2019 IEEE 35th International Conference on Data Engineering (ICDE)

# ServeDB: Secure, Verifiable, and Efficient Range Queries on Outsourced Database

Songrui Wu[1], Qi Li[1], Guoliang Li[1], Dong Yuan[1], Xingliang Yuan[2], Cong Wang[3]
[1]Tsinghua University, [2]Monash University, [3]City University of Hong Kong

***Abstract*—Data outsourcing to cloud has been a common IT practice nowadays due to its significant benefits. Meanwhile, security and privacy concerns are critical obstacles to hinder the further adoption of cloud. Although data encryption can mitigate the problem, it reduces the functionality of query processing, e.g., disabling SQL queries. Several schemes have been proposed to enable one-dimensional query on encrypted data, but multi-dimensional range query has not been well addressed. In this paper, we propose a secure and scalable scheme that can support multi-dimensional range queries over encrypted data. The proposed scheme has three salient features: (1) Privacy: the server cannot learn the contents of queries and data records during query processing. (2) Efficiency: we utilize hierarchical cubes to encode multi-dimensional data records and construct a secure tree index on top of such encoding to achieve sublinear query time. (3) Verifiability: our scheme allows users to verify the correctness and completeness of the query results to address server's malicious behaviors. We perform formal security analysis and comprehensive experimental evaluations. The results on real datasets demonstrate that our scheme achieves practical performance while guaranteeing data privacy and result integrity.**

## I. Introduction

### A. Motivation and Background

With the fast development of cloud services, many data owners outsource their data to the cloud server to save the cost, while the cloud server supports query processing for data users. However, data security is rather important. Once the cloud server leaks or tampers the data, it will face serious privacy violations and other legal issues. For example, many datasets contain sensitive data, such as the electronic medical records (EMR) with age, location, gender, and address [1].

Although data encryption can alleviate the security and privacy concerns, it reduces the query functionality, e.g., supporting SQL queries on encrypted data. Some schemes are proposed to support queries on encrypted data [2], [3], [4], which aim to enable the cloud server to search only over encrypted data and return encrypted results. Despite promising, most of current schemes focus on limited functionality, such as keyword search [5] and single dimensional range query [6]. Complex query functionality like multi-dimensional range query is rarely investigated. For example in an EMR dataset, a SQL query `select * from users where 33 ≤longitude ≤ 35 and 20 ≤ latitude ≤ 22 and 25 ≤ age ≤ 45` finds users whose ages are in a range and locations are in a range.

In this paper, we focus on the multi-dimensional range queries, which are fairly common in database. There are three challenges to design a method to protect the data security in outsourced database. (1) Privacy. The method should protect the confidentiality of data records and queries. (2) Efficiency. The method should achieve as efficient as sublinear or even constant query processing time for users. (3) Verifiability. The query result returned from the cloud server should be verifiable to guarantee its correctness and completeness.

One plausible approach is to use Order Preserving Encryption (OPE) [7] to encrypt the index structure (e.g., R-tree) in order to support multi-dimensional range query [8]. However, the order and distribution information of data records will be revealed due to the weak security notion of OPE [9]. Another approach is to employ searchable symmetric encryption (SSE) [5] that is designed for keyword search. Existing schemes [10], [11] extended SSE to support one-dimensional queries by transforming data records and queries into SSE codes and using the codes to enable query processing. An intuitive way of applying such schemes is to search for each dimension separately and return the intersection of the results in each dimension. As a result, it not only incurs significant delays but fails to ensure strong protection of multi-dimensional range queries. Specifically, each dimension is searched independently, and it might end up with revealing more matched records than desired [12]. Furthermore, most of the previous schemes assumed that the cloud server is semi-honest and follows the given rules [13], [8]. Unfortunately, it is not always true in practice since the server could be compromised and then behave maliciously. For example, the server may execute a fraction of the query operation to save its computation cost, or manipulate the query results deliberately. Therefore, it is necessary to enable verifiability in multi-dimensional range queries. In this paper, we consider the case that the cloud server could be malicious. We aim to design a framework for multi-dimensional range queries that provides privacy, efficiency, and verifiability simultaneously.

### B. Contribution

We propose an effective and efficient way to achieve verifiable multi-dimensional range query on encrypted data in the cloud server. To achieve multi-dimensional range query, we develop a hierarchical cube based encoding method to preprocess the dataset and query range respectively. During this process, we map the $d$-dimensional data to a $d$-dimensional space, and divide the space into hierarchical $d$-dimensional cubes. Accordingly, we can transform the data records and query ranges into a number of cube codes to be used in query processing. After pre-processing of the dataset at the user client, an encrypted index based on tree structure is built to achieve the sub-linear query time complexity. Each node of the tree index uses a Bloom filter to store the encrypted data, and the data records are split randomly and evenly into its child nodes. The structure of index just depends on the number of data records which will protect the information like the

2375-026X/19/$31.00 ©2019 IEEE
DOI 10.1109/ICDE.2019.00062

data distribution. To verify the correctness and completeness of results, we design a verification mechanism for our scheme. Instead of introducing a new data structure, we leverage the tree index used for range query to realize the verification at the same time. The proof information generated by the server links to the search path for each query request. Hence, the user can reproduce the query process with the proof information and verify the correctness for each step of searching on the tree nodes. Our contributions can be summarized as follows:

(1) We propose a secure, verifiable, and efficient framework, `ServeDB`, to support encrypted multi-dimensional range query on outsourced database. This is the first method to achieve privacy, efficiency and verifiability simultaneously.
(2) We use hierarchical cubes to encode the data and build a tree-structure index on top of the data encoded by cubes to support efficient query processing.
(3) We add verification information in the tree structure and develop effective methods to verify the correctness and completeness of results without introducing new structure.
(4) We formally analyze the security properties of our scheme, including the guarantees on the confidentiality of data and queries, and the correctness and completeness of query results.
(5) Experimental results on real-world datasets show the efficiency and scalability of our scheme.

## II. Problem Statement and Related Work

### A. Problem Definition

**System model.** Our scheme involves three kinds of entities: a data owner, a cloud server, and data users.

(1) Data Owner. The data owner owns a dataset with $n$ data records: $D$ = {$D_1$, ..., $D_n$}. Before uploading the dataset to the cloud server, the data owner encrypts each record $D_i$ into $E_i$ and builds a secure index $\Gamma$ to preserve the data privacy. The data owner submits the encrypted data $E$ and the corresponding index $\Gamma$ to the cloud server.

(2) Data User. Data users are trusted and authorized to have the secret keys of the data owner. The data user posts a query range $Q$, encrypts the query, and sends the query request to the cloud server. After receiving the search results, the data user verifies the correctness and completeness of the results.

(3) Cloud Server. The cloud server stores the encrypted data and index from the data owner, and processes multi-dimensional range queries for data users. In this paper, we assume that the server will not only be interested in the contents of data records and queries, but also be able to compromise the integrity of the query results.

Now we present the definitions of the data model and the query model in our scheme respectively.

**Definition 1** (Data Model)**.** *A dataset $D$ is modeled as a relational table with $d$ attributes $\{a_1, a_2, \ldots, a_d\}$ and $n$ records $\{D_1, ..., D_n\}$. We assume that each attribute $a_j$ for the data record $D_i$ has a numerical value.*

**Definition 2** (Query Model)**.** *A query $Q$ is defined as $Q = \{[a_1.low, a_1.up], [a_2.low, a_2.up], ..., [a_d.low, a_d.up]\}$, which has the same $d$ attributes with the data records. $a_i.low$ and $a_i.up$ represent the lower bound and upper bound of the search range for the attribute $a_i$.*

Our method can be extended to support arbitrary $i$-dimensional range query ($i \leq d$) as discussed in Section IV-E.

Then we formulate our query and verification scheme in Definition 3. The scheme contains five important components. The *Setup* algorithm generates keys for data owners. The *Index* algorithm builds the encrypted tree index for the data owners. The *Trapdoor* algorithm generates tokens for data users to perform range query. The *Search* algorithm allows cloud servers to search encrypted data by using the tree index and the tokens. The *Verification* algorithm that verifies the correctness and completeness of query results.

**Definition 3.** (Verifiable Multi-Dimensional Range Query). *Our verifiable multi-dimensional range query scheme is a tuple of five polynomial time algorithms:*

***Setup***$(1^\lambda) \rightarrow (SK, HK, Rand, sk)$ *: is a probabilistic key generation algorithm that is run by the data owner to generate different keys required in range query. It takes a security parameter $\lambda$ as input, and outputs a secret key $SK$ to encrypt the dataset, a set of hash keys $HK = \{k_1, k_2, ..., k_r\}$ and random numbers $Rand = \{Rand_1, ..., Rand_{2n-1}\}$ used to generate the secure index, and a secret key $sk$ used to generate proof information $\pi$.*

***Index***$(D, HK, sk) \rightarrow (\Gamma)$ *: is a probabilistic algorithm run by the data owner to build a secure tree structure to index data records and support verification. It takes dataset $D$, keys $HK$ and $sk$ as input and outputs the tree index $\Gamma$.*

***Trapdoor***$(SK, HK, Q) \rightarrow (M(Q))$ *: is a deterministic algorithm run by the data user to generate trapdoor matrix $M(Q)$ for a given queried range $Q$, which is used as tokens to perform range query. It takes a secret key $SK$, hash key set $HK$, and a range lsefamily $Q$ as input, and outputs a trapdoor matrix $M(Q)$.*

***Search***$(\Gamma, E, M(Q)) \rightarrow (R, \pi)$ *: is a deterministic algorithm run by the cloud server to search on the ciphertext set $E$ through the encrypted index $\Gamma$. It takes the encrypted tree index $\Gamma$, ciphertext set $E$ = {$E_1$, ..., $E_n$}, and the trapdoor matrix $M(Q)$ as input, and outputs the search result set $R$ and verification proof $\pi$.*

***Verification***$(Q, R, \pi, sk) \rightarrow (accept/reject)$ *: is a deterministic algorithm run by the data user to verify the correctness and completeness of results. Given a query $Q$, result set $R$, proof $\pi$, and $sk$, the user verifies the result and outputs either accept or reject.*

A verifiable multi-dimensional range query scheme should ensure correctness and completeness of query results.

