018-1-1, the trapdoor is $\{F_K(\text{'}D : 2018\text{-}01\text{-}01\text{'}), F_K(\text{'}NAME : paper.pdf\text{'})\}$.

3.4 Query processing

There are two types of searching algorithms on the VBTree, top-down search and left-right search. The time complexity of the first is sub-linear in average cases, and the second is linear.

Top-down search Given a search trapdoor $T(q)$, the cloud can top-down search over the tree within the following steps. First, the cloud checks whether the root of $path = ''$ contains the keywords or not by evaluating $H_1(''||F_K(w_1)), H_1(''||F_K(w_2)), \ldots$. If the hash table contains these entries, the cloud needs to recursively check whether the left child node of $path =$ '0' and the right child node of $path =$ '1' contain the queried keywords or not. If the hash table doesn't contain these entries, the searches of this node and all of its children are completed. Second, the cloud recursively performs these operations and recursively sets $path \leftarrow path||\text{'0'}$ or $path \leftarrow path||\text{'1'}$. If the length of the path is $L-1$, it denotes that a search reaches a leaf node and the current search is completed. Finally, the searches output all of the binary paths of length $L-1$. The cloud converts these binary string paths to numbers as file identifiers. If a node doesn't contain one of the queried keywords, all of its children nodes are ignored (in step 3 of $\mathcal{VSA}$-1). Therefore, most of irrelevant nodes are quickly discarded in the query.

Algorithm 1 $\mathcal{VB}$Tree.Search $\mathcal{A}$lgorithm ($\mathcal{VSA}$-1)

// Cloud:
$Search(\mathcal{T}; \boldsymbol{p}\text{ath}, \{F_K(w_1), F_K(w_2), \cdots, F_K(w_u)\})$

1: **for** i=1 to u **do**
2: Let $b_i \leftarrow \mathcal{T}$.ContainsKey($H_1(\boldsymbol{path}||F_K(w_i))$)
3: If b_i=false, then return 'not found'
4: **end for**
5: If the length of $\boldsymbol{p}$ath is $L-1$, then convert it to a number as a file identifier and return one result.
6: Invoke, Search($\mathcal{T}$, $\boldsymbol{path}||\text{'0'}, \{F_K(w_1), \cdots, F_K(w_u)\}$)
7: Invoke, Search($\mathcal{T}$, $\boldsymbol{path}||\text{'1'}, \{F_K(w_1), \cdots, F_K(w_u)\}$)

Algorithm $\mathcal{VSA}$-1 gives a recursive search approach on the VBTree. It is assumed that all encrypted keywords are stored in the hash table $\mathcal{T}$. Given a trapdoor $T(q)$ and a path, the algorithm recursively searches the VBTree and outputs the file identifiers $\mathcal{ID}s$ that match the query. $\mathcal{VSA}$-1 can search from any tree node by setting the path to its corre-

sponding value. 'ContainsKey' function is to check whether a key exists in a hash table.

For a single-keyword query, if the query result size is r, the query time complexity is $O(rL)$ in the average case In fact, the cloud doesn't need to search from the root. If the cloud searches from the sub-root, whose path value is a string of $L - \lceil \log_2 n \rceil - 1$ zeros, the query time is $O(r \log_2 n)$. For a u-dimensional keyword conjunctive query q, the query complexity can be written as $O(|q| \min_{w \in q}\{|DB(w)|\} \log_2 n)$. In reality, $\min_{w \in q}\{|DB(w)|\} \ll n$, as n is typically large. Therefore, our query time complexity is sub-linear in most cases.

Left-right search The VBTree can also be searched sequentially from $leaf_0$ to $leaf_{n-1}$. Algorithm $\mathcal{VSA}$-2 gives the pseudocode of a linear search over the VBTree. The linear left–right search complexity is $O(n)$. Although the linear search is inefficient, the left–right search is a reserved algorithm for the cloud.

Algorithm 2 $\mathcal{VB}$Tree.LinearSearch $\mathcal{A}$lgorithm ($\mathcal{VSA}$-2)

// Cloud:
$LinearSearch(\mathcal{T}; \{F_K(w_1), F_K(w_2), \cdots, F_K(w_u)\})$

1: **for** j=0 to n-1 **do**
2: **for** i=1 to u **do**
3: Let $b_i \leftarrow \mathcal{T}$.ContainsKey($H_1(Path(leaf_j)||F_K(w_i))$)
4: **end for**
5: If u conditions are all satisfied, output a result
6: **end for**

4 Dynamic VBTree construction

In the above section, we have proposed the static VBTree scheme. Next, our goal is to make VBTree support dynamic updates with forward privacy. We now propose a version-control-repository-based solution for VBTree. With the version control repository, an index can be updated with forward privacy, and the VBTree index can be instantiated online with strong security guarantee. At last, we discuss the challenges of branch-leaked trees.

4.1 Version control repository

Recall the notation of forward privacy, the major drawback of a non-forward-private dynamic SSE scheme is that their update procedure leaks information that is correlated with their previously queried keywords. This drawback will lead to a breakage of the SSE scheme made by an adversary, who might adaptively inject keywords into the update files, as is discussed in [4]. To avoid newly added elements intersecting with historical queries, we label keywords and updates with versions. Given a keyword w, its vth version is denoted as $w||v$, and its vth version trapdoor is denoted by $F_K(w||v)$. Historical search queries and updates are managed by a storage mechanism, which is called version control repository.

The key idea of our approach is recording whether each keyword has been leaked to the cloud or not at the client side. If a keyword has been leaked to the cloud (queried by the data users), the data owner creates a new version of this keyword for the next update procedure, as the owner cannot use the leaked keyword to update the index.

Definition 4.1 (*Version control repository*) A version control repository for a dynamic SSE scheme includes a local repository (LR) and a cloud repository (CR). LR is a client-side hash table. From LR, the client can learn whether a keyword has been leaked to the cloud or not. The client can also get or set the latest version of keywords. CR is a cloud-side hash table. From CR, given a new version trapdoor, the cloud can get all the corresponding historical trapdoors.

On the client side, $LR(w)$ denotes the usage information of a keyword w. The client includes the data owner and the data users. For simplicity, the owner and the users share the same LR. (In the next section, we will discuss how to optimize LR). For each keyword w, $LR(w)$ has three properties, (b, V_l, n_l). The number $LR(w).V_l$ denotes the latest version of the keyword w. The bit $LR(w).b$ denotes that whether the latest version of keyword w has been queried by the data users or not. The initial state of $LR(w).b$ is false, which means this keyword is non-leaked. If a search query of the keyword w has been performed, $LR(w).b$ is set to true, which means this keyword has been leaked to the cloud according to the search pattern. We assume the users always use the latest version of the keyword to search. If $LR(w).b$ is true before the update operation, the owner should update the version of w by setting $LR(w).V_l \leftarrow LR(w).V_l + 1$ to avoid the newly added keywords intersecting with the historical queries. If a new version of w has been updated, $LR(w).b$ is set to false again, which means this keyword is non-leaked again. The number $LR(w).n_l$ denotes the last file identifier that matches the keyword w.

On the cloud-side, CR can be considered as multiple encrypted singly linked lists, in which a trapdoor of the vth version links to the $(v-1)$th version and other subversions. The historical version trapdoors of keywords are stored in CR. Let H_2 and H_3 be different random oracles, the encrypted items in CR are in the key-value form of $(H_2(F_K(w||v)), H_3(F_K(w||v)) \oplus F_K(w||(v-1)))$.

Given a search trapdoor $F_K(w||v)$ for a keyword w of a version v, the cloud can get $F_K(w||(v-1)) \leftarrow CR[H_2(F_K(w||(v)))] \oplus H_3(F_K(w||(v)))$ and other subversions by $F_K(w||(v))$. The cloud now can search for all results using these trapdoors. However, the cloud cannot deduce the $(v+1)$th version of the trapdoor from the vth version. This

is the key design consideration of a forward-private dynamic SSE scheme.

Given a single-keyword trapdoor x and CR, we use $Hist(x)$ to denote a set of historical trapdoors corresponding to x from the linked list in CR.

4.2 Dynamically update a keyword

There are three kinds of update queries on a forward-private VBTree: additions, modifications and deletions. Suppose the cloud has stored n files varying from identifier 0 to identifier $n-1$.

Additions and modifications To dynamically add a keyword w of a file with identifier n, the data owner uses following steps. First, the data owner checks LR. According to $LR(w).b$, the data owner updates the version or doesn't update it. Second, the data owner creates a set of update items T_{up_1} to update the VBTree on the cloud. T_{up_1} is created as follows. As the last file identifier that matches the keyword w is stored in $LR(w).n_l$, the set of incremental update items can be written in $T_{up_1} \leftarrow WD(w,n) - WD(w, LR(w).n_l)$. If the bit $LR(w).b$ is true, the data owner should create another part of update trapdoor T_{up_2} to update CR. With the vth version and the $(v-1)$th version of w, T_{up_2} contains the key-value pair $T_{up_2} \leftarrow \{(H_2(F_K(w||v)), H_3(F_K(w||v)) \oplus F_K(w||(v-1)))\}$. Then, if a new version has been created, the bit $LR(w).b$ is set to false. $LR(w).n_l$ is set to n, which represents the last file identifier that matches w has been changed to n. The data owner pads the size of T_{up_1} to L with random values, which are indistinguishable from the entries in T_{up_1}. $T_{up} = (T_{up_1}, T_{up_2})$ are sent to the cloud for updates. Third, the cloud computes $\mathcal{T} \leftarrow \mathcal{T} \bigcup T_{up_1}$ and $CR \leftarrow CR \bigcup T_{up_2}$ to update the tree and the cloud repository respectively.

To dynamically modify a keyword w of an existing file at $leaf_i$ ($i \in [0, n-1]$), the update approach is similar to the addition algorithm. The main difference is that the modifications always change the versions of corresponding keywords. Since LR doesn't store $DB(w)$, the update requires changing the version of w regardless $LR(w).b$. Thus, the modification operations will make CR larger.

Search queries with CR With CR, the search queries become slightly complex in comparison with the operation on a static VBTree. Given a u-dimensional conjunctive query $T(q) = \{F_K(w_1), F_K(w_2), \ldots, F_K(w_u)\}$, the search query turns into two parts. One is the search on CR, and the other is on the VBTree. The cloud uses the vth version trapdoor $F_K(w||v)$ to get all of the historical trapdoors of w from the linked list in CR. Let e_{w_i} denote the number of versions of w_i in CR, i.e., $e_{w_i} = |Hist(F_K(w_i))|$. With all historical trapdoors, the conjunctive query turns to be a conjunctive normal form (CNF) $\triangle_1 \wedge \triangle_2 \wedge \cdots \wedge \triangle_u$, where each $\triangle_i = F_K(w_i||v) \vee F_K(w_i||(v-1)) \vee \cdots \vee F_K(w_i||(v-e_{w_i}))$. We can slightly modify the algorithm $\mathcal{VSA}$-1 into $\mathcal{VSA}$-3 to suit for the CNF query. Note that this algorithm is only a sub-algorithm.

