FB-Tree: Highly Efficient Tree-Based Index for Encrypted Boolean Queries in Smart Cities

Zhiqiang Wu[1], Kenli Li[2], Jin Wang[1], and Naixue Xiong[3]

Abstract

To expand capacity, many resource-constrained industrial devices encrypt and outsource their private data to public clouds, employing a searchable encryption (SE) scheme that provides efficient search service directly to encrypted data. Current tree-based SE schemes can do this and support sublinear encrypted Boolean queries. However, they all suffer from $\log n$ overhead in a search procedure. To resolve the challenge, in this paper, we propose a new tree structure called the four-branch tree (FB-tree). Our key design is to build a tree node with four branches, which helps a search reach the destination nodes with fast jumps. Based on the index tree, we setup two systems for different requirements. The first system for efficiency-first settings achieves nearly optimal search complexity and adaptive security. The second, constructed via an oblivious RAM for security-first environments, still achieves worst-case sublinear search complexity. FB-tree performance is extensively evaluated with several real datasets. The Experimental data demonstrate that the FB-tree-based systems outperform the state-of-the-art solutions in terms of efficiency and scalability when Boolean queries are issued.

Index Terms

Industrial Security, Oblivious RAM, Searchable Symmetric Encryption.

I. Introduction

Currently, many industrial devices extend their data services to cloud environment due to cheap and unlimited resources. However, the cloud possibly exist many security risks, such as hacker's attacks and leakages released from malicious cloud managers. To address these issues, researchers have proposed searchable (symmetric) encryption (SE/SSE) that allows a client to outsource an encrypted database to the cloud who can still answer the search queries efficiently.

A Boolean query over encrypted data is a two-party protocol. The client initially outsources the encrypted files as well as an index that is constructed from the files to the cloud. To perform a query, the client encrypts the conjunctive/Boolean query $\varphi(w_1, w_2, \cdots, w_u)$ and sends it to the cloud. The query will be handled by the cloud and finally a set of file identifiers $DB(\varphi)$ are matched. In the protocol, it is assumed that the client (the device) is trusted and the cloud is not fully trusted, which means that the cloud may attempt to obtain private information, such as the file contents and plain-text search queries, yet the cloud always runs the protocol correctly. For example, a device remotely searches its files from the encrypted index with some input conditions, using a Boolean SQL query "select * from data where name='dev5' or name='dev6' and type='tp100' ".

There are many tree-based SSE works that support efficient conjunctive/Boolean queries, such as KRB [3], BlindSeer [4], [5], PBTree [6], IBTree [7], and VBTree [8]. Their search efficiency benefits from the tree-based design. That is, if a tree node does not match the query, it implies that its subtrees do not match the query. This feature helps the conjunctive/boolean queries to reduce the search length significantly.

Unfortunately, these trees have the logarithmic search overhead in each query. If one tree node matches the condition, the procedure should still touch every subtree level to obtain a result. On most occasions, the algorithm accesses more nodes than necessary.

[1] The School of Computer and Communication Engineering, Changsha University of Science and Technology, Hunan, China, 410114. (e-mail: wzq@csust.edu.cn, jinwang@csust.edu.cn)

[2] The College of Information Science and Engineering, and National Supercomputing Center in Changsha, Hunan University, Hunan, China, 410082 (e-mail: lkl@hnu.edu.cn)

[3] The Department of Mathematics and Computer Science, Northeastern State University, Tahlequah, OK 74464, USA. (e-mail: xiong31@nsuok.edu)

Corresponding author: Jin Wang.

Our main targets are 1) to remove this logarithmic multiplicative overhead by designing a new tree data structure, and 2) to seek good security-efficiency trade-offs for various requirements.

A. Security Model

The IND-CKA2 security model in [12] is adopted for Boolean SSE solutions to evaluate the security level of the index. The model has two parts: IND (indistinguishability), under which the real view and the simulated view are indistinguishable in polynomial time, and CKA2, denoting that the adversary can adaptively choose keywords for attacks. If there exists a simulator that can adaptively simulate the adversary's view by using only the predefined leakage function, it implies that the adversary cannot always infer any remaining keywords in the index in polynomial time with nonnegligible probability, and the index is secure.

Oblivious RAM (ORAM), first proposed by Goldreich and Ostrovsky in [11], is a searchable encryption technique to hide data access patterns in untrusted memory. In general, ORAMs are stronger than the adaptively-secure SSE constructions due to all randomly accessed locations that are shuffled in every search/update operation. Index shuffling enables the adversary to have no cumulative knowledge about the index and queries. The adversary learns only the number of memory accesses performed on the data.

B. Limitation of Prior Art

The limitations of prior Boolean query work are described in two aspects.

First, the existing tree-based Boolean/conjunctive query SSE schemes suffer from logarithmic search overhead. Take IBTree [7] and VBTree [8] [34] as examples. The u-dimensional conjunctive query complexity is $O(u \min_{w \in \varphi} |DB(w)| \log_2 n)$, where n is the total number of files, and φ is a u-dimensional conjunctive query. If n is large, the logarithmic overhead, in general, means a multiplier of several orders of magnitude.

Second, the leakages of the search and access patterns for Boolean searches are not completely acceptable. If adequate well-correlated background knowledge is provided to the adversary, many access-pattern-based attacks can partially disclose the device's private data [22] [41] [42].

C. Proposed Approach

Two systems named FB-tree-1 and FB-tree-2 are proposed for different use cases. The first is for the efficiency-first environment, and the second is for the security-first environment.

The efficiency-first construction is based on a novel data structure called a four-branch tree that can remove the logarithmic search overhead in a search query. Informally, a four-branch tree is a particular full binary tree, in which each tree node has at most four branches. Two long branches are shortcuts to the nearest destination nodes for both left-side search and right-side search, respectively, where the destination nodes mean the tree nodes traversed in the search query. To search for files over a four-branch tree, unlike KRB, PBTree, IBTree, and VBTree, the search process can always follow the shortcuts to arrive in the desired leaves. The search process is no longer beginning from the tree root to leaves step by step.

FB-tree-1 achieves nearly optimal search efficiency for Boolean queries since all shortcuts help the search process remove the logarithmic search overhead. The tree-based structure can also help the Boolean queries to be performed efficiently. Although the operation contains controllable leakage, it does not directly reveal the query contents and other vital privacy, and the solution is still adaptively secure. In reality, since most devices want much efficiency, they will likely choose this security-efficiency trade-off.

The security-first construction is also based on the FB-tree structure that is put into an oblivious RAM. Even in the ORAM, the logarithmic search overhead can still be removed. Through making back-box use of the oblivious map proposed in [35], a root-to-leaf search will obliviously access a set of tree nodes that are completely hidden from the cloud. With these tree nodes and branches, the client can perform the Boolean query locally. An advantage of FB-tree-2 is that the number of accessed tree nodes in the search query is minimized. The saved accesses will contribute to reducing communication overhead and search time.

FB-tree-2 achieves high-level security and sublinear search complexity at the cost of many client-server interactions and some client-side computations. If the network latency is very low (e.g., $<$1 ms), this solution is a recommended option. Despite the fact that additional communication overhead and interactive rounds are induced, the hardware enclave based technique proposed in [38] will mitigate these problems.

Table I shows the comparison of current typical tree-based Boolean query schemes.

TABLE I: Comparison of current typical tree-based Boolean searchable encryption schemes

Scheme	Index size	Conjunctive query time	Boolean query time	Query leakage	Rounds
KRB [3]	$O(mn)$	$O(ux \log n)$	-	(SP, AP)	1
IBTree [7]	$O(cn \log n)$	$O(ux \log n)$	-	(SP, AP)	1
VBTree [8]	$O(N \log n)$	$O(ux \log n)$	$O(uvy \log n)$	(SP, AP)	1
FB-tree-1	$O(N \log n)$	$O(ux)$	$O(uvy)$	(SP, AP)	1
FB-tree-2	$O(N \log n)$	$O(ux \cdot t_{oram})$	$O(uvy \cdot t_{oram})$	(n_φ, n_f)	$O(uvy \cdot r_{oram})$

The table denotes by n the number of files, m is the number of keywords that can be queried, N is the number of keyword-document pairs, SP is the search pattern, AP is the access pattern, n_φ is the number of memory accesses for φ, n_f is the number of file-block accesses for φ, t_{oram} is the read/write time per access in an oblivious RAM, and r_{oram} is the number of read/write rounds per access between the server and the client in the oblivious RAM. $c = max_{i \in [1,n]} m_i$, where m_i is the number of distinct keywords in the i-th file. q is a conjunctive query, $q = w_1 \wedge w_2 \cdots \wedge w_u$. φ is a CNF query, $\varphi = \delta_1 \wedge \delta_2 \cdots \wedge \delta_u$, where each $\delta_i = w_{i,1} \vee w_{i,2} \cdots \vee w_{i,v}$. $x = \min_{w \in q} |DB(w)|$, and $y = \min_{\delta_i \in \varphi} |DB(\delta_i)|$. Assuming all disjunctions have the same size, the symbol '-' denotes that the approach was not mentioned in the papers. Note that (n_φ, n_f) is far less than AP.

D. Key Contributions

Our contributions are summarized in three aspects.

- First, FB-tree is a novel tree data structure that supports efficient Boolean queries. This tree can be encrypted by many approaches for different-level security-efficiency trade-offs.
- Second, FB-tree-1 is the first SSE-based Boolean query scheme that achieves nearly optimal search efficiency, adaptive security, and scalable index size with controllable leakage.
- Third, FB-tree-2 is the first ORAM-based Boolean query scheme that provides strong privacy protection and sublinear search efficiency. It is, to the best of our knowledge, stronger than all the prior conjunctive/Boolean constructions since the query leakage contains nothing of the search and access patterns.

E. Document Structure

The rest of this paper is organized as follows: Section II presents the related work of SSE and ORAM; Section III shows the data structures and definitions of the FB-tree designs; Sections IV and V setup the two systems: FB-tree-1 and FB-tree-2 in detail, respectively; Sections VI, VII, VIII consist of security analysis and proof, the experimental performance data, and the conclusion with future work, respectively.

II. Related Work

There is a long list of SSE-based secure keyword query schemes. Many early constructions are non-adaptively secure, such as those in [10] [14]. The adaptively-secure ones of [3] [18] [26] [12], have an unscalable index size of $O(mn)$. Later, the researchers used pseudorandom functions to achieve adaptive security, including the constructions of [13] [20] [24], etc.

Resent SSE work focuses their studies on forward and backward privacy [21] [32] [33] [39] to resist file injection attacks [28]. Wu and Li proposed the first single-round zero-user-storage forward-private solution in [8].

For more functionality, there are many tree-based constructions [3] [6] [7] [8] and the inverted-index-based ones [27] [31]. The difference is that the first exploits a search tree. Numeric data can be considered as text contents through prefix encoding methods [6], which consider a numeric range query as a Boolean expression. Li et al. proposed an SE scheme which supports multi-user voice searches in [43].

Oblivious RAMs (ORAMs) (e.g., [29] [30] [36] [37] [19]) are more secure than SSE constructions. They can do read or write operations without the access pattern leakage, but they need many client-server communication rounds per query.

Other encrypted searches include secure multi-party computation [25], homomorphic encryption [15] [16], and order-preserving encryption [17].

III. Data Structures and Definitions

In this section, a novel tree data structure called the four-branch tree (FB-tree) is proposed to create an index for Boolean query SSE schemes. Before doing so, we introduce the keyword tree and w-tree first.

A. *Keyword tree and W-tree*

The known tree-based constructions are summarized in the following form.

Definition 3.1 (*Keyword Tree*). A keyword tree for an SE scheme is a binary tree for indexing keywords such that:

1) Each tree node contains zero or more different keywords.
2) For any nonterminal tree node v, if the left child tree node is v_l and the right is v_r, then $W(v) = W(v_l) \bigcup W(v_r)$, where $W(.)$ denotes a set of distinct keywords.

A w-tree is a keyword tree containing at most the keyword w in each tree node. In the keyword tree, each keyword w exists in different tree nodes. All the keywords that equal w form an individual binary tree called w-tree. Therefore, a keyword tree is a set of w-trees.

