ECSTORE: A Secure and Compressed Encrypted Database with Logarithmic-time Indexing

Anonymous submission #xxx

Abstract

Encrypted Databases (EDBs) have gained considerable attention in securing sensitive data outsourced to the public cloud. EDBs store encrypted data and allow for searches over it without the need for decryption. However, prior cryptographic index construction in EDBs often lead to order-of-magnitude slowdown even with optimal (logarithmic) search complexity. This is primarily due to the substantial storage overhead incurred by data encryption as large indexes built on encrypted data are infeasible to compress into small-sized, yet searchable data structures. As a result, uncompressed large indexes rapidly saturate main memory and necessitate frequent accesses to slower persistent storage.

We introduce ECSTORE, the first EDB that achieves the benefits of both data encryption and compression. At the core of ECSTORE lies the *onewayness* observation on primary keys for encrypted search, which enables using secure yet lightweight one-way functions to generate compressible identifiers for the keys, then orchestrating a Merkle-tree-like index termed ECTREE using these identifiers. ECTREE supports a full set of index operations with logarithmic search complexity. A subtle case is that ECTREE can induce false positives under dynamic workloads (e.g., updates), thus we propose a detection algorithm and an index adjustment strategy to effectively identify false postives and eliminate redundant index traversals. Meanwhile, ECSTORE ensures query integrity against compromised servers using a query authentication protocol based on ECTREE. Extensive evaluations show that ECSTORE can achieve a compression ratio close to the optimal and increase the throughput by up to two orders of magnitude compared to notable EDBs.

1 Introduction

The surge in popularity of cloud computing has led to the widespread adoption of cloud storage systems [21, 47]. However, these systems have raised grave security concerns due to their processing of user data in plaintext [4, 36, 52, 53]. Recent incidents, such as the data breach exposing the personal information of 2.2 million Pakistani citizens [26], and allegations against TikTok for compromising the data of over 150 million US citizens stored in the cloud [3], highlight the severe security risks associated with such data storage systems.

Recently, *Encrypted Databases* (EDBs) emerge as a dependable solution for bolstering data security of cloud storage [19, 35]. By encrypting the data stored on servers and keeping the decryption keys at the client [12], EDBs effectively preserve data confidentiality against privileged administrators and cloud-side attackers. To process client queries on encrypted data, notable EDBs such as CryptDB [41] and HElibDB [17] employ cryptographic tools like property-preserving encryption [5, 12], to build search indexes on encrypted primary keys. Typical cryptographic indexes [41, 55], including CryptDB's, can achieve logarithmic complexity equivalent to that of unencrypted databases.

Unfortunately, despite achieving logarithmic-time indexing, these notable EDBs [17, 41] often lead to order-of-magnitude performance degradation under dynamic workloads. This deficiency arises from the lack of compression in these systems. Compression allows servers to fit more data in main memory thus decreases the number of data accesses to slower persistent storage, yielding significant performance gains by at least an order of magnitude [37, 55]. The necessity for compressing EDBs becomes particularly urgent as encryption introduces a substantial storage overhead compared to the unencrypted data (e.g., at least 5x larger for [55]). More importantly, under dynamic workloads where more and more elements are inserted, the performance of existing EDBs severely degrades as the large cryptographic index rapidly saturates main memory (as evaluated in §5).

Therefore, an ideal EDB should integrate both encryption and compression to handle dynamic workloads efficiently, and support a full set of index operations for diverse queries.

Fundamentally, there exists a tension between encryption and compression in EDBs. First, if the index is compressed and then encrypted at rest (in persistent storage), the cloud server must possess the decryption key to decrypt and then decompress the index upon receiving queries from clients, which exposes primary keys to compromised cloud servers. Second, if index is encrypted with keys inaccessible to the server and then compressed, the compression mechanisms in EDBs [41, 55] become ineffective because encryption brings pseudorandom properties to the index, and pseudorandom index can only be compressed into non-functional blobs [45].

To resolve this tension, we draw inspiration from an empirical trend of data encryption in databases. Specifically, the primary keys utilized for indexing (e.g., random identifiers, counters, timestamps) are often deemed less sensitive than the corresponding values. Furthermore, the indexes, built on encrypted primary keys, serve exclusively for lookups and are never decrypted for processing. This *onewayness* nature, wherein decryption is unnecessary, suggests that it suffices to use lightweight One-way Functions (OWFs) [7,31] to securely convert primary keys into *identifiers* for building indexes. In contrast to prior works [17,41] that rely on incompressible two-way encryption algorithms (owing to pseudorandomness), the one-way transformed identifiers adopt a boolean form, thus rendering the identifiers compressible.

Based on the above observations, we propose an **E**ncrypted and **C**ompressed Data **Store** (ECSTORE), the first EDB that achieves the benefits of both encryption and compression. At the core of ECSTORE lies a compressible tree-based index named ECTREE that provides efficient support for a full set of index operations, namely lookup query, range query, insert, update, delete, and bulkload. ECTREE introduces the concept of *membership-based* lookups by carefully orchestrating a tree structure storing data membership information and conducting searches via membership tests.

As exemplified in Figure 1, an ECTREE is initialized by organizing the identifiers (i.e., keys transformed by OWFs, such as SHA-256 [27]) at the bottom layer. The membership information of these identifiers is then recursively propagated to the root through *union* computations. We term this approach *membership propagation*, which ensures that each inner node *conservatively* encompasses the membership information of all its children for search purposes.

Membership propagation offers two crucial benefits. First, it enables logarithmic encrypted search by testing the membership of query parameters at inner nodes (i.e., by determining $\in$ or $\notin$) without revealing plaintexts. Second, the membership information is stored in a boolean form (i.e., bits) due to the use of OWFs, and we observe that only a portion of bits convey useful information for searches. Based on the observation, we compress large ECTREE nodes by eliminating redundant bits, converting remaining bits into sorted arrays, and leveraging linear approximation with machine learning models (i.e., learned indexes [30,33]) to precisely compress bit arrays within constant space occupancy, as only a few slopes and intercepts in the models need to be stored.

A key performance challenge arises from the issue of *membership hallucination*. Specifically, as more elements are inserted, the membership information of these elements is accumulated. As the accumulated information is propagated to the root, ECTREE may exhibit false-positives (FPs). Consequently, an ECTREE node may pass a membership test even the node is not on the traversal path of the search target, causing the hallucination of multiple correct search paths. To mitigate this issue, we introduce the FP-aware detection algorithm (FADA) that identifies FP-induced hallucination and eliminates redundant traversal paths. Additionally, we propose an index adjustment strategy that splits an ECTREE when necessary to keep an ECTREE's FP rate bounded.

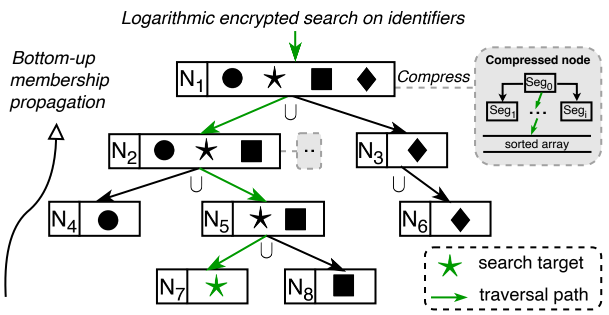


Figure 1: ECTREE propagates the membership information of identifiers (e.g., $\star$) from the bottom to the top, compresses large inner nodes that are in the boolean form (represented as sorted arrays) with learned indexes, and searches via membership tests.

Another security challenge is that a compromised server may induce integrity breaches by manipulating the database's behavior, which involves modifying, dropping, or deleting query results [51,58]. Fortunately, ECTREE's unique membership propagation construction leads to a Merkle-tree-like data structure that can facilitate query authentication by verifying if the requested data is included in the query results. Therefore, through a careful design of ECTREE, we effectively provide both data confidentiality and query integrity within an EDB in a unified manner.

We implemented ECSTORE based on CryptDB [41] and integrated with both MySQL [37] and WiredTiger [16]. ECSTORE is designed as a layer above them to make it adaptable to different data stores and leverage their performance and fault-tolerance capabilities. Our extensive evaluation shows that ECSTORE delivered significant compression while keeping the index encrypted and functional. For instance, ECSTORE supported a full set of index operations on the TPC-C dataset [32], and achieved a compression ratio of 4.7 that is close to the maximum ratio of 5.6, which can be obtained by compressing an entire encrypted index into a single non-functional blob. ECSTORE also achieved 8.9x $\sim$ 78x higher throughput compared to the encrypted baselines (CryptDB [41] and HElibDB [17]) and vanilla baselines (MySQL and WiredTiger with no encryption and compression) by fitting more index components into main memory.

In sum, we make the following three main contributions:

- We introduce ECTREE, a new membership-based index that efficiently supports a full set of index operations over encrypted data. ECTREE achieves lossless compression without compromising functionality and effectively authenticates query results to detect integrity breaches.
- We devise a false-positive-aware detection algorithm and a lightweight index adjustment strategy to facilitate logarithmic-complexity index operations over encrypted data.
- We provide a prototype implementation and conduct an extensive evaluation that demonstrates the advantages and efficacy of ECSTORE. The source code is released on `github.com/osdipxxx/ecstore`.