Algorithm $\mathcal{VSA}$-3 gives the pseudocode to search over the VBTree with the CNF query. For simplicity, we write $\triangle_i = \{x_{i1}, x_{i2}, \ldots, x_{ip(i)}\}$. Let $|\triangle_1| = p(1), |\triangle_2| = p(2), \ldots, |\triangle_i| = p(i)$, and $T(q) = \{\triangle_1, \triangle_2, \ldots, \triangle_u\} = \{\{x_{11}, x_{12}\ldots\}, \{x_{21}, x_{22}\ldots\}, \ldots, \{x_{u1}, x_{u2}\ldots\}\}$. If a sub-condition x_{ij} is not in $\mathcal{T}$, this means that all children nodes of the current node don't contain x_{ij}. Thus, for fast query processing, x_{ij} should be removed from $\triangle_i$ when x_{ij} is not matched. This can significantly reduce the amount of pseudo-random computations in the search process.

To make the tree and CR work more efficiently, the cloud create a new thread to reorganize the tree and CR after a search query. If the cloud learns that CR contains many versions of w, the cloud performs an organization operation within three steps. First, the cloud removes all of the versions of w from CR to reduce the storage. Second, the cloud removes all old version items in the tree and only preserves the new version items. Third, the cloud inserts the indexing elements of the current version to replace the old versions with the index construction algorithm. Note that the leakage of this reorganization is included in the search pattern. The leakage is as little as most of the SSE schemes. The cloud can do this work only when the server is idle.

Algorithm $\mathcal{VSA}$-4 gives the steps to search over a forward-private VBTree with a conjunctive query. $\mathcal{T}$ means the hash table of VBTree. $\mathcal{T}$ and CR all represent a set of items. $DB(t)$ denotes the result set of the single-keyword query by the trapdoor t. The data user first creates a trapdoor with LR for a conjunctive query q. The data user sends it to the cloud. Second, the cloud gets all historical trapdoors from CR by the current trapdoor. Third, the cloud searches the VBTree using a CNF query. Fourth, the cloud reorganizes the VBTree for later fast query processing.

Algorithm 3 $\mathcal{VB}$Tree.Search $\mathcal{A}$lgorithm ($\mathcal{VSA}$-3)

// Cloud:
$Search(\mathcal{T}; \boldsymbol{p}\text{ath}, \{\triangle_1, \triangle_2, \cdots, \triangle_u\})$
1: **for** i=1 to u **do**
2: **for** j=1 to $|\triangle_i|$ **do**
3: Let $b_i \leftarrow \mathcal{T}$.ContainsKey($H_1(\boldsymbol{p}ath||x_{ij})$)
4: **if** b_i=false **then**
5: Remove x_{ij} from $\triangle_i$, i.e., $\triangle_i \leftarrow \triangle_i - \{x_{ij}\}$
6: **else**
7: break
8: **end if**
9: **end for**
10: If $|\triangle_i| = 0$, return 'not found.'
11: **end for**
12: If the length of $\boldsymbol{p}$ath is $L-1$, then convert it to a number as a file identifier and return one result.
13: Invoke: Search($\mathcal{T}$, $\boldsymbol{p}ath||$'0', $\{\triangle_1, \triangle_2, \cdots, \triangle_u\}$)
14: Invoke: Search($\mathcal{T}$, $\boldsymbol{p}ath||$'1', $\{\triangle_1, \triangle_2, \cdots, \triangle_u\}$)

Algorithm 4 $\mathcal{VB}$Tree.Search $\mathcal{A}$lgorithm ($\mathcal{VSA}$-4)

// Data user:
$Trapdoor(q; K, LR)$
1: For each keyword $w \in q$, set $LR(w).b \leftarrow true$.
2: Generate a trapdoor, $T(q) \leftarrow \bigcup_{w \in q} F_K(w||LR(w).V_l)$
3: Send $T(q)$ to the cloud.
// Cloud:
$Search(T(q); \mathcal{T}, CR)$
1: parse T(q) as $\{x_1, x_2, \cdots, x_u\}$
2: initialize u empty sets for a CNF query, i.e., $\{\triangle_1, \triangle_2, \cdots, \triangle_u\} \leftarrow \{\varnothing, \varnothing, \cdots, \varnothing\}$.
3: **for** $i = 1$ to u **do**
3: get all historical version trapdoors from the linked list of CR, i.e., $\triangle_i \leftarrow Hist(x_i)$.
3: build the query, i.e., $\triangle_i \leftarrow \{x_i\} \bigcup \triangle_i$
4: **end for**
5: set $path$ to a string of $L - \lceil \log_2 n \rceil - 1$ zeros.
6: invoke: $\mathcal{VSA}$-3.Search($\mathcal{T}$; $path$; $\{\triangle_1, \triangle_2, \cdots, \triangle_u\}$).
7: send all results to the data user.
8: create a new thread to run following codes.
9: **for** $i = 1$ to u **do**
10: **for** all $y \in \triangle_i$ **do**
11: ignore the lastest trapdoor, i.e., if $y = x_i$ continue;
12: remove a historical trapdoor from CR by a key, i.e., $CR \leftarrow CR - \{H_2(y)\}$.
13: get a result set from the tree by the subquery y, i.e., $\mathcal{IDs} \leftarrow DB(y)$.
14: **for** all $id \in \mathcal{IDs}$ **do**
15: remove old version items, $\mathcal{T} \leftarrow \mathcal{T} - \{(H_1(Path(v)||y), 1)\}_{v \in Nodes(id)}$
16: insert the latest items, $\mathcal{T} \leftarrow \mathcal{T} \bigcup \{(H_1(Path(v)||x_i), 1)\}_{v \in Nodes(id)}$
17: **end for**
18: **end for**
19: **end for**

We now analyze computational complexity and storage requirements of these algorithms. In the worst case, the conjunctive query time is $O((\sum_{w \in q}(e_w + 1)) \min_{w \in q}\{|DB(w)|\} \log_2 n)$. There are three states of e_w. In the index-initialized state, the e_w is zero, as the keyword w have not been updated. In the index-modified state, the e_w is the number of previous updates of w. In the index-reorganized state, the e_w is zero, as all historical versions of w are removed from CR. Thus, the conjunctive query complexity of a forward-private VBTree is sub-linear in the average cases. The index size of CR is $O(e_{up})$, where e_{up} is the number of version updates of CR.

Deletions To dynamically delete a keyword of a file from the index, two VBTrees are created. One VBTree $tree_A$ is for saving added keywords, and the other $tree_B$ is used to save deleted keywords. The search turns to be two parts. One is the search over $tree_A$, and the other is the search over $tree_B$. The final result is the difference of the two result sets. The encoding of the path is changed to the form of TreeName:BinaryPath. At the client side, for each keyword, the owner uses two variables n_{add} and n_{del} to represent the largest file identifiers inserted for two trees, respectively. Let $LR = \{LR(w).n_{add}, LR(w).n_{del}, LR(w).b, LR(w).V_l\}_{w \in W}$. For example, an item in VBTree-A is encrypted as

$H_1(\text{`}A\text{:0010'}||F_K(w))$, and an item in VBTree-B is encrypted as $H_1(\text{`}B\text{:0010'}||F_K(w))$. Note that these two trees are stored in the same hash table, and the trees exist only in a logical view. The leakage of the deletion is as little as the addition operations. These two operations are indistinguishable from each other. The cloud can reorganize this index after a search query also.

The time complexity of all updates is $O(L)$, which is in terms of per updated keyword/document pair. These operations include additions (op=‘add’), modifications (op=‘add’), and deletions (op=‘del’).

4.3 Discussion of branch-leaked trees

A branch-leaked tree is an encrypted tree outsourced to the cloud with all branch information directly exposed such that: (1) Each tree node contains one or more encrypted blocks. (2) Following the tree branches, the cloud can learn the tree nodes that contain encrypted blocks without user requests. KRB [6], PBTree [13], IBTree [9], KBB [14] and etc. are all branch-leaked trees. Although the branch leakage is negligible to the adversary for a static SSE scheme, there are still two key challenges for an SSE scheme to exploit a branch-leaked tree.

First, it is hard for these trees to achieve scalable index size. As discussed in the first section, to avoid leaking the tree node size to the cloud, the schemes use padding, but padding makes KRB and KBB unscalable. To reduce the index size, IBTree exploits a compression algorithm, which makes it difficult to update its index dynamically. PBTree pads all tree nodes of each level to the same size (it is a numeric tree which differs from others), and this also costs much space.

Second, it is hard for branch-leaked trees to achieve sub-linear update efficiency. If the update operation has been done naively, for example, directly inserting items into the tree beginning from the root to a leaf, the update leaks all updated position information of all updated tree nodes. Since an updated non-terminal node contains information that comes from both its left sub-tree and right sub-tree, the naive updates might leak a part of distribution of the keywords. Take the full version of PBTree in [13] for example; although its update complexity is $O(\log n)$, the insertion operation leaks the distribution of an existing prefix that is same as the inserted prefix (a number is converted into a set of prefix strings), since different numbers perhaps share the same prefixes. This is disastrous for an SSE scheme. To avoid this leakage and others (e.g., the number of updated keywords in each tree node), KRB rebuilds the whole tree nodes involved in every node update, but the costly re-encryption makes its insertion complexity be $O(m \log_2 n)$. IBTree faces the same difficulty.

KBB cannot be arbitrarily updated by a (w, id) pair, since any modifications will lead to rebuilding the whole tree node. All update efficiency and other comparisons are shown in Table 1 of the first section.

5 Optimizations and extensions

5.1 Traversal width optimization

Recall the definition of VBTree, for a non-terminal node v, all keywords of the node v come from the left child v_l and the right child v_r, such that $W(v) = W(v_l) \bigcup W(v_r)$, where $W(v)$ denotes a set of all keywords in v. This union operation can be also considered as a file set partition operation. The node v is partitioned into two child nodes v_l and v_r so that, $|f(v_l)| = |f(v_r)|$, where the function $f(v)$ denotes the set of files (including empty files) in the node v. This partition is critical for the performance of query processing on VBTree because searching for the common keywords in both children will lead to the traversal of both subtrees. Intuitively, if all results cluster together, the search process will quickly discard almost half of the irrelevant results in the root; however, if the distribution of all results is uniform, the search process still needs to traverse both subtrees after accessing the root. Thus, in partitioning $f(v)$ to $f(v_l)$ and $f(v_r)$, we want to minimize the $|W(v_l) \bigcap W(v_r)|$. We call this problem a Data File Partition. We next formally define this problem and its decisional version.

Definition 5.1 The Data File Partition Problem (DFPP) is described as follows. Given a set of files $\mathcal{F}$ of even size, we want to partition $\mathcal{F}$ into a file set $\mathcal{F}_1$ and a file set $\mathcal{F}_2$, such that $|\mathcal{F}_1| = |\mathcal{F}_2|$ and $|W(\mathcal{F}_1) \bigcap W(\mathcal{F}_2)|$ is the minimized value.