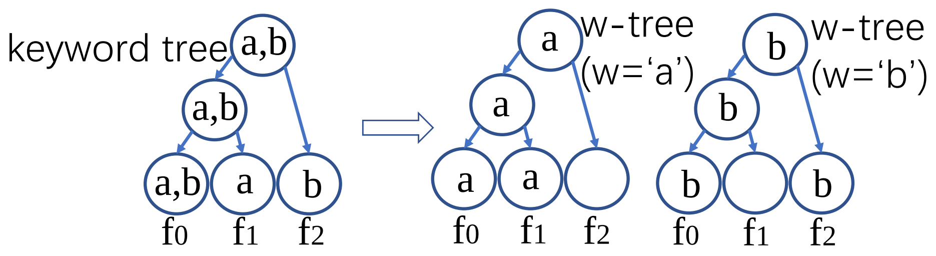


Fig. 1: Keyword tree and its w-trees

Fig. 1 shows a keyword tree and its two w-trees. File f_0 contains keywords {'a','b'}, f_1 contains keywords {'a'}, and f_2 contains {'b'}, and then the root contains keywords {'a','b'}$\bigcup$ {'a'}$\bigcup${'b'}. This keyword tree includes two w-trees (w='a', w='b').

In the keyword tree, each leaf is considered to be a different data file (may be empty) with a unique file identifier. To construct a nearly balanced tree, a set of files (n files) are sequentially inserted into the tree leaves beginning from $leaf_0$ to $leaf_{n-1}$. The client can submit a Boolean query and root-to-leaf search the tree in average logarithmic time to retrieve the desired data files. The known keyword trees, such as KRB [3], PBTree [6], IBTree [7], and VBTree [8], are capable of this work.

Each tree node is labeled with a unique path string for accessing tree nodes more efficiently. The paths of a full binary tree are regular binary strings. 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. If the keyword tree is a full binary tree, the approach to encoding the path of the tree node is stated as follows. For any nonterminal tree node, the tree branch to its left child is encoded as '0', and the right as '1'. For any tree node v, its path denotes a root-to-v branch string. The path of this tree node can also be regarded as a pointer to this node. In the special case, the root path is empty. The path of the i-th leaf (zero-based) is a binary string converted from the integer i, which can also be viewed as a file identifier. Generally speaking, an L-bit integer is enough to hold the path value.

Another important concept for a w-tree is connection node. Given a keyword w, and its corresponding w-tree, the connection node is a w-tree node with two nonempty children. A nonconnection node is a w-tree node with only one nonempty child. The number of connection nodes in the w-tree determines the search complexity of the w-tree since the search thread turns into two parts when it arrives in a connection node.

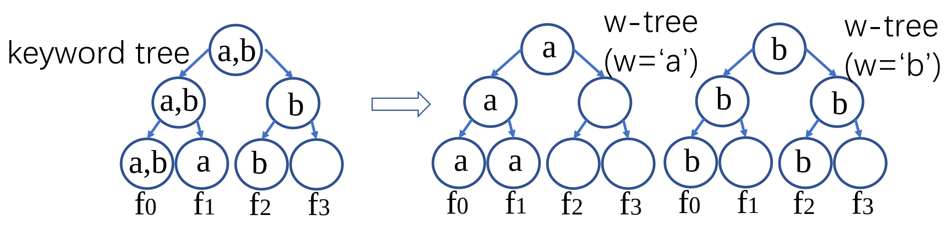


Fig. 2: Full binary keyword tree and its w-trees

Fig. 2 shows a full binary keyword tree ($L = 3$) converted from Fig. 1 as well as two w-trees. A w-tree of the full binary keyword tree is still a full binary tree.

B. Four-branch Tree

Definition 3.2 (*Four-Branch Tree*). A four-branch tree (FB-tree) is a full binary keyword tree (height=L) for a searchable encryption scheme with the following properties:

1) It includes a set of full binary w-trees.
2) Each node of a w-tree has four branches (may be empty): two short branches and two long branches.
3) Two short branches are the pointers to the left child and right child, respectively. If a child is empty, the corresponding pointer is empty.
4) The left long branch (may be empty) of the w-tree node is a pointer to the nearest left-side subtree node that is a connection node or a nonempty leaf. The right long branch is similar to the left long branch.

TABLE II: Pointers of the FB-tree

w-tree	Current	ShB1	ShB2	LB1	LB2	Type
a	‘’	‘0’	‘’	‘0’	‘’	non
a	‘0’	‘00’	‘01’	‘00’	‘01’	con
a	‘00’	”	‘’	‘’	‘’	leaf
a	‘01’	‘’	‘’	‘’	‘’	leaf
b	‘’	‘0’	‘1’	‘00’	‘10’	con
b	‘0’	‘00’	‘’	‘00’	‘’	non
b	‘1’	‘10’	‘’	‘10’	‘’	non
b	‘00’	‘’	‘’	‘’	‘’	leaf
b	‘10’	‘’	‘’	‘’	‘’	leaf

Table II shows all the pointers of the FB-tree corresponding to the full binary keyword tree of Fig. 2. The column ‘Current’ means the path value of the current tree node, ‘ShB1’ means the left short branch, ‘LB1’ means the left long branch, and so on. ‘non’ means a nonconnection node. ‘con’ means a connection node. For example, in the root of the w-tree(w=‘b’), the left long branch points to the leaf ‘00’, and the right long branch points to the leaf ‘10’.

The core concept of FB-tree is the long branches (LBs). Given a w-tree node v, if the left-side subtree of v has connection nodes, there is undoubtedly only one nearest connection node of v. The reason is that, if the subtree has two nearest connection nodes, according to the definition of keyword tree, a new nearest connection node of these two will appear, which is contrary to the assumption. If the left-side subtree of v has no connection nodes and contains nonconnection nodes, there is at most one nonempty leaf in the left subtree. If the subtree has two nonempty leaves, a new connection node will appear, which is also contrary to the assumption. The right long branch is similar to the left long branch. Intuitively, since the long branches can jump over the nonconnection nodes to arrive in the desired leaves quickly, the query processing is very efficient. This is the critical reason why the FB-tree does not suffer from the logarithmic overhead in a search query.

Two short branches exist only logically, and they seem ‘discarded’. To save an FB-tree, we store only the long branches, which can recover the short branches that make the tree compatible with the existing keyword trees.

If the height L is a constant (e.g., $L = 32$), given a keyword w and its corresponding ordered result set $DB(w)$, a full binary w-tree will be uniquely outputted.

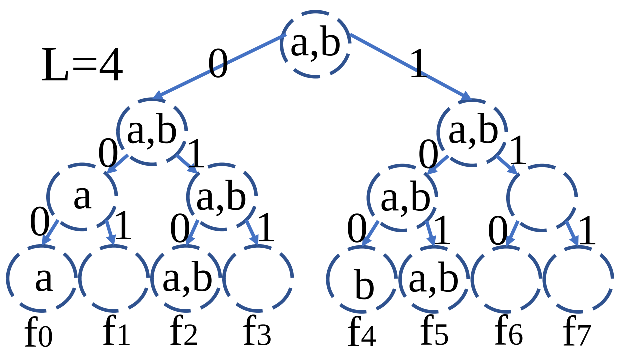


Fig. 3: Full binary keyword tree with two keywords

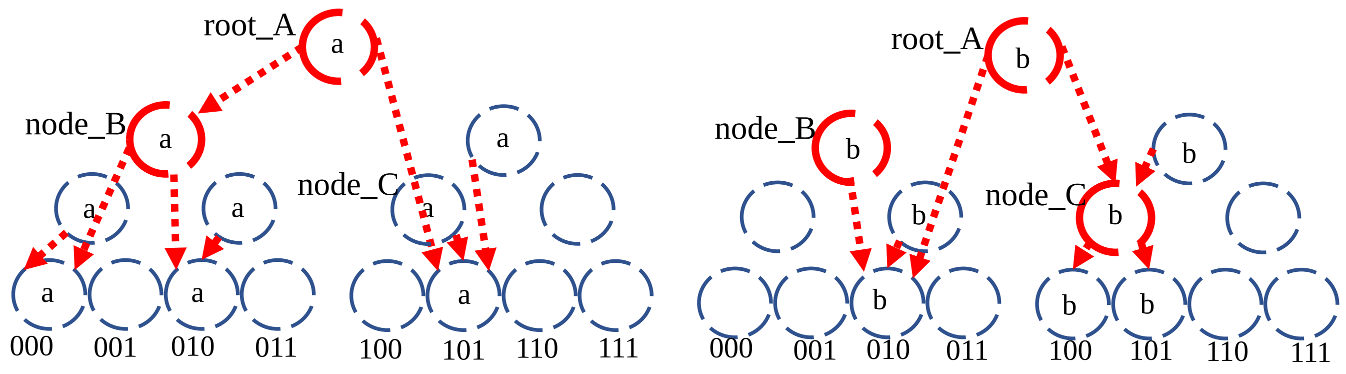


Fig. 4: FB-tree with two keywords

Fig. 3 shows a full binary keyword tree of height $L = 4$ and Fig. 4 is its corresponding FB-tree. File f_0 contains keyword {'a'}, f_2 contains keywords {'a','b'}, and so on. For each nonempty w-tree node, two long branches are added to point to its left or right nearest connection node (or leaves). All short branches are removed. Fig. 4 is an FB-tree containing two w-trees (w='a', w='b'). In w-tree(w='a'), the root node $root\_A$ and its left child $node\_B$ are connection nodes, but in w-tree(w='b'), $node\_B$ is a nonconnection node. In w-tree(w='a'), the left long branch of the root is '0', which points to $node\_B$, and the right long branch of the root is '101', which points to the 5th leaf (the path of the 5th leaf is '101'). If searching for keyword 'a' to fetch $\{f_0,f_2,f_5\}$, the search only needs to access $root\_A(\text{`'},\text{`0'},\text{`101'})_{w=\text{`}a'}$ and $node\_B(\text{`0'},\text{`000'},\text{`010'})_{w=\text{`}a'}$ without accessing nonconnection nodes and leaves, since f_0:'000', f_2:'010', and f_5:'101' are final results. The dotted lines denote that the cloud does not learn how many items there are in each tree node. The next section shows how to encrypt the tree.

An unencrypted FB-tree can be expressed by all nonempty w-tree nodes. Let W denote a set of all possible keywords that can be queried. Given a keyword $w \in W$ and a w-tree, $(p_i^0, p_i^1, p_i^2)_w$ denotes the i-th w-tree node whose left LB is p_i^1, whose right LB is p_i^2, and whose node path is p_i^0. An FB-tree is denoted by $T_{ful} \overset{def}{=} \{(p_1^0, p_1^1, p_1^2)_w, \cdots, (p_r^0, p_r^1, p_r^2)_w\}_{w\in W}$, where r is the number of nonempty nodes of the corresponding w-tree.

C. *Features of an FB-tree*

An FB-tree contains a set of full binary w-trees, where each w-tree has three critical features.

1) The total number of nonempty leaves in a w-tree is $|DB(w)|$.
2) The total number of connection nodes in a w-tree is $|DB(w)| - 1$.
3) All w-trees share the same logical view, which is the full binary tree with a height of L.

The first feature can be deduced according to the definition of the w-tree.

The second feature shows the constitution of a w-tree. If the number of file results is 1, i.e., $|DB(w)| = 1$, the number of connection nodes in the w-tree is zero. Every time one adds a different file that contains keyword w, the number of connection nodes in the w-tree increases precisely one. For example, in Fig. 3, to insert a file f_1 that contains keyword 'a', the newly added connection node is '00'. In the special case, the root node can also turn into a connection node.

The third feature helps a Boolean query to be efficiently processed between different w-trees. A w-tree comes from the corresponding keyword tree. If the original keyword tree is a full binary tree, then this w-tree is still a full binary tree. Therefore, all w-trees share this same logical view. An FB-tree node is a set of w-tree nodes that have the same path value. Note that, the FB-tree node is only a logical concept.

It is not difficult to deduce that the construction complexity of T_{ful} is $O(N \log_2 n)$, where N is the total number of keyword-document pairs (i.e., $N = \sum_{w\in W} |DB(w)|$), and n is the total number of files.