System	Data Encryption	Query Integrity	Index Compression	General Support	Lookup Complexity
HElibDB [17]	✓	✓	$n/a.^{\star}$	✗	$O(N)$
Dory [19]	✓	✓	✗	✗	$O(N)$
HybrTC [50]	✓	✓	$n/a.^{\star}$	✗	$O(N)$
CryptDB [41]	✓	✗	✗	✓	$O(\log N)$
Z-IDX [24]	✓	✗	✗	✗	$O(\log N)$
Blindseer [39]	✓	✗	✗	✓	$O(\log N)$
ECSTORE	✓	✓	✓	✓	$O(\log N)$

*: $n/a.$ means the system does not require an index for encrypted search.

Table 1: Characterization of representative EDBs.

2 Background and Preliminaries

2.1 Server-side Encrypted Search

Organizations are increasingly transferring sensitive data to cloud databases and asserting a need for data confidentiality during data processing. To meet this demand, *Encrypted Databases* (EDBs) [1, 2, 17, 41] typically employ fully homomorphic encryption algorithms [44] to encrypt value fields in data stores, allowing for arbitrary computations on encrypted values without decrypting them in an untrusted cloud. However, despite the rich computational capabilities it offers, homomorphic encryption lacks support for encrypted search.

To enable server-side encrypted search, much prior works have explored the use of searchable encryption algorithms to encapsulate primary keys into searchable ciphertexts, as shown in Table 1. CryptDB [41] uses deterministic encryption [12] and order-preserving encryption [5] to generate multiple searchable ciphertexts for a given key, then builds multiple indexes to support various query types. While CryptDB is time-efficient with logarithmic complexity, it lacks support for detecting integrity breaches against compromised servers, which is a common threat in public cloud [18]. HElibDB [17] relies on Fermat's Little Theorem [46] to directly assess the equivalence of homomorphically encrypted data without the need for indexing, but it necessitates linearly scanning the entire database. Dory [19] offers linear search functionalities specifically tailored for encrypted file stores. However, its approach to constructing file indexes using one-way functions [7, 31] is not adaptable to other database types.

ECSTORE aims to be a generalized system that can efficiently support a full set of index operations while providing both data confidentiality and query integrity guarantees against a compromised server. Nevertheless, prior works cannot fulfill all these goals in an efficient manner.

2.2 Encryption-Compression Co-designs

While some prominent EDBs (e.g., CryptDB) have achieved logarithmic search complexity, empirical evaluations reveal that the actual performance often deviates from theoretical expectations. This discrepancy primarily results from the linear expansion of ciphertext size by data encryption, consequently increasing the number of data accesses to persistent storage. Moreover, this effect becomes particularly pronounced when faced with dynamic workloads involving frequent insertions as the index rapidly saturates main memory. Compression is widely used in databases [22, 28, 34] and has potential to mitigate the performance penalty brought by encryption. We discuss two strawman designs that combines the power of encryption and compression and show their limitations.

The first approach is to encrypt over compressed data by utilizing compression techniques such as run-length encoding [25] and distributed source-coding [28]. These methods enable query execution using indexes built on encrypted and compressed data. However, their effectiveness in achieving a high compression ratio is limited to specific columns, resulting in an overall low compression ratio. For instance, reported results indicate that utilizing such techniques only yielded a compression ratio of 1.6 on the Conviva dataset [55].

An alternative approach is to encrypt the data and subsequently apply compression. Due to the pseudorandomness of encrypted data, this approach will generate non-functional blobs that are not searchable. To address it, clients are required to use a dictionary table [13] (i.e., an additional index) that maps uncompressed encrypted values to compressed codes. Concretely, clients search specific encrypted values in the table to find associated compressed code and submit queries parameterized with the code. The server conducts lookups by linearly scanning the database and comparing every code. Finally, clients decompress the retrieved code and decrypt it for results. However, the use of dictionary tables is inefficient because clients must do extra reads for compression and decompression that would reduce the lookup throughput.

ECSTORE is designed to harness the advantages of both encryption and compression by achieving a high index compression ratio while incurring minimal performance overhead. It is also independent of workloads and data types.

2.3 Preliminaries

Tree-based Index. In this paper, we aim at devising an encrypted and compressed index structure capable of supporting all operations in traditional tree-based indexes. Given x and y as keys and v as value, the index S supports the operations:

(1) *member(x)* = TRUE if $x \in S$, FALSE otherwise.
(2) *lookup(x)* returns the value associated with key $x \in S$.
(3) *range(x, y)* returns all the elements where $x \leq$ key $\leq y$.
(4) *insert(x, v)* puts an element with key x and value y to S.
(5) *delete(x)* deletes the element with key x from S.
(6) *update(x, nv)* runs *delete(x)* followed by *insert(x, nv)*.
(7) *bulkload(x[N])* is used for initializing or rebuilding index with N elements.

We assume that the keys are unique, although supporting duplicate keys is feasible by utilizing an overflow list [48]. We also assume that the index is based on a single column. In cases necessitating searches across multiple columns, ECSTORE manages disjunctive queries by independently searching each column and merging the results, but it does not support conjunctive queries currently. Implementing support for conjunctive queries would entail ECSTORE's integration with multidimensional indexes like k-d tree [57], which is orthogonal to the scope of this work.

Bloom Filter. Our membership-based index structure is based on the Bloom filter [9], a compact data structure designed for membership testing. It operates by using a single set to store the hash values of all elements. Specifically, a Bloom filter B is represented by an m-bit array and employs k independent and uniformly distributed OWFs, i.e., cryptographic hash functions $\{H_k\}$. Each hash function maps an element to a position in the array. To insert an element x into the filter, we compute its hash using $\{H_k\}$ and set all the bits at the corresponding indices in B to 1. For testing whether an element y is a member of the set, we again hash y using $\{H_k\}$. If any of the bits at the corresponding indices in B are 0, then y is definitely not in the set (i.e., zero false negatives). Conversely, if all the bits at the corresponding indices in B are 1, y is considered to have a high probability of being in the set, albeit with a small configurable false positive rate [9].

It is important to note that while prior EDBs have utilized Bloom filters to build indexes, they target specific query scenarios with high complexity. For instance, Dory [19] supports only encrypted keyword search on file stores with linear complexity. While Z-IDX [24] and Blindseer [39] achieve logarithmic complexity, they overlook the threat of integrity breaches in an outsourced database. ECSTORE differentiates itself by aiming to facilitate general queries and provide both data confidentiality and query integrity guarantees.

Learned Index. Given a key, learned index [11] maps it to the position in a sorted array of keys, which thus is considered as a trained model. Learned index builds a hierarchy of models to locate a key. During a lookup, the higher-level model predicts the model at the next level, and the leaf-level model computes the final prediction for the key's position in the sorted array. Finally, the learned index applies binary search to rectify inaccurate predictions based on a given error bound ε.

PGM [22] is a space-efficient and updatable learned index. It uses linear models (or *segments*) and divides keys into different segments to approximate the positions of all keys in the sorted array within ε distance. The space efficiency of PGM stems from the fact that linear segments require only constant space (for slopes and intercepts) and constant query time, independent of the total number of keys to be indexed. Each segment of PGM specifies the first key covered by next-level segment, enabling recursive index construction on the sorted keys of segments at lower levels. To locate a key, PGM predicts positions at each level and corrects them immediately. This process continues until convergence to a leaf-level segment. For insertions, similar to LSM-tree [38], PGM separates keys into subsets. Each insertion involves finding a series of non-empty sets, merging them into a large subset, then bulkloading a new index on the large subset.

3 Overview

3.1 System Setup

Entities. Same as existing EDBs [37, 55], ECSTORE follows a cloud-hosting model consisting two roles: the hosting server (which can be distributed) responsible for providing query services, and the client (e.g., banks and hospitals) issuing queries to the server for using the services. The client can have multiple endpoint machines sharing the same encryption keys, which are inaccessible to the server. In line with [20], we assume the cloud provider is motivated to ensure server availability, by either bringing the failed server back online or replacing it promptly [20, 29].

Data model. We employ the key-value data model to illustrate our design for its simplicity. The server-side EDB contains two columns storing the primary keys and their respective values. In ECSTORE, a key is stored as an *irreversible identifier*, achieved by applying one-way functions (OWFs) such as SHA-256 [27] to transform the original plaintext key. Irreversibility means that these identifiers cannot be reversed back to plaintext keys even if the OWFs are known. Noted that, while OWFs are deterministic schemes, they are sufficiently as secure as randomized schemes in our setting because the primary keys in an EDB are unique. This determinism allows ECSTORE to maintain a sorted index based on these identifiers, enabling the clients to directly query on the EDB. Values are encrypted via fully homomorphic encryption [54] before being inserted to the EDB, to facilitate arbitrary computations without decryption.

In ECSTORE, all key-value pairs (we call a key-value pair and an element interchangeable) are *versioned*. This involves a client maintaining the latest version of elements in a local version table (depicted in Figure 2), and keeping it synchronized with other clients. In such a case, clients are assumed to have obtained the latest version of an element before submitting a new update query to that particular element.

3.2 Threat Model and Guarantees

Similar to prominent EDBs [15, 20, 35, 41], we adopt a strong threat model in which a malicious adversary $\mathcal{A}$ can corrupt servers and arbitrarily deviate from ECSTORE's protocol to disrupt query services. Specifically, $\mathcal{A}$ can either passively observe or actively manipulate query results and tamper with stored data in the server via modification, replay, or deletion [58]. Given the growing reliance on cloud-hosted databases and dependence on third-party database administrators, we believe this threat is increasingly important. However, in line with prior research [19, 41], we assume that $\mathcal{A}$ is incapable of reversing OWFs or compromising cryptographic encryption keys. This threat includes compromises of database software and access to the RAM of physical machines.