**Definition 4** (Correctness)**.** *Given a query $Q$, the cloud server returns the result set $R = \{R_1, R_2, ...R_r\}$. If for each result $R_i$ ($i <= r$), the decrypted data $D_i$ of $R_i$ falls in the range $Q$, then the result set $R$ is correct.*

**Definition 5** (Completeness)**.** *Given a query $Q$, the cloud server returns the result set $R = \{R_1, R_2, ...R_r\}$. If each record $D_i \in D$ such that $D_i$ is in $Q$, the encrypted data $R_i$ must in $R$, then $R$ is complete.*

We aim to design a multi-dimensional range query scheme that can protect data confidentiality, query confidentiality, and search result confidentiality. Similar to existing SSE

schemes [10], [11], [14], we do not focus on defending against the side channel attacks, which are extensively addressed [15], [16]. For instance, attacks to search and access patterns can be prevented by obfuscation [15], [16].

### B. Related Work

**Encrypted Range Query.** Privacy-preserving range queries have advanced rapidly in recent years. Theoretically, it could be solved with generic yet sophisticated cryptographic tools such as ORAM [3] and garbled circuit [4]. However, these tools incurred prohibited query delays in large databases. One possible approach to improving the efficiency is to apply Order-preserving Encryption (OPE), e.g., CryptDB [7]. However, generated ciphertext preserves the ordering of the plaintext, which is vulnerable to inference attacks [8]. Efficient structures [8], [13], [17], [10] are developed for range query over encrypted data. However, they do not have confidentiality guarantee when they are applied in multi-dimensional range query. Multi-dimensional tree structures, such as kd-trees [18] and R-trees [19], are used to achieve sub-linear multidimensional range searches. However, these approaches reveal single-dimensional privacy or/and data distributions on the trees. To address these issues, extra hardware is leveraged [20]. Moreover, these schemes do not have verifiability for range query. Our scheme enables privacy-preserving range query with ensured security and verifiability.

**Encrypted Keyword Query.** Traditional SSE [12], [21], [22] has been extended to enable privacy-preserving range query. It allows controlled leakage during the query, aka the access patterns of each query, as well as the search patterns. Two recent designs [10], [11] formalized range query in the context of SSE. Demertzis et al. [11] implemented multi-keyword search using range covering techniques built upon SSE. However, these approaches only focus on one-dimensional range query and do not enable query result verification. Our scheme is designed to achieve multi-dimensional range query with efficient performance and verifiability of the results.

**Verification.** Merkle trees [23] and accumulation trees [24] were extensively studied. For instance, Zhang et al. [25] employed the bilinear accumulator primitive and utilized the accumulation tree to achieve the verifiable process. However, these schemes required exponential query complexities to enable possible combination of dimensions in search verification. Papadopoulos et al. [26] proposed the first scheme that achieved linear verification with the numbers of dimensions. Data owners built a novel authenticated structure over every database attribute separately and bound all structures using an existing set membership structure. Our scheme is independent of the number of dimensions, which has high efficiency and can protect the security for each dimension. Signature aggregation and chaining [27] was also used to verify the integrity of search results. However, these schemes required cloud providers to reply the customers about the boundary data items of the queries, which leaked sensitive information. `ServeDB` enables verification that directly leverages our query index `SVETree`. It achieves sub-linear verification performance while ensuring efficient multi-dimensional range query.

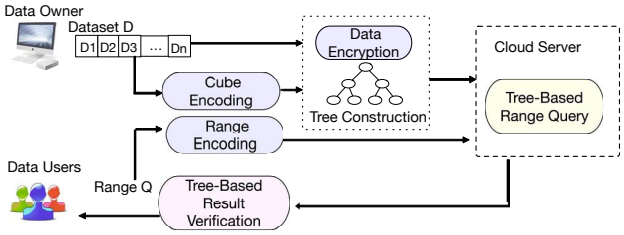


Fig. 1: Overview of `ServeDB`.

## III. Scheme Overview

The goal of our scheme is to answer privacy-preserving multi-dimensional range query with verifiability of query results. We leverage an index tree structure to perform range query, where each leaf node is indicated by an encoded cube that is mapped from different dimensions data (or ranges). Range query on the tree structure ensures the query accuracy, while each encoded node preserves the privacy of queried data. Meanwhile, we design a verification mechanism that leverages the tree index to verify the correctness and completeness of query results. The main procedure is shown in Figure 1. Note that, our scheme can be applied to query datasets with both numeric and non-numeric data. In particular, it can directly process data with non-numeric values and encode the data into different cube codes without data pre-processing. For simplicity, we do not focus on addressing query over non-numeric data in this paper.

**Data Encryption.** To preserve data security, the data owner encrypts the data and stores the encrypted data in the cloud server. When the data user gets the result from the cloud server, the data user uses its private key set to decrypt the data. Any semantic security encryption algorithm can be used here and we use Advanced Encryption Standard (AES) in this paper.

**Hierarchical Cube Based Encoding.** In order to query all data with different dimensions, we transform the data ranges into discrete cube codes so that we can compare data records and query range by comparing the cube codes. In particular, we map data with different dimensions into one dimension, which enables efficient range query by performing query operations once. Formally, for each record $D_i$, a set of codes $C_i$ is generated. Given a range query $Q$, a set of codes $C_Q$ is generated. If $C_i \cap C_Q \neq \phi$, $D_i \in Q$ matches the query. Note that it is non-trivial to decide an appropriate cube width. If the width of the cubes is too large, it achieves good efficiency because it is fast to check whether the query cube set and data cube set overlap; but it involves false positives (see Section IV-A). On the other hand, if the cube width is too small, the code set $C_Q$ for the query $Q$ contains a large number of cube codes, leading to low performance. To address this issue, we propose a hierarchical cube based encoding method and Section IV-A presents the details.

**Secure, Verifiable, and Efficient Tree.** Although there exist a variety of indexes to check $C_i \cap C_Q \neq \phi$ (e.g., inverted lists, B+ tree, and kd-tree), they do not consider security and privacy requirements. Hence, we build a dedicated index denoted as `SVETree` to guarantee the security and enable verifiability while achieving high efficiency. `SVETree` is a balanced binary tree structure to index cube codes by the data owner. The root node contains the cube codes of all the data records and each

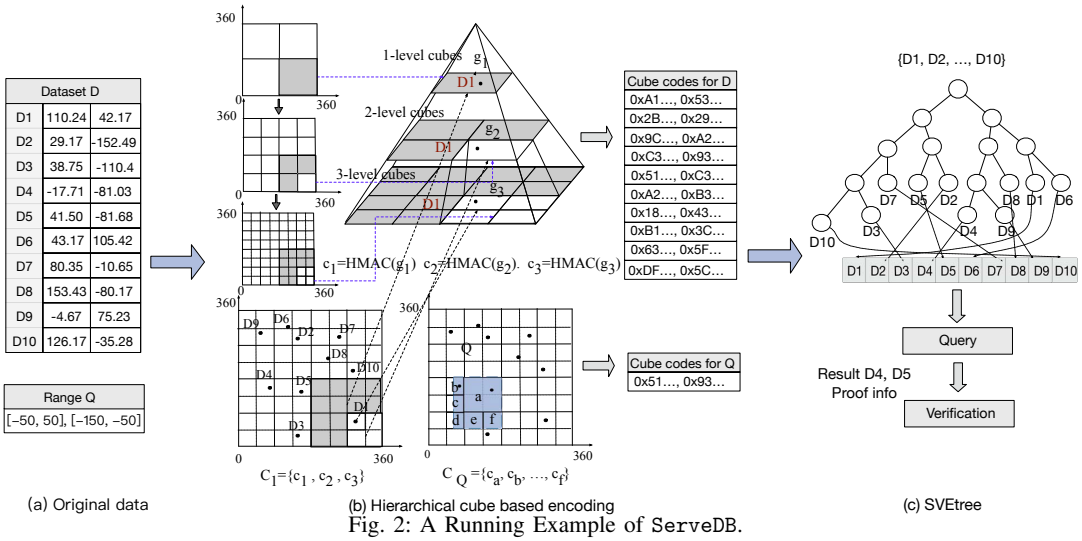

Fig. 2: A Running Example of ServeDB.

leaf node contains the cube of a single record with a pointer to the encrypted record. Each internal node contains the cubes of records in its leaf nodes. At each node, verification information is also added to enable verifiability. In brief, SVETree has the following salient features.