Definition 5.2 The Decisional version of the Data File Partition Problem (DDFPP) is described as follows. Given a set of files $\mathcal{F}$ of even size and a positive number k, we want to partition $\mathcal{F}$ into a file set $\mathcal{F}_1$ and a file set $\mathcal{F}_2$, such that $|\mathcal{F}_1| = |\mathcal{F}_2|$ and $|W(\mathcal{F}_1) \bigcap W(\mathcal{F}_2)| < k$.

Theorem 5.1 *(1) DFPP is NP-hard. (2) DDFPP is NP-complete.*

Proof DFPP is a general version of Equal Size Prefix Family Partition defined in [9,11]. DFPP slightly extends their definitions to support unstructured data files. The proof is omitted here, but 'Appendix' gives another proof. □

Next, we present our approximation algorithm to handle the data file partition problem. If we partition all files recursively, the result is an optimized order of all files. Our goal is to partition the data files to get the optimized file insertion order, which will contribute to faster query processing and smaller index size. In [9], the authors also introduce an approximation algorithm to deal with the partition problem, but their algorithm is not suitable for a large dataset. We now propose our algorithm to get the optimized order of file insertions.

Given n data files $\mathcal{F} = \{d_1, d_2, \ldots, d_n\}$, we use three steps to get an optimized file order. First, we construct a graph $G = (V, E)$ for the data files, where V is the vertex set and E is the edge set. Each vertex v_i in V has a file identifier list property $v_i.list$ to represent a list of files and a hash table $v_i.t$ to contain all of the corresponding keywords. For each vertex v_i, if the construction is in an initial stage, $v_i.list$ contains only one file identifier of $\{d_i\}$, and the hash table $v_i.t$ is filled with all distinct keywords of the file list. Between each two vertexes v_i and v_j, we construct an edge with the weight property of $|W(v_i) \bigcap W(v_j)|$. Then, the graph G is constructed. Second, we sort all of the edges by the weight property in descending order. For each of the edges (v_i, v_j), we merge the corresponding vertexes v_i and v_j into a new vertex v, i.e., $v.list \leftarrow v_i.list \cup v_j.list$, and $v.t \leftarrow v_i.t \cup v_j.t$. The new vertex v is a member of the new graph G.' We remove all edges connected with v_i and v_j from E. Repeat this process until all of the edges are removed from E. Third, go to step 1, reconstruct graph G' and merge the files again. Repeat this process until the graph G contains only one vertex, whose file list contains our optimized file order.

The computational complexity of the algorithm above is $O(n^2 \log_2^2 n)$. Building the graph G, it takes $O(n^2)$ time. Sorting all of the edges in G, it takes $O(n^2 \log_2 n)$ time. The algorithm also needs to construct the graph repeatedly. This is problematic when processing a large dataset. To process a large dataset using the algorithm above, we use another algorithm preprocessing the data files. The key idea of the algorithm is sorting all of the files first. Next, we split the sorted files into small parts. After splitting the files, we can use the graph-based algorithm to optimize each part of the files.

To index a big unstructured dataset, we use a TF-IDF-based algorithm to sort the data files first. Given a keyword w and a file d_i, $TF(w, d_i)$ means the term frequency, i.e., the number of appearances of w in this file. In this scheme, $TF(w, d_i)$ is set to $\frac{1}{m_f}$ here, where m_f is the number of distinct keywords in the file. Given a set of n files $\mathcal{F}$ each with distinct keywords, $IDF(w, \mathcal{F})$ means inverse document frequency. Here, it is set to $\log_2 \frac{n}{|DB(w)|}$. We can compute a score of the file d_i, using $score(d_i) = \sum_{w \in d_i} TF(w, d_i) IDF(w, \mathcal{F})$. The optimizing algorithm of the big dataset becomes two steps. First, we use the TF-IDF-based algorithm to sort the files and to split the sorted files into $\lceil \frac{n}{P} \rceil$ parts, where P is the size of each part. Second, we use the graph-based algorithm to optimize each of the subset files. This algorithm outputs the optimized insertion order of

the files. Thus, we can use the indexing algorithm to initialize each part of the VBTree online.

The time complexity of the graph-based algorithm is reduced to $O(\lceil \frac{n}{P} \rceil P^2 \log_2^2 P)$. In our experiments, we set $P = 1024$ or 8192, which is mainly related to the amount of all available memory of the users.

5.2 Reducing client-side storage

We remove redundant information of LR to reduce the client-side storage. Let W denote a set of all keywords that can be queried, LR_{usr} denote the storage of the data users, and LR_{onr} denote the storage of the data owner. We have $LR_{usr} = \{LR(w).b, LR(w).V_l\}_{w \in W}$ and $LR_{onr} = \{LR(w).n_l, LR(w).b, LR(w).V_l\}_{w \in W}$.
$LR(w).V_l$ and $LR(w).b$ are mainly used in search queries by the users, but $LR(w).n_l$ is only used by the owner in updates (let's consider the addition operation first).

There are three states of the data user's storage requirement. In the index-initialized state, all default versions of keywords $LR(w).V_l$ are zero, and all keyword flags $LR(w).b$ are zero. This information can be removed. Thus, the storage of the user is zero at this time. After many queries, the user requires a storage space to record all keywords leaked. Since search queries of the data user only affect $LR(w).b$, the client-side storage of the data user is $O(m_q)$, where m_q is the number of queried keywords. After many updates by the data owner, suppose the number of updated keywords is denoted by m_u, and the maximal version number of all keywords is v_u, the client-side storage of the data user is increased to $O(m_q + m_u \log_2 v_u)$, as the data user requires additional space to store the new versions. In reality, compared to m_u, m_q and v_u are typically small, we have $O(m_q + m_u \log_2 v_u) \approx O(m_u)$. In the worst case, if all keywords are queried and updated, the storage requirement of the data user is $O(m)$ (where m is the number of all keywords).

To update a keyword w, the data owner uses three steps. First, the owner gathers all leaked keywords from all of the data users. Second, the owner updates this keyword by trapdoors $T_{up_1}(w)$ and $T_{up_2}(w)$. Third, the owner notifies all of the users of new version keywords. The storage requirement of the data owner is $O(m(\log_2 n + \log_2 v_u))$ all the time. In reality, since $n \gg v_u$, then the storage requirement is $O(m \log_2 n)$.

5.3 Removing user storage

The client-side storage of the data users LR_{usr} can be removed by updating all keywords to the same version. Running after a period of time, LR_{usr} turns larger. In the worst case, the size of LR_{usr} is increased to $O(m)$. However, LR_{usr} can be dynamically removed by the data owner using algorithm $\mathcal{VUA}$-1.

Algorithm 5 $\mathcal{V}$CR.$\mathcal{U}$pdate $\mathcal{A}$lgorithm ($\mathcal{VUA}$-1)

// Data owner:
$Trapdoor(K, LR_{onr})$
1: Get the maximal version from LR_{onr}, i.e., $v_u \leftarrow max_{w \in W}\{LR_{onr}(w).V_l\}$.
2: Gather LR_{usr} of version v_u of all users.
3: For each keyword w in W, skip all keywords of version v_u
   1. Let $v_0 \leftarrow LR(w).V_l$
   2. Prepare a key-value token o for update, $o \leftarrow (H_2(F_K(w||v_u)), H_3(F_K(w||v_u)) \oplus F_K(w||v_0))$
   3. Generate a trapdoor, and add all o, $T_{up_2} \leftarrow T_{up_2} \bigcup \{o\}$ Set to the new version, i.e., $LR_{onr}(w).V_l \leftarrow v_u$.
   4. Mark w with non-leaked, i.e., $LR_{onr}(w).b \leftarrow false$.
4: Send T_{up_2} to the cloud.
5: Send v_u to the users.

// Cloud:
$Update(T_{up_2}, CR)$
1: Update CR, i.e., $CR \leftarrow CR \bigcup T_{up_2}$.

// Data users:
$Refresh(LR_{usr})$
1: Remove the storage, i.e., $LR_{usr} \leftarrow \phi$.

As shown in algorithm $\mathcal{VUA}$-1, the procedure to remove LR_{usr} of all users includes three sub-algorithms, 'Trapdoor', 'Update', and 'Refresh'. The owner updates all keywords to the same version of v_u by the sub-algorithm $Trapdoor(K, LR_{onr})$. This algorithm consists of four steps: (1) Retrieve the maximal version v_u of all keywords; (2) gather LR_{usr} of version v_u of all users; (3) generate an update trapdoor T_{up_2} for all keywords of versions less than v_u; let W' denote a set of keywords of versions less than v_u, then $T_{up_2} \leftarrow \{(H_2(F_K(w||v_u)), H_3(F_K(w||v_u)) \oplus F_K(w||(LR(w).V_l)))\}_{w \in W'}$. 4) Send the update trapdoor T_{up_2} to the cloud and send v_u to the users. If the owner updates all keywords to the $(v_u + 1)$th version instead of the v_uth version, step 2 is not required, since all keywords of the $(v_u + 1)$th version are non-leaked. The cloud updates CR by the algorithm $Update(T_{up_2}, CR)$. The users empty their LR_{usr} by the algorithm $Refresh(LR_{usr})$. Now, given a keyword w, the users can use the trapdoor $F_K(w||v_u)$ to search over the VBTree with zero local storage (except for the value v_u). $\mathcal{VUA}$-1 removes all user storages at the cost of slight reduction of search performance. However, this search performance loss is small because the CNF query of $\mathcal{VSA}$-3 is very efficient in processing empty conditions (which are sub-trapdoors with no matched results).

In the special case, if the data owner updates all keywords to the (v_u+1)th version in every index update, the user's storage LR_{usr} can be permanently deleted. Since all keywords of the $(v_u + 1)$th version are not leaked, the data users don't need to store anything forever except for the variable v_u.

In summary, updating an individual keyword or a set of files makes LR_{usr} larger, but updating all keywords to the same version will make LR_{usr} empty.

6 Security analysis

In the above sections, we have proposed a static VBTree scheme and a dynamic one. The static one can be considered as a part of the dynamic one. We now mainly analyze the leakage of the dynamic one and prove that it is IND-CKA2 secure against an adaptive adversary.

6.1 Leakage function

To formally describe the leakage of the VBTree-based schemes, we first introduce two notations: search pattern and history of a keyword. Let Q denote a two-dimensional array that stores all historical conjunctive queries issued in order of arrival, for example, $Q[0][0] = `a'$ and $Q[0][1] = `b'$ for the 0th conjunctive query $a \wedge b$. The search pattern of a keyword w is defined as follows: $SP(w) = \{(i, j) : for\ all\ Q[i][j] = w\}$. Search pattern is the repetition of trapdoors sent by the client to the cloud. The history of a keyword w is defined as $HIST(w) = (DB_0(w), UpHist(w))$, whose first element is $DB_0(w)$, the set of file identifiers matching w at the setup stage, and whose second element is the set $UpHist(w)$, called update history. An update history of the keyword w denotes all the modifications made by $DB(w)$ over the time, i.e.,

$$UpHist(w) = \{(t_0, op_0, w, id_0), (t_1, op_1, w, id_1), \ldots\},$$

where t_i is a time stamp, op_i is an operation name 'add' or 'del,' and id_i is a file identifier that matches w. The additions and modifications are all considered as $op = `add'$, and deletion operations are $op = `del'$.