We also assume that the clients are trusted and authorized to access all server-side data (e.g., because the client is the data owner). In ECStore, we focus on protecting the data from a server-side adversary. Consequently, we assume that the adversary $\mathcal{A}$ does not have control over any client, thereby lacking the capability to execute any queries (e.g., insert, delete, update) through a client.

Out-of-scope Attacks. ECStore is not designed to hide side-channel information such as data access patterns or element sizes. The security of ECStore is *not* perfect as it could reveal side-channel information that corresponds to the type of computation that queries perform, such as equality comparisons and sorting operations. Achieving 'optimal' security requires ECStore to integrate with recent theoretical cryptography works that conceal all side-channel information [40, 49], which is prohibitively expensive. However, it is notable that unlike prior compression methods (e.g., run-length encoding [25]) that expose data frequency, ECStore's compression avoids introducing additional data frequency information, thus does not augment such vulnerabilities.

ECStore's Guarantees. The confidentiality guarantee is twofold. First, ECStore ensures that all primary keys are irreversible, in the sense that a computationally bounded adversary $\mathcal{A}$ cannot reverse the OWF-transformed primary keys to retrieve plaintext keys. Second, ECStore protects values with a strong and standard encryption scheme (fully homomorphic BGV [23]), and protects the names of columns and tables with AES-256 in CBC mode, which provides semantic security [54]. Meanwhile, ECStore ensures query integrity by precisely detecting any manipulations in query results.

3.3 ECStore's Workflow Overview

The architecture of ECStore is shown in Figure 2. The server hosts a tree-based index (ECTree) in the main memory and stores elements at the leaf level physically, following the common practice of B$^+$-tree [14]. The server also hosts a list of MAC tags, each corresponding to an element. The client interacts with the server through ECStore's query manager module, consisting of three components. The profiler transforms plaintext query parameters into cipher identifiers for server-side encrypted search and maintains cryptographic keys inaccessible to the server. The ECTree cache stores a partial index, including the ECTree root for authentication and the leaf nodes (primary keys) for write operations (i.e., *insert*(x,v), *delete*(x), *update*(x,nv) and *bulkload*$(x[N])$). The version table maintains the latest version of all elements for both query authentication and write operations. ECStore's integrity monitor module detects breaches in query results for read operations (i.e., *member*(x), *lookup*(x), and *range*(x,y)) by interacting with the local ECTree cache and version table.

For read operations, like *lookup(x)* in Figure 2, the client first transforms plaintext parameter x into a secure primitive called *subfilter* (described in §4.1). This primitive, denoted as t_x, acts as a cipher identifier for x to search and is irreversible. The client encrypts query via AES, signs it, and sends to

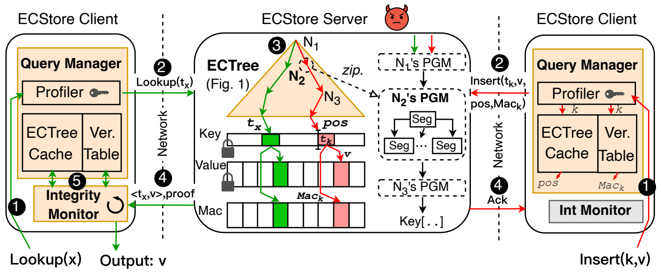


Figure 2: The architecture behind ECStore.

the server via a TLS-enabled link (❷). The server will drop the connection if malformed queries (e.g., those with invalid signatures) are received. The server traverses ECTree using t_x to locate the requested element (❸), returning both the element $<t_x, v>$ and a proof to the client (❹). The client fetches essential data from the ECTree cache and version table to verify the proof (❺, described in §4.4), and eventually outputs the decrypted value if verification succeeds.

For write operations, like *insert*(k,v) in Figure 2, the client first transforms the key k into identifier t_k, then finds k's predecessor position, *pos*, using ECTree cache, and generates the MAC tag, Mac_k, for the element. Subsequently, the client sends a query parameterized with t_k, v, pos and Mac_k to the server (❷). On the server side, the ECTree locates the insertion position for element $<t_k, v>$ by finding the successor position of *pos* then adds the element (❸). Finally, the server adds Mac_k to the Mac list and returns an Ack to the client (❹) to signify the completion of the write operation.

Overall, the highlight of ECStore lies in its ability to provide both data confidentiality and query integrity, while supporting a full set of index operations with high efficiency (with logarithmic search complexity and compression). This is primarily achieved by ECTree secure index, featuring a new insight of membership-based logarithmic search and overcoming the incompressibility of prior cryptographic methods.

4 Design

4.1 ECTree Construction

Opportunity: Bloom Filters. In the context of encrypted search, preserving the *onewayness* property of primary keys suffices as these keys are exclusively used for lookups, unlike values. Additionally, assuming the keys are unique (§2.3) and the server does not have control over any client to execute any queries such as insert, delete and update (§3.2), it is infeasible for adversaries to reverse one-way transformed primary keys by lauching brute-force or dictionary attacks [6, 13].

The conventional Bloom filter [9] relies on One-Way Functions (OWFs) like SHA-256 to uphold onewayness, as it stores only the cryptographic hashes of elements rather than the original plaintext keys (§2.3). Unfortunately, the Bloom filter is *monolithic* in nature since all elements are allocated to a single

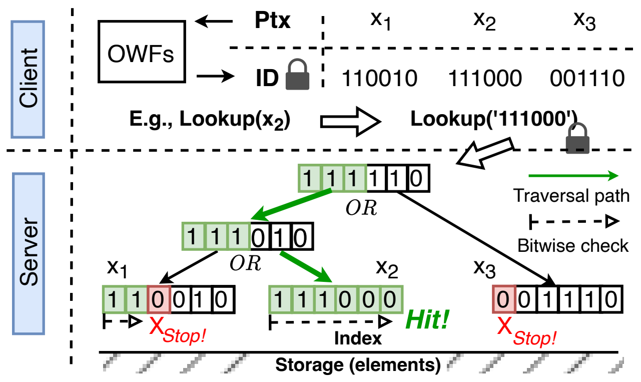


Figure 3: ECTREE's core concept of membership-based search.

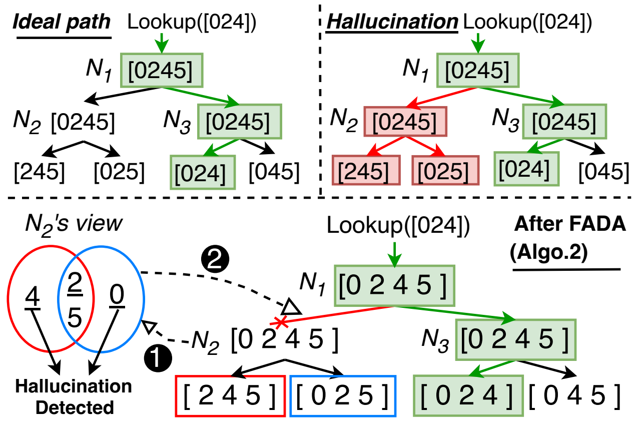


Figure 4: Membership hallucination might happen in ECTREE due to false positives, leading to redundant index traversals.

set (a bit array). As a result, it can only support membership tests, i.e., confirming the presence of an element (hash) within a set. However, it is not capable of serving as an index for precisely locating a given key.

Our approach: Subfilter-based ECTREE Index. We observe that the onewayness nature of the Bloom filter aligns perfectly with the encrypted search demand for primary keys, thus in our preliminary design, we decompose the monolithic Bloom filter into multiple *subfilters*, with each subfilter solely storing the hash (in bits) for a single element.

Definition 1 (***Subfilter***)**.** *A subfilter is the Bloom filter for an individual element, i.e., only one element is assigned to a set for membership testing.*

Figure 3 exemplifies the idea of constructing an ECTREE using subfilters. Specifically, each subfilter denotes a primary key (e.g., "111000" representing x_2 after transformation via OWFs) for indexing. These subfilters, sharing a consistent array size, are positioned at the leaf layer and arranged in an order aligned with plaintext keys. Each inner node in ECTREE is determined by performing entry-wise *OR* gate computations on all its child nodes' subfilters. Given subfilters with length m, for an inner node *inner* with p child nodes $\{cn_1, \dots, cn_p\}$, each bit in the inner node's subfilter is computed as

$$inner[i] \;=\; cn_1[i] \;OR\; ... \;OR\; cn_p[i]\;,\; \forall\, i \in [0,m] \qquad (1)$$

The design rationale behind ECTREE is that, if any entry in a child subfilter holds the "1" bit, Equation 1 ensures that the corresponding entry in its parent subfilter will also possess the "1" bit. This principle guarantees that if an element *exists* in a particular node, it must also exist in its parent node. We term this approach *membership propagation*, i.e., the membership information of primary keys (identifiers) is propagated upward to the root, and later facilitates top-down logarithmic traversal, while preserving irreversible confidentiality inherited from OWFs employed in generating subfilters.

4.2 ECTREE Operations

Next, we describe the procedures for index operations and the algorithms to dynamically adjust ECTREE's structure.

