Fast and Scalable Range Query Processing With Strong Privacy Protection for Cloud Computing

Rui Li, Alex X. Liu, Ann L. Wang, and Bezawada Bruhadeshwar

***Abstract*—Privacy has been the key road block to cloud computing as clouds may not be fully trusted. This paper is concerned with the problem of privacy-preserving range query processing on clouds. Prior schemes are weak in privacy protection as they cannot achieve index indistinguishability, and therefore allow the cloud to statistically estimate the values of data and queries using domain knowledge and history query results. In this paper, we propose the first range query processing scheme that achieves index indistinguishability under the *indistinguishability against chosen keyword attack* (IND-CKA). Our key idea is to organize indexing elements in a complete binary tree called PBtree, which satisfies *structure indistinguishability* (i.e., two sets of data items have the same PBtree structure if and only if the two sets have the same number of data items) and *node indistinguishability* (i.e., the values of PBtree nodes are completely random and have no statistical meaning). We prove that our scheme is secure under the widely adopted IND-CKA security model. We propose two algorithms, namely PBtree traversal width minimization and PBtree traversal depth minimization, to improve query processing efficiency. We prove that the worst-case complexity of our query processing algorithm using PBtree is $O(|R| \log n)$, where n is the total number of data items and R is the set of data items in the query result. We implemented and evaluated our scheme on a real-world dataset with 5 million items. For example, for a query whose results contain 10 data items, it takes only 0.17 ms.**

***Index Terms*—Cloud computing, privacy preserving.**

Manuscript received October 19, 2014; revised May 25, 2015; accepted June 28, 2015; approved by IEEE/ACM TRANSACTIONS ON NETWORKING Editor S. Chen. Date of publication August 25, 2015; date of current version August 16, 2016. This work was supported in part by the National Natural Science Foundation of China under Grants No. 61370226, No. 61472184, No. 61321491, and No. 61272546; the Young Teacher Growth Plan of Hunan University; the China Postdoctoral Science Foundation; the Jiangsu Future Internet Program under Grant No. BY2013095-4-08; the Jiangsu High-level Innovation & Entrepreneurship (Shuangchuang) Program; and the National Science Foundation under Grant No. CNS-1017598. The preliminary version of this paper titled "Fast Range Query Processing with Strong Privacy Protection for Cloud Computing" was published in the Proc. 40th International Conference on Very Large Data Bases (VLDB), Vol. 7, No. 14, Hangzhou, China, September 2014. *(Corresponding author: A. X. Liu.)*

R. Li and A. X. Liu are with the College of Computer Science and Technology, Dongguan University of Technology, Dongguan 523808, China (e-mail: ruili.hunan@yahoo.com; alexliuxy@yahoo.com).

A. L. Wang is with the Department of Computer Science and Engineering, Michigan State University, East Lansing, MI 48824 USA (e-mail: liyanwan@cse.msu.edu).

B. Bruhadeshwar is with the National Key Laboratory for Novel Software Technology, Nanjing University, Nanjing 210023, China (e-mail: bru@nju.edu.cn).

Color versions of one or more of the figures in this paper are available online at http://ieeexplore.ieee.org.

Digital Object Identifier 10.1109/TNET.2015.2457493

I. INTRODUCTION

A. Background and Motivation

DRIVEN by lower cost, higher reliability, better performance, and faster deployment, data and computing services have been increasingly outsourced to clouds such as Amazon EC2 and S3 [1], Microsoft Azure [3], and Google App Engine [2]. However, privacy has been the key roadblock to cloud computing. On one hand, to leverage the computing and storage capability offered by clouds, we need to store data on clouds. On the other hand, due to many reasons, we may not fully trust the clouds for data privacy. First, clouds may have corrupted employees who do not follow data privacy policies. For example, in 2010, a Google engineer broke into the Gmail and Google Voice accounts of several children [4]. Second, cloud computing systems may be vulnerable to external malicious attacks, and when intrusions happen, cloud customers may not be fully informed about the potential implications on the privacy of their data. Third, clouds may base services on facilities in some foreign countries where privacy regulations are difficult to enforce.

In this paper, we consider the following popular cloud computing paradigm: A data owner stores data on a cloud, and multiple data users query the data. For a simple example, a user stores his own data and queries his own data on the cloud. For another example, multiple doctors in a clinic store query patient medical records in a cloud. Fig. 1 shows the three parties in our model: a data owner, a cloud, and multiple data users. Among the three parties, the data owner and data users are trusted, but the cloud is not fully trusted. The problem addressed in this paper is range query processing on clouds in a privacy-preserving and yet scalable manner. For a set of records where all records have the same attribute $\mathbb{A}$, which has numerical values or can be represented as numerical values, given a range query specified by an interval $[a, b]$, the query result is the set of records whose $\mathbb{A}$ attribute falls into the interval. Range queries are fundamental operations for database SQL queries and big data analytics. In database SQL queries, the `where` clauses often contain predicates specified as ranges. For example, SQL query `select * from patients where 20 <= age and age <= 30}` means to find all records of the patients whose age is in the range of [20, 30]. In big data analytics, many analyses involve range queries along dimensions such as time and human age.

Given data items $d_1, \cdots, d_n$, the data owner encrypts these data using a symmetric key K, which is shared between the

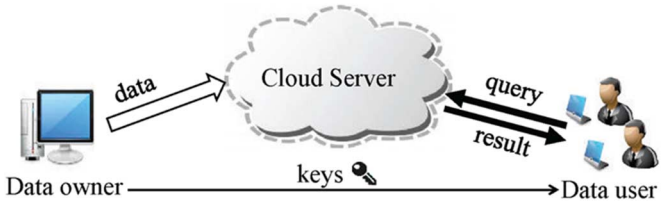

Fig. 1. Cloud computing model.

data owner and data users, generates an *index*, and then sends both the encrypted data denoted $(d_1)_k, \cdots, (d_n)_k$ and the index to the cloud. Given a query, the data user generates a *trapdoor* and then sends it to the cloud. The index and the trapdoor should allow the cloud to determine which data items satisfy the query. Yet, in this process, the cloud should not be able to infer *useful information* about the data and queries. The useful information in this context includes the values of the data items, the content of the queries, and the statistical properties of the data items. Other than encrypted data and queries, together with query results, the cloud may have information obtained from other channels, such as domain knowledge about the data (e.g., age distribution). However, even with such information, a privacy-preserving range query scheme should not allow the cloud to infer additional information about the data based on past query results.

Besides privacy guarantees, a privacy-preserving range query scheme should be efficient in terms of query processing time, storage overhead, and communication overhead. The query processing time needs to be small because many applications require real-time queries. The storage overhead refers to the data that cloud needs to store other than encrypted data items. It needs to be small because the volume of data stored on the cloud is typically large. The communication overhead refers to the data transferred between the data owner and the cloud, other than encrypted data items, and the data transferred between data users and the cloud, other than the precise query results. It needs to be small due to bandwidth limitations and the extra time involved in uploading and downloading.

B. Threat Model

For the cloud, we assume that the cloud is *semi-honest* (also called *honest-but-curious*), which was proposed by Canetti *et al.* in [13] and has been widely adopted including prior privacy-preserving range and keyword query work [8]–[11], [15], [16], [18], [23]–[26], [30], [37]. A cloud is semi-honest means that it does follow required communication protocols and execute required algorithms correctly, but it may attempt to obtain information about data items and the content of user queries with the help of domain knowledge about the data items and the queries (such as the distribution of data items and queries). For the data owner and the data users, we assume that they are trusted.

C. Security Model

We adopt the IND-CKA security model proposed in [18], which has been widely accepted in prior privacy-preserving keyword query work. This model has two key requirements: *index indistinguishability* (IND) and *security under chosen keyword attacks* (CKA). Informally, a range query scheme is secure under the IND-CKA model if an adversary $\mathcal{A}$ chooses two different sets S_1 and S_2 of data items, where the two sets have the same number of data items and they may or may not overlap, lets an oracle simulating the data owner to build indexes for S_1 and S_2, but $\mathcal{A}$ cannot distinguish which index is for which dataset. The rationale is that if the problem is distinguishing the indexes for S_1 and S_2 is hard, then deducing at least one data item that S_1 and S_2 do not have in common must also be hard. In other words, if $\mathcal{A}$ cannot determine which data item is encoded in an index with probability non-negligibly different from 1/2, then the index reveals nothing about the data items. Such indexes are called *secure indexes*. The IND-CKA model aims to prevent an adversary $\mathcal{A}$ from deducing the plaintext values of data items from the index, other than what it already knows from previous query results or from other channels. Note that secure indexes do not hide information such as the number of data items. For applications that demand the privacy of data item numbers, they can inject dummy data items into small datasets to make all datasets to have equal sizes. Also, note that we are not interested in hiding search patterns, where a search pattern is defined as the set of trapdoors corresponding to different user queries. So far, there are no searchable symmetric encryption schemes that can hide the statistical patterns of user searches because trapdoors are generated deterministically (i.e., the same trapdoor will always be generated for the same keyword) [26].

D. Summary and Limitation of Prior Art

Prior privacy-preserving query schemes fall into two categories according to their query types: range queries, which query all data items that fall into a given range, and keyword queries, which query all text documents that contain a given keyword. Privacy-preserving range query schemes can also be called *range searchable symmetric encryption schemes*, and privacy-preserving keyword query schemes can also be called *keyword searchable symmetric encryption schemes*. Prior privacy-preserving range query schemes for the single-data-owner–multiple-data-user cloud paradigm fall into two categories: bucketing schemes [23]–[25] and order-preserving schemes [9], [10], [30]. In bucketing schemes, the data owner partitions the whole data domain (e.g., [0, 150] of human ages) into multiple buckets of varying sizes (e.g., 4 buckets of [0, 12], [13, 22], [23, 60], [61, 150]). The index consists of pairs of a bucket ID and the encrypted data items in the bucket. The trapdoor of a range query (e.g., [10, 20]) consists of the IDs of the buckets that overlaps with the range (e.g., bucket IDs 1 and 2). All data items in a bucket are included in the query result as long as the bucket overlaps with the query. Bucketing schemes have two key limitations: weak privacy protection and high communication cost. Their privacy protection is weak because the cloud can statistically estimate the actual value of both data items and queries using domain knowledge and historical query results, as pointed out in [25]. Their communication cost is high because many data items in the query result do not satisfy the query. Reducing bucket sizes helps to reduce communication costs, but will worsen privacy protection because the number of buckets becomes closer to that of data items.

Order-preserving schemes use encryption functions that preserve the relative ordering of data items even after encryption. For any two data items a and b, and an order-preserving encryption function f, $a \leq b$ if and only if $f(a) \leq f(b)$. In order-preserving schemes, the index for data items $d_1, \cdots, d_n$ is $f(d_1), \cdots, f(d_n)$, and the trapdoor for query $[a, b]$ is $[f(a), f(b)]$. Order-preserving schemes have weak privacy protection because they allow the cloud to statistically estimate the actual values of both data items and queries [5].

The fundamental reason that the privacy protection provided by the above prior schemes is weak is because their indexes are distinguishable for the same number of data items but with different distributions. In bucketing schemes, for the same number of data items, different distributions in data values will cause buckets to have different distributions in sizes because they need to balance the number of items among buckets. In order-preserving schemes, for the same number of data items, different distributions in data values will cause cipher-texts to have different distribution in the projected space. Leveraging domain knowledge about data distribution, both bucketing schemes and order-preserving schemes allow the cloud to statistically estimate the values of data and queries.