(1) Privacy preservation: SVETree protects the data confidentiality. Each node in SVETree is constructed via HMAC as the pseudo-random function. It is computationally indistinguishable for any two nodes of SVETree. The cube codes for each data record is hashed by $r$ times, and each hash value is mapped into a Bloom filter to disrupt the correlation between data records. Besides, when constructing SVETree, the whole dataset is split into the tree nodes randomly and evenly from the root node. Section V presents the security analysis of SVETree.

(2) High Efficiency: The index is a balanced binary tree which achieves sub-linear query complexity. Given a query $Q$, SVETree is searched from the root to leaves and the time complexity is the height of the tree, i.e., $\log(|D|)$ . Therefore, SVETree is scalable for large datasets.

(3) Verifiability: Our verification information Ver($v$) is added to the tree node $v$ when constructing the index. During query processing, the cloud server will generate proof information $\pi$ which is related to the search path and allows data users to verify the query results using the proof $\pi$ according to SVETree. If the cloud server follows the given search rules, the results can be proved to be correct and complete.

**Verification.** ServeDB ensures the completeness and correctness of query results by verifying each query process, instead of directly verifying the query results. A verification process verifies result correctness and completeness that are defined in Definition 4 and 5. As for result correctness, we use the idea of Merkle tree to insert hash value in each tree nodes of our SVETree. If the result is correct, the hash value of them can generate the root's hash value correctly from bottom to up. To achieve result completeness, the idea is to reproduce the query process at the data user side using the proof information. For each query request, it corresponds to a set of search paths. If each step can be reproduced on these paths, the cloud server follows the given rules and the search results will be complete. Note that, one could directly utilize the existing replay-proof mechanism in SSE [5]. We do not discuss the proof replay attacks in this paper.

Let us take an example shown in Figure 2. As shown in Figure 2(a), the original dataset includes ten 2-dimensional records $\{D_1, D_2, ..., D_{10}\}$ and a 2-dimensional query Q. We use the cube encoding method to transform each data in the dataset into hierarchical cubes including 1-level cubes, 2-level cubes and 3-level cubes (see Figure 2(b)), and then map data records and the queried range into the encoded space and the corresponding cube sets. We construct an SVETree structure according to the cube sets, and then perform and verify the data query according to the SVETree. As shown in Figure 2(b), data $D_1$ is mapped into cube $g_1$, $g_2$, and $g_3$ and encoded into $C_1$=$\{c_1, c_2, c_3\}$, and range Q is mapped and encoded into $C_Q$=$\{c_a, c_b, ..., c_f\}$. We can construct an SVETree structure according to the cube sets (see Figure 2(c)), which is a balanced binary tree structure. We can perform and verify data query according to the built SVETree. Note that, our scheme can be implemented as part of an actual database such as a SQL database. We can store the encrypted data as a table and design the tree structure as an index (similar to R-tree) so that the query and verification mechanism can be performed by searching the data in the table. Also, in order to enable query verification, ServeDB incurs the extra overhead to deliver verification information.

## IV. SVETree

We first propose a hierarchical cube encoding method that transforms the data into cubes in Section IV-A. We then present the SVETree to index data records in Section IV-B. Next we discuss how to utilize SVETree to answer multi-dimensional queries and generate corresponding proof information $\pi$ in Section IV-C. Then the verification process is discussed in Section IV-D. Finally we discuss the extensions of our method in Section IV-E.

### A. Hierarchical Cube Encoding

To transform the numeric value of the data records and query range into cube codes, we need to provide an encoding

rule between the data owner and the data user. To this end, we first define a uniform cube coding system (CCS) and then use CCS to encode both the data and query in SVETree.

**Uniform Cube Coding System.** The data owner first determines the value range in each dimension based on the minimal value and maximal value, and get a $d$-dimensional multi-cube where each dimension represents the corresponding value range. It takes the cube as the root. Then it divides the cube into $2^d$ equal-size sub-cubes, takes them as children of root, and gets the first-level sub-cubes. For each $i$-level sub-cube, it recursively divides the cube into $2^d$ sub-sub-cubes as his children and gets the $i+1$ level cubes. Iteratively it can build the $k$-level hierarchical cubs until the side length is smaller than a given threshold. For example, let us follow the example of the hierarchical cube structure shown in Figure 2(c), where the data is 2-dimensional. We map the values of the attributes into the $x$- and $y$-axis in the 2-dimensional space, respectively, where the values in $x$- and $y$-axis are [-180,180]. Similarly, we can further divide each computed cube into four sub-cubes.

The next step encodes each cube in different levels and ensures the uniqueness of a code. Suppose the central point of a cube $\mathrm{g}_i$ is $(\mathrm{g}_i.x, \mathrm{g}_i.y)$. We first concatenate $\mathrm{g}_i.x$, $\mathrm{g}_i.y$ and the level of the cube $\mathrm{L}_i$. Then we compute its hash code using a one-way hash function with a secret key $K$. The final cube code of $\mathrm{g}_i$ is: $\mathrm{c}_i$ = HMAC($K$, ($\mathrm{g}_i.x$+$\mathrm{g}_i.y$+$\mathrm{L}_i$)). Based on this idea, we can transform each cube to a unique code and build a cube coding system (CCS). The data owner and data user utilize the same CCS to generate the cube set for the data and query respectively.

**Code Generation for Data Owner.** Given a data record, it should belong to $k$ cubes (from the first level cube to the $k$-th level), where $k$ is the number of levels in the cube hierarchy. Therefore, each data record $D_i$ corresponds to $k$ codes $C_i$={$c_1$, $c_2$, ..., $c_k$}. For example, in Figure 2, $D_1$, $D_2$, ...,$D_{10}$ are data records. The $x$- and $y$-value represent the two attributes of the data records. For data record $D_1$, we can easily find that $g_1$, $g_2$ and $g_3$ cover this point, then we can generate its code set $C_1$ which contains the codes of these cubes.

**Code Generation for Data User.** The aim of the data user is generating a series cubes codes that can entirely cover the given range. We generate the code set of a query $Q$ by accessing the CCS hierarchy in an up-down manner. From the root, we find the cubes that overlap with the query range. If the cube is fully covered by the range, we add its code as a code of the query; otherwise, we access its children and repeat the above steps. If the cube is in a leaf node, we add its code into the code set of the query. In Figure 2, the query range $Q$ is encoded into $C_Q$={$c_a$, $c_b$, ...,$c_f$}.

$C_Q$ is generated in an up-down manner which contains the fewest token covers the given range $Q$. The reason why we use the cube hierarchical is that it can reduce the code size of the query. For example, if an $i$-th level cube is covered by the range, we can skip its children cubes in the $i+1$ level, thus the query process will be more efficient. For a query range, we may introduce some cubes not fully covered by the query, but this method may incur false positives. Obviously, if the cube is too large, it will involve more false positives. Otherwise, it will generate many codes for a query and the query efficiency will be low. Data users can easily remove the false positives by simply removing the data records that are not in the query range. Actually, we find that false positives incurred by SVETree are similar to or lower than the existing methods (see Section VI).

### *B. SVETree Construction*

Based on cube codes of data records, we construct SVETree, which includes three steps: Build Tree, Encode Tree and Add Verification Information. The first step generates a balanced binary tree structure to index the codes. The second step encodes the tree node to make the index privacy preserving. The third step adds the verification information Ver(v) in each node $v$ to enable verification.

**Building Tree.** We first construct the root, which contains the codes of all the records $D$={$D_1$, $D_2$, ..., $D_n$}. Then we partition the set $D$ into two subsets $D^l$ and $D^r$ randomly and evenly, and take them as the two children of the root. We recursively apply the above step until each leaf node contains only one data record and the leaf nodes also index the corresponding encrypted data records. Figure 3(a) shows the detailed information of the tree structure $\Gamma$ in Figure 2(b).

**Encode Tree.** Note for each node we will not keep the codes and instead we use the Bloom filter which not only reduces the index size but also enables fast query processing. For a tree node $v$ with a set of records $D(v)$ generated from the first step, we use a Bloom filter denoted as $BL(v)$ to store each code for $D_i$ in its corresponding cube set $C_i$ (where $D_i \in D(v)$). We use $r$ hash functions for each code $c$ with $r$ hash keys $HK$={$key_1$, ..., $key_r$}. Note that, if a cube shared by two cube sets of two different nodes $v_1$ and $v_2$, for all locations computed by the hash value of the cube, Bloom filters will set the value as 1. To make the hash values unique and eliminate the correlation among different tree nodes, we introduce a random number $rand_v$ for each node $v$ as a hash key to re-hash the $r$ hash values: HMAC($rand_v$, HMAC($key_i$, $c$)). We utilize $rand_v$ to eliminate the correlations among different tree nodes to protect data privacy. If we use the same hash key to compute the hash code on each tree node when building the index of the cube code, it will be easy to find the relation between nodes by an attacker. For example, the cube $c_i$ is allocated into a left child, there must be the same hash value between the left child with the parent node. We store $rand_v$ in each node, and then insert every hash values on $BL(v)$. Given a query $Q$, we can use the Bloom filter to check whether each node contains codes falling in the range (see Section IV-C). The Bloom filter can reduce the space cost and increase randomness. Figure 3(b) shows the procedure of the tree node encoding. Node $V_2$ includes $\{D_1, D_4, D_6, D_8, D_9\}$, and each data in $V_2$ is mapped to three cube codes. Therefore, we can obtain the cube set corresponding to $V_2$, i.e., $C(V_2) = \{c_1, \cdots, c_{15}\}$. Each element $c_i$ will be encrypted 2*$k$ times by using $k$ hash keys and a random number $\mathrm{rand}_{v2}$ so that we can obtain $k$ encrypted values. These encrypted values will be mapped into the Bloom filter $BL(V2)$ for range query.