IV. FBTree-1: A Highly Efficient Boolean Query SSE System

This section shows how to encrypt an FB-tree by using only pseudorandom functions. FB-tree-1 has two variants: FB-tree-1a in Algorithm I and FB-tree-1b in Algorithm III (Appendix A). The conjunctive query SSE system FB-tree-1a consists of three polynomial time algorithms, FB-tree-1a=(Setup, Trapdoor, Search), and the CNF-query FB-tree-1b is the extension of the first.

A. *Conjunctive Queries over an FB-tree*

The conjunctive query system consists of three algorithms. 'Setup' is used to create an encrypted FB-tree for later fast query processing. 'Trapdoor' is used to build an encrypted conjunctive query, which enables the cloud to search for files over the index without learning the contents of files and queries. 'Search' is the algorithm conducted by the cloud to get the matching results.

Overview of FB-tree-1. Conjunctive query processing has two methods: jumping and checking. Recall that in an FB-tree, following the long branches, the cloud can jump over nonconnection nodes to quickly reach the target leaves. Jumping means reduced search length. If a search process reaches a leaf (i.e., arriving in a data file), since the jumps of long branches perhaps yield a false result, the cloud should check whether this file matches the conjunctive query to filter the false result. Checking means discarding the incorrect results.

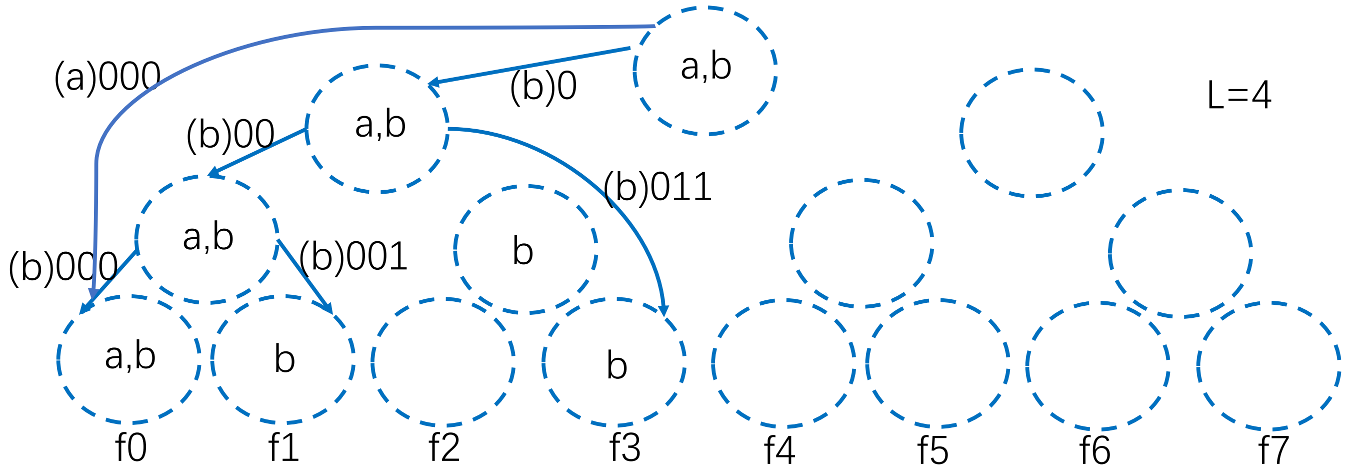


Fig. 5: An example of conjunctive queries over an FB-tree

An example. Fig. 5 shows an example FB-tree of height $L = 4$ when searching for '$a \wedge b$', where '(a)000' denotes the path '000' of the w-tree(w='a'), '(b)0' denotes the path '0' of the w-tree(w='b'), and so on. File f_0 contains two keywords {'a', 'b'}, and f_1 and f_3 contain keyword 'b', respectively. If the conjunctive query 'a $\wedge$ b' is performed, the search steps are as follows. 1) In the root, the search follows the branch '(a)000' instead of '(b)0', since '(a)000' is the longest jump from the current node. Thus, node '0' and node '00' have been jumped over. 2) In the leaf '000' (f_0), the search checks whether the keyword 'b' exists in leaf '000'. In the above processing, only two FB-tree nodes (corresponding to four w-tree nodes) are involved.

This example shows only the logical steps. In untrusted environments, all tree nodes and branches should be encrypted with the following approach.

Setup. Given a set of files with file identifiers which vary from 0 to $n-1$, these files are encrypted and inserted into leaves of an FB-tree from $leaf_0$ to $leaf_{n-1}$. The FB-tree T_{ful} is encrypted and put into a hash table. Let F be a keyed pseudorandom function with a secret key K, and H_1 be a collision-resistant hash function modeled as a random oracle that outputs $(l+2L)$-bit integers. In our experiments, l=160 and L=32. The i-th w-tree node is denoted as (p_i^0, p_i^1, p_i^2), where p_i^0 points to the current node, p_i^1 is the left long branch, and p_i^2 is the right long branch. The w-tree node p_i^0 is first encrypted into $H_1(p_i^0||F_K(w))$. Let $v \leftarrow H_1(node.p_i^0||F_K(w))$. Then, v is partitioned into three parts (a_1, a_2, a_3) in fixed size (l, L, L) (e.g., size $(160, 32, 32)$). Let e be a key-value pair of $e \leftarrow (a_1, (a_2 \oplus p_i^1)||(a_3 \oplus p_i^2))$. Now, the w-tree node is encrypted into the key-value pair e. This is repeated until all w-tree nodes are encrypted. All pairs are stored into the hash table, which forms an encrypted index I. The client finally outsources the encrypted database EDB=(I, D) to the cloud, where D is the set of files encrypted by a CPA-secure scheme.

In this process, all nonconnection nodes and leaves of w-trees are put into the index for future efficient conjunctive queries. Note that, all empty w-tree nodes are discarded, without requiring additional space to save the empty nodes. A w-tree consumes $O(|DB(w)|\log_2 n)$ space. Therefore, the index size of FB-tree-1 is $O(N \log_2 n)$.

Encrypting a Datatable. If the input is a data table of a relational database, such as a MySQL table, each record is converted into a data file with a unique file identifier which varies from 0 to $n-1$. A record with u columns is

Let $T = T_{ful}$ be an unencrypted FB-tree. Consider the following conjunctive query SSE system FB-tree-1a=(Setup, Trapdoor, Search):

```
• Setup($T$):
    1) $K \leftarrow$ KeyGen($1^\lambda$)
    2) for all $node \in T$
        a) $v \leftarrow H_1(node.p_i^0||F_K(w))$
        b) partition v into $(a_1, a_2, a_3)$
        c) $e \leftarrow (a_1, (a_2 \oplus (node.p_i^1))||(a_3 \oplus (node.p_i^2)))$
        d) update $I \leftarrow I \bigcup \{e\}$
    3) send $EDB = (I, D)$ to the cloud
• Trapdoor($\varphi; K$):
    1) parse $\varphi$ as $(w_1, w_2, \cdots, w_u)$
    2) $Q \leftarrow F_K(w_1) \wedge F_K(w_2) \cdots \wedge F_K(w_u)$
    3) send $Q$ to the cloud
• Search($path$, $Q$, $EDB$):
    1) initialize $P_l = \{\}, P_r = \{\}$
    2) for all $q \in Q$
        a) $k \leftarrow H_1(path||q)$
        b) parse $k$ as $(a_1, a_2, a_3)$
        c) $b \leftarrow EDB.I[a_1]$
        d) parse $b$ as $(b_2, b_3)$
        e) $P_l \leftarrow P_l \bigcup \{b_2 \oplus a_2\}$
        f) $P_r \leftarrow P_r \bigcup \{b_3 \oplus a_3\}$
    3) Get the longest path that contains all other paths from $P_l$, and let it be $p_1$, i.e., $p_1 \leftarrow L\_stem(P_l)$.
       If it does not exist, return 'not found'.
    4) If the length of $p_1$ is $L - 1$, do the conjunctive query at the leaf node $p_1$. If this query returns true
       ($\Phi(Q, p_1)$), convert $p_1$ to a file identifier and output a result.
    5) If $p_1$ is not null, then recursively invoke Search($p_1, Q, EDB$)
    6) Get the longest path that contains all other paths from $P_r$, and let it be $p_2$, i.e., $p_2 \leftarrow L\_stem(P_r)$.
       If it does not exist, return 'not found'.
    7) If the length of $p_2$ is $L - 1$, do the conjunctive query at the leaf node $p_2$. If this query returns true
       ($\Phi(Q, p_2)$), convert $p_2$ to a file identifier and output a result.
    8) If $p_2$ is not null, then recursively invoke Search($p_2, Q, EDB$)
```

Algorithm I: conjunctive queries for FB-tree-1a.

considered as a file consisting of distinct keywords $\{`attr_i : w_i'\}_{i \in [1,u]}$, where each $attr_i$ is the attribute name of the record cell w_i. Now, the above algorithm can encrypt the whole data table.

Trapdoor. To search over an FB-tree, a u-dimensional conjunctive query $\varphi = w_1 \wedge w_2 \cdots \wedge w_u$ is encrypted into $Q = F_K(w_1) \wedge F_K(w_2) \cdots \wedge F_K(w_u)$ by the client. The query will be submitted to the cloud later. The goal of the client is to get the resultset $DB(Q)$ (i.e., $DB(\varphi)$).

Search. With the conjunctive search trapdoor Q, the cloud searches over I and returns a set of matched file identifiers to the client. Before doing so, three algorithms are introduced for this work.

$\underline{\Phi(Q, path)}$. This algorithm takes as input an encrypted conjunctive query Q, and a path to represent a keyword tree node. This algorithm outputs whether this tree node matches query Q. Let q_i be one of the subqueries of Q. The value $H_1(path||q_i)$ is partitioned into three parts (a_1, a_2, a_3) in fixed sizes. If $EDB.I[a_1]$ is not null, then the subquery returns true; otherwise, it returns false. The final result of this conjunctive query can be computed by all the results of the subqueries. For example, if a tree node with a corresponding path '001' contains keywords

$\{a, b, c, d\}$, let $Q = F_K(a) \wedge F_K(b) \wedge F_K(c)$. $\mathit{\Phi}(Q,$'001'$)$ returns true. In the special case, if this tree node is a leaf, the query Q with a true result will yield a matched file identifier converted from the path.

This algorithm can be extended to support Boolean queries also. For example, in the above tree node, $\mathit{\Phi}(($'a'$\vee$'e'$)\wedge$'d',$'001')$ returns true.

$\underline{L\_Stem(P)}$. Given a set of paths P, this algorithm outputs the longest path that contains all the other paths by left-aligned matching. This algorithm returns nothing if no paths are matched. For example, $L\_Stem(\{$'01','011','0110'$\})$ returns '0110', but $L\_Stem(\{$'01','011','010'$\})$ returns nothing. To traverse from one tree node to another, a jump of the longest path means the optimal shortcut that can be chosen.

$\underline{S\_Stem(P)}$. Given a set of paths P, this algorithm outputs the longest stemming path that is a substring of all the other paths by left-aligned matching. For example, $S\_Stem(\{$'01','011','0110'$\})$ returns '01', and $S\_Stem(\{$'01','011','010'$\})$ also returns '01'. To traverse from one tree node to another, the longest stemming path means the maximally reduced search length.

Next, the recursive search algorithm over the FB-tree is defined as follows. Given an encrypted conjunctive query Q, the cloud runs Q at the root first by the algorithm $\mathit{\Phi}(Q, path_r)$, where $path_r$ denotes the path of the root. For each subquery in Q, the cloud obtains all left-side long branches P of all subqueries at this node. The cloud uses $L\_Stem(P)$ to determine the next left-side path p_1 to search. The right-side search is the same as the left-side search, and it yields a node of path p_2 to traverse. The cloud recursively conducts this until the search reaches a leaf node whose path length is $L-1$. Let the path of this leaf node be $path_{lf}$. Then, the cloud can use $\mathit{\Phi}(Q, path_{lf})$ to test whether this node is a matched result. If the function returns true, the cloud converts $path_{lf}$ into an integer as a file identifier and outputs a result. This continues recursively until all results are matched. Obviously, the long branch that has reached a leaf does not need to be recomputed in a leaf node. For example, in Fig. 5, the computation of testing whether keyword 'a' exists in leaf f_0 is redundant since the left long branch at the root directly shows that 'a' exists in this leaf.