E. Proposed Approach

In this paper, we propose the first privacy-preserving range query scheme that achieves index indistinguishability. Our key idea for achieving index indistinguishability is to organize all indexing elements in a complete binary tree where each node is represented using a Bloom filter, which we call a *PBtree* (where "P" stands for privacy and "B" stands for Bloom filter). PBtrees allow us to achieve index indistinguishability because they have two important properties. First, a PBtree has the property of *structure indistinguishability*, that is, two sets of data items have the same PBtree structure if and only if the two sets have the same number of data items. The structure of the PBtree of a set of data items is determined solely by the set cardinality, not the value of data items. Second, a PBtree has the property of *node indistinguishability*, that is, for any two PBtrees constructed from datasets of the same cardinality, which have the same structure, and for any two corresponding nodes of the two PBtrees, the values of the two nodes are not distinguishable. Thus, our scheme prevents a cloud from performing statistical analysis on the index even with domain knowledge.

F. Technical Challenges and Solutions

There are two key technical challenges. The first challenge is the *construction* of PBtrees by data owners. We address this challenge by first transforming less-than and bigger-than comparisons into set membership testing (i.e., testing whether a number is in a set), which involves only equal-to comparisons, and then organize all the sets hierarchically in a PBtree. This transformation helps us to achieve node indistinguishability because the less-than or bigger-than relationship among PBtree nodes is no longer statistically meaningful. The second challenge is the *optimization* of PBtrees for fast query processing on the cloud. We address this challenge by two ideas: *PBtree traversal width minimization* and *PBtree traversal depth minimization*. The idea of *PBtree traversal width minimization* is to minimize the number of paths that the cloud needs to traverse for processing a query. We prove that the PBtree traversal width minimization problem is NP-hard, and propose an efficient approximation algorithm. The idea of *PBtree traversal depth minimization* is to minimize the traversal depth of the paths that the cloud needs to traverse for processing a query; in other words, we want the traversal of many paths to terminate as early as possible.

G. Key Contributions

We make three key contributions. First, we propose the first privacy-preserving range query scheme and prove that it is secure under the widely adopted IND-CKA model. Second, we propose PBtrees, basic PBtree construction and query processing algorithms, and two PBtree optimization algorithms. Third, we implemented and evaluated our scheme on a large real-world dataset with 5 million data items. Experimental results show that our scheme is both fast and scalable. For example, for a query whose results contain 10 data items, it takes only 0.17 ms.

II. Related Work

There are some privacy-preserving range query works that do not fit into our cloud computing paradigm and cannot be used to solve the problem addressed in this paper. In the public-key domain, the approach in [36] supports range querying using identity-based encryption primitives [12], [35]. Their encryption scheme allows a network gateway to encrypt summaries of network flows before submitting them to an untrusted repository; when a network operator suspects that an intrusion happens, a trusted third party can release a key to the operator to allow the operator to decrypt flows whose attributes fall within specified ranges, but not other flows. However, the user query privacy is not preserved.

A significant amount of work has been done in privacy-preserving keyword and ranked keyword queries [7], [8], [11], [14]–[18], [20], [21], [26], [27], [34], [37], [39]. However, these solutions are not optimized for range queries.

Prior work on outsourced databases has addressed problems such as secure kNN processing [31], [38], [40], privacy-preserving data mining [6], [33], and query result integrity verification [29]. In [31], [38], and [40], order-preserving encryption techniques were used to compute the k-nearest neighbors of a given encrypted query point in an encrypted database. For the privacy-preserving clustering mechanisms in [6] and [33], certain confidential numerical attributes are perturbed in a uniform manner so as to preserve the distances between any two points. Significant work has been done on query result integrity verification [29]. The basic idea is to include verifiable digital signatures for each returned tuple, which allow the client to verify the integrity of query results.

III. PBtree Construction

In this section, we first present our PBtree construction algorithm, which is executed by the data owner. This algorithm consists of three steps: prefix encoding, tree construction, and node randomization using Bloom filters. Second, we present our algorithm for computing the trapdoor for a given query, which is

executed by the data users. With the PBtree of n data items and the trapdoor for a given query, the cloud is able to process the query on the PBtree without knowing the value of the data items and the query.

A. Prefix Encoding

The key idea of this step is to convert the testing of whether a data item falls into a range to the testing of whether two sets have common elements, where the basic step is testing whether two numbers are equal. To achieve this, we adopt the prefix membership verification scheme in [32]. Given a number x of w bits whose binary representation is $b_1b_2\cdots b_w$, its prefix family denoted as $F(x)$ is defined as the set of $w+1$ prefixes $\{b_1b_2\cdots b_w, b_1b_2\cdots b_{w-1}*, \cdots, b_1*\cdots *, * * \ldots *\}$, where the ith prefix is $b_1b_2\cdots b_{w-i+1}*\cdots *$. For example, the prefix family of number 6 of 5 bits is $F(6) = F(00110) = \{00110, 0011^*, 001^{**}, 00^{***}, 0^{****}, ^{*****}\}$. Given a range $[a, b]$, we first convert the range $[a, b]$ to a minimum set of prefixes, denoted $S([a, b])$, such that the union of the prefixes is equal to $[a, b]$. For example, $S([0, 8]) = \{00^{***}, 1000\}$. Given a range $[a, b]$, where a and b are two numbers of w bits, the number of prefixes in $S([a, b])$ is at most $2w - 2$ [22]. For any number x and range $[a, b]$, $x \in [a, b]$ if and only if there exists prefix $p \in S([a, b])$ so that $x \in p$ holds. Furthermore, for any number x and prefix p, $x \in p$ if and only if $p \in F(x)$. Thus, for any number x and range $[a, b]$, $x \in [a, b]$ if and only if $F(x) \cap S([a, b]) \neq \emptyset$. From the above examples, we can see that $6 \in [0, 8]$ and $F(6) \cap S([0, 8]) = \{00^{***}\}$. In this step, given n data items $d_1, \cdots, d_n$, the data owner computes the prefix families $F(d_1), \cdots, F(d_n)$; given a range $[a, b]$, the data user computes $S([a, b])$.

B. Tree Construction

To achieve sublinear search efficiency, we organize $F(d_1), \cdots, F(d_n)$ in a tree structure that we call *PBtree*. We cannot use existing database indexing structures like $B+$ trees because of two reasons. First, searching on such trees (such as $B+$ trees) requires the operation of testing which of two numbers is bigger. However, PBtrees cannot support such operations for the cloud because otherwise PBtrees will share the same weaknesses with prior order-preserving schemes [23]–[25]. Second, their structures for different sets of data items are often different even if the two sets have equal sizes. However, for any two sets of the same size, their PBtrees are required to have the same structure, i.e., the two PBtrees are indistinguishable. In this paper, we organize $F(d_1), \cdots, F(d_n)$ using our PBtree structure.

Definition 3.1 (PBtree): A PBTree for n data items is a full binary tree with n terminal nodes and $n-1$ nonterminal nodes, where all n terminal nodes form a linked list from left to right and each node is represented using a Bloom filter. Each terminal node contains one data item, and each nonterminal node contains the union of its left and right children. For any nonterminal node, the size of its left child either equals that of its right child or exceeds by one.

According to this definition, a PBtree is a highly balanced binary search tree. The height of the PBtree for n data items is $\lfloor \log n \rfloor + 1$. We construct the PBtree from $F(d_1), \cdots, F(d_n)$ in a top–down fashion. First, we construct the root node, which is labeled with the n prefix families $\{F(d_1), \cdots, F(d_n)\}$. Second, we partition the set of n prefix families $\{F(d_1), \cdots, F(d_n)\}$ into two subsets of prefix families $S_{\texttt{left}}$ and $S_{\texttt{right}}$ such that $|S_{\texttt{left}}| = |S_{\texttt{right}}|$ if n is even and $|S_{\texttt{left}}| = |S_{\texttt{right}}| + 1$ if n is odd, and then construct two child nodes for the root, where the left child is labeled with $S_{\texttt{left}}$ and the right child is labeled with $S_{\texttt{right}}$. We recursively apply the above step to the left child and the right child, respectively, until every terminal node contains only one prefix family. At the end, we link all terminal nodes by a linked list. Fig. 2 shows the PBtree for the set of prefix families $S = \{F(1), F(6), F(7), F(9), F(11), F(12), F(13), F(16), F(20), F(25)\}$.

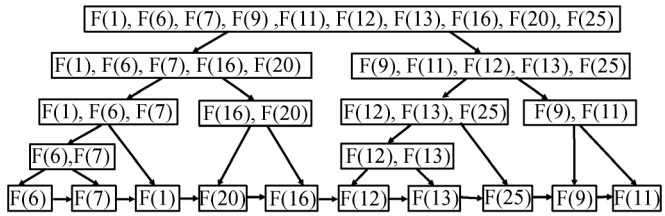

Fig. 2. PBtree example.

The key property of PBtrees is stated in Theorem 3.1, which is straightforward to prove according to its construction algorithm. Note that the constraint $0 \leq |S_{\texttt{left}}| - |S_{\texttt{right}}| \leq 1$ makes the structure of the PBtree for a set of data items to solely depend on the number of data items.

Theorem 3.1 (Structure Indistinguishability): For any two sets of data items S_1 and S_2, their PBtrees have exactly the same structure if and only if $|S_1| = |S_2|$.

We now describe the query processing algorithm on the above tree. For a PBtree T, we use $T.root$ to denote the root node of T, $T.left$ to denote the left subtree of T, and $T.right$ to denote the right subtree of T. For a node v, we use $L(v)$ to denote the label of v, which is a set of prefix families, and $U(v)$ to denote the union of all prefix families in $L(v)$. For example, if $L(v) = \{F(6), F(7)\}$, then $U(v) = F(6) \cup F(7)$. Given a range $[a, b]$, starting from the root $T.root$ of a PBtree T, where $L(T.root) = \{F(d_1), \cdots, F(d_n)\}$ and $U(T.root) = F(d_1) \cup \cdots \cup F(d_n)$, we check whether $U(T.root) \cap S([a, b]) = \emptyset$. If $U(T.root) \cap S([a, b]) = \emptyset$, then none of the n data items $d_1, \cdots, d_n$ falls into the range of $[a, b]$ and therefore we do not need to continue searching tree T. If $U(T.root) \cap S([a, b]) \neq \emptyset$, then there exists at least one of the n data items $d_1, \cdots, d_n$ falls into the range of $[a, b]$; thus, we need to continue to recursively conduct the same search on $T.left$ and $T.right$, if they exist.

We now analyze the time complexity of our query processing algorithm and show that it is sublinear in the number of data items. Let n be the number of data items indexed by the PBtree, $[a, b]$ be the query, and R be the query result. The average runtime of the search algorithm depends on $|R|$, query result size. Theoretically, if $|R| = 0$, then only the root of the PBtree needs to be checked and the time complexity is $O(1)$; if $|R| = n$, then all data items indexed by the PBtree need to be traversed via the linked list, and the time complexity is $O(n)$. In reality, we have $|R| \ll n$ as n is typically large. For each data item in R,

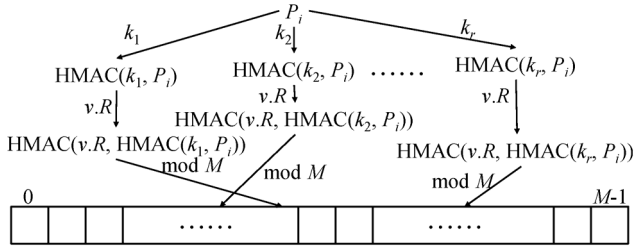

Fig. 3. Secure hashing in Bloom filters.

we need to traverse at most $2\log n - 1$ nodes. Thus, the time complexity is $O(|R|\log n)$.

C. Node Randomization Using Bloom Filters

Next, we present a solution based on secure keyed hash functions (HMAC) and Bloom filters to make our PBtree privacy-preserving. For each node v, we use a Bloom filter denoted by $v.B$ to store the prefixes of a node's prefix families. We assume that the data owner and the users share r secret keys, denoted $k_1, \cdots, k_r$, other than the symmetric key for encrypting and decrypting data items. Consider a PBtree node v, where set $L(v)$ consists of n prefix families and set $U(v)$ consists of m prefixes $p_1, \cdots, p_m$. Let w be the number of bits that each data item contains. Our node randomization algorithm consists of three steps.