Lookup Queries. The search process in ECTREE begins at the root node and involves conducting membership tests by comparing subfilters to determine the traversal path. As depicted in Figure 3, to look up a key, the client transforms the query parameter into a subfilter to locate the corresponding value via ECTREE. On the server side, during the traversal of inner nodes, ECTREE determines the path (shown in green) with subfilter comparisons. These comparisons verify if every "1" bit in the parameter's subfilter exists in the current child node's subfilter. Successful verification leads ECTREE to progress to that child node until reaching the bottom layer. At the bottom leaf layer, a leaf node validates the match of *every* bit, as each subfilter exclusively contains one key. Our methodology is outlined in Algorithm 1.

Challenge: Membership Hallucination. One subtle case is that, unlike a B$^+$-tree that relies on numerical comparisons, ECTREE's subfilter comparisons may inadvertently generate false positives (FPs). These FPs can lead to redundant index traversals wherein multiple ECTREE nodes pass membership tests, despite there being only one correct traversal path. We call this phenomenon *membership hallucination*. In the worst-case scenario, if a lookup navigates through an exponential number of paths, the resulting complexity would degrade from logarithmic to linear.

To illustrate, Figure 4 shows a hallucination example. Assuming the utilization of 6-bit subfilters with three OWFs, for clarity, subscripts denote the positions of "1" bits within each subfilter (e.g., [245] signifies a subfilter "001011"). Suppose a client submits a lookup query parametered with [024]. Ideally, an ECTREE should follow the path: $N_1 \rightarrow N_3 \rightarrow [024]$ (in green). However, since both N_2 and N_3 pass the membership test for [024] (i.e., the 0th, 2nd and 4th bits are "1"), the ECTREE mistakenly searches the entire left branch, showing that hallucination leads to redundant lookups (in red).

We observe that such redundancy stems from the *undirected* membership propagation. Specifically, N_2 in Figure 4 inadvertently includes [024] by simply taking the union of its child nodes, resulting in a FP where not all "1" bits origi-

Algorithm 1: ECTREE Lookup

```
Input: $\tau$: the root node of ECTREE, $k$: the lookup key
Output: $isFound$: indicates whether $k$ is found, $v$: the value
Function Lookup($\tau$, $k$) do
    if $isLeaf(\tau)$ then
        ▽ Every bit should match (Def 1)
        if $k.subfilter == \tau.subfilter$ then
            return <$True$, $\tau.value$>
    if $isInner(\tau)$ then
        $child[] \leftarrow$ the child nodes of $\tau$
        for each $c$ in $child$ do
            ▽ Match '1' bits and identify membership hallucination
            if $\tau \in c$ && !IsMemHall($c$, $k$) then
                return Lookup($c$,$k$)
        return <$False$, $Null$>
```

Algorithm 2: FADA (Hallucination Detection)

```
Input: $N$: the node of ECTREE for check, $k$: the lookup key
Output: $isFound$: indicates whether $k$ is a hallucination of $N$
Function IsMemHall($N$, $k$) do
    $child[] \leftarrow$ $N$'s child nodes
    $pos[] \leftarrow$ positions of "1" bits in $k$'s subfilter
    isMem $\leftarrow$ $True$
    for each $c$ in $child$ do
        for each $p$ in $pos$ do
            if $c.subfilter[p] != $ "1" then
                $isMem \leftarrow False$
                $break$
        ▽ Not a hallucination (CCS is not $\phi$)
        if $isMem$ then
            return $False$                ▷ Algo 1 continues
        $isMem \leftarrow True$
    return $True$
```

nate from a single child node. Thus, we detect FPs by letting ECTREE's inner nodes store an additional *direction view*, indicating the source of every "1" bit. In Figure 4, the view of N_2 shows that the 4^{th} bit originates from its left child, the 2^{nd} bit is from both child nodes and the 0^{th} bit is from its right child. By analyzing this direction view, ECTREE will stop traversing N_2's branch even if N_2 contains the requested [024] because N_2's view reveals that the 0^{th} bit and 4^{th} bit originate from different sources, indicating that the requested key is not a member in N_2's branch.

Based on the above observation, we introduce a FP-aware Hallucination Detection Algorithm (FADA) to identify false postives and eliminate redundant traversal paths for ECTREE lookups. Specifically, the direction view of an ECTREE node is formed as a table T, where the rows of T represent subscripts of "1" bits in N's subfilter, and the columns denote identifiers of child nodes containing these specific "1" bits. FADA takes the query parameter's subfilter Sub as input and outputs whether Sub is a hallucination of N, by verifying the *Common Child Set* (CCS) for Sub in T.

Definition 2 (***Common Child Set (CCS)***)**.** *Given a requested subfilter* `Sub`*, the CCS of an* ECTREE *node* `N` *comprises a collection of* `N`*'s child nodes, wherein each node in the set encompasses all* "1" *bits present in* `Sub`*.*

To compute the CCS (parameterized with node N and target Sub), we simply enumerate the child nodes and checking corresponding bits. Specifically, if all the "1" bits in Sub are present in a child node's subfilter, the identifier of that node is added to the CCS. To determine the occurrence of hallucination, the size of the resulting CCS is checked. An empty CCS implies the absence of N's child nodes that encompass all "1" bits present in Sub, indicating that N is not part of the traversal path for the current data lookup; otherwise, N is on the traversal path. Algorithm 2 shows our methodology.

In summary, ECTREE supports lookup queries by first comparing subfilters, and then employing FADA to detect membership hallucination to ensure correct index traversal paths (line **10** in Algorithm 1). Consequently, as a tree-based index, the resulting complexity of ECTREE's encrypted search scales logarithmically with respect to the size of the EDB.

Index Inserts. Apart from lookup queries, the index should handle insert operations, while maintaining the strict order guarantee for the index (keeping subfilters in order at the leaf level like a B^+-tree). Since subfilters are OWF-transformed identifiers without orders, to achieve this requirement, we let the client maintain an ECTREE cache to locate positions for insertions on the server side.

Specifically, the client-side ECTREE cache retains a partial index, including the root node of ECTREE and the primary keys associated with leaf nodes. The cache may contain multiple root nodes because the sever might maintain multiple splitted ECTREEs due to *index adjustments* under dynamic workloads (discussed next). Each root node in the cache serves as an identifier to locate a specific index for insertions and is also used for query authentication (§4.4). To insert an element keyed with k, the client requests the local cache to retrieve a root node r (indicating the index where the insertion should occur) and the predecessor of k, followed by transforming the predecessor into subfilter S_{pre}. The server-side ECTREE (rooted at r) locates S_{pre} at the leaf nodes and inserts the element next to S_{pre}, thus keeping the leaf nodes monotonically increase, which is crucial for facilitating range queries.

After finishing the insert operation, we backtrack along the search traversal path in the opposite direction and update the subfilters and direction views associated with the nodes along this path. Additionally, we judge whether the ECTREE should trigger an adjustment to keep the FP rate bounded.

ECTREE's Adjustment Strategy. Recall that, FADA (Algorithm 2) effectively detects FPs in a static setting where only read operations occur. However, its effectiveness diminishes under dynamic workloads (e.g., inserts). We can think of an extreme case when all bits in top-level subfilters eventually become "1". This case arises when more and more newly inserted elements propagate their membership to the root (Equation 1), given that subfilters have a constant size. Consequently, this leads to increased FPs for data lookups. To maintain a bounded FP rate within ECTREE, we trigger the splitting of nodes when certain conditions are satisfied. We focus on two core issues: When to adjust and How to adjust?

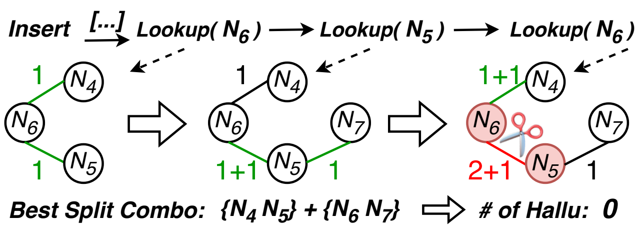


Figure 5: An adjustment example based on edge-weighted graph.

• ***When to adjust?*** We propose two main criteria to decide whether to trigger adjustment on an ECTREE node.

First, the number of inserted elements in an ECTREE rooted at r is at least α times larger than the elements in the last adjustment. This condition is expressed as $\frac{r.insert\_num}{r.total\_num} \geq \alpha$, where *r.insert_num* increments with each insertion and *r.total_num* is the number of elements by the time of the last adjustment. α is set to 2 by default, which is derived from the logarithm methods of PGM [22], i.e., always triggering the split process for an index when the inserted elements is twice as much as elements contained in the previously adjusted index. This criterion effectively reduces the frequency of adjustments.

Second, the ratio between the number of FADA-detected hallucinations on a node n and the insertions in an ECTREE rooted at node r exceeds a given threshold β, i.e., $\frac{n.hall\_num}{r.insert\_num - r.total\_num} \geq \beta$. Hallucinations negatively impact performance by requiring direction view checks for lookup operations. Therefore, an adjustment should be triggered if an excessive number of hallucinations are observed on a node. By default, we set the threshold β = 0.1 to achieve an appropriate adjust rate. To further choose proper parameters, we study the parameter sensitivity for α and β in §5.

• ***How to adjust?*** Upon detection of a hallucination by FADA, we catalog the nodes related to the hallucination as an edge-weighted graph. The second condition, triggered by an over-weighted edge (i.e., excessive hallucinations) and a certain number of insertions causing FPs, is exemplified in Figure 5.