**Adding Verification Information.** Verification process has two aspects according to Definition 4 and Definition 5. As for data correctness, we follow the idea of Merkle tree to insert the data hash in each tree nodes. To achieve result

completeness, the idea is to reproduce the query process on each data user side using the proof information related to the query request. The cloud server needs to return part of the tree node information which will be introduced in Section IV-C (i.e. bloom filter segments). To guarantee the information in bloom filter, we introduce the hash signature for each bloom filter. The verification information $\texttt{Ver(v)}$ for each tree node $v$ includes: hash label of the node ($HL(v)$) and HMAC value of the Bloom filter ($HB(v)$). Figure 3(c) illustrates the verification information.

(1) Hash label of node ($HL(v)$): Similar to the idea of Merkle tree [23], for leaf node $v_l$ that indexes an encrypted data $E_i$=Enc($D_i$), we calculate a hash value on $E_i$ as the hash label of node $v_l$, $HL(v_l) = hash(E_i)$. For a non-leaf node $v_n$, we first concatenate the hash label of its children's, and then calculate $HL(v_n)$=$hash(HL(v_n.left)$ + $HL(v_n.left))$. From bottom to up, we generate the hash label $HL(v_{root})$ for the root, and the root's hash label is shared with data users.

(2) HMAC value of Bloom filter ($HB(v)$): To reproduce the query process on data user, only several bits on bloom filter should be used. Therefore, to reduce the communication cost, we segment $BL(v)$ and add the HMAC value of each segment as the verification information (See Section IV-E).

We use a running example to show how to construct a $\texttt{SVETree}$. As shown in Figure 3(a), we take ten data records as an example. We first construct a balanced binary tree structure with ten records as input. Each leaf node contains one record and each non-leaf node contains the dataset which is the union of its children's. The distribution of each record is randomly and unordered. Then we encode each node in Figure 3(b). Here we take $v_2$ as an example. $v_2$ contains five records $\{D_1, D_4, D_6, D_8, D_9\}$, and each of them has three cube codes. We use a random number $rand_{v2}$ and $r$ hash keys to calculate $r$ hash values on each cube and map these values to a Bloom filter $BL(v_2)$. Then we generate the verification information $Ver(v_2)$ for it: hash label of $v_2$ ($HL(v_2)$) and HMAC value of Bloom filter ($HB(v_2)$). Then we only store the Bloom filter $BL(v_2)$, random number $rand_{v2}$ and verification $Ver(v_2)$ in each node and delete other information to protect privacy.

### C. Query Processing

We introduce a fast query processing method based on $\texttt{SVETree}$, and discuss how to generate the proof $\pi$ which will be used in the verification step.

**Fast Query Processing.** We perform query by looking up the codes in Bloom filters of the nodes in the tree. When a data user posts a query request, the user needs to first encode the search range $Q$ into $C_Q$ using CCS (see Section IV-A). Then, the query process obtains desired data records by searching the data records having the common cube code with $C_Q$. Because the data owner hashes each cube code $r$ times when constructing the Bloom filter, the data user should use the same functions to hash $C_Q$. For each cube $g_j$ in $C_Q$, we calculate $r$ hash values and define the trapdoor set $T_j$ with $r$ hash values. Then the cube set $C_Q$ is transformed into a matrix $M(Q)$ with $|C_Q|$ trapdoor sets where each trapdoor set has $r$ hash values.

After receiving a trapdoor matrix $M(Q)$ from the data user, the cloud server uses $M(Q)$ to search over $\texttt{SVETree}$ from the root in a top-to-bottom manner. For each node $v$, we aim to check whether the cube set $C_Q$ has the common element with the codes in node $v$ through mapping each element in trapdoor on the Bloom filter of the tree node. Since we have added different random number $rand_v$ for each node, the cloud server should first hash each element in $T_i$ using $rand_v$ for node $v$, and then map the final hash value to $BL(v)$. If there exists a trapdoor set $T_i$ ($1 \leq i \leq |C_Q|$) in matrix $M(Q)$ that for every $j$ ($0 \leq j \leq r$), the corresponding positions of the hash values on the Bloom filter are all 1, then $C_Q \cap C(v) \neq \varnothing$. It means the trapdoor $T_i$ matches the Bloom filter of node $v$, and there are some data records with the common code with the query range in this subtree. Therefore, we can continue our search on this branch and visit its children. On the other hand, for $T_i$, if there exists an element that the corresponding position on the Bloom filter is 0, then $T_i \cap C(v)$=$\varnothing$. For the descendent node $v'$ of $v$, we have $T_i \cap C(v')$=$\varnothing$. Thus, the trapdoor $T_i$ is an un-matched trapdoor and we remove the trapdoor $T_i$ from the trapdoor matrix $M(Q)$ when we continue to search the descendant nodes of $v$. The searching process terminates if (1) $M(Q)$ becomes empty or (2) we finish searching the leaf node. As shown in Figure 3, the cube of $C(Q)$ is encoded into $M(Q)$, and the tokens will be matched by the corresponding Bloom filters, e.g., the token of $D_4$ matches $V_2$ and the query process will continue in $V_2$'s child nodes $V_5$ and $V_6$ until we find the final result in $V_{17}$.

**Generating Proof $\pi$.** As introduced before, if a trapdoor $T_i$, which is an unmatched trapdoor of node $v$, will be removed from $M(Q)$, there is no data record in the cube $g_i$ in the corresponding sub-tree. In particular, if the cloud server removes a matched trapdoor maliciously, the search results will be incomplete. To address this issue, we design a verification mechanism to track the searching processes of each trapdoor. If a trapdoor $T_i$ does not match a tree node $v$, the proof information in node $v$ should be returned to verify this removal operation. Note that, if the trapdoor $T_i$ matches a leaf node $v$, the proof information in node $v$ should also be returned to verify whether the data record indexed by $v$ is a correct result. To sum up, there are two conditions of tree nodes should be returned:

*Condition 1:* The matched leaf nodes that satisfy the query range. All the query answers are included in these nodes.

*Condition 2:* The unmatched tree nodes that do not satisfy the query range. If a tree node is unmatched, the sub-tree nodes will be ignored directly.

The tree nodes that satisfy the above conditions are defined as the *key nodes*. Figure 3 gives an example of generating key node set $K(Q)$ for query $Q$. For each tree node $v$, the $id$s of the unmatched trapdoors form the unmatched trapdoor set $UMT(v)$, and the $id$s of the matched trapdoors form a matched trapdoor set $MT(v)$.

For example, considering the query range $Q$ in Figure 2, the trapdoor matrix $M(Q)$=$\{T_a, T_b, ..., T_f\}$ and the data records $D_4$, $D_5$ are in this range. $D_4$ satisfies $T_a$ and $D_5$ satisfies $T_b$. We first check $M(Q)$ with root $v_0$. The trapdoor $T_a$ and $T_b$ match the Bloom filter $BL(v_0)$, and the rest trapdoors are removed from $M(Q)$. The $id$s of the unmatched trapdoors compose $UMT(v_0)$=$\{c, d, e, f\}$. Then the new trapdoor matrix $M(Q) = \{a, b\}$ continues searching on child node $v_1$

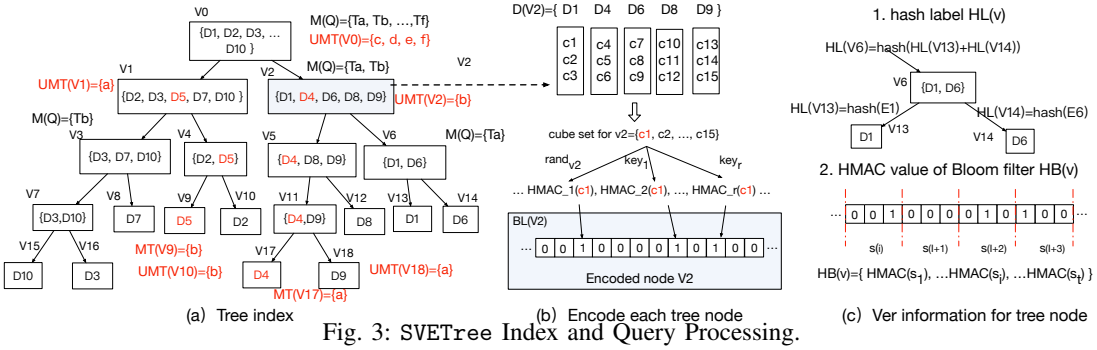


Fig. 3: SVETree Index and Query Processing.

and $v_2$. Similarly, we get $UMT(v_1)$={$a$} and $UMT(v_2)$={$b$} since $D_5$ in the sub-tree of $v_1$ and $D_4$ in the sub-tree of $v_2$. We continue the query process on each branch with a new trapdoor matrix until $M(Q)$ is empty or it obtains the leaf node. We can easily find that $UMT(\mathrm{v}_{10})$={$b$}, $UMT(\mathrm{v}_{18})$={$a$} and $v_0$, $v_1$, $v_2$, $\mathrm{v}_{10}$, $\mathrm{v}_{18}$ are key nodes for the second condition. The leaf nodes $v_9$ and $\mathrm{v}_{17}$ are key nodes for the first condition and $MT(v_9)$={$b$}, $MT(\mathrm{v}_{17})$={$a$}. The key nodes and their $UMT(v)$ and $MT(v)$ are given in Figure 4.