One-Wayness: For each prefix p_i, we use the r secret keys to compte r hashes: $\texttt{HMAC}(k_1, p_i)$, $\cdots$, $\texttt{HMAC}(k_r, p_i)$. The purpose of this step is to achieve one-wayness, that is, given prefix p_i and the r secret keys, it is computationally efficient to compute the r hashes; but given the r hashes, it is computationally infeasible to compute the r secret keys and p_i; furthermore, even given the r hashes and p_i, which is the case in chosen plaintext attacks (CPA), it is still computationally infeasible to compute the r secret keys.

Decorrelation: For node v, we generate a random number $v.R$, which has the same number of bits as a secret key. We use $v.R$ to compute r hashes: $\texttt{HMAC}(v.R, \texttt{HMAC}(k_1, p_i))$, $\cdots$, $\texttt{HMAC}(v.R, \texttt{HMAC}(k_r, p_i))$. For each prefix p_i and for each $1 \le j \le r$, we let $v.B[\texttt{HMAC}(v.R, \texttt{HMAC}(k_j, p_i)) \ mod \ M] := 1$. The purpose of the random number that is unique for each node is to eliminate the correlation among different Bloom filters for different nodes. For the same prefix p, this random number allows us to hash p independently for different Bloom filters. Without the use of this random number, if prefix p_i is shared by $U(v_1)$ and $U(v_2)$ of two different nodes v_1 and v_2, then for all the r locations $\texttt{HMAC}(k_1, p_i) mod M$, $\cdots$, $\texttt{HMAC}(k_r, p_i) mod M$, both Bloom filters have the value 1. Although two Bloom filters both having 1 for all these r locations does not necessarily mean that $U(v_1)$ and $U(v_2)$ share a common prefix, without the use of this random number, if two Bloom filters have more 1's at the same locations than other pairs, then the probability that they share common prefixes is higher. Fig. 3 shows the above hashing process for Bloom filters.

Padding: If $m < (w+1) * n$, which means that some prefix families share common prefixes, we generate $((w+1)*n-m)*r$ random numbers and for each number x, $v.B[x \ mod \ M] := 1$. At last, we use this Bloom filter together with the random number $v.R$ to replace the label of v. The purpose of this step is to avoid a Bloom filter to expose the information how much its prefix families share common prefixes. Without the padding, some Bloom filters are inserted with less number of elements than others, which will cause it to have less 1's than others in the statistical sense.

By now the PBtree is fully constructed from data items $d_1, \cdots, d_n$ by the data owner. The data owner sends the encrypted data items and the PBtree to the cloud.

D. Trapdoor Computation

Given a query $[a, b]$, suppose $S([a, b])$ consists of z prefixes $p_1, \cdots, p_z$, for each prefix p_i, $1 \le i \le z$, the data user computes r hashes: $\texttt{HMAC}(k_1, p_i)$, $\cdots$, $\texttt{HMAC}(k_r, p_i)$. The trapdoor for query $[a, b]$, denoted as $M_{[a,b]}$, is a matrix of $z * r$ hashes: $\texttt{HMAC}(k_1, p_1)$, $\cdots$, $\texttt{HMAC}(k_r, p_1)$, $\cdots$, $\texttt{HMAC}(k_1, p_z)$, $\cdots$, $\texttt{HMAC}(k_r, p_z)$. We organize these $z * r$ hashes in a matrix because the cloud needs to know which r hashes are all correspond to the same prefix. The trapdoor of p_i corresponds to the ith row of the trapdoor matrix. After the computation, the data user sends $M_{[a,b]}$ to the cloud.

E. Query Processing

After receiving a query represented as a trapdoor, the cloud uses the trapdoor to search over the PBtree. The query processing algorithm on PBtrees still applies except that the checking of whether $U(v) \cap S([a, b]) \neq \emptyset$ is implemented as checking whether there exists a row $i (1 \le i \le z)$ in matrix $M_{[a,b]}$ so that for every j $(1 \le j \le r)$ we have $v.B[\texttt{HMAC}(v.R, \texttt{HMAC}(k_j, p_i)) \ mod \ M] = 1$. The straightforward implementation of the above query processing algorithms requires to check each row of $M_{[a,b]}$ at each visited PBTree node. For a row i in $M_{[a,b]}$, if there exists j $(1 \le j \le r)$ so that $v.B[\texttt{HMAC}(v.R, \texttt{HMAC}(k_j, p_i)) \ modM] = 0$, then $U(v) \cap p_i = \emptyset$. If $U(v) \cap p_i = \emptyset$, then for any descendent node v' of node v, we have $U(v') \cap p_i = \emptyset$ because $U(v') \subset U(v)$. Thus, when we take $M_{[a,b]}$ to search over the PBtree, for any such row in $M_{[a,b]}$, we remove it from $M_{[a,b]}$ when we continue to search the descendent nodes of v. The searching process terminates when $M_{[a,b]}$ becomes empty or we finishes searching terminal nodes.

F. False Positive Analysis

As each node in a PBtree is represented by a Bloom filter, which inherently has false positives, the query result on a PBtree may contain false positives. For simplicity, consider a PBtree with $n = 2^h$ leaf nodes, where the height of the PBtree is $h+1$. Let R be the query result, which is a set of data items. We color all the terminal and nonterminal nodes on the path from a data item in R to the root of the PBtree to be gray and others to be white. Fig. 4 shows such a marked PBtree where $d_j \in R$. Let f be the false positive rate of a Bloom filter in the PBtree. Note that although nodes of different levels in a PBtree may have a Bloom filter of different length, we always choose the number of hash functions r to be $(m/n) \times \ln 2$ to minimize the false positive rate to be $(1 - (1 - (1/m)^{rn})^r \approx (1 - e^{-rn/m})^r = 2^{-r} \approx 0.6185^{m/n}$; thus, by choosing the same m/n value for each node, the false positive of the Bloom filter at each node is the same. For any node $d_i \notin R$, let $\texttt{len}(d_i, R)$ be the number of white nodes on the path from d_i to the root, the probability that

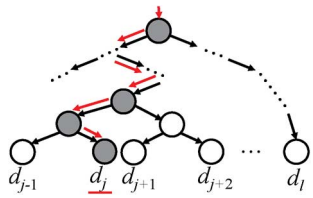

Fig. 4. PBtree example.

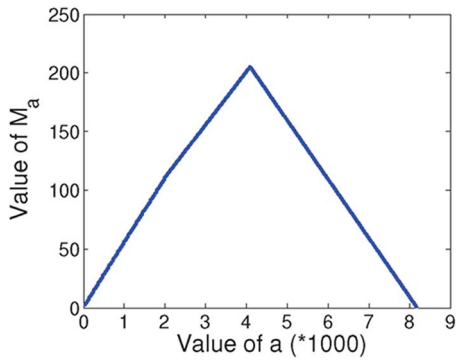

Fig. 5. Relation between M_a and a.

d_i is a false positive is $f^{\mathtt{len}(d_i,R)}$. Thus, the expected number of false positives is $\Sigma_{d_i \notin R} f^{\mathtt{len}(d_i,R)}$. Among all possible query result sets R of the same size a, we use M_a be denote the maximum expected number of false positives. Thus

$$M_a = \max_{\forall R} \left(\sum_{d_i \notin R} f^{\mathtt{len}(d_i,R)} \right). \tag{III.1}$$

For $a = 0$, we have

$$M_0 = 2^h \times p^{h+1}. \tag{III.2}$$

For $a = 1$, say $d_j \in R$ as illustrated by Fig. 4, the values of $\mathtt{len}(d_i, R)$ for $d_i \notin R$ are $1, 2, 2, 3, 3, 3, 3, \cdots$. Thus, we have

$$M_1 = f + 2f^2 + \cdots + 2^{h-1} f^h = f \frac{1-(2f)^h}{1-2f}. \tag{III.3}$$

For $1 < a \leq n$, according to (III.1), M_a corresponds to the case where in the $(\lceil \log a \rceil + 1)$th layer there are a nodes colored gray and for each subtree rooted at these a nodes, there is one and only one terminal node is colored gray. Considering the $2^{\lceil \log a \rceil}$ subtrees rooted at the $(\lceil \log a \rceil + 1)$th layer, the a subtrees have only one gray terminal node each and the rest $2^{\lceil \log a \rceil} - a$ subtrees have no gray terminal nodes. For each of the a subtrees, we can calculate the maximum expected number of false positives based on (III.3); similarly, for each of the rest $2^{\lceil \log a \rceil} - a$ subtrees, we can calculate that based on (III.2). Thus, M_a can be calculated as follows:

$$M_a = af \times \frac{1-(2f)^{h-\lceil \log a \rceil}}{1-2f} + \left(2^{\lceil \log a \rceil} - a\right) f(2f)^{h-\lceil \log a \rceil}.$$

Fig. 5 shows the relation between M_a and a, where we choose $f = 0.05$ and $h = 13$.

IV. PBtree Search Optimization

In this section, we optimize PBtree searching efficiency by minimizing the number of nodes that a query needs to traverse both horizontally and vertically.

A. Traversal Width Optimization

Recall that in the PBtree construction algorithm in Section III-B, for a nonterminal node with prefix family set S, we partition this node into two child nodes S_1, S_2 so that $0 \leq |S_1| - |S_2| \leq 1$. This partition is critical for the performance of query processing on the PBtree because querying the common prefixes that both S_1 and S_2 share will lead to the traversal of both subtrees. Thus, in partitioning S into S_1, S_2, besides satisfying the condition $0 \leq |S_1| - |S_2| \leq 1$, we want to minimize $Max\{F_i \cap F_j | F_i \in S_1, F_j \in S_2\}$, which is the maximum number of prefixes in the intersection of two prefix families that one from S_1 and the other from S_2. This condition is to let those prefix families that share more prefixes to be partitioned in the same set. We call this problem *Equal Size Prefix Family Partition*. We next formally define this problem and prove that it is NP-hard.

Definition 4.1 (Equal Size Prefix Family Partition): Given a set S of prefix families, we want to partition S into S_1, S_2, such that the following two conditions are satisfied:

1) $0 \leq ||S_1| - |S_2|| \leq 1$;
2) $Max\{F_i \cap F_j | F_i \in S_1, F_j \in S_2\}$ is minimized.

Theorem 4.1: The Equal Size Prefix Family Partition problem is NP-hard.

Proof: The decision version of the Equal Size Prefix Family Partition Problem is the following: "Is it possible to partition a set S of prefix families into S_1 and S_2 such that $0 \leq ||S_1| - |S_2|| \leq 1$ and $Max\{F_i \cap F_j | F_i \in S_1, F_j \in S_2\} < k$?" We reduce the *Set Partition Problem*, a known NP-Complete problem, to the decision version of Equal Size Prefix Family Partition Problem. The Set Partition Problem is as follows: "For a multiset of positive numbers $A = \{a_1, a_2, \cdots, a_n\}$, is it possible to partition A into A_1 and A_2 such that $\sum_{a_i \in A_1} a_i = \sum_{a_j \in A_2} a_j$."

Given an instance of the set partition problem with positive number multiset $A = \{a_1, a_2, \cdots, a_n\}$, we convert it to an instance of our Equal Size Prefix Family Partition Problem with prefix family set S as follows. Let $a_{\max}$ be the largest number in A. For each number a_i in A, we first generate a_i data items $d_1, d_2, \cdots, d_{a_i}$ where each data item has $\lceil \log n \rceil + \lceil \log a_{\max} \rceil$ bits, and for each data item d_j $(1 \leq j \leq a_i)$, the value of the first $\lceil \log n \rceil$ bits is i and the value of the last $\lceil \log a_{\max} \rceil$ bits is $j - 1$. For example, suppose $A = \{2, 3, 4\}$, for number 2 in A, we generate 2 data items 0000 and 0001 in their binary representation. Second, for each data item d_j $(1 \leq j \leq a_i)$, we generate its prefix family $F(d_j)$. Finally, we map each number a_i in A to a_i prefix families $F(d_1), F(d_2), \cdots, F(d_{a_i})$ in S, and let $k = \lceil \log n \rceil$.