Through all the jumps, the logarithmic search overhead $(\log n)$ is completely removed. The search complexity does not rely on the number of files n.

Theorem 4.1 (*FB-tree traversing*). Let T_{ful} be an unencrypted FB-tree. For any u-dimensional conjunctive query $\varphi = w_1 \wedge w_2 \cdots \wedge w_u$ to search over T_{ful}, the search process can obtains all final results by traversing at most $u \min_{w\in\varphi} |DB(w)|$ w-tree nodes.

Proof: Consider the following two observations. First, a connection node of a w-tree may be a nonconnection node of another w-tree. According to Feature 2 of the FB-tree, there are u w-trees and at most $\sum_{w\in\varphi}(|DB(w)|-1)$ connection nodes involved in the conjunctive query. Note that, an FB-tree node is a logical concept that corresponds to a set of w-tree nodes that have the same path value. Second, in the worst case, the search process will traverse at most all w-tree connection nodes of a subquery w_x, whose query result size is the smallest among $\{|DB(w)|\}_{w\in\varphi}$. If a searching thread reaches an FB-tree node that does not match the query, this thread completes immediately. Then, the total number of traversed w-tree nodes is no more than $u \min_{w\in\varphi} |DB(w)|$ (corresponding to $\min_{w\in\varphi} |DB(w)|$ FB-tree nodes).

In the special case, if $u = 1$, which denotes a single-keyword query, simply traversing only the connection nodes of the w-tree and the root, the search will fetch all the final results.

B. Search Complexity of FB-tree-1a

According to Theorem 4.1, the search complexity of the u-dimensional conjunctive query is $O(u \min_{w\in\varphi} |DB(w)|)$, which is the nearly optimal search complexity of a conjunctive query SE system. Note that Theorem 4.1 shows only the complexity of an unencrypted FB-tree. If it is encrypted by pseudorandom functions, the complexity is the same as the original FB-tree, because the hash table has $O(1)$ access time. In reality, since $\min_{w\in\varphi} |DB(w)| \ll n$, the search complexity of FB-tree-1a is sublinear.

From the above, the logarithmic multiplicative overhead has been completely removed. If the load factor of the hash table is not considered, the search time is independent of the total number of files. Furthermore, the load factor of the hash table can be dynamically changed when the server is free. That is, to dynamically improve the search efficiency, the cloud can enlarge the hash table by itself without learning any contents of the files.

C. Boolean Queries over an FB-tree

Algorithm III is the extension of Algorithm I to support Boolean queries. For simplicity, a Boolean query is converted to a conjunctive normal form (CNF) $\varphi = ((w_{1,1} \vee w_{1,2} \vee \cdots) \wedge \cdots \wedge (w_{u,1} \vee w_{u,2} \vee \cdots))$. FB-tree-1b includes three algorithms, FB-tree-1b=(Setup, Trapdoor, CNFSearch).

Overview of FB-tree-1b. Recall that in a keyword tree, if a Boolean query matches a tree node, it implies that at least a child tree node (left or right child) matches the Boolean query. Then, the search can be recursively performed until it reaches the leaves.

Two types of jumping algorithms are involved in reducing the search length that begins from the root to leaves. One is L_Stem, which is used in the above conjunctive queries, and the other is S_Stem, which is used for disjunctive queries. Intuitively, a stemming path of a set of paths means that all individual terms of the disjunction share the same traversed tree nodes to arrive in the potential target leaves. The shared tree nodes should be skipped by jumping to the node of the longest stemming path. In summary, L_Stem outputs the longest jump to a target tree node for a conjunctive query, and S_Stem outputs the longest jump to a target tree node for a disjunctive query. A CNF query can use these two algorithms alternately.

An example. Fig. 6 presents a simple FB-tree of height $L = 4$ with two keywords, where file '010' contains {'a', 'b'}, and file '011' contains only {'b'}. All empty tree nodes are ignored. If searching for '$a \wedge b$', the conjunctive query has only two steps. 1) Access Root_A. Since '(a)010' is longer than '(b)01', the next node will be '010'. 2) Access the leaf '010'. Since 'b' is in this node, the search successfully outputs a result of '010'. If searching for '$a \vee b$', the disjunctive query has three steps. 1) Access the root_A. Since '(a)010' and '(b)01' have stemming path '01', the next node will be '01'. 2) Access node '01', jumping over node '0'. 3) Access two nodes {'010', '011'}, and yield the final results.

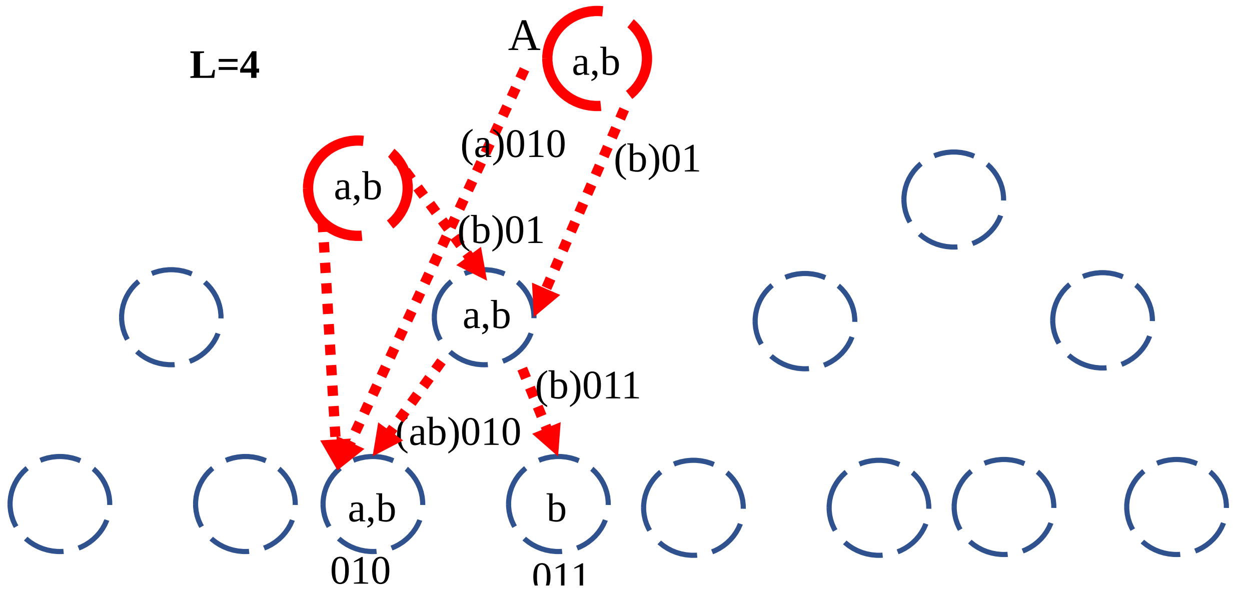


Fig. 6: Example jumping algorithms

Setup. This algorithm is the same as FBTree-1a.Setup.

Trapdoor. Let the CNF query be the following form, $\varphi = \delta_1 \wedge \delta_2 \cdots \wedge \delta_u$, where each $\delta_i = w_{i,1} \vee w_{i,2} \cdots \vee w_{i,v}$. The client encrypts the query into $Q = \Delta_1 \wedge \Delta_2 \cdots \wedge \Delta_u$, where each $\Delta_i = q_{i,1} \vee q_{i,2} \vee \cdots$, and $q_{i,j} = F_K(w_{i,j})$. Δ_i is called an encrypted disjunction of this CNF query. The client sends this query Q to the cloud for matching results $DB(Q)$ (i.e., $DB(\varphi)$).

CNFSearch. The CNF-search pseudocode in Appendix A (Algorithm III) is extended from Algorithm I by slightly modifying the jumping algorithm in each tree node.

D. Search Complexity of FB-tree-1b

Compared with the conjunctive queries, the CNF queries need to process more disjunctions in each tree node. If each disjunction Δ_i is considered as a 'term', the CNF queries can still be viewed as conjunctive queries. In the worst case, the search process traverses all the tree nodes of the disjunction query, whose result set size is the smallest among $\{|DB(\Delta_1)|, |DB(\Delta_2)|, \cdots, |DB(\Delta_u)|\}$. Let v denote the average number of disjunctive terms in each disjunction. There are at most $2uv$ long branches to be processed in each FB-tree node. Then, the search complexity of a u-dimensional CNF query is $O(uv \min_{i \in [1,u]} |DB(\Delta_i)|)$, which is also the nearly optimal search complexity of a CNF-query SSE system.

V. FB-tree-2: Boolean queries via Oblivious RAM

We propose FB-tree-2=(Setup, Search, Update), a new Boolean query system on oblivious RAM that reveals almost nothing to the cloud in any operations (e.g., searches and updates). Although FB-tree-1 is highly efficient, it still leaks the search and access patterns of individual queries. Therefore, stronger constructions should be built to protect device privacy.

A. Sublinear Boolean Queries over Oblivious RAM

FB-tree-2 uses the FB-tree structure and oblivious maps to achieve both good searching/updating efficiency and strong security.

Oblivious Maps. An oblivious map (OMAP) is a privacy-preserving data structure implemented with oblivious RAM techniques that support certain types of operations such as reading and writing. OMAP aims to hide the name and content of the sequence of operations performed on the data structure. The OMAP is secure on the condition that for any same-length two possible arrays of actions, their resulting access patterns (i.e., the sequence of memory addresses accessed) are indistinguishable. We use the OMAP implementation proposed in [35].

OMAP provides the following abstract data reading/writing APIs. The API $\mathbf{v} \leftarrow OMAP.read(\mathbf{k})$ denotes obliviously reading a value $\mathbf{v}$ from $OMAP$ with the hash-key $\mathbf{k}$, and the API $OMAP.write(\mathbf{k}, \mathbf{v})$ denotes obliviously writing the key-value pair $(\mathbf{k}, \mathbf{v})$ into $OMAP$. Assume an FB-tree is fully stored in $OMAP$. That is, $OMAP$ stores $\mathbb{N}$ nonempty w-tree nodes. Note that, here $\mathbb{N} = O(N \log n)$, where N is the number of keyword-document pairs. In the state of the art [35], the data reading or writing time per operation is $t_{oram} = O(\log^2 \mathbb{N})$, and each operation requires $r_{oram} = O(\log \mathbb{N})$ rounds between the server and the client.

Setup. The setup procedure can be implemented within two steps. The client first initializes an oblivious map with size $O(\mathbb{N})$. Next, for each w-tree node $(p_0, p_1, p_2)_w$ in the FB-tree T_{ful}, it invokes $OMAP.write(p_0||w, 1||p_1||p_2)$ to put the tree items into the $OMAP$, where '1' is the indication to denote that the value has not been deleted and $0||p_1||p_2$ means that this pair has been removed. The index-building time is $O(t_{oram} \cdot N \log n)$. This solution still requires $O(r_{oram} \cdot N \log n)$ round interactions between the server and the client.

To easily build the index, the client can create an unencrypted oblivious RAM locally by batch insertion, and next, it encrypts and outsources the server-side ORAM to the cloud. It preserves the client-side ORAM (i.e., the stash) for building search/update queries. In [35], the stash takes $O(\log^2 \mathbb{N})$ client-side storage space.

Search. The Boolean search algorithm via oblivious RAM logically involves two stages, traversing and checking. According to Theorem 4.1, accessing $O(uv \min_{\delta \in \varphi} |DB(\delta)|)$ w-tree nodes, the search will get all pointers needed. It then alternately uses L_STEM and S_STEM algorithms. When reaching a leaf, the search algorithm checks whether this leaf matches the query and outputs a correct result if it returns success. The logical view and jumping algorithms of the oblivious search are the same as those in FB-tree-1b.