Consider the leaf nodes N_4 to N_7 of the ECTREE in Figure 4. Following a series of insertions, three lookups are conducted, and FADA detects and records hallucinations for each lookup: *Lookup(N_6)*, or *Lookup([024])*, establishes two edges between N_4 and N_5 as these nodes propagate a hallucination of N_6 ([024]). Similarly, *Lookup(N_5)* increases the weight of the edge between N_5 and N_6 and adds an edge with N_7. Subsequent *Lookup(N_6)* further raises the weight of the edge between N_5 and N_6, triggering the second condition, thus N_5 and N_6 need to be split into two indexes.

To identify the best split combination, we employ Kruskal's algorithm [43] to compute the Minimum Spanning Trees of the graph, aiming to minimize the total number of hallucinations (weights) in each split index. Ultimately, we determine the combination of $\{N_4, N_5\}$ and $\{N_6, N_7\}$ and execute *bulk-load* to generate two new ECTREEs. This division is chosen because it results in the lowest total number of weights (hallucinations) in these two split indexes (i.e., 0).

More Operations. *Range queries* identify elements whose keys fall within a specified range by converting the range parameters into subfilters, locating the corresponding leaf nodes using lookup operations, and retrieving all elements positioned between these leaf nodes due to ECTREE's monotonic nature. *Delete queries* are supported by looking up the entry of the element and marking the type of that entry as NULL. Subsequent lookup queries for this deleted key lead to traversal to a NULL entry, indicating the non-existence of the key. *Update queries* come in two forms. One involves modifying the key itself, executed by combining a delete operation with an insert operation. The other type involves solely modifying the payload and is supported by searching for the key and overwriting the existing value.

4.3 ECTREE Compression TODO:(1 fig, 1 algo).

A core index challenge: size and cost. Despite ECTREE's efficient tree-based index structure, which facilitates logarithmic search complexity, achieving a low search complexity alone is insufficient for achieving high-performance encrypted search. The reasons are two fold. Firstly, as the total number of items increases, each ECTREE node experiences exponential growth in membership information storage due to the membership propagation design (§**??**). Consequently, the index size significantly expands, adversely affecting search efficiency. Secondly, even in a static setting, the false positive issue, attributed to the fundamental nature of Bloom filters, further amplifies the size of each minifilter node.

To illustrate, let's consider a database with 2^{20} static items and the user aims for a low false positive rate of 1×10^{-3}, same as the configuration in [50], each minifilter in ECTREE would need to be approximately 1.79MB in size according to the Bloom filter theory [9]. Consequently, the entire ECTREE would require a substantial amount of storage space, around 3.57TB, which is impractical. The significant increase in space occupancy compared to a plaintext database index results in a notable performance decline due to expensive I/O operations involving secondary storage.

Facing this challenge, our contention is that relying solely on achieving theoretical logarithmic search complexity is insufficient for achieving high-performance encrypted search, even though it represents a significant improvement over linear search methods [17, 19]. Instead, an encrypted search index should adopt a compact design to ensure that the overall query performance remains manageable and cost-effective.

The compact design. As depicted in Figure 6, during index construction, ECTREE preserves light index nodes and compacts giant index nodes that exceed a user-defined memory budget B using two steps below.

• **Step 1 (From bits to offsets).** Recall that in ECTREE, each index node (i.e., minifilter) is represented as a one-item Bloom filter (§**??**), which is a bit array consisting of only '`0`' and '`1`'

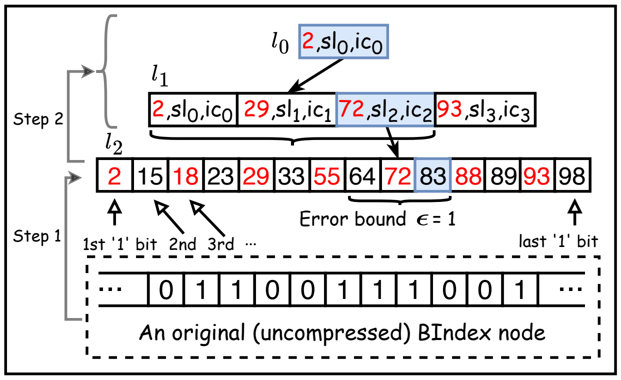


Figure 6: The construction and a running example that show our compact design for ECTREE. **Step 1** extracts an original ECTREE node into offset representation wherein the ith entry is the position of the ith '1' bit. In **Step 2**, we uses the Piece-wise Linear Approximation model [22] to compact and search the extracted offset array.

bits. However, only the '1' bits contain informative data since membership tests only involve checking if all '1' bits are present in a filter. Thus in **step 1**, we extract the positions of all '1' bits from the filter, creating a new offset array. This extraction process allows us to discard unnecessary bits and avoid wasting primary storage resources.

• **Step 2 (Linear Approximation using PGM [22]).** Although **step 1** reduces the size of giant ECTREE nodes, the size still increases exponentially with the total number of records n. Fortunately, we find that the linear compression technique known as Piece-wise Linear Approximation, particularly the PGM [22], is highly suitable for further compressing the extracted offset array in our specific setting.

Running example. Figure 6 provides a running example of searching a compressed ECTREE node using Algorithm **??**. Each accessed node in the search path is highlighted in blue. We assume the compressed ECTREE has an error bound $\varepsilon = 1$ and the key being searched is $k = 83$.

The search begins from the root segment $s' = l_0[0][0]$. The next position, $\lfloor fs'(k) \rfloor = \lfloor k * sl_0 + ic_0 \rfloor$, is computed to be 1 for the next level. The search then proceeds to locate k within the range $[1-\varepsilon, 1+\varepsilon]$ in l_1, considering the keys $[2, 29, 72]$. It is determined that the next segment for the search is $s'' = l_1[2]$ because $k > 72$. Finally, the position for the next (leaf) level, l_2, which represents the extracted offset array, is computed by evaluating $\lfloor fs''(k) \rfloor = 8$. Consequently, a binary search is performed to search for k within the range $[8-\varepsilon, 8+\varepsilon]$ in $l2$. Ultimately, it is found that k is located at position 9 since $A[9]$ = 83.

In the compacted ECTREE, Algorithm **??** functions as the internal search API of `minifilter.contain(k)` in Algorithm 1. Combined, they constitute the complete encrypted search algorithm (without metadata protection) of ECSTORE.

4.4 ECTREE-driven Query Authentication

So far, we have assumed that the server adheres to our protocol in an honest-but-curious manner. Next, we show how we leverage ECTREE to detect malicious manipulations, including modifications, replays, and dropping (deletions) of query results through the enforcement of two authentication rules.

• ***Rule 1: Freshness Validation.*** This rule serves to detect any query result modification or replay attempts by using MACs. Two key conditions define the validity of a MAC tag. First, a MAC tag should only be valid for the latest update to prevent replay attacks. Second, a the MAC tag should not apply to other elements. Thus, we compute the MAC over both the key (subfilter) and the version of the element obtained from the local version table. For each insert or update, the client includes an additional MAC tag for the element by appending a 256-bit tag to the tail of the subfilter.

• ***Rule 2: Completeness Validation.*** This rule aims to detect drops and deletions within query results by validating that the results align with the specified range and that the recomputed ECTREE root matches the ground-truth root stored in the local cache. Specifically, using a range query as an example, the server first searches for elements within the specified range (r_l and r_r). It then generates a proof that includes a node's subfilter to the immediate left of the lower-bound (R_l) and another subfilter to the immediate right of the upper-bound (R_r). Additionally, the server retrieves the subfilters of all left sibling nodes in the left traversal path and the subfilters of all right sibling nodes in the right traversal path.

Upon receiving the proof, the client validates it via two aspects. First, it confirms that the range of R_l is smaller than the left boundary, i.e., $R_l.range < r_l.range$, and it also verifies that $R_r.range > r_r.range$. Second, thanks to the Merkle-tree-like construction of ECTREE, the client recomputes the ECTREE root in a Merkle-tree fashion using the requested query results and their siblings. Then, it compares this recomputed root with the ground-truth root stored in the ECTREE cache, which was pre-stored by the client and synchronized for each update and index adjustment. Any mismatch between the two subfilters signifies drops or deletions within the query results.

In sum, **Rule 1** detects modifications or replays and **Rule 2** detects drops or deletions in query results. Together, they form a cohesive query authentication protocol, achieved by the unified utilization of ECTREE for encrypted search.

5 Evaluation TODO:add comp ratio and para sens study.

Testbed. All experiments were done on our cluster machines, each equipped with a 2.60GHz Intel E5-2690 V3 CPU, 64GB memory, 40Gbps NIC, and 24 cores. Each node, including clients, databases, and the proxy, was executed in a docker container. The average ping latency between the nodes was set at 0.17ms with the aid of Linux traffic control [8].

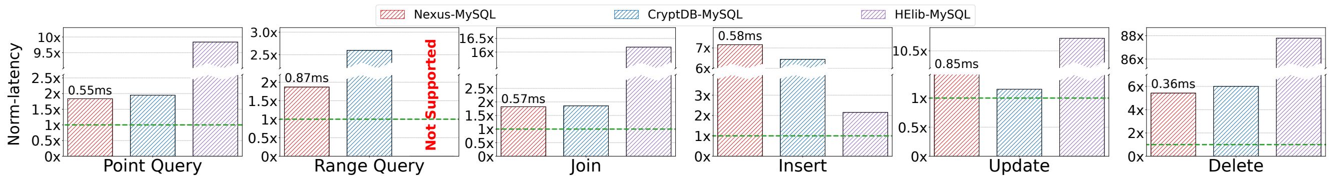


Figure 7: Normalized end-to-end latency of running TPC-C on all baselines in SQL instances. All systems' latency results are normalized to vanilla insecure MySQL (in green dashed line). Values on each red bar indicate ECSTORE's average latency.