Suppose the prefix family set S constructed above has an equal-size prefix family partition solution S_1 and S_2 with $Max\{F_i \cap F_j | F_i \in S_1, F_j \in S_2\} \leq k = \lceil \log n \rceil$. We next prove that A has a set partition solution. Note that if $\sum_{a_i \in A} a_i$ is odd, then the set partition problem has no solution. Thus, we only need to consider cases where $\sum_{a_i \in A} a_i$ is

even, which means that $|S_1| = |S_2|$. A notable property of the a_i prefix families $F(d_1), F(d_2), \cdots, F(d_{a_i})$ constructed above is that any two of these prefix families share at least $\lceil \log n \rceil$ prefixes. For example, $F(0001)$ and $F(0001)$ share 3 prefixes. Thus, $Max\{F_i \cap F_j | F_i \in S_1, F_j \in S_2\} \leq k = \lceil \log n \rceil$ implies that for any a_i in A, the constructed a_i prefix families $F(d_1), F(d_2), \cdots, F(d_{a_i})$ are either all in S_1 or all in S_2. Otherwise, suppose $F(d_1) \in S_1$ and $F(d_2) \in S_2$, then $|F(d_1) \cap F(d_2)| \geq \lceil \log n \rceil = k$. Thus, $|S_1|$ is equal to the sum of some numbers in A and $|S_1|$ is equal to the sum of the remaining numbers in A. Finally, $|S_1| = |S_2|$ implies that A has a set partition solution. Thus, the Set Partition Problem $\leq_p$ the decision version of Equal Size Prefix Family Partition Problem, which means that the Equal Size Prefix Family Partition Problem is NP-hard. ■

Next, we present our approximation algorithm to the equal size prefix family partition problem. Our algorithm consists of two phases: *partition phase* and *re-organization phase*. In the partition phase, we partition the input prefix family set into two or three subsets so that the size of each subset is no larger than $\lceil n/2 \rceil$, where n is the size of the prefix family set. In the reorganization phase, if the first phase outputs two subsets, then we do nothing because the two subsets must satisfy the condition of $0 \leq ||S_1| - |S_2|| \leq 1$; if the first phase outputs three subsets, then we first choose one subset to split into multiple smaller subsets, and then merge these new subsets with the two other subsets to obtain two subsets that satisfy the condition of $0 \leq ||S_1| - |S_2|| \leq 1$.

To help to present the details of these two phases, we first define two concepts: *longest common prefix* and *child prefixes*. The longest common prefix of a set of prefix families $S = \{F_1, F_2, \cdots, F_n\}$, denoted by $LCP(S)$, is the longest prefix in $F_1 \cap F_2 \cap \cdots \cap F_n$. For example, the longest common prefix of $\{F(1101), F(1100)\}$ is 110^*. Note that for any set of prefix families, it has only one longest common prefix because no prefix family consists of two prefixes of the same length. For any prefix $b_1 b_2 \cdots b_{w-i+1} * \cdots *$, it has two child prefixes $b_1 b_2 \cdots b_{w-i+1} 0 * \cdots *$ and $b_1 b_2 \cdots b_{w-i+1} 1 * \cdots *$, which are obtained by replacing the first $*$ by 0 and 1 and called child-0 and child-1 prefixes, respectively.

For example, prefix 11^{**} has two child prefixes 110^* and 111^*. For any prefix p, we use p^0 and p^1 to denote p's child-0 and child-1 prefixes, respectively. Given a set of prefix families $S = \{F_1, F_2, \cdots, F_n\}$, the partition phase of our approximation algorithm works as follows. First, we compute $LCP(S)$, the longest common prefix of S. Second, we partition S into two subsets, one subset whose each prefix family contains $LCP(S)^0$ and one subset whose each prefix family contains $LCP(S)^1$. If any of the two subsets has a larger size than $\lceil n/2 \rceil$, then we recursively apply the above two partition process to that subset. This process terminates when all subsets have a smaller size than $\lceil n/2 \rceil$. Third, for any two subsets whose union has a size smaller than $\lceil n/2 \rceil$, we call them *mergeable* and we merge them (i.e., union them into one set). This process terminates when no two subsets are mergeable. Thus, we result in either two subsets or three subsets. It is impossible to result in four subsets or more because otherwise there are at least two subsets can be merged.

If the partition phase results in two subsets, then the reorganization phase does nothing. Let S_1 and S_2 be the two subsets. Because $|S_1| + |S_2| = n$, $|S_1| \leq \lceil n/2 \rceil$, and $|S_2| \leq \lceil n/2 \rceil$, we have $0 \leq ||S_1| - |S_2|| \leq 1$. Thus, S_1 and S_2 represent the final partition result.

If the partition phase results in three subsets, then the reorganization phase chooses one subset to split into multiple smaller subsets, and then union these new subsets with the two other subsets to obtain two subsets S_1 and S_2 that satisfy the condition of $0 \leq ||S_1| - |S_2|| \leq 1$. Let S_1, S_2, and S_3 be the three subsets. We choose the subset whose longest common prefix is the smallest, that is, the subset whose prefix families share the least number of prefixes. Let S_3 be the subset that we choose. We first compute its longest common prefix $LCP(S_3)$. Note that for any prefix family in S_3, it contains either $LCP(S_3)^0$ or $LCP(S_3)^1$. Second, we split S_3 into two subsets S_{31}, whose each prefix family contains $LCP(S_3)^0$, and S_{32}, whose each prefix family contains $LCP(S_3)^1$. Without loss of generality, we suppose $|S_{31}| \leq |S_{32}|$. Thus, either S_1 or S_2 can be merged with S_{31}. Otherwise, if both $|S_1| + |S_{31}| > \lceil n/2 \rceil$ and $|S_2| + |S_{31}| > \lceil n/2 \rceil$, then $|S_1| + |S_2| + |S_3| = |S_1| + |S_2| + |S_{31}| + |S_{32}| \geq |S_1| + |S_2| + |S_{31}| + |S_{31}| = (|S_1| + |S_{31}|) + (|S_2| + |S_{31}|) > \lceil n/2 \rceil + \lceil n/2 \rceil \geq n$. Again, suppose $|S_1| \geq |S_2|$ and S_1 can be merged with S_{31}, we merge S_1 with S_{31}. After merging S_1 with S_{31}, we check whether S_2 can be merged with S_{32}. If they can, we merge them and output the partition result $S_1 \cup S_{31}$ and $S_2 \cup S_{32}$. If S_2 and S_{32} cannot be merged, we further split S_{32}, and repeat the above process. If S_1 cannot be merged with S_{31}, we merge S_{31} with S_2 and split S_{32}, and then repeat the above process.

We now analyze the worst-case computational complexity of our prefix family partition algorithm. Let n be the size of the input set of prefix families and $T(n)$ be the corresponding time complexity. Each set partition operation takes $O(n)$ time and each subset merging takes $O(1)$ time. The worst-case computational complexity is when each set partition operation produces two subsets where the size of one subset is one. Thus, we have: $T(n) = T(n-1) + O(n)$. The computational complexity of this algorithm is therefore $O(n^2)$ in the worst case.

Set splitting and merging operations are performed in both phases of our prefix family partition algorithm. To analyze the partition results, we combine these two phases together and view the whole algorithm as two parts: set splitting and subsets reorganization. The subsets are produced in a top–down fashion. For any prefix family $F_i \in S_i$, $F_j \in S_i$, and $F_k \notin S_i$, we have $|F_i \cap F_j| > |F_i \cap F_k|$. We state the properties of the subsets generated by our algorithm in Theorem 4.2. Before we present Theorem 4.2, we first introduce and prove Lemma 4.1.

Lemma 4.1: For any three prefix families $F(d_1)$, $F(d_2)$, and $F(d_3)$, if $|F(d_1) \cap F(d_2)| \geq |F(d_1) \cap F(d_3)|$ and $|F(d_1) \cap F(d_2)| \geq |F(d_2) \cap F(d_3)|$, then $F(d_1) \cap F(d_3) = F(d_2) \cap F(d_3) \subseteq F(d_1) \cap F(d_2)$.

Proof: Let $|F(d_1) \cap F(d_3)| = k \geq 0$ and $F(d_1) \cap F(d_2)| = m \geq k$, which means the first k bits of d_1 and d_3 are the same and the first m bits of d_1 and d_2 are the same. Because $m \geq k$, the first k bits of d_1 and d_3 share is also the first k bits of d_1 and d_2 share, which means $F(d_1) \cap F(d_3) \subseteq F(d_1) \cap F(d_2)$. Similarly, $F(d_2) \cap F(d_3) \subseteq F(d_1) \cap F(d_2)$. Because d_1 and d_3

share exactly the first k bits of d_1, d_1 and d_2 share exactly the first m bits of d_1, and $m \geq k$, d_2 and d_3 also share exactly the first k bits of d_1. Thus, $F(d_1) \cap F(d_3) = F(d_2) \cap F(d_3)$. ■

Theorem 4.2: For any three prefix family sets S_1, S_2, and S_3 produced in our prefix family partition algorithm, we have the following.

1) $|F_p \cap F_l| = |F_q \cap F_l| = |\cup_{F_i \in S_i} \cap \cup_{F_j \in S_j}|$, where $F_p \in S_i$, $F_q \in S_i$, and $F_l \in S_j$.
2) if $|\cup_{F \in S_1} F \cap \cup_{F \in S_2} F| \geq |\cup_{F \in S_1} F \cap \cup_{F \in S_3} F|$ and $|\cup_{F \in S_1} F \cap \cup_{F \in S_2} F| \geq |\cup_{F \in S_2} F \cap \cup_{F \in S_3} F|$, then $\cup_{F \in S_1} F \cap \cup_{F \in S_3} F = \cup_{F \in S_2} F \cap \cup_{F \in S_3} F \subseteq \cup_{F \in S_1} F \cap \cup_{F \in S_2} F$.

Based on Lemma 4.1 and its proof, it is straightforward to prove Theorem 4.2. According to Theorem 4.2, subset production is optimized in our partition algorithm, but subset reorganization is not because the number of common prefixes shared between subsets is not considered.

B. Traversal Depth Optimization

Our idea for optimizing searching depth is based on the following observation: for any internal node v with label $\{F(d_1), F(d_2), \cdots, F(d_m)\}$ that a query prefix p traverses, if $p \in F(d_1) \cap F(d_2) \cap \cdots \cap F(d_m)$, then all terminal nodes of the subtree rooted at v satisfy the query; thus, we can directly jump to the left most terminal node of this subtree and collect all terminal nodes using the linked list, skipping the traversal of all nonterminal node under v in this subtree. This optimization opportunity is the motivation that we chain the terminal nodes in PBtrees. Note that here $F(d_1) \cap F(d_2) \cap \cdots \cap F(d_m) \neq \emptyset$ because it must contain the prefix of w *s. Furthermore, our searching width optimization technique significantly increases the probability that the prefix families in a nonterminal node share more than one common prefix.

For a node v labeled with $\{F(d_1), F(d_2), \cdots, F(d_m)\}$, we split $\bigcup_{i=1}^{m} F(d_i)$ into two sets: the *common set* $\mathbb{C} = \bigcap_{i=1}^{m} F(d_i)$ and the *uncommon set* $\mathbb{N} = \bigcup_{i=1}^{m} F(d_i) - \bigcap_{i=1}^{m} F(d_i)$. With this splitting, query processing at node v is modified to be the following. First, we check whether $p \in \mathbb{N}$. If $p \in \mathbb{N}$, then we continue to use the query processing algorithm in Section III-E to search p on v's left and right child nodes. If $p \notin \mathbb{N}$, then we further check $p \in \mathbb{C}$; if $p \notin \mathbb{N}$ but $p \in \mathbb{C}$, then we directly jump to the bottom to collect all terminal nodes in the subtree rooted at v; if p is in neither set, then we skip the subtree rooted at v.