The algorithm Search in FB-tree-2 shows the pseudocode to search for files over the oblivious RAM. P_l is the set of left-side pointers of the current node for the Boolean formula φ, and P_r is the set of right-side pointers. The algorithm $choose(.)$ takes as input a set of pointers corresponding to the current FB-tree nodes, and outputs the longest path that can be chosen, as stated in FB-tree-1b. For any Boolean formula $\varphi = \delta_1 \wedge \delta_2 \cdots \wedge \delta_u$, where each $\delta_i = w_{i,1} \vee w_{i,2} \cdots \vee w_{i,v}$, the search of FB-tree-2 consumes $O(t_{oram} \cdot uv \min_{\delta \in \varphi} |DB(\delta)|)$ time and requires $O(r_{oram} \cdot uv \min_{\delta \in \varphi} |DB(\delta)|)$ round interactions. In reality, since $t_{oram} << n$, $r_{oram} << n$, and $\min_{\delta \in \varphi} |DB(\delta)| << n$, the Boolean search complexity is sublinear.

B. Oblivious FB-tree node Updates

This section presents a privacy-preserving dynamic update approach. In FB-tree-1, since the long branches (LBs) should be modified when the corresponding connection nodes are changed, the naive update approach to modifying the encrypted pointers will induce extra leakage that violates the forward and backward privacy. We show, however, that this challenge can be easily addressed via oblivious RAM.

To update a (w, id) pair, the w-tree can be obliviously modified through $O(L)$ node modifications, where L is the height of the tree, w is a keyword, and id is the corresponding file identifier. That is, only the corresponding root-to-leaf nodes are involved, and other nodes remain the same. Adding a new (w, id) pair means that a new connection node appears. Deleting an existing (w, id) pair means that only one connection node disappears. In the deletion, if $n > 1$, only one connection node is changed into a nonconnection node according to Feature 2 of the FB-tree. This node exists in the deepest position along all the root-to-leaf connection nodes. All the old pointers that point to this node should be modified into new values.

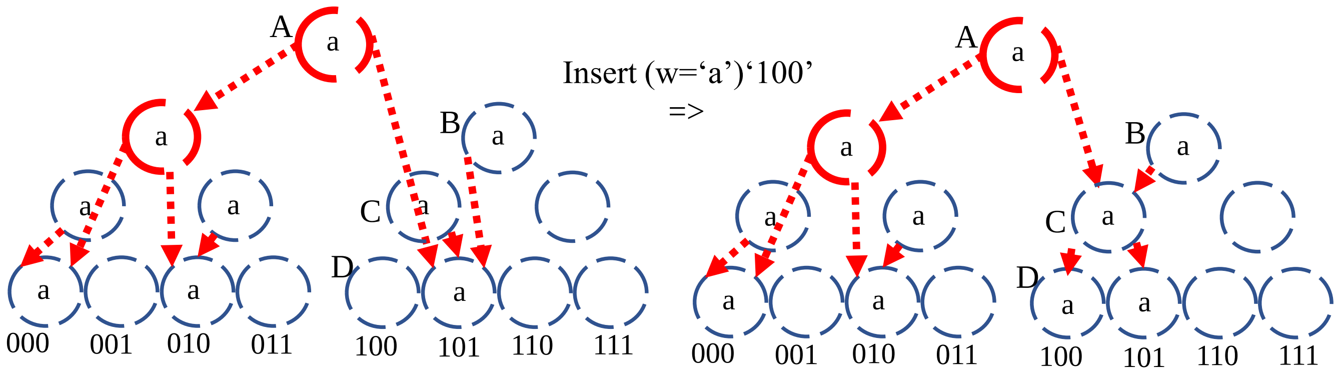


Fig. 7: Insertions via an oblivious RAM

An example. As shown in Fig. 7, the tree contains only one keyword and three files {‘000’,‘010’,‘101’} as well. If a file {‘a’} at ‘100’ is inserted, only four tree nodes {A, B, C, D} are involved. Some LBs of {A, B, C, D} should be modified to satisfy the FB-tree definition since node C turns into a connection node. The deletion can be considered as the addition in reverse. If file (‘a’,‘100’) is deleted from the right-part of Fig. 7, node C is changed into a nonconnection node.

The algorithm Update in FB-tree-2 gives the pseudocode to add a (w, id) pair with op=‘add’ and to delete a (w, id) pair with op=‘del’. This algorithm mainly consists of two steps. First, it retrieves all the root-to-leaf LBs. Second, it finds all the new connection nodes and rebuilds all the changed LBs.

The algorithm $Path(id, i)$ takes as input a file identifier id and the height of the node i in the root-to-leaf path and outputs the path value of the current node. The algorithms $LeftPointer$ and $RightPointer$ are used to rebuild the long-branches involved in additions and deletions, respectively. $LeftPointer$ takes as input the current node value p_0 and the new w-tree logical view after the update, and outputs the left LB of this node. The new w-tree logical view can be constructed from the old items in $OMAP$ and the $(op, (w, id))$ operation. $RightPointer$ is similar to the left-side LB.

When a nonterminal w-tree node has two empty LBs after deletions, it should be logically removed since it becomes an empty w-tree node. Note that, the code $OMAP.write(\mathbf{k}, 0||0||0)$ is used to delete an item by a hash-key **k**, simply marking the node as deleted. When reading the node, the search should first test whether this bit is zero.

Given a (w, id) pair with op=‘add’ or op=‘del’, the update complexity of FB-tree-2 is $O(t_{oram} \cdot L) \approx O(t_{oram} \cdot \log n)$. Updating a pair still requires $O(\log n \log \mathbb{N})$ round interactions, assuming we use $OMAP$ in [35] [2].

VI. Security Analysis

In this section, all possible leakages of FB-tree-1 and FB-tree-2 are analyzed. Scheme security is formally proven later.

A. Leakage Function

The leakage information $\mathcal{L}$ includes two parts: setup leakage $\mathcal{L}_1$, which is leaked to the adversary from the index and files in the setup stage, and query leakage $\mathcal{L}_2$, which is leaked to the adversary in all search queries.

Let T_{ful} be an unencrypted FB-tree, and $OMAP$ be a global oblivious map. FB-tree-2=(Setup, Search, Update) is defined as follows:

```
• Setup($T = T_{ful}$, $OMAP$)
    1) for all $node \in T$
        a) parse $node$ into $((p_0, p_1, p_2), w)$
        b) invoke $OMAP.write(p_0||w, 1||p_1||p_2)$
• Search($path$, $\varphi = \delta_1 \wedge \delta_2 \cdots \wedge \delta_u$, $OMAP$)
    1) initialize $P_l = \{\}, P_r = \{\}$
    2) read $P \leftarrow \{OMAP.read(path||w_{i,j})\}_{i\in[1,u],j\in[1,v]}$
    3) parse $P$ as $(P_l, P_r)$
    4) If $path$ reaches a leaf, test whether $path$ matches $\varphi$. If success, return a result; if failure, return 'not
       found'.
    5) compute $p_1 \leftarrow choose(P_l)$
    6) If $p_1$ is zero, return 'not found'.
    7) compute $p_2 \leftarrow choose(P_r)$
    8) If $p_2$ is zero, return 'not found'.
    9) invoke Search($p_1$,$\varphi$,$OMAP$)
   10) invoke Search($p_2$,$\varphi$,$OMAP$)
• Update($op$, $(w, id)$, $OMAP$)
    1) $T_{up} \leftarrow \{\}$
    2) $P \leftarrow \{Path(id, i)\}_{i\in[0,L-1]}$
    3) Retrieve all nodes of $P$ to determine the new connection nodes and new pointers.
    4) $for\ all\ p_0 \in\ P$
        a) invoke $p_1 \leftarrow LeftPointer(op, (w, id), p_0, OMAP)$
        b) invoke $p_2 \leftarrow RightPointer(op, (w, id), p_0, OMAP)$
        c) if $p_1$ is $empty$ and $p_2$ is $empty$
           if $op$='del', invoke $OMAP.write(p_0||w, 0||0||0)$.
           else
           insert $T_{up} \leftarrow T_{up} \bigcup \{(p_0, p_1, p_2)_w\}$.
    5) invoke Setup($T_{up}, OMAP$)
```

Algorithm II: encrypted Boolean queries via oblivious RAM for FB-tree-2.

Leakage of FB-tree-1. FB-tree-1 has the following leakages, $(\mathcal{L}_1, \mathcal{L}_2)$, where $\mathcal{L}_1$ appears in the setup stage, and $\mathcal{L}_2$ the search stage.

$\mathcal{L}_1$=$\mathcal{L}^{setup}(I)$=$\{\mathbb{N}, L\}$. Given an index I constructed from an FB-tree, this function outputs $\mathbb{N}$, i.e., the number of nonempty w-tree nodes, and L, i.e., the height of the FB-tree.

$\mathcal{L}_1$=$\mathcal{L}^{setup}(D)$=$\{n, \{id_0, id_1, \cdots\}, \{size_0, size_1, \cdots\}\}$. Given an encrypted file set D, this function outputs the number of files n, the file identifiers ranging from 0 to n-1, i.e., $\{id_0, id_1, \cdots\}$, and the sizes of outsourced files encrypted by CPA-secure algorithms, i.e., $\{size_0, size_1, \cdots\}$.

$\mathcal{L}_2$=$\mathcal{L}^{search}(I, \varphi)$=$\{B(\varphi), SP, AP\} \approx \{SP, AP\}$. This function takes as input the index I and a boolean query φ. The function outputs three types of information: the logical operators $B(\varphi)$ of φ (e.g., $\wedge$, $\vee$), the search pattern SP, which indicates whether the same search was performed or not, and the access pattern AP, which is the set of file identifiers matched by each query term of φ. Assume there is a two-dimensional array $\mathbf{A}$ that stores all the historical Boolean query terms in order of arrival. For example, if the first Boolean query is $x \wedge y$, we have $\mathbf{A}[0][0]$ ='x' and $\mathbf{A}[0][1]$ ='y'. Formally, $SP = \{(i, j, k)|for\ all\ i, j, k\ \mathbf{A}[i][j] = \varphi[k]\}$, and $AP = \{DB(w)|for\ all\ w \in \varphi\}$. Note that the traversed paths in a search query are included in AP.

Leakage of FB-tree-2. The leakage of $\mathcal{L}_{oram}$ contains two parts $\mathcal{L}_{oram}$=$(\mathbb{N}, n_\varphi, \mathbb{N}_f, n_f)$. The first part $(\mathbb{N}, n_\varphi)$

comes from the FB-tree ORAM, where $\mathbb{N}$ is the total number of nonempty w-tree nodes in the tree, and n_φ is the total number of memory accesses for a Boolean formula φ. The leakage of $\mathbb{N}$ is unavoidable because it is related to the ORAM size. Exposing n_φ is also unavoidable in a sublinear ORAM scheme because the execution time is proportional to this variable. The second part $(\mathbb{N}_f, n_f)$ comes from the usage of an oblivious file retrieving scheme, where $\mathbb{N}_f$ is the number of file blocks, and n_f is the number of file accesses for φ. Since $\mathbb{N}$ and $\mathbb{N}_f$ can be constants, the leakage function is written as $\mathcal{L}_{oram} \approx (n_\varphi, n_f)$.

B. FB-tree Security

Theorem 7.1 (*IND-CKA2 Security*). If F is a pseudorandom function, and H_1 is a random oracle, FB-tree-1 is IND-CKA2 $(\mathcal{L}_1, \mathcal{L}_2)$-secure.

Proof: Scheme security is proven at a high level. Let us consider a simulator S_1 that can use the leakage $(\mathcal{L}_1, \mathcal{L}_2)$ to adaptively simulate the adversary view, including historical queries, encrypted files, and encrypted index. S_1 simulates the adversary's view through three steps.

In step 1, S_1 first simulates the encrypted index. S_1 creates a simulated index, which is padded with random values to the same size as the original index.

In step 2, S_1 simulates encrypted files. S_1 creates a set of simulated files filled with random values, where the sizes and file identifiers are the same as the originals.

In step 3, S_1 adaptively simulates the query $F_K(w)$, which is one of the trapdoors of the full Boolean query. If the adversary queries the index the i times, the simulator simulates the trapdoor the i times. If this query has appeared before, according to the search pattern $\mathcal{L}_{sp}$, S_1 uses this historical trapdoor for simulation. If the query has not appeared in history, S_1 generates a never used random value q^* using the same size as the original trapdoor to simulate the original.