Baseline. We implemented a SQL instance using MySQL [37] and a NoSQL instance using WiredTiger [16]. We compared ECSTORE with three baselines: CryptDB [41], HElibDB [?], and vanilla insecure databases (i.e., MySQL and WiredTiger). CryptDB and HElibDB are imperative HE databases. CryptDB cryptographically enables point queries with deterministic encryption (DET) [12] and range queries with order-preserving encryption (OPE) [5], with separate indexes. HElibDB utilizes Fermat's Little Theorem [?] to directly evaluate the equivalence of HE data but requires scanning the entire database.

For an apple-to-apple comparison, we expanded upon baselines in three ways. Firstly, we upgraded CryptDB's HE algorithm from partial HE to fully HE BGV [?] to enable arbitrary computation on ciphertexts as in ECSTORE. Secondly, all baselines were extensively implemented on both SQL and NoSQL instances. Thirdly, we integrated ECSTORE's core proxy components into CryptDB and HElibDB to support multi-party collaborative analytics as in ECSTORE.

Workload and default setting. Since there is no standardized benchmark for evaluating collaborative SQL analytics on EDBs, for a fair comparison, we evaluated workloads from recent EDBs [?, 41, 51, 58], which includes the TPC-C benchmark [32] for relational databases and a customized workload for NoSQL key-value stores.

Specifically, we followed the multi-party deployment approach [49] by first partitioning TPC-C randomly and uniformly across databases based on warehouse ID, and assigned clients to parties based on their associated warehouse IDs. For customized NoSQL workloads, we simulated governmental statistics and medical workloads (as described in §**??**) by generating 2^{10} fixed-size keys (8 bytes) and values (64 bytes) in a manner similar to [56], and horizontally partitioned all key-value pairs across databases as in TPC-C.

To match the setup of existing multi-party query systems [19, 49], we equipped each party with two databases. Unless for scalability experiments, we simulated 10 parties. For the homomorphic setting, we employed HElib's BGV implementation [?], using a plaintext prime modulus of 29 and a ciphertext modulus of 1009 to compress ciphertexts with an expansion coefficient of 4. We ran each experiment for 60 seconds and collected the result in the middle 30s (i.e., 15s∼45s) to avoid the disturbance caused by system start-up and cool-down. We focus on the following questions:

§5.1 How efficient is ECSTORE compared to baselines?

Distributed query latency (in milliseconds)				
Systems (in SQL instance)	**ECSTORE**		**CryptDB**	
Query type	PQ	RQ	PQ	RQ
P1 (***client***): Parameter transform	0.11	0.13	0.19	0.32
P2 (***proxy***): Query dispatch	0.12	0.18	0.06	0.1
P3 (***database***): Data lookup	4e-4	1e-3	2e-4	1e-3
P4 (***proxy***): Query authenticate	1e-3	1e-3	N/S	N/S
P5 (***proxy***): Key switch & aggregate	0.17	0.21	0.12	0.24
P6 (***client***): Result decrypt	0.15	0.35	0.21	0.52
End-to-end ($\sum P_i$)	0.55	0.87	0.58	1.18

Table 2: Breakdown and comparison of query latency. **PQ** and **RQ** mean point and range query respectively. **P1** (***client***) means the first phase of parameter transformation occurs at the client, etc.

§5.2 Can ECSTORE scale to more parties and larger datasets?

§5.3 Does ECSTORE exhibit extensibility?

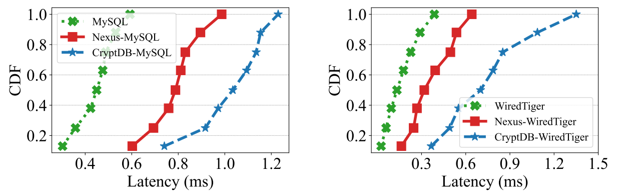
§5.4 What are the lessons we learned?

5.1 End-to-end Performance

We first evaluated the performance of ECSTORE and baselines in a fault-free scenario where no integrity breaches occurred in the query results. The results presented in Figure 7 indicate that ECSTORE demonstrated a lower latency (between 5.4x to 16.6x, excluding *insert* queries) compared to HElibDB. This is attributed to the use of MFT index by ECSTORE, which provides logarithmic search complexity as opposed to HElibDB that requires linear scans to search for equivalent encrypted data, leading to reduced latency for all *read* queries.

Priority: read query latency. HElibDB achieved 3x to 3.3x lower *insert* latency compared to ECSTORE and CryptDB. This is because HElibDB simply appends new records to its storage by trading off the linear search latency on *read* queries. In contrast, ECSTORE and CryptDB both maintain sorted records for encrypted search, leading to higher *insert* latency by first looking up an insert position via indexing before writing a new record to that position. Luckily, it is worth noting that in the joint query scenario of ECSTORE, *read* could occur more frequently (refer to §**??**), and hence, we deem it acceptable to make such a performance trade-off, which has been a common practice in existing insecure databases [16, 37].

General capabilities with breakdown. On the TPC-C workload depicted in Figure 7, ECSTORE achieved 6% to 28% lower latency. For the representative queries (§**??**) running on the NoSQL workload with varying numbers of key-value pairs, ECSTORE had 32% to 55% lower latency shown in Figure 10(a). Q2 exhibited higher latency than Q1 because



(a) Run in a SQL instance (b) Run in a NoSQL instance

Figure 8: Distributions of end-to-end range query latency running TPC-C and customized key-value workload on all baselines.

Q2 involved range searches and a *sum* with aggregates while Q1 simply counted the total number. Both ECSTORE and CryptDB supported point and range queries with logarithmic search complexity; interestingly, ECSTORE still outperformed CryptDB in terms of both query latency (verified in Table 2) and throughput (shown in Figure 9), despite sharing the same complexity.

Table 2 shows the performance breakdown. CryptDB's primary overhead was observed to be in **P1** and **P6** for cryptographic operations. In contrast, ECSTORE incurred additional latency in **P2** for subfilter comparison at leaf nodes (the time is covered by index traversal, analyzed in §**??**), and **P4** for query authentication. Nonetheless, the cryptographic overhead of CryptDB surpassed that of ECSTORE, ultimately leading to the overall performance advantages exhibited by ECSTORE.

We then evaluated the latency distribution by running range queries that accessed 20%∼80% of records on all baselines, using various workloads. Figure 8 shows that ECSTORE reduced the latency for all workloads in comparison to CryptDB. The latency distribution was affected by the length of queries, where longer query ranges led to higher latency of lookup, key switch, aggregation, and decryption. Notably, insecure databases do not require key switch and decryption. HElibDB was not evaluated as it does not support range queries.

Performance on mixed queries. Figure 9 shows that ECSTORE achieved 1.31x to 1.88x higher throughput over CryptDB on the customized workload with mixed point and range queries. This improvement is mainly because CryptDB's design necessitates frequent shuffling between the point search index and range search index, whereas ECSTORE uniformly tackled both query types through a single MFT index abstraction. The throughput gain is also bolstered by our homomorphic implementation that employs the batching strategy (§**??**).

Robustness under attacks. Next, to assess the robustness of ECSTORE, we simulated attacks by randomly manipulating half of the queries running on the KV workload, involving either modifying, replaying, or dropping (deleting) query results. The stable effective throughput in Figure 10(b) demonstrates ECSTORE's 100% detection rate, whereas other baselines with no defenses had significant performance drops. This is due to ECSTORE's two-step integrity check (§4.4): the MAC check detects any modifications or replays on versioned data; the MFT check detects any drops or deletions of query results.

In sum, ECSTORE presents a robust solution for enabling

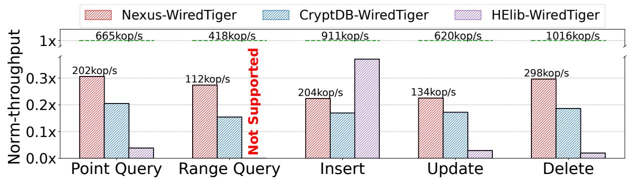
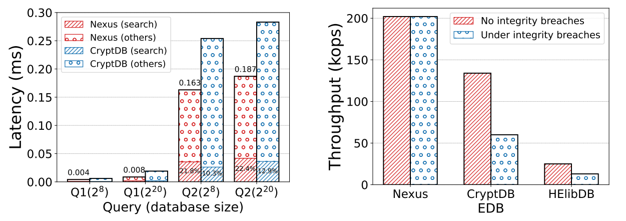


Figure 9: Normalized throughput in NoSQL instances. The green dashed line is vanilla (insecure) WiredTiger's averaged throughput.



(a) Example query in §2.2 (b) Effective throughput

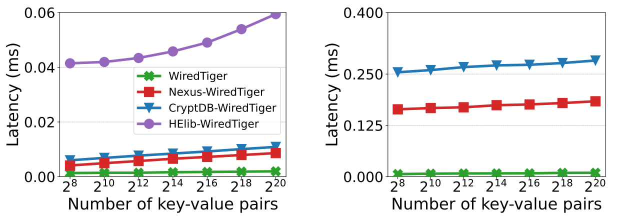
Figure 10: Comparison of example query latency and effective query throughput under integrity breaches.

general queries among multiple collaborative parties, with low latency and high throughput. ECSTORE favors *read* query performance and tolerates moderate write performance downgrades. ECSTORE is most suitable for applications that prioritize *read* performance and require end-to-end security, which is common in collaborative scenarios such as financial statistics across banks [**?**] and medical studies across hospitals [**?**].