The key technical challenge in searching depth optimization is how to store the common set $\mathbb{C}$ and the uncommon set $\mathbb{N}$ for each nonterminal node using Bloom filters. The straightforward solution is to use two Bloom filters, storing $\mathbb{C}$ and $\mathbb{N}$, respectively. However, this will not be space-efficient as we need two bit vectors. In this paper, we propose an space efficient way to represent two Bloom filters using two sets of k hash functions $\{hc_1, hc_2, \cdots, hc_r\}$ and $\{hn_1, hn_2, \cdots, hn_r\}$ but only one bit vector B of m bits. In the PBtree construction phase, for a prefix $p \in \mathbb{C} \cup \mathbb{N}$, if $p \in \mathbb{C}$, we set $B[hc_1(p)]$, $B[hc_2(p)], \cdots, B[hc_r(p)]$ to be 1; if $p \in \mathbb{N}$, we set $B[hn_1(p)]$, $B[hn_2(p)], \cdots, B[hn_r(p)]$ to be 1. Thus, we check whether a $p \in \mathbb{C}$ by checking whether $\wedge_{i=1}^{r}(B[hc_i(p)] == 1)$ holds and check whether $p \in \mathbb{N}$ by checking whether $\wedge_{i=1}^{r}(B[hn_i(p)] == 1)$ holds.

Next, we analyze the false positives of this Bloom filter with two sets of r hash functions and a bit vector B of m bits. Suppose we have inserted n elements (i.e., $|\mathbb{C}| + |\mathbb{N}| = n$) into this Bloom filter. Recall that our query processing algorithm conducts two times of set membership testing of a query prefix p at node v: first, we test whether $p \in \mathbb{N}$; second, on the condition that $p \notin \mathbb{N}$, we test whether $p \in \mathbb{C}$. Let $f_{\mathbb{N}}$ be the probability of a false positive occurs at the first membership testing, and $f_{\mathbb{C}}$ be the probability of a false positive occurs at the second membership testing. As the n elements are randomly and independently inserted into the bit vector B, the false positive probability at the first membership testing is the same as the false positive probability of the standard Bloom filter. Thus, we have

$$f_{\mathbb{N}} = \left(1 - (1 - \frac{1}{m})^{rn}\right)^r = \left(1 - e^{-\frac{rn}{m}}\right)^r. \quad \text{(IV.1)}$$

As the second testing is only performed on the condition that $p \notin \mathbb{N}$, and similarly, when the condition $p \notin \mathbb{N}$ holds, the false positive probability at testing whether $p \in \mathbb{C}$ is the same as that at testing whether $p \in \mathbb{N}$, we have

$$\begin{aligned} f_{\mathbb{C}} &= (1 - f_{\mathbb{N}}) \times \left(1 - (1 - \frac{1}{m})^{rn}\right)^r \\ &= \left(1 - (1 - e^{-\frac{rn}{m}})^r\right) \times \left(1 - e^{-\frac{rn}{m}}\right)^r. \end{aligned} \quad \text{(IV.2)}$$

To further reduce the false positive probability in testing $p \in \mathbb{C}$ at node v, when we collect the leaves of the subtree rooted at v, we can randomly choose x leaf nodes to test whether they indeed match p; if any of the leaf nodes does not match p, which means that $p \notin \mathbb{C}$, then we exclude all leaves of the subtree rooted at v from the query result. Thus, with the testing of the x leaf nodes, the false positive probability in testing whether $p \in \mathbb{C}$ becomes the following:

$$f_{\mathbb{C}} \times \left(1 - e^{-\frac{rn}{m}}\right)^{rx} = \left(1 - \left(1 - e^{-\frac{rn}{m}}\right)^r\right) \times \left(1 - e^{-\frac{rn}{m}}\right)^{rx+r}. \quad \text{(IV.3)}$$

Note that we test $p \in \mathbb{N}$ first and only when $p \notin \mathbb{N}$ we test $p \in \mathbb{C}$. Otherwise, if we use the above mentioned leaf testing method to further reduce the false positive probability in testing $p \in \mathbb{C}$, we may introduce false negatives, which is not allowed in our scheme. Suppose we first test $p \in \mathbb{C}$ first and only when $p \notin \mathbb{C}$ we test $p \in \mathbb{N}$. For a query prefix p, if $p \in \mathbb{N}$ and $p \notin \mathbb{C}$, but false positive occurs in testing $p \notin \mathbb{C}$, when we collect the leaves of the subtree rooted at v, if we test a leaf node and find it is not in $\mathbb{C}$, according to the above leaf testing method, we exclude all leaves of the subtree rooted at v from the query result. However, as $p \in \mathbb{N}$, some of these excluded leaves should be included in the query result, which are false negatives.

V. PBtree Update

Databases are typically subject to changes including record insertion, modification, and deletion. In this section, we present algorithms for inserting a new data item into a PBtree, modifying an existing data item, and deleting an existing data item from a PBtree. Note that all these operations are taken place on the data owner side, not in the cloud. After finishing the update

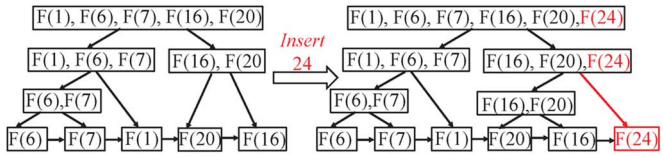

Fig. 6. PBtree insertion example.

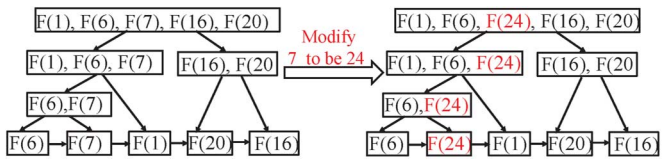

Fig. 7. PBtree modification example.

operations, the data owner sends the modified part of the PBtree to the cloud. The cloud updates PBtree index correspondingly. Although we support PBtree update, after a number of changes to a PBtree, we suggest to reconstruct the PBtree offline to improve query processing efficiency.

A. PBtree Insertion Algorithm

According to the definition of PBtrees, given a new data item and a PBtree, the location of the new data item on each level of the new PBtree is predetermined. In the process of inserting this new data item, the prefixes of the new data item need to be inserted into the Bloom filers along the path from the root to the corresponding leaf node. Fig. 6 shows an example PBtree and that after inserting item 24.

B. PBtree Modification Algorithm

In the process of modifying an existing old data item to be a new data item, we first compute the different prefixes between the old data and the new data item. In the Bloom filters along the path from the root to the corresponding leaf node, the prefixes that only belong to the old data item are deleted and the prefixes that only belong to the new data item are inserted. Fig. 7 shows an example PBtree and that after modifying data item $F(7)$ to $F(24)$.

C. PBtree Deletion Algorithm

As deletion can happen to any data item in a PBtree, the key technical challenge is to preserve the structural indistinguishability of the PBtree after deletion; that is, for a PBtree constructed from n data items, after deleting one data item, the structure of the resulting PBtree should be the same as any PBtree constructed from $n-1$ data items. To address this challenge, instead of directly deleting the data item d that we want to delete, we first delete a data item d' in the original PBtree so that after deleting d', the structural indistinguishability of the PBtree after deletion is preserved; second, we modify d to be d' in the resulting PBtree after deleting d'. Semantically, this two-step process is equivalent to directly delete d; structurally, we preserve the indistinguishability of the PBtree after deletion. Fig. 8 shows an example PBtree and that after deleting one data item 7, during which process we first delete 24 from the PBtree

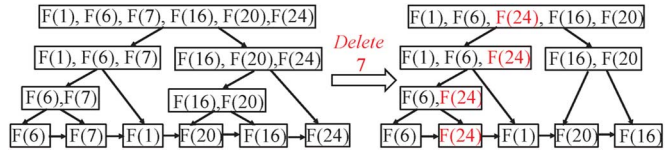

Fig. 8. PBtree deletion example.

and then use 24 to replace 7 in the PBtree and modify the corresponding Bloom filters

Bloom filters only support data insertion directly. When processing modification and deletion on the PBtree, the corresponding Bloom filters need to be rebuilt. Let n be the number of prefix families in the root. For insertion algorithm, we only need to insert the prefix family of the inserted data item in the corresponding Bloom filters. The number of Bloom filters is equal to the height of the PBtree, which is $\log n$. Thus, the complexity of insertion is $O(\log n)$. The computation complexities of modification and deletion algorithms are similar. For each of these two algorithms, we need to rebuild the Bloom filters along the root to one or two terminals. The total number of prefix families that we need to deal with is $n + n/2 + n/4 + \cdots + 1 = O(n)$. As the number of prefixes in a prefix family and the number of hash function in a Bloom filter are two constants, the complexity of each of these two algorithms is $O(n)$. Note that the updating operations reveal the hash locations of updated data items in the related Bloom filters, which can be partially hidden by choosing new padding bits for the updated Bloom filters.

VI. Security Analysis

A. Security Model

To achieve IND-CKA security, our PBtree uses `HMAC` as the pseudo-random function, which is a *keyed* hash function whose output cannot be distinguished from the output of a truly random function with nonnegligible probability [28]. Let $g : \{0,1\}^n \times \{0,1\}^s \rightarrow \{0,1\}^m$ be a keyed function, which takes as input n-bit strings and s-bit keys and maps to m-bit strings. Let $G : \{0,1\}^n \rightarrow \{0,1\}^m$ be a random function, which maps n-bit strings to m-bit strings. Function g is pseudo-random if satisfies the following two conditions. First, $g(x,k)$ can be computed efficiently from input $x \in \{0,1\}^n$ and $k \in \{0,1\}^s$. Second, for any probabilistic polynomial-time (PPT) adversary $\mathcal{A}$, $|Pr[\mathcal{A}^{g(\cdot,k)} = 1 | k \leftarrow \{0,1\}^s] - Pr[\mathcal{A}^r | r \leftarrow G : \{0,1\}^n \rightarrow \{0,1\}^m]| \leq negl(m)$, where $negl(m)$ is a negligible function. The IND-CKA secure model captures the notion that an adversary cannot deduce the contents of a dataset from its index, other than what it already knows from previous query results or from other channels. Let (`keygen, BuildIndex, Trapdoor, SearchIndex`) be an index scheme. The set difference of A and B is defined as $A \bowtie B = (A - B) \cup (B - A)$. We use the following game between a challenger $\mathcal{C}$ and a PPT adversary $\mathcal{A}$ to define IND-CKA security for a dynamic range query scheme.

Setup: $\mathcal{C}$ creates a set S of data items and gives it to $\mathcal{A}$. $\mathcal{A}$ generates a collection S^* of subsets by choosing a number of subsets from S. $\mathcal{A}$ sends S^* to $\mathcal{C}$. $\mathcal{C}$ runs `keygen` to generate

the private key set K_{priv} and builds indexes for each subset in S^* using `BuildIndex`. Finally, $\mathcal{C}$ sends all indexes with their associated subsets to $\mathcal{A}$.

Queries: $\mathcal{A}$ issues a range query $[a, b]$ to $\mathcal{C}$. $\mathcal{C}$ computes a trapdoor $t_{[a,b]}$ for $[a, b]$ via `Trapdoor` and returns it to $\mathcal{A}$. With $t_{[a,b]}$, $\mathcal{A}$ searches an index $\mathcal{I}$ via `SearchIndex` and acquires the corresponding query result.

Updates: $\mathcal{A}$ issues an update request to $\mathcal{C}$ on index $\mathcal{I}$ for inserting, modifying, or deleting a specified data item. $\mathcal{C}$ updates $\mathcal{I}$ to $\mathcal{I}'$ and returns $\mathcal{I}'$ to $\mathcal{A}$.