Now we discuss the proof information $\pi$ for each key node. There are four kinds of proof information that should be returned to enable verification:

(1) Hash label for each key node $v$: The data user should use hashed label $HL(v)$ for key nodes to generate the hash value of the root's label from bottom-up verify the set membership for each key node.

(2) The Bloom filter for each key node: The data user should re-map the trapdoor to the Bloom filter to verify the query processing on the cloud server during verification process. Therefore, the cloud server should return the Bloom filter of each key node $BL(v)$ and its HMAC value $HB(v)$ to the data user. If the Bloom filter of each node is too large, we will segment the Bloom filter values, which will be introduced in Section IV-E.

(3) Unmatched trapdoor set $UMT(v)$ and matched trapdoor set $MT(v)$. To re-map the trapdoor, the cloud server needs to return the $id$s of the trapdoor to the data user. Therefore, for the key nodes holding the first condition, the cloud server returns the matched trapdoor set $MT(v)$ and the unmatched trapdoor set $UMT(v)$ if they are not empty. For key nodes of the second case, the cloud server returns the unmatched trapdoor set.

(4) The random number $rand_v$. We introduce a random number $rand_v$ to eliminate the correlation among different tree nodes during SVETree construction. $rand_v$ is generated by data owner and stored in each node of the tree index. Each trapdoor should be re-hashed using $rand_v$ before mapping to the Bloom filter. If the data user needs to remap trapdoors during verification, the cloud server should return the random number for each key node.

### D. Verification

The cloud server returns the encrypted results and the proof $\pi$ to the data user. The data user uses proof $\pi$ to verify the correctness and completeness of the results.

**Verifying Correctness.** According to Definition 4, two aspects should be considered. On the one hand, the verification mechanism needs to check whether the query results fall into the given range. For this aspect, data user can easily decrypt the result set and check each data with range $Q$ in plaintext. If the results are outside the query range, the data user can delete them. On the other hand, the verification mechanism also needs to check the set membership for query results and key nodes to determine whether they are published by the data owner rather than making up by the cloud server. For this aspect, we use the hash label $HL(v)$ for each key node to generate a hash value $h_{root}$ of root from bottom to up. Similar to Merkle Tree [23], if $h_{root}$ equals to hash label of the root $HL(root)$, we can verify the results are members of the data set and the key nodes are members of the SVETree nodes.

**Verifying Completeness.** According to Definition 5, the basic idea is that if the data user can use the query results $R$ and proof information $\pi$ to reproduce a correct query process, the query results are complete and accepted; otherwise the query results are incomplete and rejected.

We first introduce the principle of our verification idea. In query processing, through observing each query path of the tree from root to a leaf node, we can find that the complete trapdoor matrix $M(Q)$ is first checked with the root and the query process terminates until all the trapdoors are removed from $M(Q)$ or a trapdoor matches a leaf node. Thus if we use the unmatched trapdoors $UMT$ and matched trapdoors $MT$ for leaf nodes to reproduce the query process from bottom to up in each path, we obtain the original trapdoor matrix $M(Q)$.

As shown in Figure 4, we show the $UMT(v)$ for each node and the $MT(v)$ for leaf nodes for the query example in Section IV-C. We use the path $P_4$ as an example, during query process from $v_0$ to $v_9$, the unmatched trapdoor $T_c$, $T_d$, $T_e$, $T_f$ are deleted from root $v_0$, and then $T_a$ is deleted in node $v_1$, and $T_b$ matches the leaf node $v_9$. The union of these sets: $UMT(v_0)$, $UMT(v_1)$ and $MT(v_6)$ can generate the original trapdoor matrix $M(Q)$. We have defined nodes $v_0$, $v_1$ and $v_6$ are key nodes in Section IV-C. Thus for each search path $P_i$ of the tree, the union of the $UMT(v_j)$ for each key node $v_j$ in path $P_i$ and $MT(v_l)$ for leaf nodes must obtain the complete trapdoor matrix $M(Q)$. Otherwise, we can determine that the query process is incorrect and the cloud server may delete the trapdoor deliberately in this path which may lead the incompleteness of the results.

We summarize the three steps of verifying completeness:

*Step - 1:* Verifying the correctness of Bloom filter for each key node. Before remapping the trapdoor to each Bloom filter segment, we should make sure the Bloom filter is correct and not be tampered in some bits. We use the HMAC value to verify it. Since HMAC is a one-way function, the cloud server

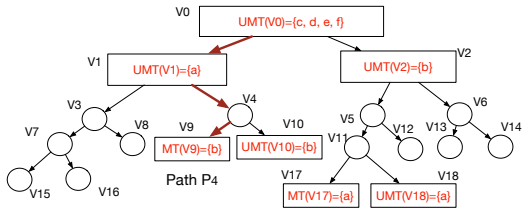


Fig. 4: Verification Tree for UMT(v) and MT(v)

could not tamper both of the Bloom filter and the HMAC value, and data user will easily find the mismatching for them. If the information of Bloom filter is correct, the data user can continue the verification process. Otherwise data user will reject the results.

*Step - 2:* Re-mapping the trapdoors for each key node $v$ to verify the correctness of $UMT(v)$ set and $MT(v)$ set. For each key node, using random number $rand_v$, the data user can hash the corresponding trapdoors on the Bloom filter segments to check whether the unmatched trapdoor set $UMT(v)$ and the matched-trapdoor set $MT(v)$ are correct. If $UMT(v)$ sets and $MT(v)$ sets are correct, the data user can continue the verification process. Otherwise data user will reject the results.

*Step - 3:* Verifying the query process in each search path. For each search path $P_i$, we union the $UMT(v_j)$ of each key node $v_j$ and the $MT(v_l)$ of leaf node $v_l$ (if $v_l$ is a leaf node). If the union set $U(P_i)$ contains all trapdoor's $id$ for the original trapdoor matrix, the query process in this path is correct. Only if every path is correct, we can determine our query result is complete. Note that, we can verify each path in parallel to reduce the verification delay.

### E. Optimizing SVETree

We discuss how to further optimize SVETree.

**Segments of Bloom Filter.** If the Bloom filter of each node is large, it is rather expensive to transmit the Bloom filters from the server to the client. To address this issue, we can segment the Bloom filter when constructing the index: $BL(v)$=$\{s_1$, $s_2$, ... $s_t\}$. The length of each segment $s_i$ is $Len$, which will significantly affect the verification performance. To add the verification information $Ver(v)$, the data owner should generate the HMAC value H.sg$_i$ for each segment $s_i$. Thus the cloud server can only transmit the relevant segments $s_i$ with H.sg$_i$ as proof $\pi$.

As introduced before, each trapdoor has $r$ elements, and these values will be mapped to $r'$ ($r' \leq r$) bits on $BL(v)$, therefore only the segments that contain these bits should be returned. Moreover, as long as there exists a hash value $h_j$ ($1 \leq r$) in trapdoor $T_i$ that misses the Bloom filter (the corresponding bit is 0), $T_i$ is an unmatched trapdoor and only the segment containing $h_i$ should be returned. Thus the cloud server just needs to return one segment for each unmatched trapdoor. It will reduce the communication cost significantly.

**Length of Bloom Filter.** To reduce the false positive, we set the length of the Bloom filter in each tree node $LB(v) = 10 * r * n_g$, where $n_g$ is the number of codes that should be hashed $r$ times before mapping. For low-level node, there are few cubes and the length of Bloom filter should be small. For upper-level node, there are many cubes and the length of Bloom filter should be large.

**Supporting Arbitrary $i$-dimensional Query.** We can extend our methods to support arbitrary $i$-dimensional range query ($i \leq d$). We provide several solutions and users can select one of them based on their preferences.
(1) Multiple SVETree Tree. Given a query $Q_i$, let $\pi_i$ denote the set of attributes in $Q_i$. For each distinct $\pi_i$ in the query workload, we build an SVETree tree. Then given a query $Q_i$, we use the corresponding SVETree tree to answer the query. This method consumes large space overhead in the cloud server. (2) Query Extension. Given a query $Q_i$ with $i$ attributes, we can use SVETree with $d$ attributes to answer the query. We add the other $(d-i)$ dimensions into the query range to form an extended query and use this query to search the results. Note the data user can know the CCS and thus can add other dimensions into the query. This method may incur larger communication overhead and longer search delay. (3) Hybrid Method. We select some queries to build multiple SVETree trees such that we can trade off between the storage overhead and search delay. Then given an online query, we can construct an SVETree tree such that (i) its nodes contain the attributes in the query and (ii) it has the smallest number of nodes among all the trees satisfying (i). Note that, in order to enable privacy-preserving i-dimensional queries, the incurred overhead is inevitable.

### F. Updating SVETree

Our SVETree can efficiently support the updates of data, including record insertion and deletion. (Modification can be supported by first deletion and then insertion.) To protect the privacy, update is allowed only for data owner. The cloud server just needs to change the modified part of SVETree.
*Insertion.* When inserting a new record $D_{in}$, the data owner first computes its code and insert the code into SVETree. It ensures the tree balanced after a node is inserted. If there is an empty node (due to deletion), the data owner can insert $D_{in}$ in this empty node. The cube set $C_{in}$ for $D_{in}$ should be added into Bloom filter along the path from the root to the corresponding leaf node. Besides, the verification (i.e., $HL(v)$, $HB(v)$) for these nodes should also be updated.
*Deletion.* Suppose the data owner deletes a record $D_{del}$. For the corresponding leaf node that indexes $D_{del}$, we clear the Bloom filter and update $HL(v)_{del} = hash(\varnothing)$. We modify the verification information along the path from it to the root. We only modify the Bloom filter in the non-leaf nodes when there is too many "1" similar to modification operation. To preserve index indistinguishable, we adjust the structure of the tree using modification operation between tree nodes.