5.2 Scalability

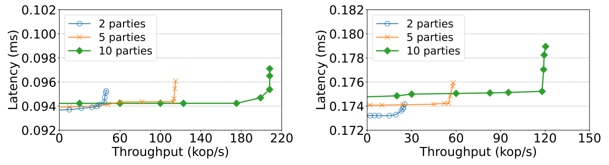
Scale to larger databases. We evaluated baselines with varying database sizes on the KV workload. As depicted in Figure 11, ECSTORE's latency scales logarithmically, matching our complexity analysis in §**??**. For point queries, both ECSTORE and CryptDB incurred moderate overhead when compared to the insecure WiredTiger, while HElibDB had significantly higher latency due to its linear scanning approach. For range queries, ECSTORE achieved a greater performance advantage than baselines compared to point queries because ECSTORE's membership-based indexing is more efficient than prior cryptographic approaches (as also evident in Table 2). HElibDB was not evaluated because it is designed solely for point searches.

Scale to more parties. We varied the number of collaborative parties in ECSTORE, ranging from 2 to 10, which is considered sufficient for collaborative analytics in real-world scenarios [49]. Figure 12 depicts ECSTORE's throughput-latency (99th% tail latency) performance on the KV workload. As the number of parties increased, both ECSTORE's throughput and latency increased. Note that the latency gap between different numbers of parties was more pronounced in range queries compared to point queries, because the range queries typically searched, verified, and aggregated more data, leading to increased query processing time.



(a) Point Query (b) Range Query

Figure 11: Performance comparison with varying database sizes.



(a) Point Query (b) Range Query

Figure 12: Performance comparison with varying numbers of parties.

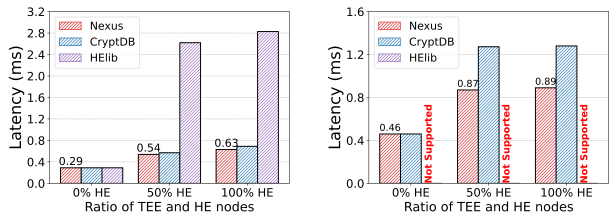
5.3 Extensibility Study

One of the fundamental capabilities of ECSTORE lies in its use of the MFT secure index to dispatch general queries to execute in query-dependent HE databases. This query routing capability is essential for a secure query system and enables ECSTORE to extend its functionality to work seamlessly with TEE databases [?, ?, 42]: ECSTORE can be extended to allow collaborative parties to outsource their data to TEE databases and perform joint queries alongside other parties' HE databases. Concretely, in this study, the clients initialized MFTs for both HE and TEE databases and uploaded MFTs to the proxy for query dispatching. When dealing with queries in TEE databases, the workflow remained unchanged, involving parameter decryption within TEE databases, searching plaintexts in a TEE-shielding B$^+$-tree, and ultimately verifying plaintext results within the TEE (§??).

To validate this extensibility, we conducted experiments by varying the proportion of HE and TEE databases in our default ten-party setting and tested the query latency. We ran TEE databases using the open-sourced Azure EdgelessDB [?], which equips a TEE-shielded B$^+$-tree with optimized index performance. Figure 13 confirms the feasibility of running a combination of HE and TEE databases for collaborative query analytics. Furthermore, the comparison between Figure 13(a) and Figure 13(b) highlights ECSTORE's primary advantage in efficiently supporting range queries when compared to baselines, corroborating the findings from previous experiments.

5.4 Lessons Learned

Cryptographic hash-based security. Cryptographic hash has been widely adopted in security-critical applications such as password managers [?, ?] and phishing detectors [?], showcasing its effectiveness in protecting sensitive information. ECSTORE builds the MFT secure index using cryptographic



(a) Point Query (b) Range Query

Figure 13: Extensibility study of running different ratios of HE and TEE databases in SQL instances distributedly.

hash functions like SHA-256 and inherits the irreversible security [?], in the sense that the encrypted data cannot be reversed to plaintexts even with the knowledge of the hash functions. This design choice enables practical performance and rich query capabilities with query integrity, all achieved through MFT. Unlike previous research (e.g., CryptDB and HElibDB) that prioritizes a cryptographically stronger confidentiality of IND-CPA [?], which often sacrifices query capability or efficiency and overlooks query integrity, the ECSTORE design strikes a balance between practicality and security. As such, it is well-suited to connect EDBs for collaborative query analytics [?, ?, ?, ?] in a malicious threat environment.

Dismissing an alternative design. One may think of using solely monotonic bias with ORE for building an encrypted index, which poses two problems. Firstly, this approach limits query capability to range queries, akin to CryptDB's OPE solution. Secondly, it fails to support query authentication for detecting integrity breaches. In contrast, ECSTORE enables both data confidentiality and query integrity in a unified manner.

Limitations. ECSTORE has two limitations. Firstly, its security is *not* perfect as it does not hide side-channel information related to the specific type of computation performed by queries, such as equality comparisons and sorting operations. This is an inherent issue in most practical encrypted databases that are not integrated with expensive oblivious algorithms [?, 41, 42].

Secondly, ECSTORE's MFT secure index is built on a single searchable attribute, making it unsuitable for databases that search over multiple attributes such as graph databases [?, 10]. For these databases, while ECSTORE is capable of handling disjunctive queries by searching each attribute independently, ECSTORE cannot handle conjunctive queries without integrating MFT with multidimensional indexes such as *k*-d tree [57], which is an interesting direction for future work.

6 Conclusion

We present ECSTORE, the first secure query system that supports general collaborative SQL analytics on EDBs, providing both data confidentiality and query integrity guarantees. ECSTORE leverages the new Merkle rangeFilter Tree (MFT) secure index to efficiently dispatch encrypted queries to query-dependent EDBs, authenticate, and aggregate query results. Extensive results on both SQL and NoSQL instances shown that ECStore is secure, highly efficient, scalable, and extensible compared to baselines. ECStore is open-sourced and its code is released on github.com/2024asplos405/Nexus.

References

[1] Tresorit, 2011. Website: https://tresorit.com/.

[2] Keybase, 2014. Website: https://keybase.io/.

[3] Tiktok data breach timeline, May 15 2023.

[4] Ankit Agarwal, Manju Khari, and Rajiv Singh. Detection of ddos attack using deep learning model in cloud storage application. *Wireless Personal Communications*, pages 1–21, 2021.

[5] Rakesh Agrawal, Jerry Kiernan, Ramakrishnan Srikant, and Yirong Xu. Order preserving encryption for numeric data. In *Proceedings of the 2004 ACM SIGMOD international conference on Management of data*, pages 563–574, 2004.

[6] Ruqayah R Al-Dahhan, Qi Shi, Gyu Myoung Lee, and Kashif Kifayat. Survey on revocation in ciphertext-policy attribute-based encryption. *Sensors*, 19(7):1695, 2019.

[7] Muhammad Bello Aliyu. Efficiency of boolean search strings for information retrieval. *American Journal of Engineering Research*, 6(11):216–222, 2017.

[8] Werner Almesberger et al. Linux network traffic control—implementation overview, 1999.

[9] Fabio Angius, Mario Gerla, and Giovanni Pau. Bloogo: Bloom filter based gossip algorithm for wireless ndn. In *Proceedings of the 1st ACM workshop on Emerging Name-Oriented Mobile Networking Design-Architecture, Algorithms, and Applications*, pages 25–30, 2012.

[10] Renzo Angles and Claudio Gutierrez. Survey of graph database models. *ACM Computing Surveys (CSUR)*, 40(1):1–39, 2008.

[11] Andrew W Appel and David McAllester. An indexed model of recursive types for foundational proof-carrying code. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 23(5):657–683, 2001.

[12] Mihir Bellare, Marc Fischlin, Adam O'Neill, and Thomas Ristenpart. Deterministic encryption: Definitional equivalences and constructions without random oracles. In *Advances in Cryptology–CRYPTO 2008: 28th Annual International Cryptology Conference, Santa Barbara, CA, USA, August 17-21, 2008. Proceedings 28*, pages 360–378. Springer, 2008.

[13] Weicheng Cai, Zexin Cai, Xiang Zhang, Xiaoqi Wang, and Ming Li. A novel learnable dictionary encoding layer for end-to-end language identification. In *2018 IEEE international conference on acoustics, speech and signal processing (ICASSP)*, pages 5189–5193. IEEE, 2018.

[14] Shimin Chen and Qin Jin. Persistent b+-trees in non-volatile main memory. *Proceedings of the VLDB Endowment*, 8(7):786–797, 2015.

[15] Weikeng Chen, Thang Hoang, Jorge Guajardo, and Attila A Yavuz. Titanium: A metadata-hiding file-sharing system with malicious security. *Cryptology ePrint Archive*, 2022.

[16] Rupali Chopade and Vinod Pachghare. Mongodb indexing for performance improvement. In *ICT Systems and Sustainability*, pages 529–539. Springer, 2020.

[17] Homomorphic Encryption Library Contributors. Helib: Homomorphic encryption library. https://github.com/homenc/HElib, 2021.