S_1 now simulates the w-trees. According to the access pattern $\mathcal{L}_{ap}$, S_1 learns $DB(w)$, which can recover a w-tree T. For each of the tree nodes in this w-tree, let the node be v, whose current node path is p_0, whose left long branch is p_1, and whose right long branch is p_2. S_1 randomly chooses a never used key-value pair stored in the simulated hash table, and let this pair be (c_1^*, c_2^*). S_1 partitions c_2^* into two parts c_a^* and c_b^*, i.e., c_2^*=$c_a^*||c_b^*$. Now, simulator S_1 programs the random oracle H_1. S_1 let the output of the random oracle $H_1(p_0||q^*)$ be $(c_1^*||(c_a^* \oplus p_1)||(c_b^* \oplus p_2))$. Then, a w-tree node is simulated. S_1 repeatedly programs the random oracle and chooses entries in the hash table until all the w-tree nodes of T are simulated. S_1 now has generated a simulated w-tree T^*. The simulated trapdoor q^* yields the same outputs as the original trapdoor $F_K(w)$.

Simulator S_1 simulates other parts of the Boolean query by using a similar approach as above.

According to the pseudorandom functions and CPA-secure algorithms, the simulated view, including the index, the files, and the queries, and the original index, cannot be distinguished from the real view, including the files, the index, and the queries, in polynomial time with non-negligible probability. This conclusion implies that the adversary learns only $(\mathcal{L}_1, \mathcal{L}_2)$. Therefore, the theorem is proven.

Theorem 7.2 (*ORAM Security*). If $OMAP$ is an oblivious map, FB-tree-2 is $\mathcal{L}_{oram}$-secure.

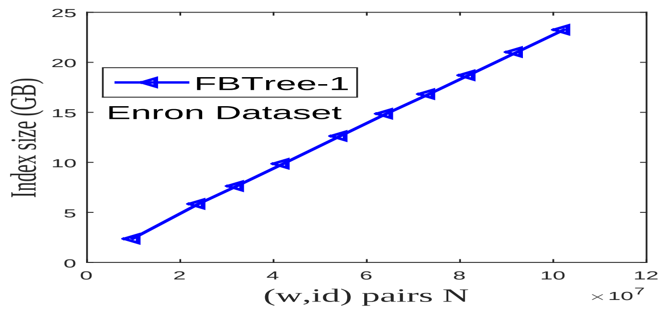
Proof: The security of FB-tree-2 is straightforward since the oblivious map was utilized in a black-box way. In overall FB-tree-2 execution, the cloud observes only a sequence of positions, each chosen uniformly at random. During the Boolean search, since each accessed location is statistically and computationally indistinguishable from random, the server learns only the number of memory accesses n_φ for φ. Since all data files are also put into the obliviously accessing memory, no access patterns are revealed in data file retrieving. Therefore, FB-tree-2 is $\mathcal{L}_{oram}$-secure.

VII. Performance Analysis

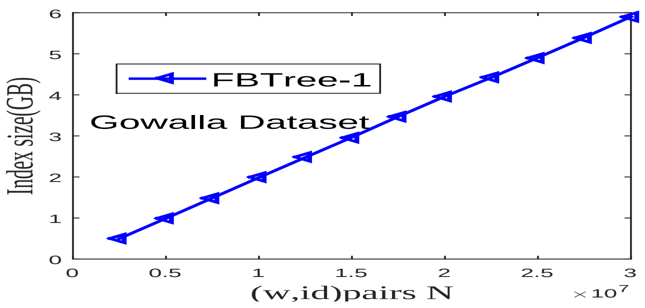
A. Experimental Methodology

The experiments consider three factors: 1) the Boolean query φ; 2) the distribution of the result set; 3) some parameters including the number of files n, the number of keywords m, the number of keyword-document pairs N, and the number of entries in the hash table $\mathbb{N}$. Based on these considerations, several testing cases are used to comprehensively evaluate the construction time, index size, and query time of FB-tree.

Datasets: Four datasets are chosen to evaluate the systems. The first dataset is the Enron email dataset [1], which is also used in [13] and [23]. The Enron dataset contains 517,401 email files, where each file has size varying



(a) Enron dataset



(b) Gowalla dataset

Fig. 8: Index Size of FB-tree-1

from 1 KB to 400 KB. For each document, the text contents are split by the space character, and then $m = 26E5$ distinct keywords are extracted. The second Gowalla dataset [9] consists of 6 millions records, where each record is considered as a file, and the number of (w, id) pairs N is five times the number of records n. The other two datasets come from the commonly-used MySQL sample databases. One is the world-city data table that consists of 4,080 records, and the other is the Sakila-payment data table with 16,050 records.

For simplicity, the symbol $DB(\varphi) = 15$ denotes a query of the expression φ with 15 matched results. 'F1a' denotes FB-tree-1a, and 'F2' indicates FB-tree-2, assuming the index is loaded into memory before evaluations in all the experiments.

Implementation Details: The experiments are conducted on a desktop computer running Windows 10 with 40 GB memory and one Intel(R) Core(TM) i7-6700 CPU @ 3.40GHz processor. Blake2b is chosen as the pseudorandom function. FB-tree-1 and FB-tree-2 are completely implemented in C++. We use the source code provided by Bost in [2] to construct an oblivious map, which is also used in [39].

The height of the FB-tree L is set to 32. Therefore, the tree can hold 2^{31} files at most. The bit length of the pseudorandom function H_1 is 160, and the bit length of the long branches equals 32. Each entry in the hash table consumes 28 bytes. The maximum load factor of the hash table is set to α =80%. Therefore, given 16 GB memory to build an in-memory FB-tree, the tree can be filled with $\mathbb{N} = 4.9E8$ items at most.

Assuming the communication time in an ORAM can be ignored, we test only the computational overhead of FB-tree-2, since the modern technique of hardware enclaves introduced in [38] will help us to handle the communication overhead efficiently.

If the client has at least $O(N \log n)$ memory, it is not hard to build the in-memory tree. If the client is a resource-constrained device, the tree is constructed as follows. First, the client creates an unencrypted inverted index DB for all the files (assuming the client has enough disk space). Second, for each keyword w in dictionary W of DB, the client builds a w-tree from $DB(w)$ and encrypts it. The client submits the encrypted key-value pairs to the cloud in batch mode, with each group the same size.

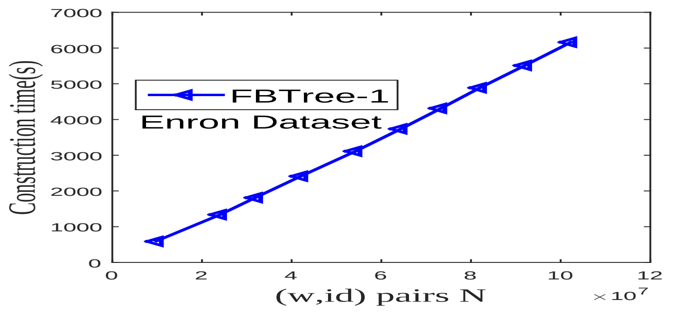
B. *Index Construction Evaluations*

TABLE III: Index size evaluation

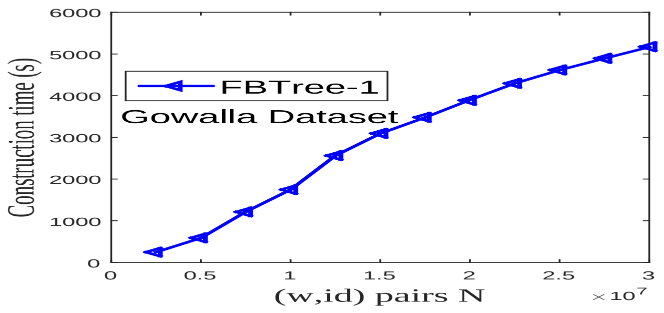
n	m	N	$\mathbb{N}$	I	Dataset
$51E4$	$26E5$	$10E7$	$69E7$	25 GB	FB-tree-1(Enron)
$64E5$	$13E5$	$32E6$	$18E7$	6.7 GB	FB-tree-1(Gowalla)
4080	$13E3$	$2E4$	$45E4$	25 MB	FB-tree-1(MySQL w. city)
16050	$49E3$	$11E4$	$17E5$	100 MB	FB-tree-1(MySQL Sakila)

$\mathbb{N}$ means the number of entries in the hash table. N is the total number of keyword-document pairs. I is the encrypted index.

Testing results in Table III and Fig. 8 show that FB-tree-1 has a practical index size for all the datasets, where Record 1 uses the full Enron dataset with $51E4$ files, Record 2 uses the full Gowalla dataset with $64E5$ data rows, and Records 3 and 4 come from the MySQL sample databases. The FB-tree-2 construction consumes no more disk space than that in FB-tree-1, as shown in Fig. 10(a).

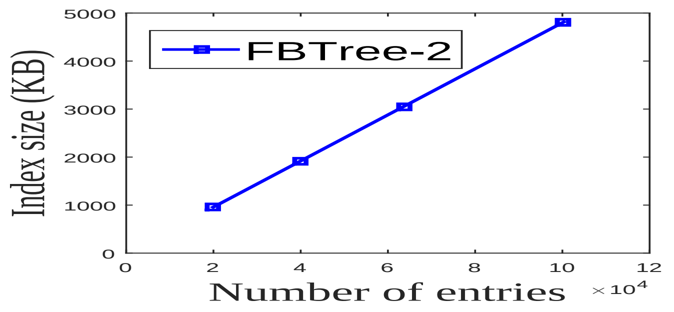


(a) Enron dataset

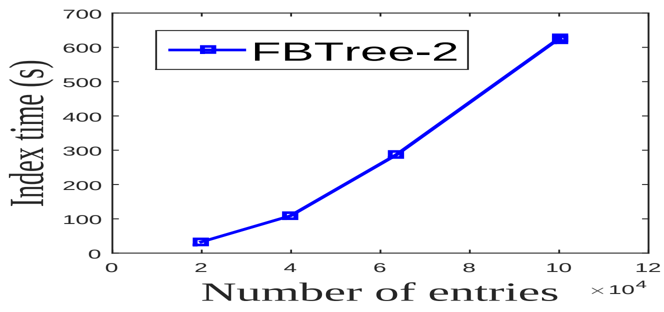


(b) Gowalla dataset

Fig. 9: Construction Time of FB-tree-1



(a) Index size of FB-tree-2



(b) Index time of FB-tree-2

Fig. 10: Index construction of FB-tree-2

Fig. 9 and Fig. 10(b) show the indexing speed. All the experiments are performed by using a single thread. To index the full Enron dataset, the indexing algorithm of FB-tree-1 takes 6,162 seconds with all disk access times considered. The indexing speed of FB-tree-1 is almost 56,072 (w, id) pairs per second. The indexing speed of FB-tree-2 is only 351 (w, id) pairs per second (in batch insertion mode), which is mainly determined by the efficiency of $OMAP$. Optimizing the $OMAP$ in [2] is left to our future work.

The data in Table III prove that MySQL data tables can be practically encrypted by using FB-tree-1. Encrypting the whole sample database of Sakila requires only a few seconds, where 16 relational data tables are involved. An encrypted keyword w at the tree node $path$ has the form of F_K(“$dbName : tbName : clName : w : path$”), where $dbName$, $tbName$, and $clName$ are the database name, the data table name, and the data-column name, respectively.

C. Query Processing Evaluations

Conjunctive queries. The testing results in Fig. 11 show that the query times of FB-tree-1 and FB-tree-2 are practically acceptable. Fig. 11 presents four lines, where ‘F1a,15’ denotes the query ‘x AND w’ with $DB(w) = 15$ and a variable selectivity of x, and so on. If $DB(w) = 15$, with the selectivity of x increasing from 1 to $1E5$, the query processing time of FB-tree-1a increases slightly. If $DB(w) = 2000$, the worst-case upper-bound query time of FB-tree-1a is not large. Although FB-tree-2 is approximately two orders of magnitude slower than FB-tree-1, the above conjunctive query takes only 100 ms, which is far better than the naive idea, downloading the whole database and handling the query locally.