Challenge: After making some queries and updates, $\mathcal{A}$ decides on a challenge by picking a nonempty subset $S_0 \in S^*$ and generating another subset S_1 from S such that $|S_0 - S_1| \neq 0$, $|S_1 - S_0| \neq 0$, and $|S_0| = |S_1|$. Note that $\mathcal{A}$ must not have queried $\mathcal{C}$ for the trapdoor of any range query $[a, b]$ such that $\exists d_i \in S_0 \bowtie S_1$ and $a \leq d_i \leq b$, and $\mathcal{A}$ also must have not updated data item d_j such that $d_j \in S_0 \bowtie S_1$. $\mathcal{A}$ sends S_0 and S_1 to $\mathcal{C}$. $\mathcal{C}$ choose $b \leftarrow \{0, 1\}$ and builds an index $\mathcal{I}_b$ for S_b via `BuildIndex` and sends $\mathcal{I}_b$ to $\mathcal{A}$. After the challenge is issued, $\mathcal{A}$ is not allowed to query $\mathcal{C}$ for the trapdoor of any query $[a, b]$ such that $\exists d_i \in S_0 \bowtie S_1$, $a \leq d_i \leq b$ and to update d_j such that $d_j \in S_0 \bowtie S_1$.

Response: $\mathcal{A}$ guesses which subset that $\mathcal{I}_b$ is built for and outputs a bit b' as its guess for b. The advantage of $\mathcal{A}$ in winning this game is defined as $\mathtt{Adv}_{\mathcal{A}} = |Pr[b = b'] - 1/2|$.

We say that $\mathcal{A}$ breaks an index if $\mathtt{Adv}_{\mathcal{A}}$ is not negligible. We show next that no PPT adversary $\mathcal{A}$ can break the PBtree scheme and, thereby, prove its security under the IND-CKA model.

B. Security Proof

Theorem 6.1: The PBtree scheme is IND-CKA secure.

Proof: We prove this theorem by contradiction. Suppose PBtree is not IND-CKA secure, then there exists a PPT algorithm $\mathcal{A}$ that can break it. We construct a PPT algorithm $\mathcal{B}$ using $\mathcal{A}$ as a subroutine to distinguish a pseudo-random function from a truly random function with nonnegligible probability. The algorithm $\mathcal{B}$ works as follows. Given an unknown function g, g is either truly random or pseudo-random, a challenger $\mathcal{C}$ replaces `HMAC` with g in the PBtree scheme. Note that the challenger $\mathcal{C}$ has access to an Oracle O_g for the unknown function g that takes as input $\{0, 1\}^n$ and returns $g(x) = \{0, 1\}^m$. The algorithm $\mathcal{B}$ calls the algorithm $\mathcal{A}$ as a subroutine in the following game.

PBtree Construction: The challenger $\mathcal{C}$ creates a dataset S by randomly picking up a number of data items from $\{0, 1\}^n$ and sends to $\mathcal{B}$. $\mathcal{B}$ calls $\mathcal{A}$ to generates a collection S^* of subsets by choosing polynomial number of subsets of S and returns to $\mathcal{C}$. After choosing r secret keys, $\mathcal{C}$ builds a PBtree for each subset $S_i \in S^*$ using the PBtree Construction algorithm. $\mathcal{C}$ sends all PBtrees with their associated subsets to $\mathcal{B}$.

Queries: $\mathcal{B}$ calls $\mathcal{A}$ to generate a range query $[a, b]$ to $\mathcal{C}$. $\mathcal{C}$ computes a trapdoor $M_{[a,b]}$ for $[a, b]$ and returns it to $\mathcal{B}$. With $M_{[a,b]}$, $\mathcal{B}$ calls $\mathcal{A}$ to search an index $\mathcal{I}$ using PBtree searching algorithm and gets the corresponding query results.

Updates: $\mathcal{B}$ calls $\mathcal{A}$ to generate update operations to $\mathcal{C}$ on a PBtree $\mathcal{I}$ of inserting new data item d_i, deleting existing data item d_i, or modifying existing data item d_i to new data item d_j. $\mathcal{C}$ executes the corresponding updating algorithms and returns the updated index $\mathcal{I}'$ to $\mathcal{B}$.

Challenge: After making some queries and updates, $\mathcal{B}$ calls $\mathcal{A}$ to pick a nonempty subset $S_0 \in S^*$ and generates another subset S_1 from S such that $|S_0 - S_1| \neq 0$, $|S_1 - S_0| \neq 0$, $|S_1| = |S_0|$, no such queries $[a, b]$ have been queried such that $\exists d_i \in S_0 \bowtie S_1$ and $a \leq d_i \leq b$, and no such data item d_j is involved in the updating processes such that $d_j \in S_0 \bowtie S_1$. $\mathcal{B}$ sends S_0 and S_1 to $\mathcal{C}$. $\mathcal{C}$ randomly chooses a subset $S_b (b = 0, 1)$ and builds a PBtree $\mathcal{I}_b$ for S_b. $\mathcal{C}$ sends $\mathcal{I}_b$ to $\mathcal{B}$ and $\mathcal{B}$ is not allowed to query $\mathcal{C}$ for the trapdoor of any query $[a, b]$ such that $\exists d_i \in S_0 \bowtie S_1$ and $a \leq d_i \leq b$, and $\mathcal{B}$ is not allowed to update $\mathcal{I}_b$ involving data item d_j such that $d_j \in S_0 \bowtie S_1$.

Response: $\mathcal{B}$ calls $\mathcal{A}$ to guess which subset that $\mathcal{I}_b$ is built for. $\mathcal{A}$ outputs a bit b' as its guess for b. If $b' = b$, $\mathcal{B}$ outputs 1, indicating that it guesses that g is pseudo-random. Otherwise, $\mathcal{B}$ outputs 0.

Next, we show that $\mathcal{B}$ can determine whether g is a pseudo-random function or a random function with advantage greater than $negl(m)$ by proving the following claims.

Claim 1: When g is a pseudo-random function, then $|Pr[\mathcal{B}^{g(\cdot,k)} = 1 | k \leftarrow \{0, 1\}^s] - (1/2)| > negl(m)$.

Claim 2: When g is a random function, then $Pr[\mathcal{B}^g(n) = 1 | g \leftarrow \{G : \{0, 1\}^n \rightarrow \{0, 1\}^m\}] = 1/2$.

The proof of Claim 1 is directly. When g is pseudo-random, $\mathcal{B}$ calls $\mathcal{A}$ as a subroutine on IND-CKA game. Therefore, Claim 1 follows by the definition of $\mathcal{A}$.

We prove Claim 2 from following aspects.

First, the PBtrees of other subsets in S^* reveal no information about S_0 and S_1. Recall that each node v in a PBtree is assigned a different random number $v.R$ and g uses $v.R$ to map a hash string s into the Bloom filter B_v of node v. We view B_v which s has been mapped into by g as a string output by g. If g is pseudo-random function, then for any PPT algorithm $\mathcal{A}'$, $|Pr[\mathcal{A}'^{(g(s))} = 1 | s \leftarrow \{0, 1\}^n] - Pr[\mathcal{A}'^r = 1 | r \leftarrow \{0, 1\}^m]| < negl(m)$, which means a string output by a pseudo-random function is computationally indistinguishable from a random string. Thus, given two different Bloom filters B_i and B_j, each of which a string has been mapped into by g, it is infeasible for any PPT algorithm to distinguish whether a same string or two different strings are mapped into them. Note that it does not matter that whether B_i and B_j belong to a same PBtree or two different PBtrees. Since g is a random function, it is impossible for $\mathcal{A}'$ to correlate codewords across different Bloom filters. That is, $\mathcal{A}'$ learns nothing about $S_0 \bowtie S_1$ from other subsets in S^* and their corresponding PBtrees.

Second, the issued trapdoors reveal no information about the difference between the challenge subsets. With the restriction on the choice of queries during the game, no data item $d_i \in S_0 \bowtie S_1$ falls in any query result. Thus, no issued trapdoors reveal the difference between S_0 and S_1.

Third, the updating operations reveal no useful information for distinguishing the challenge subsets. Given a PBtree, the updating algorithms do not change its structure and, also, no data item $d_j \in S_0 \bowtie S_1$ is involved in the updating operations. Thus, no useful information for distinguishing challenge subsets is revealed from the tree structure and the updated data items. When performing an updating operation, some changes are made to the related Bloom filters, and the adversary $\mathcal{A}'$ can learn these changes by comparing corresponding Bloom filters. Without

losing generality, we assume that $\mathcal{A}'$ learns the hash locations of a data item d_i in Bloom filter B_x. When g is pseudo-random, $\mathcal{A}'$ cannot predict the hash locations of d_i in a new Bloom filter B_y, also, $\mathcal{A}'$ cannot predict the hash locations of $d_j (d_j \neq d_i)$ in B_x, which follows by the definition of a pseudo-random function. Thus, $\mathcal{A}'$ cannot predict hash locations for any data item $d_k \in S_0 \bowtie S_1$ in any Bloom filter of $\mathcal{I}_b$. Since g is truly random, it is impossible for $\mathcal{A}'$ to predict hash locations for any data item $d_i \in S_0 \bowtie S_1$ in any Bloom filter of $\mathcal{I}_b$. That is, $\mathcal{A}'$ learns nothing to distinguish S_0 and S_1 from updating operations.

Now, we only need to consider the challenge subsets to prove Claim 2. As $|S_0| = |S_1|$, according to Theorem 3.1, no information is revealed from the structure of $\mathcal{I}_b$. Without loss of generality, we assume that $S_0 \bowtie S_1$ only contains two data items d_i and d_j, where $d_i \in S_0$ and $d_j \in S_1$. Suppose $\mathcal{A}$ can guess b correctly with advantage θ, then at least there exists one Bloom filter B_i in $\mathcal{I}_b$ that $\mathcal{A}$ can determine d_i or d_j in B_i with advantage θ, which is impossible. Therefore, $\mathcal{A}$ guesses b correctly with probability 1/2. It follows that $Pr[\mathcal{B}^g = 1 | g \leftarrow \{G : \{0,1\}^n \rightarrow \{0,1\}^m\}] = 1/2$, thus proving Claim 2.

It follows from Claim 1 and Claim 2 that $|Pr[\mathcal{B}^{g(\cdot,k)} = 1 | k \leftarrow \{0,1\}^s] - Pr[\mathcal{B}^g = 1 | g \leftarrow \{G : \{0,1\}^n \rightarrow \{0,1\}^m\}]| > negl(m)$, i.e. $\mathcal{B}$ can distinguish a pseudo-random function from , which is impossible. Thus, we proved that our PBtree scheme is IND-CKA secure. ■

VII. Experimental Evaluation

A. Experimental Methodology

To evaluate the performance of PBtree, we considered four factors and generated various experimental configurations. The metrics considered are the following: the datasets, the type of PBtree construction, the type of queries, and the operations of updating. Based on these metrics, we have comprehensively evaluated the construction cost of the PBtree, the query evaluation time, the observed false positive rates, and the time and the communication cost for updating.

1) Datasets: We chose the Gowalla [19] dataset, which consists of 6 442 890 checkin records of users, over the period of February 2009 to October 2010, and extracted the timestamps. Now, given that each timestamp is represented as a tuple: $\langle year, month, date, hour, minute, second \rangle$, we performed a binary encoding for each of these attributes and treated the concatenation of the respective binary strings as a 32-bit integer value, while ignoring the unused bit positions. The details of encoding are as follows: $year$ is represented with a single bit as the value for $year$ is either 2009 or 2010; $month$, $date$, and $hour$ are represented using 5 bits each; and $minute$, $second$ are represented using 6 bits each. We perform the construction and querying experiments on 10 fixed-size datasets varying from 0.5 to 5 million records with a scaling factor of 0.5 million records, respectively, chosen uniformly at random from the 6-million-plus total records in the Gowalla dataset. For the update experiments, we choose data items randomly from the dataset records, which have not been used during the PBtree construction phase.