We define the leakage function of the VBTree-based scheme: $\mathcal{L} = (\mathcal{L}_1, \mathcal{L}_2, \mathcal{L}_3) = (\mathcal{L}^{Setup}, \mathcal{L}^{Search}, \mathcal{L}^{Update})$, where

$$\mathcal{L}_1 = \mathcal{L}^{Setup}(\mathcal{T}, D, CR) = (M, L, M_c, n, sizes, ids),$$
$$\mathcal{L}_2 = \mathcal{L}^{Search}(T(q)) = \{(SP(w), HIST(w))\}_{w \in q}, \quad \text{and}$$
$$\mathcal{L}_3 = \mathcal{L}^{Update}(T_{up}(op, w, id)) = \perp .$$

Given an encrypted tree $\mathcal{T}$, a set of encrypted files D, and an encrypted CR, $\mathcal{L}_1$ leakage contains M, i.e., the number of entries in the hash table of the VBTree, L, i.e., the height of the VBTree, n, i.e., the number of files, M_c, i.e., the number of entries in CR, $sizes$, i.e., the set of file sizes of the files encrypted by a CPA-secure scheme, and ids, the set of file identifiers of the files varying from 0 to $n - 1$.

Given a conjunctive query $T(q)$, $\mathcal{L}_2$ leakage function outputs all of the search patterns of subqueries and all of the history of subqueries. This leakage includes all historical modifications of queried keywords. Note that the access pattern, which is the information about all search results of all historical subqueries, has been included in $\mathcal{L}_2$ leakage. The number of query terms u is also in this leakage.

Given an update query $T_{up}(op, w, id)$ to add a pair (w, id) with $op = `add'$ or to delete a pair $(w, id)$ with $op = `del'$, $\mathcal{L}_3$ leakage function outputs nothing.

6.2 IND-CKA2 security

Recall the notation of forward privacy in [7]. An $\mathcal{L}$-adaptively secure SSE scheme is forward private if the update leakage function outputs at most the operation name, the updated document identifiers, and the number of modified keywords in each update. Then, if $\mathcal{L}_3$ leakage function outputs nothing, the scheme is forward private (i.e., forward secure). We now analyze indistinguishability of the VBTree-based scheme.

The IND-CKA2 security model has been discussed in [5,25], and later work. Let's consider a stateful and adaptive adversary $\mathcal{A}$, who has been given an ability to get search and update trapdoors, which can be based on historical queried results, over the encrypted index and files without accessing the secret keys K. To prove that an SSE scheme is adaptively secure, we should show the existence of a probability polynomial-time simulator that can adaptively simulate the adversary's view, including historical queries, encrypted files, and the encrypted index, using only the leakage $\mathcal{L} = (\mathcal{L}_1, \mathcal{L}_2, \perp)$. The simulator is prohibited from accessing any other resources, for example, the secret keys K, the original files, and the unencrypted conjunctive queries.

Theorem 6.1 *(*IND-CKA2 Security*) If F is a pseudo-random function, and H_1, H_2, and H_3 are different random oracles, then the dynamic VBTree-based SSE scheme is IND-CKA2 $(\mathcal{L}_1, \mathcal{L}_2, \perp)$-secure against an adaptive adversary.*

Proof We prove the scheme security at a high level. Let's consider such a stateful and efficient simulator S_1, who can adaptively simulate the adversary's view including historical queries, encrypted files, and the encrypted index, using only the leakage $\mathcal{L}$. We now prove that the adversary $\mathcal{A}$ cannot distinguish the real view from the simulated view with non-negligible probability. S_1 simulates the adversary's view within three steps.

In Step 1, S_1 first simulates the encrypted tree, the encrypted files, and the encrypted cloud repository (CR). According to the $\mathcal{L}_1$ leakage, S_1 creates a simulated hash table to accommodate the tree, which is padded with random values to the same size as the original hash table. S_1 creates a group of randomly generated files, which use the same sizes and same file identifiers as the original ones. S_1 simulates CR. S_1 creates a simulated CR^*, which is padded with random values to the same size as the original CR.

In step 2, S_1 adaptively simulates the search trapdoor $T = \{F_K(w_1), F_K(w_2), \ldots, F_K(w_u)\} = \{z_1, z_2, \ldots, z_u\}$.

If the adversary queries the index i times, the simulator simulates the trapdoor i times. If parts of conjunctive query $\{z_{i1}, z_{i2}, \ldots, z_{ix}\}$ have appeared before according to the search pattern of the $\mathcal{L}_2$ leakage, S_1 uses these historical parts of trapdoor $\{z^*_{i1}, z^*_{i2}, \ldots, z^*_{ix}\}$ for simulation. If another part of the conjunctive query $\{z_{j1}, z_{j2}, \ldots, z_{jy}\}$ has not appeared in history, where $y = u - x$, S_1 generates y never-used random values $\{z^*_{s1}, z^*_{s2}, \ldots, z^*_{sy}\}$ using the same size as the original trapdoor items. The adaptively simulated trapdoor will be $T^* = \{z^*_{i1}, z^*_{i2}, \ldots, z^*_{ix}, z^*_{s1}, z^*_{s2}, \ldots, z^*_{sy}\}$. Then, the simulator S_1 programs the random oracle H_1. S_1 first programs H_1 for z^*_{s1}. For each of the tree node v in the VBTree, S_1 gets a unique path string $Path(v)$ of the node v. If the value $H_1(Path(v)||z_{j1})$ exists in the original hash table according to the access pattern of $\mathcal{L}_2$, S_1 chooses a never-used key-value pair $\{s_{j1}, 1\}$ in the simulated hash table for simulation. As the value $Path(v)||z^*_{s1}$ has never appeared before, S_1 lets the random oracle $H_1(Path(v)||z^*_{s1})$ output the value that equals s_{j1}. If the value $H_1(Path(v)||z_{j1})$ doesn't exist in the original hash table, S_1 ignores the current node v. The simulator repeatedly programs all of the tree nodes for z^*_{s1}. Now, the sub-trapdoor z_{j1} is simulated. Repeat programing the oracle H_1 y times, until the set $\{z_{j1}, z_{j2}, \ldots, z_{jy}\}$ is simulated.

Next, S_1 adaptively simulates the cloud repository (CR). S_1 programs z^*_{s1} for CR first. According to the $\mathcal{L}_2$ leakage, S_1 gets all historical trapdoors of z_{j1}. Suppose there are e_w historical trapdoors of z_{j1} from CR. S_1 chooses e_w never-used random key-value pairs $\{(a_1, b_1), (a_2, b_2), \ldots, (a_{e_w}, b_{e_w})\}$ from CR^* for simulation. S_1 now programs the random oracles H_2 and H_3. Using the similar approach above, S_1 lets $\{c_1, c_2, \ldots, c_{e_w}\}$ be historical trapdoors of z^*_{s1}, where $a_i = H_2(c_i), b_i = H_3(c_i) \oplus c_{i-1}$ and $a_{e_w} = H_2(z^*_{s1}), b_{e_w} = H_3(z^*_{s1}) \oplus c_{e_w-1}$.

Repeat this process until all of the trapdoors $\{z^*_{s1}, z^*_{s2}, \ldots, z^*_{sy}\}$ are programmed. Now, the simulated trapdoor T^* will yield the same values as the outputs of the original trapdoor T.

In step 3, the simulator S_1 adaptively simulates the update trapdoors T_{up_1} and T_{up_2}. If the adversary updates the index i times, the simulator simulates it i times. The adaptively simulated update trapdoors are $T^*_{up_1}$ and $T^*_{up_2}$, which are filled with never-used random values to the same sizes as the original ones.

According to the pseudo-random functions and the CPA-secure encryption algorithm, the simulated index and the original index, the simulated CR and the original CR, the simulated files and the original files, the adaptively simulated search trapdoors and the original search trapdoors, and the adaptively simulated update trapdoors and the original update trapdoors cannot be distinguished in polynomial time with a non-negligible probability. This implies that except for the leakage $\mathcal{L}$, the adversary $\mathcal{A}$ learns nothing about the encrypted index, the encrypted files, and the trapdoors. Thus, the dynamic VBTree-based scheme is secure against the adaptive adversary. Furthermore, the $\mathcal{L}_3$ leakage is nothing. According to the definition of forward privacy by Bost in CCS'16 [7], the scheme is also forward secure. Therefore, the theorem is proved. □

7 Experimental evaluation

7.1 Experimental methodology

The key factors that affect the search performance of a VBTree include the dimension of the query u, the number of files n, and the distribution of the result set. The key factors that affect the construction and update performance include the number of files n, the number of distinct keywords m, the number of (w, id) pairs N, and the number of items in the hash table M. Based on these factors, we generate various experimental configurations and comprehensively evaluate construction time, index sizes, query time, and update efficiency.

DataSets We choose two datasets to evaluate our schemes. One dataset is the Enron e-mail dataset [41], which is also used by Kamara et al. [25] and Naveed et al. in [12]. The unstructured Enron dataset consists of 517,401 e-mail files in total, with sizes varying from 1k to 391k. Contents and metadata of e-mails are all considered as keywords with prefix attribute strings.

The other dataset that we choose is the Gowalla dataset [42], which is also used in [9,11]. The Gowalla dataset consists of 6,442,890 check-in records of users from the period of February 2009 to October 2010. A record consists of a time stamp DATE, a user identifier ID, a location identifier LOC, a position coordinate XPOS, and a position coordinate YPOS. Thus, the Gowalla dataset can be considered as a five-dimensional data table. For easily matching data records, decimal parts of the numeric attributes are ignored. In the Gowalla dataset, a record can be considered as a file, and the number of (w, id) pairs N is five times the number of records n. We generate ten sub-datasets chosen from the Gowalla dataset for testing scalability and efficiency of the VBTree scheme. The dataset sizes range from 0.5 million to 5 million.