The testing results in Fig. 12 demonstrate that FB-tree is scalable in the number of query terms. Even if the number of query terms reaches $1E5$, the conjunctive queries with form $\varphi = x$ AND $(y_1$ AND $\cdots$ AND $y_u)$, take only 702 ms in FB-tree-1a, where $u = 1E5$, $DB(x) = 2000$, and each $DB(y_i) = 15$.

Fig. 14 demonstrates the high scalability of FB-tree-1. With the number of files growing from 1 million to 6 million, the query time increases little. This is due to the logarithmic search overhead that has been completely removed.

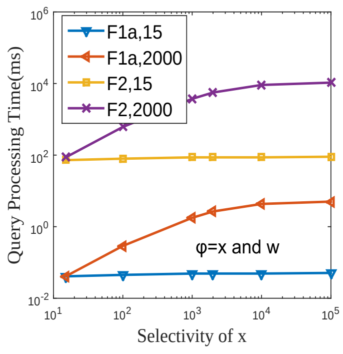


Fig. 11: Conjunctive queries with variable subquery sizes. Number '15' denotes $DB(w) = 15$, and so on.

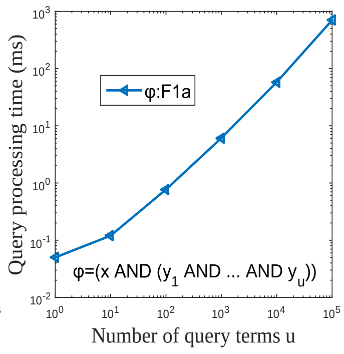


Fig. 12: Conjunctive queries with variable query terms

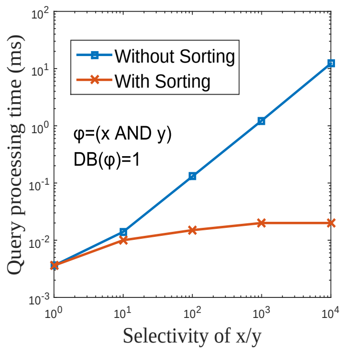


Fig. 13: Data table sorting

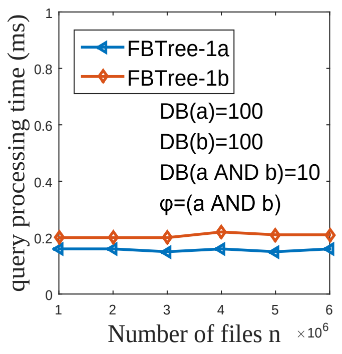


Fig. 14: Conjunctive queries with different index sizes

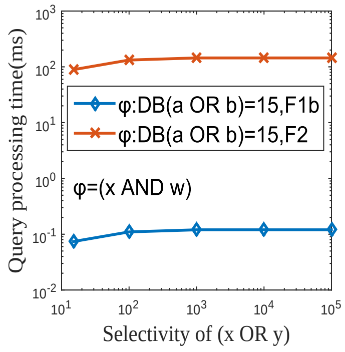


Fig. 15: Boolean Queries

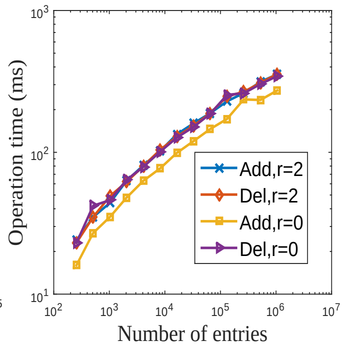


Fig. 16: Add/Delete a (w, id) pair with FB-tree-2

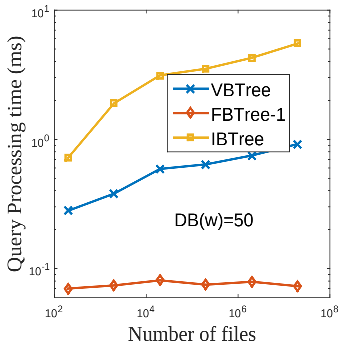


Fig. 17: Compared with VBTree and IBTree on single-keyword queries

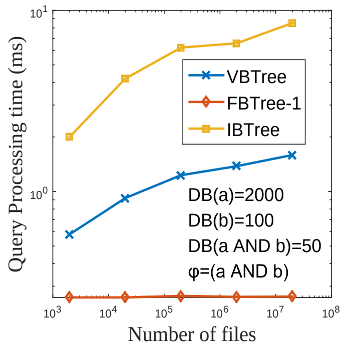


Fig. 18: Compared with VBTree and IBTree on conjunctive queries

Boolean queries. The experimental results in Fig. 15 demonstrate the high scalability of FB-tree-1 and FB-tree-2 regarding Boolean queries. The Boolean queries are of the form φ=((a OR b) AND (x OR y)). This figure shows that the Boolean query time is independent of the selectivity of (x OR y). Although FB-tree-2 uses an oblivious RAM, it is still scalable when the selectivity of one disjunction is not large (e.g., $DB(a \vee b) = 15$).
Sorting the data table. The key difference between an FB-tree and an inverted index is that the tree can be sorted when initializing the index. Consider the following query $\varphi = (x\ AND\ y)$ in Fig. 13, where $DB(x) = DB(y)$ varies from 1 to $1E4$ but $DB(\varphi) = 1$. In the best case, if the data table is sorted initially for the queried data columns, all the results of each query term will be clustered together. The root-to-leaf search will complete quickly after accessing several FB-tree nodes. In the worst case, if all the results are randomly distributed, the root-to-leaf search will run step by step (with none of the logarithmic overhead). Since optimizing the index involves an NP-hard problem of data file partition (DFPP) in [8] [40], we use a simple method of sorting the frequently queried columns for the data table. Fig. 13 shows the advantage of data table sorting.
Data Updates. Fig. 16 demonstrates three points. First, the update time of FB-tree-2 is mainly determined by the efficiency of $OMAP$ either for additions or deletions. Updating a (w, id) pair over an ORAM of capacity 2^{20} takes 357 ms. Second, additions perform almost the same as deletions since these two operations are indistinguishable in an ORAM. Inserting or deleting a (w, id) pair will involve 32 FB-tree nodes since $L = 32$. Third, when a w-tree is empty, it is easier to insert a (w, id) pair into $OMAP$, as shown in Fig. 16, where $r = 2$ denotes there are two nonempty leaves in the w-tree.

FB-tree-1 appears to be a static index. Note that FB-tree-1 can still be updated after a search query since the cloud has the corresponding trapdoors. This feature of FB-tree-1 will help the cloud combine two w-trees that share the same keyword into one w-tree with no additional leakage.

D. Compared with VBTree and Other Trees

Logarithmic search overhead. The key difference between FB-tree-1 and VBTree in [8] is the logarithmic search overhead. As shown in Fig. 17 and Fig. 18, FB-tree is about one order of magnitude faster than VBTree when $1E6$ files are inserted, either for a single-keyword query with 50 results returned in Fig. 17 or for a conjunctive query with 50 identifiers matched in Fig. 18. The comparison is made on the same hardware environment, assuming all the result sets are in the worst-case distribution. IBTree in [7] suffers not only logarithmic overhead but also pseudorandom computational overhead due to Indistinguishable Bloom filters (IBFs). All the other trees still suffer from this logarithmic search overhead (e.g., [3] [6]).

Leaf-to-leaf search. IBTree [7] has a leaf-to-leaf linked list. According to their design, a search can be performed efficiently from one leaf to another step by step when its results cluster together. The leaf-to-leaf search is not a good choice for a keyword tree since most of the results are distributed randomly due to encryption. On most occasions, the leaf-to-leaf search will degrade to a linear scan even if the tree is optimized for searching depth. To avoid this, the search should be performed from top to bottom with all possible jumps.

Index size. The KRB in [3] is inherently unscalable due to the padding in each tree node that makes its index size increases when a large keyword space is provided, as has been deeply discussed in [7] and [8]. For example, a keyword space of $1E6$ keywords and $1E6$ data files of KRB will consume approximately 64 TB disk space. IBTree in [7] is unscalable for unstructured datasets. This stems from the fact that IBTree should pad all the tree nodes (except the root) to the size of the largest file. According to the experimental data in [8], VBTree performs better than KRB and IBTree in terms of index size. The index size of FB-tree is only 1.4 times as much as VBTree (if $L = 32$).

Security. FB-tree-1, KRB, IBTree, and VBTree are all adaptively secure, yet PBTree [6] is non-adaptively secure. Undoubtedly, FB-tree-2 is stronger than all the prior trees. Some comparisons are provided in Table I of the first section.

VIII. Conclusions and Future Work

In this paper, we present a four-branch tree structure and two SE algorithms for different security-efficiency trade-offs. The first is highly efficient and adaptively secure at the cost of the leakage of the search and access patterns, and the second is as strong as an oblivious RAM, but it sacrifices some performance. The designs were tested on several real datasets. The experimental results show that the encryption schemes can gain much efficiency from the FB-tree structure in smart cities.

For future work, we will further improve the performance of FB-tree-2 in reality, including reducing index-building time and update time.

Acknowledgments

This work is supported by the Research Foundation of Education Bureau of Hunan Province, China, under Grant 19C0085, and the National Natural Science Foundation of China (61772454, 61811530332, 61811540410).

References

[1] Enron email dataset. [Online]. Available: http://www.cs.cmu.edu/˜enron/, 2015.

[2] R. Bost, “OpenSSE Schemes,” [Online]. Available: https://github.com/OpenSSE/, 2019.

[3] S. Kamara, and C. Papamanthou, “Parallel and dynamic searchable symmetric encryption,” In International Conference on Financial Cryptography and Data Security, pp. 258-274. Springer, Berlin, Heidelberg, 2013.

[4] V. Pappas, F. Krell, B. Vo, V. Kolesnikov, T. Malkin, S. G. Choi, et al., “Blind seer: A scalable private dbms,” In Symposium on Security and Privacy (S&P), pp. 359-374. IEEE, 2014.

[5] B. A. Fisc, B. Vo, F. Krell, A. Kumarasubramanian, V. Kolesnikov, et al., “Malicious-Client Security in Blind Seer: A Scalable Private DBMS,” In IEEE Symposium on Security and Privacy (S&P). IEEE Computer Society, 2015.

[6] R. Li, A. X. Liu, A. L. Wang, and B. Bruhadeshwar, “Fast range query processing with strong privacy protection for cloud computing,” In Proceedings of the VLDB Endowment, vol. 7, no. 14, pp. 1953-1964, 2014.

[7] R. Li, and A. X. Liu, “Adaptively secure conjunctive query processing over encrypted data for cloud computing,” In 33rd International Conference on Data Engineering (ICDE), pp. 697-708. IEEE, April 2017.

[8] Z. Wu and K. Li, “VBTree: forward secure conjunctive queries over encrypted data for cloud computing,” The VLDB Journal—The International Journal on Very Large Data Bases, vol. 28, no. 1, pp. 25-46, 2019.

[9] E. Cho, S. A. Myers, and J. Leskovec, “Friendship and mobility: user movement in location-based social networks,” In the 17th international conference on Knowledge Discovery and Data mining (SIGKDD), pp. 1082-1090. ACM, 2011.

[10] E. J. Goh, “Secure Indexes,” IACR Cryptology ePrint Archive, Tech. Rep. 2003/216, 2003. [Online]. Available: http://eprint.iacr.org/2003/216

[11] O. Goldreich and R. Ostrovsky, “Software protection and simulation on oblivious rams,” ACM, 1993.

[12] R. Curtmola, J. Garay, S. Kamara, and R. Ostrovsky, “Searchable symmetric encryption: improved definitions and efficient constructions,” In Proc. of Computer and Communications Security (CCS), pp. 79-88. ACM, 2006.

[13] S. Kamara, C. Papamanthou, and T. Roeder, “Dynamic searchable symmetric encryption,” In Proceedings of Computer and communications security (CCS), pp. 965-976, 2012.

[14] B. Bezawada, A. X. Liu, B. Jayaraman, A. L. Wang, and R. Li, “Privacy preserving string matching for cloud computing,” In 35th International Conference on Distributed Computing Systems (ICDCS), pp. 609-618. IEEE, 2015.