2) PBtree Types: We performed experiments with three variants of the PBtree: the basic PBtree without any optimizations, denoted as PB_B; the PBtree with width optimization, denoted as PB_W; and the PBtree with both depth and width optimizations, denoted as PB_WD. We have not performed experiments for the case of the PBtree with only depth optimization due to the following reasoning: When searching on a Bloom filter, we may need to perform two checks, which is twice the effort. If a query prefix is not found in the Bloom filter, then we need to perform a second check, using a different set of hash functions, to check if the prefix is a common prefix in the Bloom filter. As a result, depth optimization is more effective when combined with width optimization because width optimization aggregates the common prefixes in a systematic manner. Therefore, we focus only on the performance evaluation of PB_B, PB_W, and PB_WD.

3) Query Types: The performance evaluation of PBtree is dependent on two factors: query types and query results size. We consider two query types: prefix and range queries. A prefix query is a query specified as a single binary prefix, whereas a range query is specified as a numerical range and is likely to generate more than one binary prefixes. The prefix queries are effective in evaluating the performance of PBtree under the two types of optimizations we have described, and the range queries are effective to evaluate the performance of PBtree against other known approaches in literature. For each dataset, we generate a distinct collection of 10 prefix query sets, where each prefix set contains 1000 prefixes, and similarly, we generate 10 distinct range query sets, where each set contains 1000 range queries. The average number of prefixes for denoting a range in our range query sets varies from 5.93 to 9.6 prefixes, respectively.

The query result size is another important factor since the worst-case run-time search complexity of PBtree is given by $O(r. \log N)$, where r is the query result size. However, the challenge is that since the data values are not in any particular sequence, it is difficult to know which range queries can generate the desired query result sizes after the PBtree is built. To handle this issue, prior to the PBtree construction, we sort the data items and determine the appropriate range queries, which will result in the desired query result sizes, and use these queries in our experiments. For our experiments, we chose query ranges that result in query result sizes varying from 10 to 90 data items.

4) Implementation Details: We conducted our experiments on a desktop PC running Windows 7 Professional with 32 GB memory and a 3.5-GHz *Intel Core i7-4770k* processor. We used *HMAC-SHA1* as the pseudo-random function for the Bloom filter encoding and implemented the PBtree using C++. We set the Bloom filter parameter, $m/n = 10$, where m is the Bloom filter size and n is the number of elements, and the number of Bloom filter hash functions as 7. Although we have also experimented with other values of m/n, because of the limited space, we only show the results for $m/n = 10$.

B. Evaluation of PBtree Construction

Our experimental results show that the cost of PBtree construction is reasonable, both in terms of time and space. For the chosen datasets, Fig. 9(a) shows that the average time for generating, the PB_B is 276 to 3443 s, the PB_W is 338 to 7500 s,

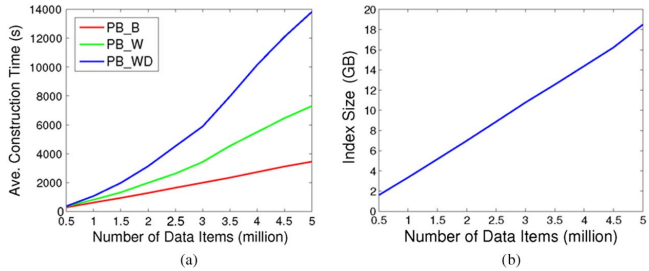

Fig. 9. Time and size. (a) Construction time. (b) Index size.

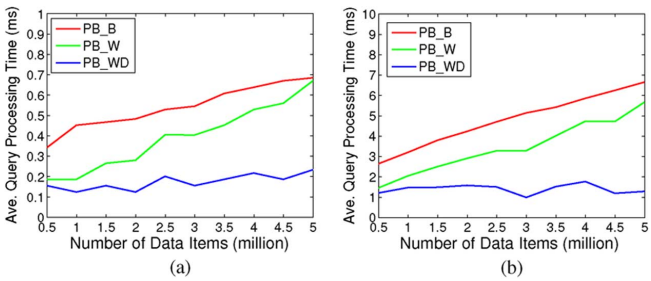

Fig. 10. Average query time of prefix queries. (a) R. size = 10. (b) R. Size = 90.

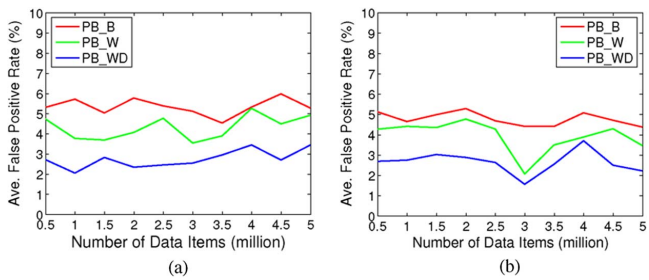

Fig. 11. Average false positive rate of prefix queries. (a) R. Size = 10. (b) R.Size = 90.

and the PB_WD is 357 to 14027 s, respectively. The average time required for the PB_WD construction is higher due to the equal-size partition algorithm and common prefix computation overhead involved. However, as we show later, the query processing time for PB_WD is smaller compared to the other two variants of the PBtree, and the false positive rate is lower as well. Fig. 9(b) shows that the PBtree sizes range from 1.598 to 18.494 GB for the datasets, and also, for a specific dataset size, the PB_B, PB_W, and PB_WD index structures are of the same size, respectively. Finally, the PBtree construction incurs a one-time offline construction overhead.

C. Query Evaluation Performance

1) Prefix Query Evaluation: Our experimental results show that the width and the depth optimizations are highly effective in reducing the query processing time and the false positive rates. We denote the average query result size as "R.Size" in the figures. Figs. 10 and 11 show the average prefix query processing times and false positive rates, respectively, on different datasets, for prefix queries issued on the corresponding PB_B, PB_W, and PB_WD structures.

The PB_WD structure exhibits higher query processing efficiency and records lower false positive among all PBtree structures. From the figures, we note that, for the same the query

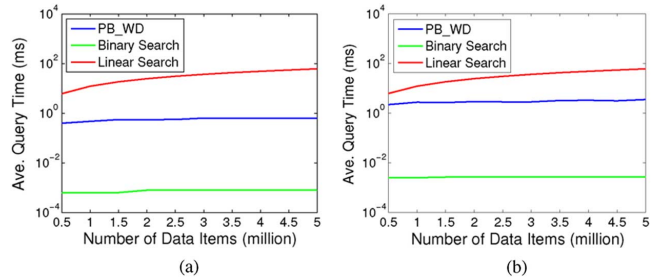

Fig. 12. Average query time of range queries. (a) R. size = 10. (b) R. Size = 90.

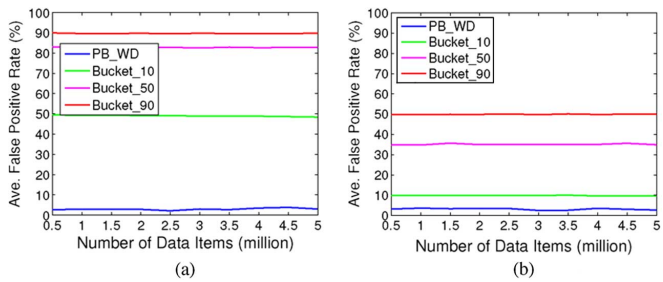

Fig. 13. Average false positive rate of range queries. (a) R. Size = 10. (b) R. Size = 90.

result sizes, PB_WD executes 2.153 and 2.533 times faster than PB_B, respectively; the corresponding false positive rates in PB_WD are 0.88 and 0.83 times smaller than in PB_B. In comparison, for the same query result sizes, PB_W executes 0.516 and 0.444 times faster than PB_B, respectively; the corresponding false positive rates in PB_W are 0.25 and 0.21 times smaller than PB_B.

2) Range Query Evaluation: Compared to existing schemes, our experimental results show that PB_WD has smaller range query processing times and lower false positive rates. We compared the speed of PB_WD to two plaintext schemes: *linear search*, in which we examine each item from the unsorted dataset to match the range query, and *binary search*, in which we execute the range query over the sorted data using the binary search algorithm. To evaluate the accuracy of PB_WD, we compared the recorded false positive rates to those observed in the *bucket* scheme of [25]. In our experiments, both the data items and the queries follow uniform distribution and, hence, each bucket contains same number of data items with bucket sizes ranging from 10 to 90. Figs. 12 and 13 show the average range query processing time and the false positive rates, respectively, for different query result sizes on the experimental datasets. We observed that, for the three query result sizes, the plaintext binary search is, respectively, 116 and 110 times, faster than the corresponding search results on the PB_WD structure. On the other hand, PB_WD performs, 14.8 and 1.748 times, faster query processing than the linear search scheme. Note that we use logarithmic coordinates in Fig. 12.

In terms of accuracy, PB_WD outperforms the bucket scheme [25] by orders of magnitude. For instance, for the maximum query result size of 90 in our experiments, the false positive rates recorded by PB_WD are 2.12 and 39.96 times lesser than the bucket scheme with respective bucket sizes being 10 and 90.

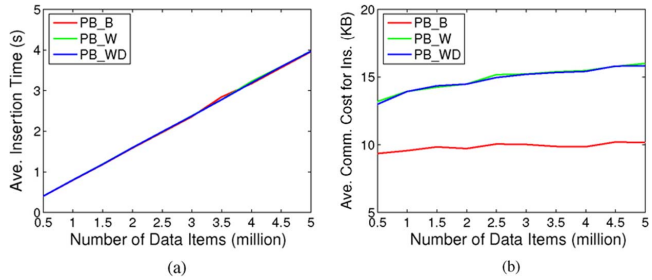

Fig. 14. Insertion. (a) Average time. (b) Average communication cost.

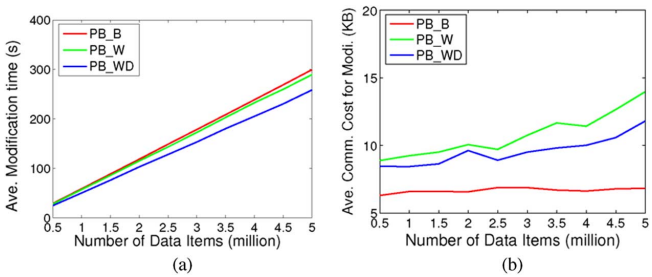

Fig. 15. Modification. (a) Average time. (b) Average communication cost.

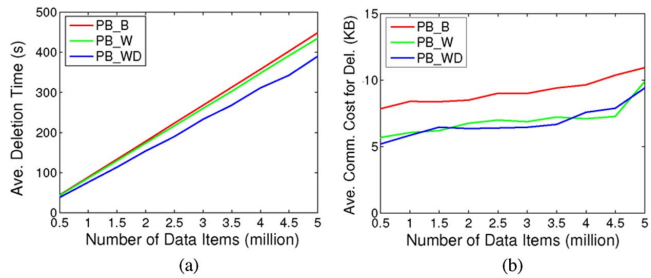

Fig. 16. Deletion. (a) Average communication cost. (b) Average communication cost.

D. Experimental Results on Updating

We measured the average update time and the communication cost for inserting, modifying, and deleting one data item on the PBtree structures built on the datasets. Note that, in our experiments, we only measure the update cost for the data owner and not for the cloud server. This is because the cloud server only needs to change the Bloom filter values specified by the data owner, and this is a negligible overhead for the cloud server.

1) Average Time of Updating: Our experiments confirm that the update time increases in proportion to the size of the dataset. Figs. 14(a), 15(a), and 16(a) show the average time of inserting, modifying, and deleting one data item on the corresponding PBtree, respectively. Among the three operations, due to the design of the Bloom filter, inserting an element is the most efficient, while modifying and deleting elements are the most expensive operations.