We use two algorithms to find insertion orders for the unstructured dataset and the structured dataset, respectively. To get an optimized insertion order of the full Enron dataset, we first sort the files by the TF-IDF algorithm. Next, we split the files into multiple parts with each group size $P = 8192$. Now we can use the graph-based algorithm to optimize each file group. To get an optimized insertion order of the full Gowalla dataset, which is a structured data table, we only need to sort the data table by attributes YPOS, XPOS, DATE, ID, and LOC in turn. Building the graph is not required since

Table 2 Evaluation database sizes

	n	m	N	M	Owner (MB)	EDB (GB)
(1) EN	10E4	97E4	41E6	17E7	55	4.0
(2) EN	50E4	26E5	16E7	67E7	157	15.5
(3) EO	50E4	26E5	16E7	48E7	157	11.1
(4) GN	64E5	14E5	32E6	20E7	65	4.6
(5) GO	64E5	14E5	32E6	17E7	65	4.0

'E' means the Enron dataset. 'G' means the Gowalla dataset. 'O' means an optimized index, and N means non-optimized. 'M' is the number of items in the hash table of the VBTree. 'Owner' denotes the storage size of the data owner

this dataset is only a five-dimensional data table. Intuitively, an attribute with more high-frequency words means more redundant information in the index. Thus, we should first sort the table by this attribute.

With the optimized file groups, we can incrementally instantiate the VBTree online or off-line.

Implementation details We conduct our experiments mainly on a PC server running Windows Server 2012 R2 Enterprise with 128 GB memory and two Intel(R) Xeon(R) E5-2670 2.6 GHz processors. We choose Blake2b as the pseudo-random function. Blake2b is also used to emulate the random oracles in the scheme. The VBTree scheme is fully implemented in C++. We also write several testing cases to evaluate the overall performances.

The height of the VBTree L is set to 32. Therefore, the tree can hold 2^{31} files at most. If we set $L = 64$, the index construction time will turn about two times larger than the time of $L = 32$. The additional time is payed for pre-computing the entries in the sub-roots (whose height are smaller than $(L - \lfloor \log n \rfloor)$) for later fast insertions. The bit length of the output of H_1 is 160. The maximum load factor of the hash table is set to $\alpha = 80\%$. Thus, given 16 GB memory, we can fill $M = 6.8$E8 items into the VBTree at most.

7.2 Index construction evaluations

The experimental results show that the index size of VBTree and the storage size of the data owner are practically acceptable. Table 2 includes five records, where records 1–3 use the Enron dataset and records 4–5 use the Gowalla dataset. Records 3 and 5 are optimized indexes, but the others are not. In Table 2, M is the number of items in the hash table, and EDB means the size of the encrypted hash table in GB. The non-optimized index of the whole Enron dataset of 50E4 files consumes 15.5 GB storage space, and the owner storage is 157 MB also. The index of the non-optimized whole Gowalla dataset of 64E5 records requires 4.6 GB disk space, and the storage of the data owner LR_{onr} is 65 MB.

The storage size of the data user LR_{usr} is usually small. In the setup stage, LR_{usr} is nothing. After many queries, the users only need to store the updated versions and all leaked keywords. In our experiments, if the data user has queried the data table of the Enron dataset 1000 times in total with different keywords, and if the data table has been dynamically updated by 8000 new e-mails with each e-mail file about 4 KB after the setup stage, her local storage is about 1MB. This information still can be compressed.

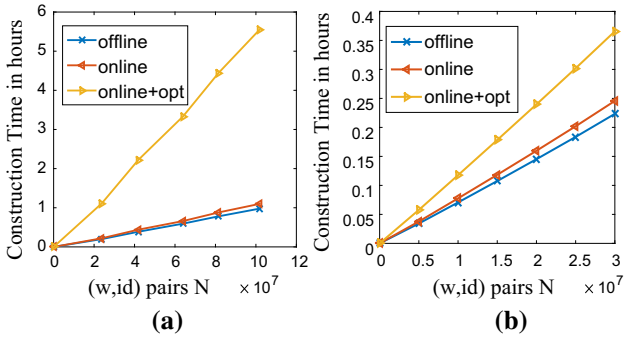


Fig. 4 Off-line and online constructions. **a** Enron dataset, **b** Gowalla dataset

Table 2 also shows that the optimization algorithm can significantly reduce the index size of the Enron dataset. For the unstructured Enron dataset, there are many nearly duplicated e-mails, which means much redundant information exists in the index, and we can optimize this for faster query processing and smaller index size. The optimized index size of the Enron dataset is 70% of the non-optimized one. For the Gowalla dataset, which is a partially ordered data table by the attribute ID, its redundant information is relatively small. The index size of the Gowalla dataset can only be reduced from 4.6 to 4.0 GB.

Figure 4 gives three possible options of index constructions. Off-line construction means indexing work is done locally. Online construction means the index is created incrementally on the cloud. The tag 'opt' means the index is optimized by the graph-based algorithm in figure (a) or is optimized by the sorting algorithm in figure (b). As shown in Fig. 4a, indexing the full Enron dataset without index optimization only takes 1 h, which is approximately $35\mu s$ per keyword/document pair. In Fig. 4b, indexing the full Gowalla dataset takes 0.3 h, which is approximately $36\mu s$ per pair. Though instantiating a tree online without optimization

spends slightly more time than that of off-line instantiating, off-line creating a big index (e.g., $N = 10E9$) is very difficult on a resource-restrained client in most cases. In the line of 'online+opt', optimizing the tree of the Enron dataset and the Gowalla dataset consumes 5 h and 7.8 min, respectively, since the algorithms are different. Although the graph-based indexing algorithm is time-consuming, the time is paid for later faster conjunctive queries.

7.3 Dynamically sorting the data table

One would wonder why we don't sort the data table dynamically. It is an attractive and challenging job to sort the encrypted data table privately. Intuitively, since we can dynamically add a file and remove a file with forward privacy, we can swap two file positions with fixed file identifiers. Then, we have the ability to remotely sort the forward-private data table by one or more attributes. Unfortunately, there are still two challenges in dynamically sorting the data table.

First, we should reduce the storage size of the data owner. To build an update trapdoor, the owner requires learning the corresponding keywords. To sort the whole data table, the owner requires learning all the files. Thus, the storage size of the owner is $O(N)$, which is the whole unencrypted dataset.

Second, we should improve search efficiency of the dynamically sorted data table. Even if we can dynamically sort the data table, the newly added encrypted contents are still in unsorted linked lists. That is to say, the dynamically sorted data table is only in a logical view, and the actual table is still not sorted. Thus, the search performance has not been improved by this approach.

Therefore, we recommend that the data owner handle the index optimization work locally if the data users want to gain good search performance.

7.4 Query processing evaluations

The evaluation in Fig. 5 shows that the two-dimensional conjunctive query time is mainly related to the minimum size of subqueries. The conjunctive query takes at most 100 ms if one of the subqueries outputs 2000 results, and takes at most 1 ms if one of the subqueries outputs 15 results. This figure shows two lines with respect to the conjunctive query 'x AND w' with the size of the subquery x varying from 10 to 1E5. One is the line of $DB(w) = 15$, and the other is the line of $DB(w) = 2000$. For simplicity, we write $DB(w) = 15$ to denote 15 results of the query. The upper boundary of the query time is about 25 ms.

From experiments in Figs. 6 and 7, we can see that the index optimization algorithm can significantly reduce the query time. There are three distribution cases of a keyword w: worst case, best case, and optimized case. To get all results of w, the worst-case search will traverse

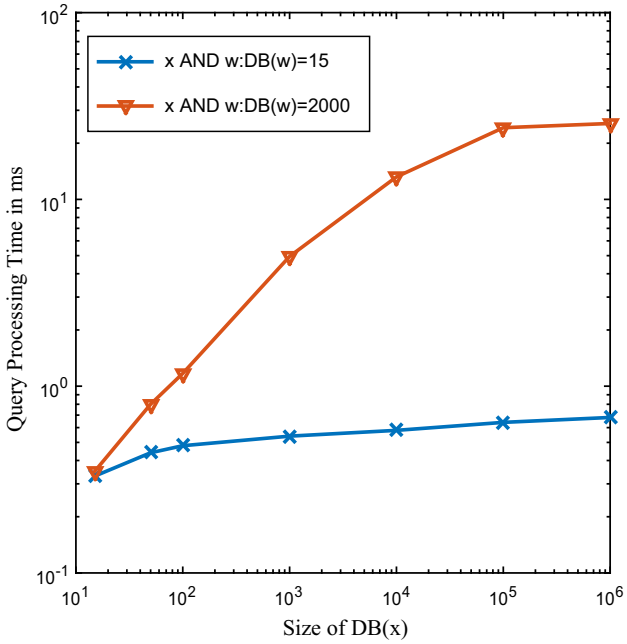


Fig. 5 Two-dimensional conjunctive queries

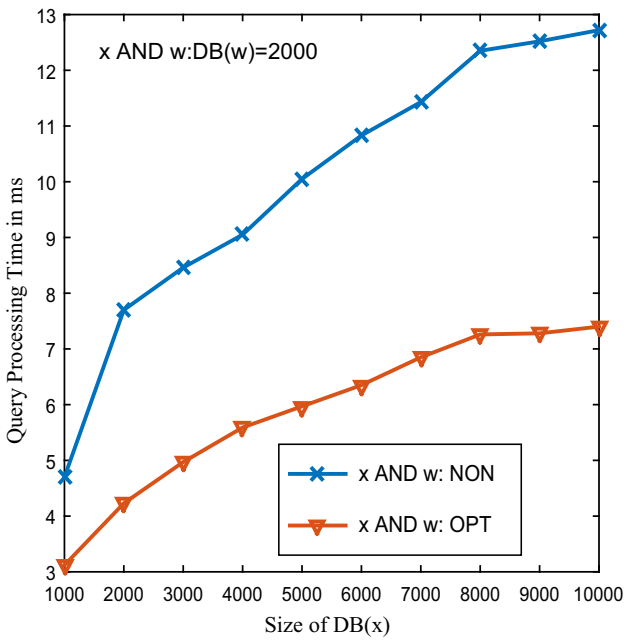


Fig. 6 Optimized two-dimensional queries, Enron dataset

$O(|DB(w)| \log_2 n)$ nodes, the best-case search will traverse $O(2|DB(w)|)$ nodes, and the number of traversed tree nodes in the optimized case is between them. In our experiments, except for Figs. 6, 7, and 8, all search queries of all figures are in the worst case.