The time complexity of these three operations is $O(\log(|D|))$. The tree update will not leak any information, which will be proved in Section V. Therefore, our SVETree is efficient and secure for dynamic database.

## V. Analysis

In our scheme, cloud servers are assumed to be malicious and may construct different attacks: (i) cloud servers can analyze and inspect data submitted by data owners; (ii) cloud servers can analyze and inspect query ranges submitted by data users; (iii) cloud servers do not follow the query procedures and return incorrect query results. We aim to ensure security

under such attacks. We analyze the security in our scheme under the first attacks in Section V-A and then analyze the verification correctness in Section V-B.

### A. Security Analysis

To formulate the security model during query processing, we utilize the framework introduced in [28], following the seminal ideal-real paradigm by [29]. We first define a leakage function $L$ to describes what a secure protocol is allowed to leak. And then we define two games, aka real and simulated games. The real game is essentially the execution in reality and the simulated one is a simulation by the simulator, which attempts to execute the multi-dimensional range query only knowing the leakage function and make it indistinguishable from the real one. Finally, we prove that an adversary cannot distinguish the output of the real game from that of the simulated game except with only negligible probability.

Specifically, the leakage functions $L_1$, $L_2$ we consider in this work are defined as follows. $L_1$ is associated with what is leaked from the index alone, whereas $L_2$ accounts for the leakage from the queries and verification operation:

• $L_1(D) = \langle n, Len(BL(v_i)), \nabla(\Gamma)\rangle$. $n$ is the size of dataset $D$, $Len(BL(v_i))$ is the bit length of bloom filter for node $v_i$. $\nabla(\Gamma)$ is the structure of tree index $\Gamma$. Since our index is a balanced binary tree, if the size of data set $n$ is determined, the bit length for each node $Len(BL(v_i))$ and index structure $\nabla(\Gamma)$ is fixed. In our scheme, the distribution of data set can be protected. And for each node, the dataset is split into two child nodes randomly and evenly. Given two data sets $D_0$ and $D_1$ in the same size ($|D_0| = |D_1|$), the two indexes for them cannot be distinguished by adversary according to Theorem 1.

• $L_2(D, Q)$=$\langle$ $\alpha(Q)$, $\sigma(Q)$, $\rho(Q)$, $(\alpha(c_i), \sigma(c_i), \rho(c_i)_{c_i \in C_Q})$ $\rangle$. In the process of range query, we transform the query $Q$ into a series cube codes $c_1, c_2, ...c_i$ (where $c_i \in C_Q$). $\alpha(\cdot)$ is the access pattern. $\alpha(W)$ denotes the data id returned by each query $Q$. And $\alpha(c_i)$ denotes the result id for each cube code $c_i$ for $Q$. $\sigma(\cdot)$ is the search pattern. $\sigma(Q)$ describes the difference between two different queries $Q_i$ and $Q_j$. $\sigma(c_i)$ is the difference for $c_i$ and $c_j$, where $c_i \in Q$ and $c_j \in Q$. $\rho(\cdot)$ is the path pattern. $\rho(Q)$ is the search path on index $\Gamma$ for query $Q$ and $\rho(c_i)$ is the search path for each cube code $c_i$.

Then we define $\text{Real}^A_\Pi(1^\lambda)$ and $\text{Sim}^A_{L,S}(1^\lambda)$ as follows.

**$\text{Real}^A_\Pi(1^\lambda)$:** the challenger runs $\text{Setup}(1^\lambda)$ to generate the key. $A$ outputs a data subset $D'$ and receives encrypted index $\Gamma \leftarrow BuildIndex\ (D',\ HK,\ sk)$ from the challenger. The adversary makes a polynomial number of queries $Q_1, Q_2, ...Q_t$. For each query $Q_i$ (where $i <= t$), the adversary receives from the challenger a trapdoor matrix $M(Q_i) \leftarrow GenTrapdoor(SK, HK, Q_i)$. Finally, $A$ returns a bit $b$ that is output by the experiment.

**$\text{Sim}^A_{L,S}(1^\lambda)$**: $A$ outputs a data subset $D'$. Given $L_1(D')$, $S$ generates and sends the index $\Gamma$ and the encrypted dataset $C'$ to $A$. The adversary makes a polynomial number of queries $Q_1, Q_2, ...\text{Q}_{t'}$. Then for each query $Q_i$ (where $i <= t'$), the simulator is given $L_2(D', Q_i)$=$\langle$ $\alpha(Q_i)$, $\sigma(Q_i)$, $(\alpha(c_j), \sigma(c_j)_{c_j \in C_Q i})\rangle$. The simulator returns an appropriate trapdoor matrix $M(Q_i)$. Finally, A returns a bit b that is output by the experiment.

**Definition 6.** *The verifiable multi-dimensional range query scheme $\Pi$ is L-secure against adaptive attacks if for all efficient adversary $A$ there exists an efficient simulator $S$ such that $\textbf{\textit{Adv}}^{adap}_{\Pi,A}(1^\lambda)$ = Pr[$\text{Real}^A_\Pi(1^\lambda)$=1] - Pr[$\text{Sim}^A_{L,S}(1^\lambda)$=1] is a negligible function.*

Now we can prove that the `SVETree` indexes of two datasets are computationally indistinguishable if the two datasets have the same size. Hence, based on the size information of the dataset, the simulator can simulate a random index which is indistinguishable from the real one. In our design, a balanced binary tree structure is used to organize the dataset. We divide the element randomly and evenly, and index the data records by using Bloom filter. If two datasets have the same size, the resulting trees will have the same size and the identical structure. Therefore, we can obtain that the tree indexes of `ServeDB` are indistinguishable if the underlying datasets have the same size. The detailed proofs can be found in [30].

**Theorem 1.** `ServeDB` *achieves indexes indistinguishability if two datasets have the same size.*

According to leakage functions defined above, if there exist a polynomial-time simulator $S$ such that for all probabilistic polynomial time (PPT) adversaries $A$, the output between the real game $\text{Real}^A_\Pi(1^\lambda)$ and a simulation game $\text{Adv}^{n-adap}_{\Pi,A}(1^\lambda)$ is computationally indistinguishable. Thus, the adversary $A$ cannot differentiate ($\Gamma$; $C$) from the real game when query. Thus, we can obtain the following theorem.

**Theorem 2.** `ServeDB` *is L-secure against adaptive attacks if F and G are pseudo-random functions.*

In order to enhance security with ensured search and query pattern protection, padding can be used to obfuscate search and access patterns [15], [16]. For example, a trapdoor can be mapped to a set of different trapdoors [15], where each is associated with a ciphertext copy of a plaintext cube. As a result, given a range of a dimension, different queries on this range can be issued with different trapdoor matrices, which match different sets of ciphertexts that point to the same plaintext data records. Thus, such approaches consume more memory for security. Meanwhile, re-encryption can be utilized to reset the leakage functions periodically such that it enables stronger search and access pattern protection.

### B. Verification Analysis

According to Definition 4 and 5, we prove the correctness and completeness of our verification scheme. Firstly, the data user can check whether the result $R_i$ falls in the range after decryption, however the cloud server may return a data record that falls in the range $Q$ but does not belong to the dataset $D$ (e.g., an old data record has been deleted from $D$). Therefore, we calculate the hash value for each data in $D$ and store it into the corresponding leaf node in tree index as the hash label. The correct hash label of root node can only be generated by the correct hash label from leaf nodes and key nodes. Therefore, the hash label can prevent being tampered of query results.

**Theorem 3.** *If each result $R_i$ fall in the range Q, and the hash value of key nodes can generate the hash label of root nodes correctly, the result set is correct.*

In ServeDB, a data user does not verify the completeness of query results. Instead, the data user reproduces the query process to verify each step of searching on tree nodes and detect incomplete results. Therefore, ServeDB ensures the completeness of range results.

**Theorem 4.** *For each search path, if the search operation is correct on each node, and it terminates when trapdoor matrix is empty or obtain the leaf node, the search result is complete.*

## VI. Experimental Study

### A. Experiment Setup

**Datasets.** We used the real Twitter dataset crawled from Twitter and the Foursquare dataset to evaluate our scheme. The Twitter dataset contained 5 million tweets and the Foursquare dataset contained 0.5 million items. Both datasets have real geo-locations, time, age, picture, and other non-numerical data. Moreover, we extend the dimensions of the datasets by randomly including the data of experience of paramedics from the datasets of California Health and Human Services Agency.

**Baselines.** To evaluate the query performance of secure multi-dimension range query frameworks, we compared our scheme with four baselines: (1) PBtree [10] only supported secure range query on single dimensional data. We used this method to achieve $d$-dimensional range query by first performing a query on each attribute and then calculating the intersection of the $d$ result sets on each attribute to find the final results. (2) log-SRC-i [11] devised numerous ranked searchable symmetric encryption (RSSE) schemes and offered a trade-off between security and efficiency for one dimension range query. We used a similar approach to enable multiple dimension range query by using log-SRC-i. (3) R-tree supported multi-dimensional range query, which was widely adopted by the existing schemes [8]. Note that, R-tree and its variance leaked the distribution of data and the original R-tree leaked the single-dimensional privacy. (4) Scan searched each dimension based on ciphertext respectively and intersected the result sets on each dimension. This method leaked the data information by observing the search patterns.