[18] Henry Corrigan-Gibbs and Dmitry Kogan. Private information retrieval with sublinear online time. In *Advances in Cryptology–EUROCRYPT 2020: 39th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Zagreb, Croatia, May 10–14, 2020, Proceedings, Part I 39*, pages 44–75. Springer, 2020.

[19] Emma Dauterman, Eric Feng, Ellen Luo, Raluca Ada Popa, and Ion Stoica. Dory: An encrypted search system with distributed trust. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, pages 1101–1119, 2020.

[20] Emma Dauterman, Eric Feng, Ellen Luo, Raluca Ada Popa, and Ion Stoica. Dory: An encrypted search system with distributed trust. In *Proceedings of the 14th USENIX Conference on Operating Systems Design and Implementation*, pages 1101–1119, 2020.

[21] INC Dropbox. Dropbox. *http://www. dropbox. com*, 2014.

[22] Paolo Ferragina and Giorgio Vinciguerra. The pgm-index: a fully-dynamic compressed learned index with provable worst-case bounds. *Proceedings of the VLDB Endowment*, 13(8):1162–1175, 2020.

[23] Caroline Fontaine and Fabien Galand. A survey of homomorphic encryption for nonspecialists. *EURASIP Journal on Information Security*, 2007:1–10, 2007.

[24] Eu-Jin Goh. Secure indexes. *Cryptology ePrint Archive*, 2003.

[25] Solomon Golomb. Run-length encodings (corresp.). *IEEE transactions on information theory*, 12(3):399–401, 1966.

[26] IT Governance. List of data breaches and cyber attacks in 2023, 2023. Accessed on November 2, 2023.

[27] Shay Gueron, Simon Johnson, and Jesse Walker. Sha-512/256. In *2011 Eighth International Conference on Information Technology: New Generations*, pages 354–358. IEEE, 2011.

[28] Mark Johnson, Prakash Ishwar, Vinod Prabhakaran, Daniel Schonberg, and Kannan Ramchandran. On compressing encrypted data. *IEEE Transactions on Signal Processing*, 52(10):2992–3006, 2004.

[29] Seny Kamara and Kristin Lauter. Cryptographic cloud storage. In *Financial Cryptography and Data Security: FC 2010 Workshops, RLCPS, WECSR, and WLC 2010, Tenerife, Canary Islands, Spain, January 25-28, 2010, Revised Selected Papers 14*, pages 136–149. Springer, 2010.

[30] Tim Kraska, Alex Beutel, Ed H Chi, Jeffrey Dean, Neoklis Polyzotis, and Lianghong Zhu. Consistency guarantees for parallel incremental data processing. In *Proceedings of the 2019 International Conference on Management of Data*, pages 1271–1288. ACM, 2019.

[31] Arash Habibi Lashkari, Fereshteh Mahdavi, and Vahid Ghomi. A boolean model in information retrieval for search engines. In *2009 International Conference on Information Management and Engineering*, pages 385–389. IEEE, 2009.

[32] Scott T Leutenegger and Daniel Dias. A modeling study of the tpc-c benchmark. *ACM Sigmod Record*, 22(2):22–31, 1993.

[33] Chunbin Li, Wenfei Ma, and Lifang Qin. Efficient and accurate approximate query processing on data warehouses with learned indexes. In *Proceedings of the 2019 International Conference on Management of Data*, pages 1717–1734. ACM, 2019.

[34] Wei Liu, Wenjun Zeng, Lina Dong, and Qiuming Yao. Efficient compression of encrypted grayscale images. *IEEE Transactions on Image Processing*, 19(4):1097–1102, 2009.

[35] Yiping Ma, Ke Zhong, Tal Rabin, and Sebastian Angel. Incremental {Offline/Online}{PIR}. In *31st USENIX Security Symposium (USENIX Security 22)*, pages 1741–1758, 2022.

[36] Martin Mulazzani, Sebastian Schrittwieser, Manuel Leithner, Markus Huber, and Edgar Weippl. Dark clouds on the horizon: Using cloud storage as attack vector and online slack space. In *20th USENIX Security Symposium (USENIX Security 11)*, 2011.

[37] AB MySQL. Mysql, 2001.

[38] Patrick O'Neil, Edward Cheng, Dieter Gawlick, and Elizabeth O'Neil. The log-structured merge-tree (lsm-tree). *Acta Informatica*, 33:351–385, 1996.

[39] Vasilis Pappas, Fernando Krell, Binh Vo, Vladimir Kolesnikov, Tal Malkin, Seung Geol Choi, Wesley George, Angelos Keromytis, and Steve Bellovin. Blind seer: A scalable private dbms. In *2014 IEEE Symposium on Security and Privacy*, pages 359–374. IEEE, 2014.

[40] Rishabh Poddar, Sukrit Kalra, Avishay Yanai, Ryan Deng, Raluca Ada Popa, and Joseph M Hellerstein. Senate: a {Maliciously-Secure}{MPC} platform for collaborative analytics. In *30th USENIX Security Symposium (USENIX Security 21)*, pages 2129–2146, 2021.

[41] Raluca Ada Popa, Catherine MS Redfield, Nickolai Zeldovich, and Hari Balakrishnan. Cryptdb: Protecting confidentiality with encrypted query processing. In *Proceedings of the twenty-third ACM symposium on operating systems principles*, pages 85–100, 2011.

[42] Christian Priebe, Kapil Vaswani, and Manuel Costa. Enclavedb: A secure database using sgx. In *2018 IEEE Symposium on Security and Privacy (SP)*, pages 264–278. IEEE, 2018.

[43] Dipu Sarkar, Abhinandan De, Chandan Kumar Chanda, and Sanjay Goswami. Kruskal's maximal spanning tree algorithm for optimizing distribution network topology to improve voltage stability. *Electric Power Components and Systems*, 43(17):1921–1930, 2015.

[44] Damien Stehlé and Ron Steinfeld. Faster fully homomorphic encryption. In *Advances in Cryptology-ASIACRYPT 2010: 16th International Conference on the Theory and Application of Cryptology and Information Security, Singapore, December 5-9, 2010. Proceedings 16*, pages 377–394. Springer, 2010.

[45] Hoeteck Wee. On pseudoentropy versus compressibility. In *Proceedings. 19th IEEE Annual Conference on Computational Complexity, 2004.*, pages 29–41. IEEE, 2004.

[46] Eric W Weisstein. Fermat's little theorem. *https://mathworld. wolfram. com/*, 2004.

[47] Kevin Wilson and Kevin Wilson. Onedrive. *Everyday Computing with Windows 8.1*, pages 71–74, 2015.

[48] Jiacheng Wu, Yong Zhang, Shimin Chen, Jin Wang, Yu Chen, and Chunxiao Xing. Updatable learned index with precise positions. *arXiv preprint arXiv:2104.05520*, 2021.

[49] Pengfei Wu, Jianting Ning, Jiamin Shen, Hongbing Wang, and Ee-Chien Chang. Hybrid trust multi-party computation with trusted execution environment.

[50] Pengfei Wu, Jianting Ning, Jiamin Shen, Hongbing Wang, and Ee-Chien Chang. Hybrid trust multi-party computation with trusted execution environment. In *The Network and Distributed System Security (NDSS) Symposium*, 2022.

[51] Yu Xia, Xiangyao Yu, Matthew Butrovich, Andrew Pavlo, and Srinivas Devadas. Litmus: Towards a practical database management system with verifiable acid properties and transaction correctness. In *Proceedings of the 2022 International Conference on Management of Data, Philadelphia, PA, USA*, pages 12–17, 2022.

[52] Liang Xiao, Dongjin Xu, Caixia Xie, Narayan B Mandayam, and H Vincent Poor. Cloud storage defense against advanced persistent threats: A prospect theoretic study. *IEEE Journal on Selected Areas in Communications*, 35(3):534–544, 2017.

[53] Kaiping Xue, Weikeng Chen, Wei Li, Jianan Hong, and Peilin Hong. Combining data owner-side and cloud-side access control for encrypted cloud storage. *IEEE Transactions on Information Forensics and Security*, 13(8):2062–2074, 2018.

[54] Xun Yi, Russell Paulet, Elisa Bertino, Xun Yi, Russell Paulet, and Elisa Bertino. *Homomorphic encryption*. Springer, 2014.

[55] Wenting Zheng, Frank Li, Raluca Ada Popa, Ion Stoica, and Rachit Agarwal. Minicrypt: Reconciling encryption and compression for big data stores. In *Proceedings of the Twelfth European Conference on Computer Systems*, pages 191–204, 2017.

[56] Yuhong Zhong, Haoyu Li, Yu Jian Wu, Ioannis Zarkadas, Jeffrey Tao, Evan Mesterhazy, Michael Makris, Junfeng Yang, Amy Tai, Ryan Stutsman, et al. {XRP}:{In-Kernel} storage functions with {eBPF}. In *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*, pages 375–393, 2022.

[57] Kun Zhou, Qiming Hou, Rui Wang, and Baining Guo. Real-time kd-tree construction on graphics hardware. *ACM Transactions on Graphics (TOG)*, 27(5):1–11, 2008.

[58] Wenchao Zhou, Yifan Cai, Yanqing Peng, Sheng Wang, Ke Ma, and Feifei Li. Veridb: An sgx-based verifiable database. In *Proceedings of the 2021 International Conference on Management of Data*, pages 2182–2194, 2021.