Figure 6 shows two lines of the two-dimensional query 'x AND w' using the full Enron dataset. One is the non-optimized line labeled with 'NON', and the other is the

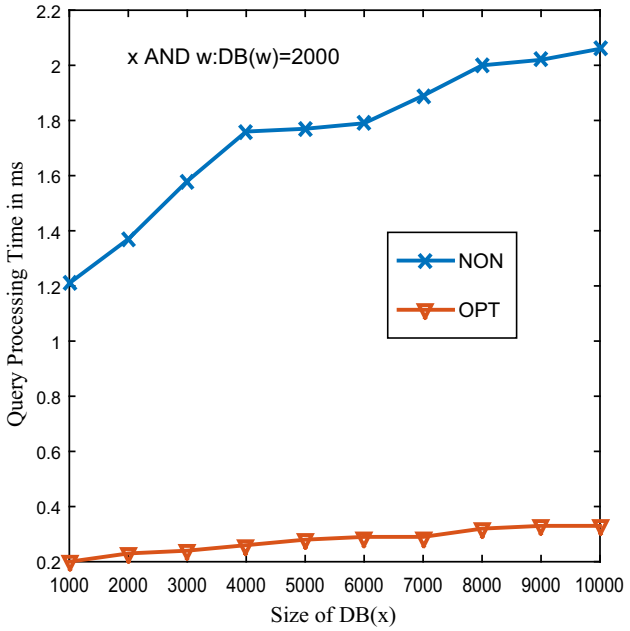


Fig. 7 Optimized two-dimensional queries, Gowalla dataset

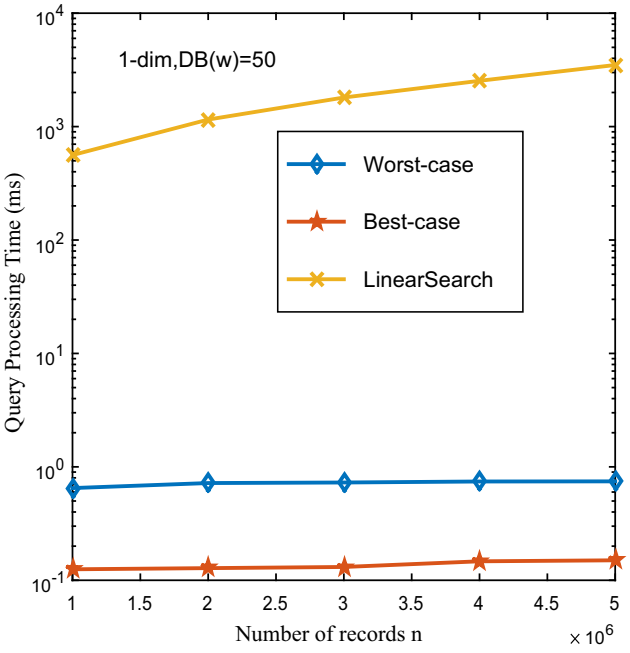


Fig. 8 DB$(w) = 50$, one-dimensional queries

optimized line labeled with 'OPT.' The keyword w of 'OPT' is randomly sampled from the optimized index with a fixed result set size of 2000, yet the result set of the subquery x is in the worst-case distribution. The distribution of w of 'OPT' is optimized with the graph-based algorithm. We do these two experiments over two indexes of the same dataset. With the size of the subquery x growing from 1 to 1E6, the two-dimensional query time of 'NON' increases from 4.7 to

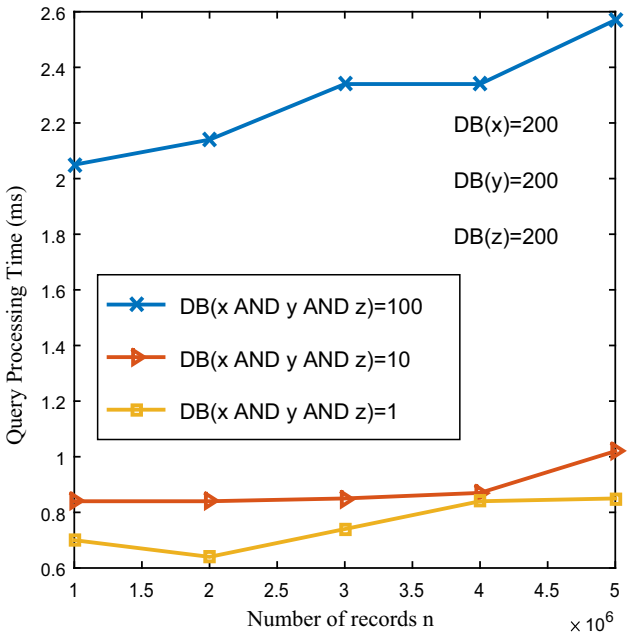


Fig. 9 Three-dimensional queries

12.7 ms, whereas the query time of 'OPT' increases from 3.1 to 7.4 ms. The queries of the optimized index are about two times efficient than those of the non-optimized index. If the distribution of x is also optimized, these conjunctive queries will perform better.

The experiments in Fig. 7 show a comparison between the fully ordered data table and the original data table over the full Gowalla dataset. Figure 7 is similar to Fig. 6. The difference is that the distribution of w, which is randomly chosen from the XPOS attribute with a fixed result set size of 2000, is optimized with a simple sorting algorithm by five attributes. We can see that the fully ordered data table is more efficient than the partially ordered one (which is initially sorted by ID).

Experimental results in Figs. 8 and 9 show that VBTree is scalable in the total number of records. In Fig. 8, there are three one-dimensional query lines: worst case, best case, and linear search. Their result sizes are all 50. The linear search uses algorithm $\mathcal{VSA}$-2, and the others use $\mathcal{VSA}$-1. $\mathcal{VSA}$-2 takes 3.4 s to search over 5 million records. The linear search is costly compared to the top-down search algorithm. With the number of records growing from 1 million to 5 million, the query time of worst-case $\mathcal{VSA}$-1 only increases from 0.65 to 0.75 ms. The best case of $\mathcal{VSA}$-1 takes only 0.15 ms to search over 5 million records. Figure 9 shows two-dimensional queries and three-dimensional queries, respectively. Distributions of x, y, and z are all in the worst case. Experimental results show that increasing a dimensional query affects a little the conjunctive query time. A smaller final result size will make the conjunctive query more efficient. With the final

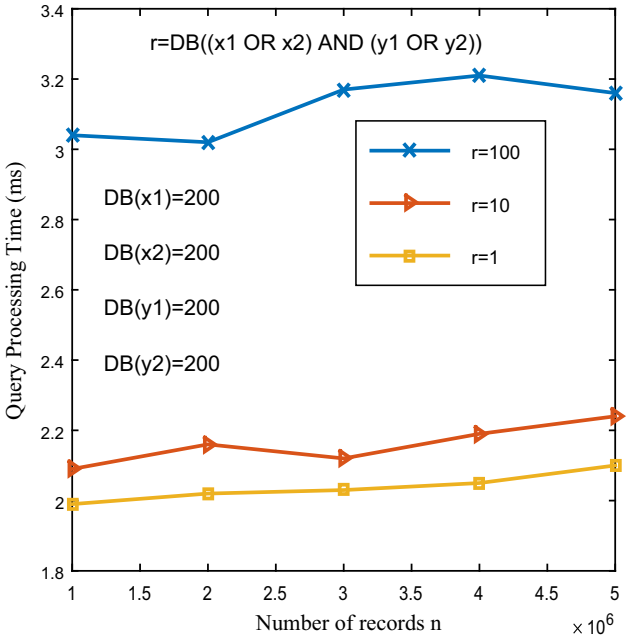


Fig. 10 CNF queries after updates

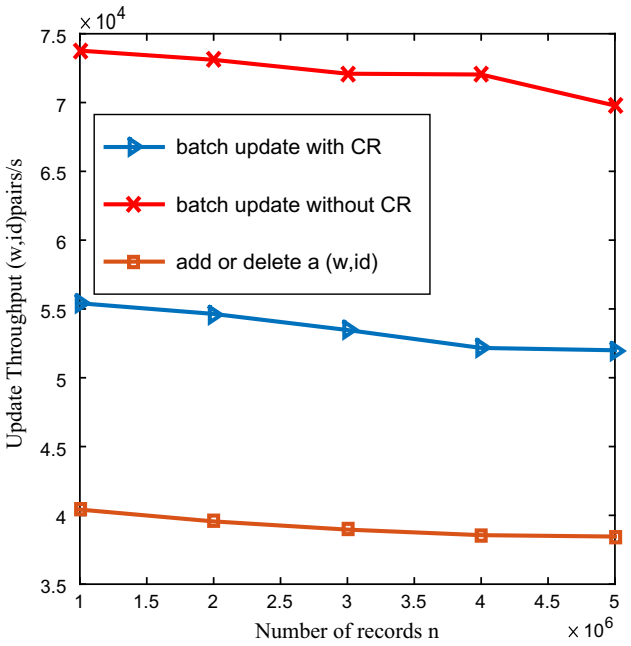


Fig. 11 Update efficiency

result size decreasing from 100 to 1, the three-dimensional query time decreases from 2.5 to 0.9 ms.

7.5 Update evaluations

We evaluate the performances of addition operations and deletion operations in both single-keyword update mode and batch update mode. To add a (w, id) pair to the full Gowalla index, the time consumed is about 2.6 µs, which includes building trapdoor, the work on LR, and insertions on the cloud without accounting for RPC, nearly 40E4 pairs per second. To update a group of files in batch, as shown in Fig. 11, the batch update throughput is 55, 414–73, 106 (w, id) pairs per second, where 'with CR' denotes the update procedure including updating both CR and the tree, which means modifications to the existing files. Each update to CR means a new version trapdoor generated. Batch updating only the tree means addition operations of new files, whose keywords are not queried by the users. With the number of data records growing from 1 million to 5 million, the update throughput of 'with CR' decreases from 55,414 pairs/s to 51,997 pairs/s. The decline is mainly due to the changed load factor of the hash table. The deletion efficiency is same as insertions since the deletion operations are all considered as additions to the $tree_B$. The bottleneck of the update procedure is clearly the pseudo-random computation. From this figure, we can conclude that the update complexity of VBTree is sub-linear.

To test search efficiency after updates, we generate conjunctive normal form (CNF) queries to evaluation Algorithm $\mathcal{VSA}$-3. Since the thread of the index reorganization is running in the background, which takes nothing of the user's time, the search efficiency of Algorithm $\mathcal{VSA}$-4 is almost equal to that of $\mathcal{VSA}$-3. As shown in Fig. 10, we create three CNF queries, as the forms of $(x1$ OR $x2)$ AND $(y1$ OR $y2)$. $x1$ and $x2$ mean two different versions of x, and so on. Experimental results show that the search after updates is still efficient. It takes only 3.6ms to search over 5 million records with each subquery 200 results using CNF queries.

Figure 12 gives the evaluation of search efficiency after updating all keyword versions to the maximal version v_u. The line 'Update 5 times' means that all keywords including x and y are continuously upgraded to a series of versions $\{v_u+1, v_u+2, \ldots, v_u+5\}$ using algorithm $\mathcal{VUA}$-1 5 times, and so on. We can see that even if the data owner updates all keywords ten times, this brings only about $3x$ search performance loss, in return for the zero local storage of the users per times. Moreover, the data owner can do an upgrade only when the user storage is large. The line 'repeated search' denotes the search time of the duplicate searches for the same conjunctive query using $\mathcal{VSA}$-4. It is efficient since the previous search queries have reorganized the tree.