**Metrics.** We measured the delays of range query and verifying query results, and evaluated the accuracy and communication overhead of range query. Communication overhead was obtained by measuring the size of the delivered proof information. We focus on the Twitter results in this paper. More Foursquare results can be found in [30].

**Parameters.** We evaluated the following settings. (1) Number of records from 0.5 to 5 million, and 1 million by default. (1) Number of dimensions from 1 to 5, and 3 by default. (3) Bloom filter (BL) length from 20 to 140, and 20 by default.

**Setup.** We implemented our framework in thousands LOCs of C++ and conducted our experiments on a Windows 7 Professional machine with 32 GB memory and a 3.5-GHz Intel Core i7-4770k processor. We used HMAC-SHA1 as the pseudo-random function to implement the hash functions of Bloom filter. Note that, in our experiments, we recursively generated the hierarchy cubes. If all the leaf nodes in a level had less than 10 data records, we terminated to split the nodes.

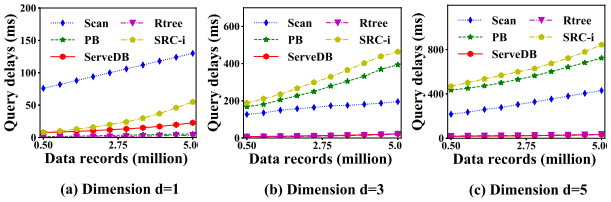

Fig. 5: Query delays with different numbers of data records.

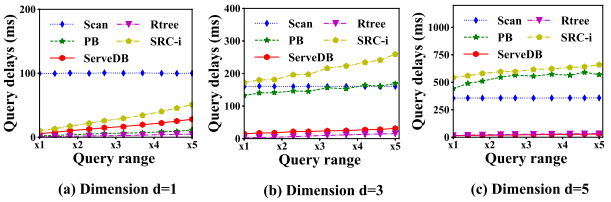

Fig. 6: Query delays with different query ranges (1M records).

### B. Query Delays

*1) Query Processing Delays in The Server:* We evaluate the query delays in the server with respect to different sizes of query ranges and different numbers of data records.

**Varying dataset size.** We evaluated the impact of the number of data records on query delays. We fixed the sizes of query range and changed the number of data records from 0.5 to 5 million. Figure 5 showed the query delays with respect to different numbers of data records. When we only queried one-dimension data, as shown in Figure 5(a), the queries performed by ServeDB were slightly slower than PBtree and R-tree, but significantly faster than Scan and log-SRC-i. The query delays of PBtree and log-SRC-i increased quickly with the increase of the number of data items when the search dimensions were larger than one (see Figure 5(b) and (c)). Although R-tree achieved relatively short query delays, the scheme has low security (see Section II-B). On average, ServeDB achieved at least 19 times faster than PB-tree and log-SRC-i. The reason is that both PB-tree and log-SRC-i required extra intersection operations. In particular, log-SRC-i needed extra interaction between cloud server and data user to compute the final results.

**Varying query ranges.** We evaluated the impact of query range on query delays. We fixed the data size with 1 million data records. Since different dimensions had different units of ranges, we first chose a fixed range in each dimension as the base range and then measured the average delays with the ranges increased by the base range. Figure 6 showed the query delays of one-dimension, three-dimension, and five-dimension with respect to different sizes of query ranges. For example, for $d = 5$ and $*4$ in the figure, we randomly selected on dimension, and increased the query range by 4 times. We repeated ten times to report the average results. The average delay for each query in ServeDB was 30 *ms*. The query delays of ServeDB was significantly reduced compared with the existing schemes. ServeDB is slightly slower than PBtree and R-tree when the dimension is one because it needs to generate extra proof information during searching. With the increase of searched data dimensions, query delays of ServeDB are relatively stable. On average, ServeDB achieved at least nine times faster than other secure query schemes in both three-dimension and five-dimension range query.

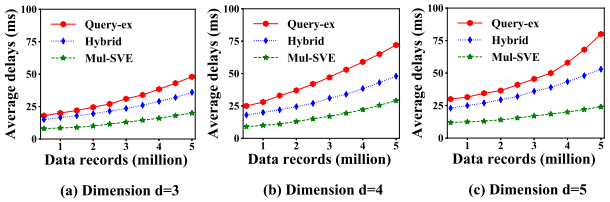


Fig. 7: Query delays with arbitrary $i$-dimensional queries.

**Varying query dimensions.** We evaluated the query delays for different arbitrary dimensional queries. As we discussed in Section IV-E, to implement arbitrary dimensional queries, `ServeDB` allows data owners to build multiple `SVETree` or data users to extend their query dimensions. We set the maximum query dimensions to be three, four, and five, respectively, and generated ten queries with arbitrary dimensions. In the multiple `SVETree` methods, 7, 17, and 31 `SVETree` were built, respectively. Figure 7(a) showed the query delays when the query dimension was set to three, and the multiple `SVETree` incurred the smallest query delays that on average was less than 12 *ms*. However, multiple `SVETree` incurred more storage to store the trees. While query delays with query extension were a bit longer and the average delays were around 35 *ms*, which generated more trapdoors for queries and introduced more communication overheads. The hybrid method enabled two dimensional range query with six `SVETree` and extended query dimensions for queries on the rest dimensions, which could trade off between delays and overheads. It incurred average 27 *ms* query delays. Figure 7(b) and (c) illustrated the delays with four and five dimensional range queries, respectively. The multiple `SVETree` method built 17 and 31 `SVETree`, respectively, and the hybrid only used two `SVETree`. The delay distribution was similar to that with 3-d query and the overall delays were larger.

**Comparison of query delays with two datasets.** We evaluated the query delays with two datasets with varied query ranges and varied numbers of dimensions, where we select 0.5 million data records. We observe that the results with the two datasets are similar. Figure 8 shows the query delays with the Foursquare dataset are slightly higher because the data in the dataset is denser. Figures 8 (a) and 8 (b) illustrate the query delays with 1-dimensional and 3-dimensional data. The delays increase with the increase of queried ranges. Figure 8 (c) shows query delays on various the dimensional data, where the dimension ranges from one to nine. On each dimension, we randomly choose ten ranges. We observe that the query delays slightly increase when the dimensions increase. The reason is that the query delays are mainly effected by the size of query ranges and the query results. Note that, since the query accuracy of these two datasets is almost the same, we do not present results due to the page limitations.

*2) Verification Delays:* We evaluated the verification delays with respect to the number of data records, the sizes of query range, and the sizes of Bloom filter segments. Firstly, we evaluated the verification delays for different numbers of query results. We choose 1 million data records from the original dataset randomly. We changed the scope of the search range, and the number of the query results ranged from 54 to 340. We recorded the time of verification on data users. Figure 9(a) showed the verification delays with respect to different number

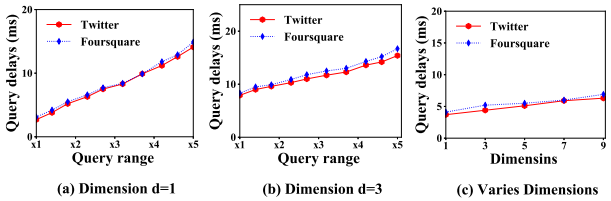


Fig. 8: Query delays with two datasets.

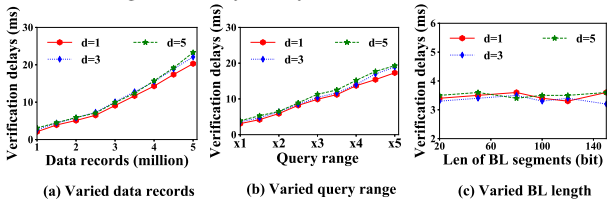


Fig. 9: Verification delays.

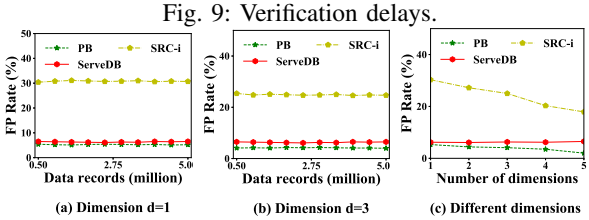


Fig. 10: Query accuracy of multi-dimensional query schemes.

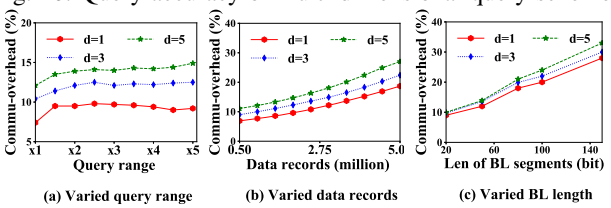


Fig. 11: Communication overheads incurred by verification.

of data items in the datasets. The verification delays were similar with different dimension queries, and increased when the size of datasets increased. `ServeDB` incurred around 12 *ms* verification delays. Figure 9(b) illustrated the verification delays with varied query ranges. Overall, the delays were not significantly impacted by the query ranges, and the delays with different query ranges were very similar. Since `ServeDB` verified the correctness of the Bloom filter segments, the size of the segments effected verification delays. Figure 9(c) depicted the verification delays with different sizes of Bloom filter segments. We observed that the length of segments did not significantly affect the verification delays. With different dimensions, the verification delays were very similar. More query dimensions with different sizes of Bloom filter segments introduced slightly longer verification delays. For simplicity, in the following experiments, we set the size of Bloom filter segments to 20 bits.