Regardless of the type of PBtree, PB_WD, PB_W, or PB_B, the insertion time is observed to be identical, i.e., as the dataset sizes progressively increase from 0.5 to 5 million data items, the insertion time increases from 0.396 to 3.96 s, respectively. We note that a similar trend is observed for modifying an element, i.e., the modification time is nearly similar across all three PBtree variants with minor differences. For the experimental datasets, with increasing dataset sizes, the modification time of PB_WD increases from 25.011 to 258.687 s, and the modification time of PB_W and PB_B increases from 28.388 to 289.666 s. However, deleting an element from PB_WD is slightly faster than deleting an element from PB_W and PB_B. As the dataset sizes increase, the average time for deleting an element from PB_WD increases from 37.656 to 389.484 s, and the deletion time for PB_W and PB_B increases from 42.597 to 434.183 s.

2) Communication Cost of Updating: Our experiments show that the communication cost for the update operation is practical and grows slowly with the dataset size. The communication cost is measured by the information size that the data owner sends to the cloud server to modify the specified Bloom filter locations in the PBtree. Figs. 14(b), 15(b), and 16(b) show the average communication cost of inserting, modifying, and deleting one data item on the corresponding PBtree, respectively. To inform the cloud server of the update, the data owner uses a 9-B metadata structure, in which the first 4 B specify the Bloom filter ID, the next 4 B specify the cell location in the Bloom filter, and the last byte contains the new bit value to be updated to the cell location. The update communication costs for PB_WD and PB_W are almost similar and are slightly smaller when compared to communication cost for updating PB_B. For inserting a data item, as the dataset sizes increase, the observed communication cost for PB_WD and PB_W increases from 12.98 to 15.82 kB, and the communication cost for PB_B increases from 9.34 to 10.16 kB. Similarly, the communication cost for modifying an item in PB_WD and PB_W increases from 8.47 to 11.82 kB, and the communication cost for PB_B increases from 6.32 to 6.9 kB. Finally, as the dataset sizes increase, the communication cost of deleting an item from PB_WD and PB_W increases from 5.186 to 9.39 kB, and the cost for PB_B increases from 7.84 to 10.93 kB.

VIII. Conclusion

In this paper, we propose the first range query processing scheme that achieves index indistinguishability, under the IND-CKA [18], which provides strong privacy guarantees. The key novelty of this paper is in proposing the PBtree data structure and associate algorithms for PBtree construction, searching, optimization, and updating. We implemented and evaluated our scheme on a real-world dataset. The experimental results show that our scheme can efficiently support real-time range queries with strong privacy protection.

Acknowledgment

The authors would like to thank the anonymous reviewers for their valuable suggestions and feedback for improving the paper significantly. They also like to thank M. Canim and M. Kantarcioglu for providing the source code of their work in [25].

References

[1] Amazon, "Amazon Web services," [Online]. Available: aws.amazon.com
[2] Google, "Google App Engine," [Online]. Available: code.google.com/appengine
[3] Microsoft, "Microsoft Azure," [Online]. Available: http://www.microsoft.com/azure
[4] T. Krazit, "Google fires engineer for privacy breach," 2010 [Online]. Available: http://www.cnet.com/news/google-fired-engineer-for-privacy-breach/

[5] R. Agrawal, J. Kiernan, R. Srikant, and Y. Xu, "Order preserving encryption for numeric data," in *Proc. SIGMOD*, 2004, pp. 563–574.
[6] R. Agrawal and R. Srikant, "Privacy-preserving data mining," in *Proc. ACM SIGMOD*, 2000, pp. 439–450.
[7] L. Ballard, S. Kamara, and F. Monrose, "Achieving efficient conjunctive keyword searches over encrypted data," *Inf. Commun. Security*, vol. 3783, pp. 414–426, 2005.
[8] M. Bellare, A. Boldyreva, and A. Oeill, "Deterministic and efficiently searchable encryption," in *Proc. CRYPTO*, 2007, pp. 535–552.
[9] A. Boldyreva, N. Chenette, Y. Lee, and A. O'Neill, "Order-preserving symmetric encryption," in *Proc. EUROCRYPT*, 2009, pp. 224–241.
[10] A. Boldyreva, N. Chenette, and A. O'Neill, "Order-preserving encryption revisited: Improved security analysis and alternative solutions," in *Proc. CRYPTO*, 2011, pp. 578–595.
[11] D. Boneh, G. D. Crescenzo, R. Ostrovsky, and G. Persiano, "Public key encryption with keyword search," in *Proc. EUROCRYPT*, 2004, pp. 506–522.
[12] X. Boyen and B. Waters, "Anonymous hierarchical identity-based encryption (without random oracles)," in *Proc. CRYPTO*, 2006, pp. 290–307.
[13] R. Canetti, U. Feige, O. Goldreich, and M. Naor, "Adaptively secure multi-party computation," in *Proc. 28th ACM Symp. Theory Comput.*, 1996, pp. 639–648.
[14] N. Cao, C. Wang, M. Li, K. Ren, and W. Lou, "Privacy-preserving multi-keyword ranked search over encrypted cloud data," in *Proc. IEEE INFOCOM*, 2011, pp. 829–837.
[15] Y.-C. Chang and M. Mitzenmacher, "Privacy preserving keyword searches on remote encrypted data," in *Proc. 3rd ACNS*, 2005, pp. 442–455.
[16] R. Curtmola, G. A. J. S. Kamara, and R. Ostrovsky, "Searchable symmetric encryption: Improved definitions and efficient constructions," *J. Comput. Security*, vol. 19, pp. 895–934, 2011.
[17] E. Damiani, S. C. Vimercati, S. Jajodia, S. Paraboschi, and P. Samarati, "Balancing confidentiality and efficiency in untrusted relational dbmss," in *Proc. CCS*, 2003, pp. 93–102.
[18] E.-J. Goh, "Secure indexes," Stanford University, Stanford, CA, USA, Tech. Rep., 2004.
[19] S. A. Eunjoon Cho and J. Leskovec, "Friendship and mobility: User movement in location-based social networks," in *Proc. 17th ACM SIGKDD KDD*, 2011, pp. 1082–1090.
[20] O. Goldreich and R. Ostrovsky, "Software protection and simulation on oblivious rams," *J. ACM*, vol. 43, no. 3, pp. 431–473, May 1996.
[21] P. Golle, J. Staddon, and B. Waters, "Secure conjunctive keyword search over encrypted data," *Appl. Cryptography Netw. Security*, vol. 3089, pp. 31–45, 2004.
[22] P. Gupta and N. McKeown, "Algorithms for packet classification," *IEEE Netw.*, vol. 15, no. 2, pp. 24–32, Mar.–Apr. 2001.
[23] H. Hacigumus, B. Iyer, C. Li, and S. Mehrotra, "Executing sql over encrypted data in the database-service-provider model," in *Proc. SIGMOD*, 2002, pp. 216–227.
[24] B. Hore, S. Mehrotra, M. Canim, and M. Kantarcioglu, "Secure multidimensional range queries over outsourced data," *VLDB J.*, vol. 21, no. 3, pp. 333–358, Jun. 2012.
[25] B. Hore, S. Mehrotra, and G. Tsudik, "A privacy-preserving index for range queries," in *Proc. VLDB*, 2004, pp. 720–731.
[26] S. Kamara, C. Papamanthou, and T. Roeder, "Dynamic searchable symmetric encryption," in *Proc. ACM CCS*, 2012, pp. 965–976.
[27] M. Kantarcioglu and C. Clifton, "Security issues in querying encrypted data," in *Proc. DBSec*, 2005, pp. 325–337.
[28] J. Katz and Y. Lindell, *Introduction to Modern Cryptography*. London, U.K.: Chapman & Hall/CRC Press, 2007.
[29] F. Li, M. Hadjieleftheriou, G. Kollios, and L. Reyzin, "Authenticated index structures for outsourced databases," in *Handbook of Database Security*. New York, NY, USA: Springer, 2008, pp. 115–136.
[30] J. Li and E. R. Omiecinski, "Efficiency and security trade-off in supporting range queries on encrypted databases," in *Proc. DBSec*, 2005, pp. 69–83.
[31] N. Li, T. Li, and S. Venkatasubramanian, "T-closeness: Privacy beyond k-anonymity and l-diversity," in *Proc. 23rd IEEE ICDE*, Apr. 2007, pp. 106–115.
[32] A. X. Liu and F. Chen, "Collaborative enforcement of firewall policies in virtual private networks," in *Proc. ACM PODC*, 2008, pp. 95–104.
[33] S. R. M. Oliveira and O. R. Zaïane, "Privacy preserving clustering by data transformation," *J. Inf. Data Manage.*, vol. 1, no. 1, pp. 37–52, 2010.
[34] D. Park, K. Kim, and P. Lee, "Public key encryption with conjunctive field keyword search," *Inf. Security Appl.*, vol. 3325, pp. 73–86, 2005.
[35] A. Shamir, "Identity-based cryptosystems and signature schemes," in *Proc. CRYPTO 84*, 1985, pp. 47–53.
[36] E. Shi, J. Bethencourt, T.-H. H. Chan, D. Song, and A. Perrig, "Multi-dimensional range query over encrypted data," in *Proc. IEEE S&P Symp.*, 2007, pp. 350–364.
[37] D. Song, D. Wagner, and A. Perrig, "Practical techniques for searches on encrypted data," in *Proc. IEEE S&P Symp.*, 2000, pp. 44–55.
[38] L. Sweeney, "K-anonymity: A model for protecting privacy," *Int. J. Uncertain. Fuzziness Knowl.-Based Syst.*, vol. 10, no. 5, pp. 557–570, Oct. 2002.
[39] C. Wang, N. Cao, K. Ren, and W. Lou, "Enabling secure and efficient ranked keyword search over outsourced cloud data," *IEEE Trans. Parallel Distrib. Syst.*, vol. 23, no. 8, pp. 1467–1479, Aug. 2012.
[40] W. K. Wong, D. W.-L. Cheung, B. Kao, and N. Mamoulis, "Secure knn computation on encrypted databases," in *Proc. SIGMOD*, 2009, pp. 139–152.

Rui Li received the M.S. degree from Central South University, Changsha, China, in 2004, and the Ph.D. degree from Hunan University, Changsha, China, in 2012, both in computer science.

He is currently an Associate Professor with Dongguan University of Technology, Dongguan, China. His research interests are in cloud computing, sensor networks, and security.

Dr. Li received the Outstanding Young Teacher Award of Hunan Province in 2009 and the Distinguished Young Teacher Award by Luanxiong Liu Funding in 2010.

Alex X. Liu received the Ph.D. degree in computer science from the University of Texas at Austin, Austin, TX, USA, in 2006.

His research interests focus on networking and security.

Dr. Liu is an Associate Editor of the IEEE/ACM TRANSACTIONS ON NETWORKING, an Editor of the IEEE TRANSACTIONS ON DEPENDABLE AND SECURE COMPUTING, and an Area Editor of *Computer Communications*. He received the IEEE & IFIP William C. Carter Award in 2004, a National Science Foundation CAREER Award in 2009, and the Michigan State University Withrow Distinguished Scholar Award in 2011. He received Best Paper awards from ICNP 2012, SRDS 2012, and LISA 2010.

Ann L. Wang received the B.S. degree in information engineering and M.S. degree in computer science from Beijing University of Posts and Telecommunications, Beijing, China, in 2009 and 2012, respectively, and is currently pursuing the Ph.D. degree in computer science and engineering at Michigan State University, East Lansing, MI, USA.

Her research interests include networking, security, and privacy.

Bezawada Bruhadeshwar received the B.E. degree in electronics and communications from Osmania University, Hyderabad, India, in 1998, the M.S. degree in electrical and computer engineering and Ph.D. degree in computer science and engineering from Michigan State University, East Lansing, MI, USA, in 2000 and 2005, respectively.

He is currently a Post-Doctoral Researcher with the Department of Computer Science and Technology, Nanjing University, Beijing, China. His research interests are in security, networking, and dependable systems.