7.6 Compared with IBTree and KRB

We compare IBTree in [9] with VBTree. For evaluating the search efficiency, we adopt the Gowalla dataset for both IBTree and VBTree. Testing results show that VBTree is much more efficient than IBTree. The conjunctive queries are the form of 'x AND y', with $DB(x) = 100DB(y)$=100 and $DB(x\ AND\ y) = 50$. The keywords are randomly generated

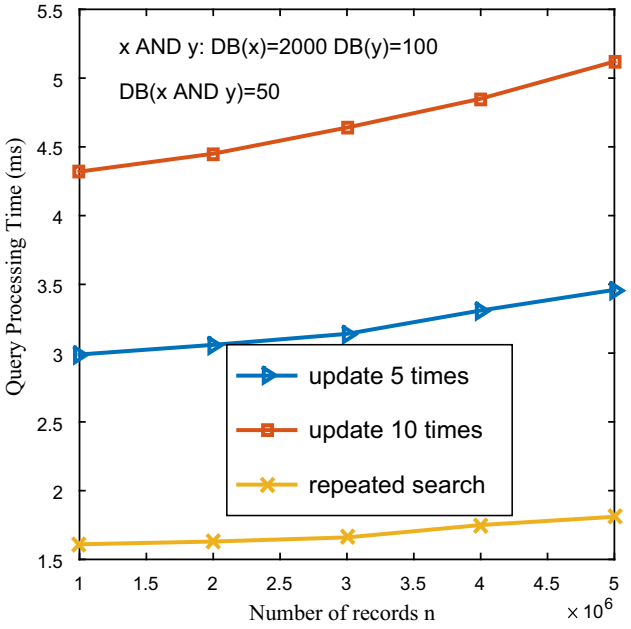


Fig. 12 Search efficiency after updates

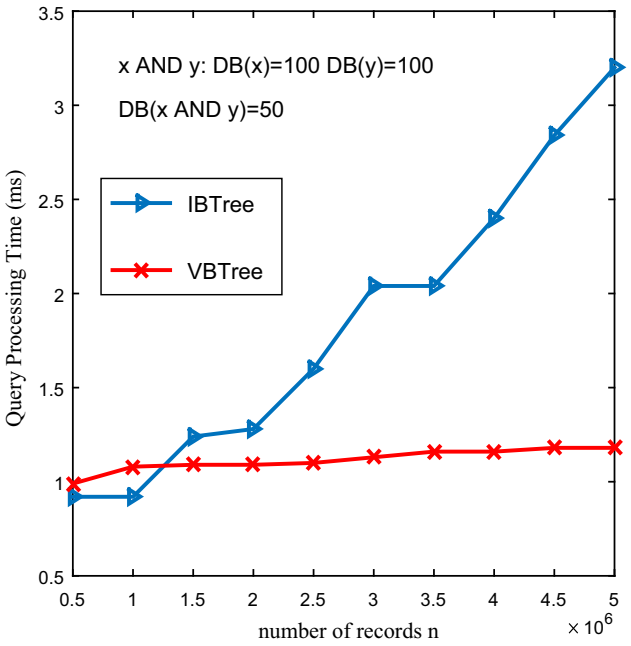


Fig. 13 Compared with IBTree (query time)

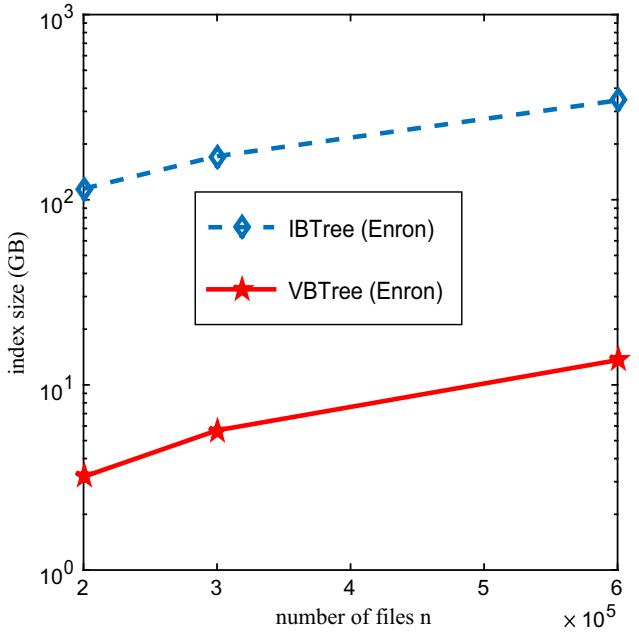


Fig. 14 Compared with IBTree (index size)

and inserted into the Gowalla index. As shown in Fig. 13, with the Gowalla dataset size growing from 1 million to 5 million, the query time of VBTree increases from 0.99 to 1.18 ms, whereas IBTree increases from 0.92 to 3.23 ms. The main reason is that in each of tree node processing, IBTree costs t pseudo-random computations to test whether a keyword is contained in a tree node or not, where t is the number of pseudo-random functions of the Bloom filter (IBF), but VBTree costs only one pseudo-random operation in each tree node, approximatively a k times improvement. For evaluating the index size of the tree, we adopt the Enron dataset. Recall that for all branch-leaked trees, it is hard for these trees to achieve scalable index size. As shown in Fig. 14, the index size of IBTree on the Enron dataset is unsalable. The index size of IBTree is proportional to the size of the largest data file in the dataset, for example, the file of size 391 KB of the Enron dataset. This makes IBTree support only structured datasets, for example, the Gowalla dataset, whose largest data files are small.

We compare KRB in [6] with VBTree for evaluating efficiency and scalability. As shown in Fig. 16, the KRB tree has an unscalable index size for the Enron dataset. To index the full dataset, the KRB tree will consume about 30TB space, where each keyword in each tree node is encrypted into two strings, but VBTree requires at most 15 GB space. To test the conjunctive query time with KRB, we implement the search algorithm of KRB tree by an algorithm that is similar to $\mathcal{VSA}$-1. We only test the query time over 10,000 files, due to limited memory. As shown in Fig. 15, VBTree is more efficient than KRB. This is because in the KRB tree, for matching in each tree node, KRB uses a pseudo-random function and a symmetric-decryption function, which brings the reduction of performance.

7.7 Compared with Sophos and FFSSE

We compare the VBTree scheme with the forward-private schemes Sophos in [7] and FFSSE in [8]. They both have studied how to reduce the client-side storage. In VBTree, the

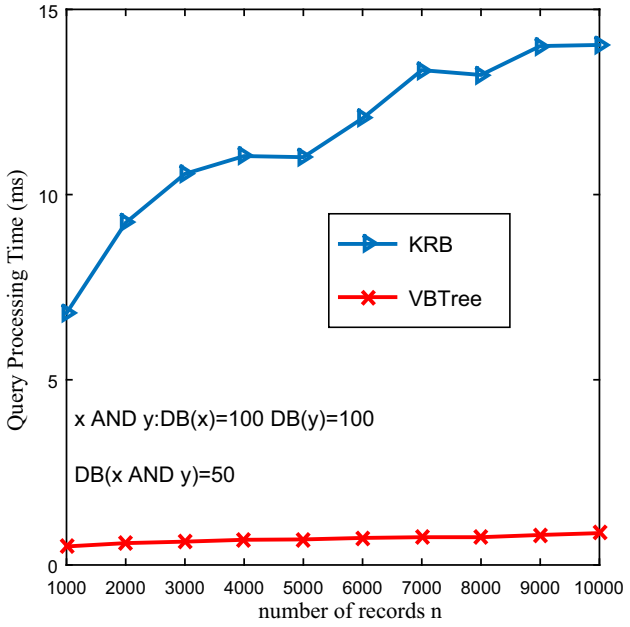


Fig. 15 Compared with KRB (query time)

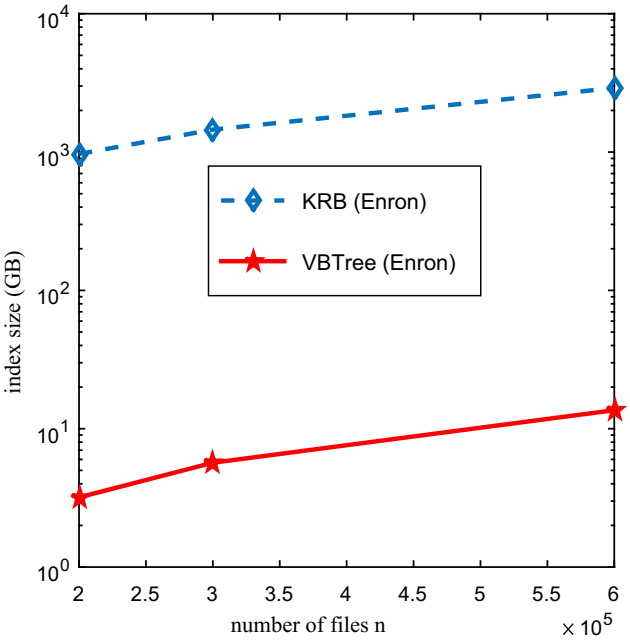


Fig. 16 Compared with KRB (index size)

client-side storage of the users LR_{usr} is mainly related to the amount of newly updated files. Furthermore, the user storage can be repeatedly removed or permanently deleted in VBTree by updating all keywords to the same versions. However, in Sophos and FFSSE, the map information of all keywords must be stored on the users' disk no matter whether these keywords will be used in the future. This map information of the full Enron dataset, whose dictionary size is $m = 26E5$, will consume about 100 MB space (approximately 4% of the size of all original uncompressed files). The data owner's storage sizes of these three schemes are all $O(m \log_2 n)$, but this information is only used for updates.

8 Conclusions

In this paper, we outsource an encrypted data table to the public clouds with SSE techniques. The data table is mainly based on a novel tree data structure called virtual binary tree (VBTree). Our scheme achieves adaptive and forward security, scalable index size, and efficient query processing. To achieve forward privacy, we also propose a version control repository to manage versions of keywords and queries. We test our scheme on two datasets. The experimental results show that the VBTree-based data table supports efficient conjunctive queries and sub-linear time updates with strong privacy guarantee. The results also show that our scheme achieves the forward privacy with small client-side storage.

Appendix: Proof of Theorem 5.1

Proof Let's first study DDFPP. First, given a partition instance, the solution can be verified in polynomial time and then DDFPP is NP. Next, we reduce a known NP-complete problem, the Subset Sum Problem, to DDFPP. The Subset Sum Problem is as follows: "For a multiset of positive numbers $A = \{a_1, a_2, \ldots, a_n\}$, given a positive number t, is there a set B such that $\sum_{a_i \in B} a_i = t$ and $B \subseteq A$?". □

Considering an instance of the Subset Sum Problem with a positive number multiset $A = \{a_1, a_2, \ldots, a_n\}$ and a positive number t, we convert it to an instance of DDFPP using following construction. If n is odd, we insert a zero number into A first. We create two variables a_{n+1} and a_{n+2}, and let $\sum_{a_i \in A} a_i = b$, $a_{n+1} = 2b - t$, and $a_{n+2} = b + t$. Then, a new set C is created as follows, $C = A \cup \{a_{n+1}, a_{n+2}\}$. For each number a_i in C, we generate a file group $G_{a_i} = \{d_1, d_2, \cdots, d_{a_i}\}$. All generated files are $G_{a_1} \bigcup G_{a_2} \bigcup \cdots \bigcup G_{a_{n+2}}$, which have $\sum_{a_i \in C} a_i = b + (2b - t) + (b + t) = 4b$ files in total. For each data file group G_{a_i}, we create k unique keywords first and insert these k keywords into every file in the group. That is to say, in the a_ith file group, a_i files share k same keywords. Next, we insert other randomly generated keywords into every file. Repeat this process until all of the file groups are initialized.