### C. Query Accuracy

In this experiment, we focused on analyzing incurred false positives. Note that all range query frameworks did not introduce *false negatives*. Figure 10 showed the false positive rate (FP) of range queries. Since R-tree and Scan did not incur false positives, we focus on comparing PBtree, log-SRC-i, and `ServeDB`. As shown in Figure 10(a), PBtree had around 5% FP that were incurred by Bloom filters when query dimension was one. `ServeDB` incurred slightly higher FP since cubes

based coding in ServeDB introduced more false positives. Log-SRC-i had the around 30% FP, which was highest among the schemes. Figure 10(b) illustrated the FPs when dimensions were set to three. FP of ServeDB was stable. Note that, FP incurred by PBtree and log-SRC-i reduced because intersection operations with different dimension query results eliminated the wrong queried results. Figure 10(c) showed the FP with the increase of dimensions. ServeDB had a constant FP because ServeDB did not differentiate data dimensions. Similarly, FP of PBtree and log-SRC-i decreased because of the intersection operations. Note that, since ServeDB ensured the completeness of query results, the incurred FP did not significantly impact the results returned to the data users since the wrong results are smaller which can be easily verified and removed by the users. We could further reduce the FP of ServeDB by optimizing the number of hash functions adopted in Bloom filters.

### D. Communication Overhead

In the experiments, we evaluated the communication overheads incurred by query verification. Figure 11(a) illustrated the communication overheads with varied sizes of datasets. The size of datasets ranged from 0.5 million to 5 million. Verification incurred around extra 100 *kb* compared with the returned query results, which introduced about 15% communication costs. The difference among different query dimensions was not significant because the verification information only depended on the query results and the height of the SVETree. Figure 11(b) showed the communication overheads with varied query range sizes when the size of dataset was 1 million. With different query dimensions, the communication overheads were similar. The communication overheads with one, three, and five dimensions were around 9%, 11%, and 14%, which was acceptable. With the increase of dataset sizes, the communication overheads because there would be more results. Figure 11(c) illustrated the communication cost by varying the size of Bloom filter segments with different queried ranges. The communication overhead slightly increased as the Bloom filter took more space but the overhead did not significantly increased with the increase of Bloom filter segment length. The average overhead is around 15%, which is acceptable for real deployment.

## VII. Conclusion

In this paper, we proposed ServeDB that supported secure, verifiable, efficient multi-dimensional range query on outsourced database. In particular, we developed SVETree in ServeDB that organized multi-dimensions data, which enabled security, verifiability, and efficiency of range query. We proved that ServeDB achieved the security properties against various attacks. We evaluated ServeDB on real datasets and experimental results show that ServeDB achieved high performance while providing security and verifiability.

## Acknowledgment

Qi Li was supported by NSFC (61572278 and U1736209), Guoliang Li was supported by 973 Program (2015CB358700), NSFC (61632016, 61521002, 61472198, and 61661166012), BHJ14L010, Huawei, and TAL, Xingliang Yuan was supported by the Oceania Cyber Security Centre POC scheme, and Cong Wang was supported by RGC of Hong Kong (CityU 11276816, CityU 11212717, and CityU C1008-16G) and NSFC (61572412). Qi Li and Guoliang Li are the corresponding author of this paper.

## References

[1] C. Y. D. Kim S, Lee H, "Privacy-preserving data cube for electronic medical records: An experimental evaluation." *International Journal of Medical Informatics.*, vol. 97, pp. 33–42, 2017.
[2] R. Agrawal, J. Kiernan, R. Srikant, and Y. Xu, "Order preserving encryption for numeric data," in *SIGMOD*, 2004, pp. 563–574.
[3] H. G. Do and W. K. Ng, "Multidimensional range query on outsourced database with strong privacy guarantee," in *PST*, 2016, pp. 555–560.
[4] V. Pappas, F. Krell, B. Vo, V. Kolesnikov, T. Malkin, S. G. Choi, W. George, A. Keromytis, and S. Bellovin, "Blind Seer: A scalable private DBMS," in *IEEE Sypm. SP*, 2014, pp. 359–374.
[5] E. Stefanov, C. Papamanthou, and E. Shi, "Practical dynamic searchable encryption with small leakage." in *NDSS*, vol. 71, 2014, pp. 72–75.
[6] C. Mavroforakis, N. Chenette, A. O'Neill, G. Kollios, and R. Canetti, "Modular order-preserving encryption, revisited," in *SIGMOD*, 2015, pp. 763–777.
[7] R. A. Popa, C. Redfield, N. Zeldovich, and H. Balakrishnan, "Cryptdb: protecting confidentiality with encrypted query processing," in *SOSP*, 2011, pp. 85–100.
[8] P. Wang and C. V. Ravishankar, "Secure and efficient range queries on outsourced databases using rp-trees," in *ICDE*, 2013, pp. 314–325.
[9] B. Yao, F. Li, and X. Xiao, "Secure nearest neighbor revisited," in *ICDE*, 2013, pp. 733–744.
[10] R. Li, A. X. Liu, A. L. Wang, and B. Bruhadeshwar, "Fast and scalable range query processing with strong privacy protection for cloud computing," *IEEE/ACM ToN*, vol. 24, no. 4, pp. 2305–2318, 2016.
[11] I. Demertzis, S. Papadopoulos, O. Papapetrou, A. Deligiannakis, and M. Garofalakis, "Practical private range search revisited," in *SIGMOD*, 2016, pp. 185–198.
[12] D. Cash, S. Jarecki, C. Jutla, H. Krawczyk, M.-C. Roşu, and M. Steiner, "Highly-scalable searchable symmetric encryption with support for boolean queries," in *CRYPTO*, 2013, pp. 353–373.
[13] P. Karras, A. Nikitin, M. Saad, R. Bhatt, D. Antyukhov, and S. Idreos, "Adaptive indexing over encrypted numeric data," in *SIGMOD*, 2016, pp. 171–183.
[14] E.-J. Goh, "Secure indexes," Cryptology ePrint Archive, 2003.
[15] J. Sherry, C. Lan, R. A. Popa, and S. Ratnasamy, "BlindBox: Deep packet inspection over encrypted traffic," in *SIGCOMM*, 2015, pp. 213–226.
[16] A. Papadimitriou, R. Bhagwan, N. Chandran, R. Ramjee, A. Haeberlen, H. Singh, A. Modi, and S. Badrinarayanan, "Big data analytics over encrypted datasets with seabed." in *OSDI*, 2016, pp. 587–602.
[17] E. Damiani, S. Vimercati, S. Jajodia, S. Paraboschi, and P. Samarati, "Balancing confidentiality and efficiency in untrusted relational dbmss," in *CCS*, 2003, pp. 93–102.
[18] Y. Lu, "Privacy-preserving logarithmic-time search on encrypted data in cloud." in *NDSS*, 2012.
[19] J. Chi, C. Hong, M. Zhang, and Z. Zhang, "Fast multi-dimensional range queries on encrypted cloud databases," in *DASFAA*, 2017, pp. 559–575.
[20] C. Priebe, K. Vaswani, and M. Costa, "EnclaveDB: A secure database using SGX," in *IEEE Sypm. SP*, 2018, pp. 405–419.
[21] D. Cash, J. Jaeger, S. Jarecki, C. S. Jutla, H. Krawczyk, M.-C. Rosu, and M. Steiner, "Dynamic searchable encryption in very-large databases: Data structures and implementation." in *NDSS*, 2014, pp. 23–26.
[22] S. Kamara, C. Papamanthou, and T. Roeder, "Dynamic searchable symmetric encryption," in *CCS*, 2012, pp. 965–976.
[23] M. Szydlo, "Merkle tree traversal in log space and time," in *Eurocrypt*, 2004, pp. 541–554.
[24] C. Papamanthou, R. Tamassia, and N. Triandopoulos, "Authenticated hash tables," in *CCS*, 2008, pp. 437–448.
[25] Y. Zhang, J. Katz, and C. Papamanthou, "Integridb: Verifiable sql for outsourced databases," in *CCS*, 2015, pp. 1480–1491.
[26] D. Papadopoulos, S. Papadopoulos, and N. Triandopoulos, "Taking authenticated range queries to arbitrary dimensions," in *CCS*, 2014, pp. 819–830.
[27] H. H. Pang, A. Jain, K. Ramamritham, and K. L. Tan, "Verifying completeness of relational query results in data publishing," in *SIGMOD*, 2005, pp. 407–418.
[28] R. Curtmola, J. Garay, S. Kamara, and R. Ostrovsky, "Searchable symmetric encryption: improved definitions and efficient constructions," *Journal of Computer Security*, vol. 19, no. 5, pp. 895–934, 2011.
[29] O. Goldreich, *Foundations of cryptography: volume 2, basic applications*. Cambridge university press, 2009.
[30] S. Wu, Q. Li, G. Li, D. Yuan, X. Yuan, and C. Wang, "Servedb: Secure, verifiable, and efficient range queries on outsourced database," http://ics.netlab.edu.cn/~qli/pdf/servedb-tr.pdf.