[15] M. Van Dijk, C. Gentry, S. Halevi, and V. Vaikuntanathan, “Fully homomorphic encryption over the integers,” In Annual International Conference on the Theory and Applications of Cryptographic Techniques, pp. 24-43. Springer, Berlin, Heidelberg, May 2010.

[16] K. Xie, X. Ning, X. Wang, S. He, Z. Ning, X. Liu, et al., “An efficient privacy-preserving compressive data gathering scheme in WSNs,” Information Sciences, vol. 390, pp. 82-94. Elsevier, 2017.

[17] C. Mavroforakis, N. Chenette, A. O’Neill, G. Kollios, and R. Canetti, “Modular order-preserving encryption, revisited,” In Proceedings of the SIGMOD International Conference on Management of Data, pp. 763-777. ACM, 2015.

[18] M. Chase, and S. Kamara, “Structured encryption and controlled disclosure,” In International Conference on the Theory and Application of Cryptology and Information Security, pp. 577-594. Springer, Berlin, Heidelberg, December 2010.

[19] Z. Chang, D. Xie, and F. Li, “Oblivious ram: a dissection and experimental evaluation,” In Proceedings of the VLDB Endowment, vol. 9, no.12, pp. 1113-1124, 2016.

[20] D. Cash, J. Jaeger, S. Jarecki, C. S. Jutla, H. Krawczyk, M. C. Rosu, and M. Steiner, “Dynamic searchable encryption in very-large databases: data structures and implementation,” In Network & Distributed System Security Symposium (NDSS), vol. 14, pp. 23-26, February 2014.

[21] R. Bost, “$\Sigma o\varphi o\varsigma$: Forward Secure Searchable Encryption,” In Sigsac Conference on Computer and Communications Security (CCS), pp. 1143-1154. ACM, 2016.

[22] M. S. Islam, M. Kuzu, and M. Kantarcioglu, “Access Pattern disclosure on Searchable Encryption: Ramification, Attack and Mitigation,” In Network & Distributed System Security Symposium (NDSS), vol. 20, pp. 12, February 2012.

[23] M. Naveed, M. Prabhakaran, and C. A. Gunter, “Dynamic searchable encryption via blind storage,” In Symposium on Security and Privacy, pp. 639-654. IEEE, May 2014.

[24] S. Kamara, and T. Moataz, “SQL on structurally-encrypted databases,” In International Conference on the Theory and Application of Cryptology and Information Security, pp. 149-180. Springer, Cham, 2018.

[25] A. Ben-David, N. Nisan, and B. Pinkas, “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, October 2008.

[26] K. Kurosawa, and Y. Ohtaki, “UC-secure searchable symmetric encryption,” In International Conference on Financial Cryptography and Data Security, pp. 285-298. Springer, Berlin, Heidelberg, February 2012.

[27] D. Cash, S. Jarecki, C. Jutla, H. Krawczyk, M. C. Rosu, and M. Steiner, “Highly-scalable searchable symmetric encryption with support for Boolean queries,” In Annual Cryptology Conference, pp. 353-373, Springer, Berlin, Heidelberg, August 2013.

[28] Y. Zhang, J. Katz, and C. Papamanthou, “All your queries are belong to us: The power of file-injection attacks on searchable encryption,” In 25th USENIX Security Symposium, pp. 707-720, 2016.

[29] S. Garg, P. Mohassel, and C. Papamanthou, “TWORAM: efficient oblivious RAM in two rounds with applications to searchable encryption,” In Annual International Cryptology Conference, pp. 563-592. Springer, Berlin, Heidelberg, August 2016.

[30] R. Bost, P. A. Fouque, and D. Pointcheval, “Verifiable Dynamic Symmetric Searchable Encryption: Optimality and Forward Security,” IACR Cryptology ePrint Archive, 2016.

[31] S. Kamara and T. Moataz, “Boolean Searchable Symmetric Encryption with Worst-Case Sub-linear Complexity,” In European Cryptology Conference (EUROCRYPT). Springer, 2017.

[32] K. S. Kim, M. Kim, D. Lee, J. H. Park, and W.-H. Kim. “Forward secure dynamic searchable symmetric encryption with efficient updates,” In Proc. of SIGSAC Conference on Computer and Communications Security (CCS), pp. 1449-1463. ACM, 2017.

[33] X. Song, C. Dong, D. Yuan, Q. Xu, and M. Zhao,“Forward private searchable symmetric encryption with optimized i/o efficiency,” Transactions on Dependable and Secure Computing (TDSC). IEEE, 2018.

[34] Z. Wu, K. Li, Keqin Li, J. Wang, “Fast Boolean Queries With Minimized Leakage for Encrypted Databases in Cloud Computing,” IEEE Access, vol. 7, pp. 49418-49431, 2019.

[35] X. S. Wang, K. Nayak, C. Liu, T. H. Chan, E. Shi, E. Stefanov, and Y. Huang, “Oblivious data structures,” In Proc. of the SIGSAC Conference on Computer and Communications Security, pp. 215-226. ACM, November 2014.

[36] D. S. Roche, A. Aviv, and S. G. Choi, “A practical oblivious map data structure with secure deletion and history independence,” In Symposium on Security and Privacy (S&P), pp. 178-197. IEEE, May 2016.

[37] C. Hao, C. Ilaria, and R. Ling, “Efficient Constant Bandwidth Oblivious RAM from (Leveled) TFHE,” in Proc. of the 2019 ACM SIGSAC Conference on Computer and Communications Security (CCS), 2019, pp. 345-360.

[38] S. Eskandarian and M. Zaharia, “An efficient oblivious database for the public cloud,” In Proc. of the VLDB Endowment, vol. 11, no. 4, 2017.

[39] J. G. Chamani, et al., “New Constructions for Forward and Backward Private Symmetric Searchable Encryption,” In Proc. of SIGSAC Conference on Computer and Communications Security (CCS). ACM, 2018.

[40] W. Li, Z. Chen, X. Gao, et al., “Multimodel Framework for Indoor Localization Under Mobile Edge Computing Environment,” IEEE Internet of Things Journal (IoT), vol. 6, no. 3, pp. 4844-4853, June 2019.

[41] M. S. Lacharite, B. Minaud, and K. G. Paterson, “Improved reconstruction attacks on encrypted data using range query leakage,” In Symposium on Security and Privacy (S&P), pp. 297-314. IEEE, May 2018.
[42] V. Bindschaedler, P. Grubbs, D. Cash, T. Ristenpart, and V. Shmatikov, “ The tao of inference in privacy-protected databases,” In Proceedings of the VLDB Endowment, vol. 11, no. 11, pp. 1715-1728, 2018.
[43] W. Li, Y. Xiao, C. Tang, et al., “Multi-user searchable encryption voice in home IoT,” Internet of Things (IoT), 2020, vol. 11. https://doi.org/10.1016/j.iot.2020.100180.

APPENDIX A
CNF-SEARCH ALGORITHM OF FB-TREE-1B

Let $Q = \Delta_1 \wedge \Delta_2 \cdots \wedge \Delta_u$ be an encrypted CNF query, where each $\Delta_i = q_{i,1} \vee q_{i,2} \vee \cdots$. The CNF query algorithm over an encrypted FB-tree is defined as follows:

```
• CNFSearch($path$, $Q = \Delta_1 \wedge \Delta_2 \cdots \wedge \Delta_u$, $EDB$):
    1) initialize $P_l = \{\}, P_r = \{\}$
    2) for all $\Delta_i \in \Delta_1 \wedge \Delta_2 \cdots \wedge \Delta_u$
        a) initialize $P_i^l = \{\}, P_i^r = \{\}$
        b) for all $q_{i,j} \in \Delta_i$
            i) $k \leftarrow H_1(path||q_{i,j})$
            ii) parse $k$ as $(a_1, a_2, a_3)$
            iii) $b \leftarrow EDB.I[a_1]$
            iv) if $b$ does not exist, remove $q_{i,j}$ from $\Delta_i$, i.e., $\Delta_i \leftarrow \Delta_i - \{q_{i,j}\}$
            v) parse $b$ as $(b_2, b_3)$
            vi) $P_i^l \leftarrow P_i^l \bigcup \{b_2 \oplus a_2\}$
            vii) $P_i^r \leftarrow P_i^r \bigcup \{b_3 \oplus a_3\}$
        c) Get the longest stemming path from $P_i^l$, let it be $p_i^l$, i.e., $p_i^l \leftarrow S\_Stem(P_i^l)$.
        d) $P_l \leftarrow P_l \bigcup \{p_i^l\}$
        e) Get the longest stemming path from $P_i^r$, let it be $p_i^r$, i.e., $p_i^r \leftarrow S\_Stem(P_i^r)$.
        f) $P_r \leftarrow P_r \bigcup \{p_i^r\}$
        g) if $|\Delta_i| = 0$, return 'not found'.
    3) Get the longest path that contains all other paths from $P_l$, and let it be $p_1$, i.e., $p_1 \leftarrow L\_Stem(P_l)$.
       If it does not exist, return 'not found'.
    4) If the length of $p_1$ is $L-1$, do the Boolean query at the leaf node $p_1$. If this query returns true
       ($\Phi(Q, p_1)$), convert $p_1$ to a file identifier and output a result.
    5) If $p_1$ is not null, then recursively invoke CNFSearch($p_1$, $\Delta_1 \wedge \Delta_2 \cdots \wedge \Delta_u$, $EDB$).
    6) Get the longest path that contains all other paths from $P_r$, and let it be $p_2$, i.e., $p_2 \leftarrow L\_Stem(P_r)$.
       If it does not exist, return 'not found'.
    7) If the length of $p_2$ is $L-1$, do the Boolean query at the leaf node $p_2$. If this query returns true
       ($\Phi(Q, p_2)$), convert $p_2$ to a file identifier and output a result.
    8) If $p_2$ is not null, then recursively invoke CNFSearch($p_2$, $\Delta_1 \wedge \Delta_2 \cdots \wedge \Delta_u$, $EDB$).
```

Algorithm III: CNF queries for FB-tree-1b.

CNFSearch. Algorithm III gives the pseudocode to search for files over an FB-tree with CNF queries. The algorithm mainly contains three parts: conduct the CNF query at a tree node; search for the left part and right part longest paths to jump over more tree nodes; recursively invoke this algorithm at the left and right sides.

The cloud conducts the CNF query at the root first by applying algorithm $\Phi(\Delta_1 \wedge \Delta_2 \cdots \wedge \Delta_u, path)$, where path=‘’ means the root node. For each disjunction Δ_i, the cloud processes every disjunctive query. From these queries, the cloud gets all the left-side long branches P_i^l in the current node, and gets all the right-side long branches P_i^r in the current node. It computes the stemming path by algorithm S_Stem, i.e., $p_i^l \leftarrow S\_Stem(P_i^l)$ and $p_i^r \leftarrow S\_Stem(P_i^r)$. The path p_i^l or p_i^r represents the jump destination to the left-side node or the right-side node for disjunctive queries of Δ_i, respectively. Next, it processes all conjunctive queries. Let $P_l \leftarrow P_l \bigcup\{p_i^l\}$ and $P_r \leftarrow P_r \bigcup\{p_i^r\}$. It now has a set of paths P_l for the left-side subtree, and a set of paths P_r for the right-side subtree. Get the longest path for the shortest search length in both left-side and right-side, i.e., $p_1 \leftarrow L\_Stem(P_l)$ and $p_2 \leftarrow L\_Stem(P_2)$. With these two paths, the cloud can recursively search the left-side node p_1 and the right-side node p_2. If a search process reaches a leaf whose path length is $L-1$, the cloud should use the algorithm Φ to test whether this CNF query matches the leaf. If it returns true, it outputs a file identifier converted from the current path. Note that, if a long branch reaches a leaf, the corresponding disjunctive query should be removed from Q to reduce the computations in step (4) $\Phi(Q, p_1)$ and step (7) $\Phi(Q, p_2)$.

If one disjunctive query $q_{i,j}$ of Δ_i does not match a tree node, according to the definition of the keyword tree, the left-side subtree and right-side subtree of this node certainly match nothing of $q_{i,j}$. For more efficiency, this disjunctive query should be removed from the full CNF query for the next searches, i.e., $\Delta_i \leftarrow \Delta_i - \{q_{i,j}\}$.