Suppose the file groups constructed above have a data file partition solution, and the file groups can be partitioned into $\mathcal{F}_1$ and $\mathcal{F}_2$ in polynomial time, such that $|\mathcal{F}_1| = |\mathcal{F}_2|$ and $|W(\mathcal{F}_1) \bigcap W(\mathcal{F}_2)| < k$. We now prove that A has a subset sum solution. Note that, for each of the file group, the files share k common keywords. It implies that for any file group

G_{a_i} constructed from the number a_i, the files of G_{a_i} are either all in $\mathcal{F}_1$ or all in $\mathcal{F}_2$. Otherwise, suppose there exist two files d_i and d_j with $d_i \in \mathcal{F}_1$ and $d_j \in \mathcal{F}_2$, then we have $|d_i \bigcap d_j| \geqslant k$ and $|W(\mathcal{F}_1) \bigcap W(\mathcal{F}_2)| \geqslant k$. This contradicts the fact that we assume. Since $|\mathcal{F}_1| = |\mathcal{F}_2|$, we have $|G_{x_1}| + |G_{x_2}| + \cdots + |G_{x_r}| = |G_{y_1}| + |G_{y_2}| + \cdots + |G_{y_r}|$, where $G_{x_i} \subseteq \mathcal{F}_1, G_{y_i} \subseteq \mathcal{F}_2$, and $2r = n+2$. As $|G_{x_i}| = x_i$ and $|G_{y_i}| = y_i$, the following equation holds, $x_1 + x_2 + \cdots + x_r = y_1 + y_2 + \cdots + y_r$. Let $B = \{x_1, x_2, \ldots, x_r\}$ and $B' = \{y_1, y_2, \ldots, y_r\}$. Note that, a_{n+1} and a_{n+2} cannot coexist in the set B or in the set B', otherwise, $\sum x_i \neq \sum y_i$. If a_{n+1} is in B, then the subset sum $\sum_{i \in (B - \{a_{n+1}\})} a_i = (2b - (2b - t)) = t$. If a_{n+1} is in B,' then $\sum_{i \in (B' - \{a_{n+1}\})} a_i = (2b - (2b - t)) = t$. Now, we have a subset sum solution of $B - \{a_{n+1}\}$ or $B' - \{a_{n+1}\}$. Finally, the Subset Sum Problem $\leq_p$ DDFPP, which means DDFPP is NP-complete. Since DDFPP is a special case of DFPP, then DFPP is NP-hard.

References

1. Amazon: "Amazon Web services" (2017). http://aws.amazon.com
2. Microsoft: "Microsoft Azure" (2017). http://www.microsoft.com/azure
3. Google: "Google App Engine" (2017). http://code.google.com/appengine
4. Zhang, Y., Katz, J., Papamanthou, C.: All your queries are belong to us: the power of file-injection attacks on searchable encryption. In: 25th USENIX Security Symposium (USENIX), pp. 707–720. USENIX Association (2016)
5. Curtmola, R., Garay, J., Kamara, S., et al.: Searchable symmetric encryption: improved definitions and efficient constructions. In: Proceedings of the 13th ACM Conference on Computer and Communications Security (CCS), vol. 95, No. 5, pp. 79–88. ACM (2006)
6. Kamara, S., Papamanthou, C.: Parallel and dynamic searchable symmetric encryption. In: Sadeghi, A.R. (ed.) Financial Cryptography and Data Security FC 2013. Lecture Notes in Computer Science, vol. 7859, pp. 258–274. Springer, Berlin, Heidelberg (2013)
7. Bost, R.: $\Sigma o\varphi o\varsigma$: forward secure searchable encryption. In: ACM Sigsac Conference on Computer and Communications Security (CCS), pp. 1143–1154. ACM (2016)
8. Liu, Z., Lv, S., et al.: FFSSE: flexible forward secure searchable encryption with efficient performance. ACR Cryptology ePrint Archive (2017)
9. Li, R., Liu, A.X.: Adaptively secure conjunctive query processing over encrypted data for cloud computing. In: International Conference on Data Engineering (ICDE), pp. 697–708. IEEE (2017)
10. Goh, E.J.: Secure indexes. IACR Cryptology ePrint Archive (2003)
11. Li, R., Liu, A.X., Wang, A.L., et al.: Fast range query processing with strong privacy protection for cloud computing. In: International Conference on Very Large Data Bases (VLDB), pp. 1953–1964 (2014)
12. Naveed, M., Prabhakaran, M., Gunter, C.A.: Dynamic searchable encryption via blind storage. In: Security and Privacy (S&P), pp. 639–654 (2014)
13. Li, R., Liu, A.X., Wang, A.L., et al.: Fast and scalable range query processing with strong privacy protection for cloud computing. In: Transactions on Networking (TON), pp. 2305–2318 (2016)
14. Xia, Z., Wang, X., Sun, X., Wang, Q.: A secure and dynamic multi-keyword ranked search scheme over encrypted cloud data. Trans. Parallel Distrib. Syst. (TPDS) **27**(2), 340–352 (2016)
15. Song, D.X., Wagner, D., Perrig, A.: Practical techniques for searches on encrypted data. In: IEEE Symposium on Security and Privacy (S&P), pp. 44–55 (2000)
16. Chang, Y.C., Mitzenmacher, M.: Privacy preserving keyword searches on remote encrypted data. In: Applied Cryptography and Network Security (ACNS), pp. 442–455. Springer, Berlin (2005)
17. Bezawada, B., Liu, A.X., Jayaraman, B., et al.: Privacy preserving string matching for cloud computing. In: IEEE International Conference on Distributed Computing Systems (ICDCS), pp. 609–618 (2015)
18. Chase, M., Kamara, S.: Structured encryption and controlled disclosure. In: Conference on the Theory and Application of Cryptology and Information Security (ASIACRYPT), pp. 577–594. Springer, New York (2010)
19. Kurosawa, K., Ohtaki, Y.: UC-secure searchable symmetric encryption. In: Financial Cryptography and Data Security (FC), pp. 285–298. Springer, New York (2012)
20. Liesdonk, P.V., Sedghi, S., Doumen, J., Hartel, P., Jonker, W.: Computationally efficient searchable symmetric encryption. In: VLDB Conference on Secure Data Management (SDM), pp. 87–100. Springer, New York (2010)
21. Cash, D., Jarecki, S., Jutla, C., et al.: Highly-scalable searchable symmetric encryption with support for Boolean queries. In: International Cryptology Conference (CRYPTO), pp. 353–373. Springer, New York (2013)
22. Pappas, V., Krell, F., Vo, B., et al.: Blind seer: a scalable private DBMS. In: Security and Privacy (S&P), pp. 359–374 (2014)
23. Ishai, Y., Kushilevitz, E., Lu, S., et al.: Private large-scale databases with distributed searchable symmetric encryption. In: Cryptographers 'Track at the RSA Conference, pp. 90–107. Springer, New York (2016)
24. Kamara, S., Moataz, T.: SQL on structurally-encrypted databases. IACR Cryptology ePrint Archive (2016)
25. Kamara, S., Papamanthou, C., Roeder, T.: Dynamic searchable symmetric encryption. In: Proceedings of the 2012 ACM Conference on Computer and Communications Security (CCS), pp. 965–976. ACM (2012)
26. Cash, D., Jaeger, J., Jarecki, S., et al.: Dynamic searchable encryption in very-large databases: data structures and implementation. In: Network and Distributed System Security (NDSS), pp. 23–26. ISOC (2014)
27. Kamara, S., Moataz, T.: Boolean searchable symmetric encryption with worst-case sub-linear complexity. In: European Cryptology Conference (EUROCRYPT). Springer, New York (2017)
28. Wang, B., Yu, S., Lou, W., et al.: Privacy-preserving multi-keyword fuzzy search over encrypted data in the cloud. In: INFOCOM, pp. 2112–2120 (2014)
29. Fu, Z., Wu, X., Guan, C., et al.: Toward efficient multi-keyword fuzzy search over encrypted outsourced data with accuracy improvement. In: Transactions on Information Forensics and Security (TIFS), pp. 2706–2716
30. Stefanov, E., Papamanthou, C., Shi, E.: Practical dynamic searchable encryption with small leakage. In: Network and Distributed System Security (NDSS), pp. 23–26. ISOC (2014)
31. Garg, S., Mohassel, P., Papamanthou, C.: TWORAM: round-optimal oblivious RAM with applications to searchable encryption. IACR Cryptology ePrint Archive (2015)
32. Bost, R., Fouque, P.A., Pointcheval, D.: Verifiable dynamic symmetric searchable encryption: optimality and forward security. IACR Cryptology ePrint Archive (2016)
33. Chang, Z., Xie, D., Li, F.: Oblivious RAM: a dissection and experimental evaluation. In: International Conference on Very Large Data Bases (VLDB), pp. 1113–1124 (2016)

34. Islam, M.S., Kuzu, M., Kantarcioglu, M.: Access pattern disclosure on searchable encryption: ramification, attack and mitigation. In: Network and Distributed System Security (NDSS). ISOC (2012)
35. Popa, R.A., Redfield, C., Zeldovich, N., et al.: CryptDB: protecting confidentiality with encrypted query processing. In: Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles (SOSP), pp. 85–100. ACM (2011)
36. Mavroforakis, C., Chenette, N., O'Neill, A., et al.: Modular order-preserving encryption, Revisited. ACM International Conference on Management of Data (SIGMOD), pp. 763–777. ACM (2015)
37. Naveed, M., Kamara, S., Wright, C.V.: Inference attacks on property-preserving encrypted databases. In: ACM Sigsac Conference on Computer and Communications Security (CCS), pp. 644–655. ACM (2015)
38. Yao, A.C.: Protocols for secure computations. In: Foundations of Computer Science (SFCS), pp. 160–164 (1982)
39. Ben-David, A., Nisan, N., Pinkas, B.: FairplayMP: a system for secure multi-party computation. In: Proceedings of the 15th ACM Conference on Computer and Communications Security (CCS), pp. 257–266. ACM (2008)
40. Dijk, M.V., Gentry, C., Halevi, S., et al.: Fully homomorphic encryption over the integers. In: Advances in Cryptology – EUROCRYPT, pp. 24–43. Springer, Berlin, Heidelberg (2010)
41. Enron email dataset (2015). http://www.cs.cmu.edu/~enron/
42. Cho, E., Myers, S.A., Leskovec, J.: Friendship and mobility: user movement in location-based social networks. In: Proceedings of the 17th International Conference on Knowledge Discovery and Data mining (SIGKDD), pp. 1082–1090. ACM (2011)