Design & Analysis of OST-1

Contents

Note that this is work in progress. Version 1 of this document includes:

(a) a detailed description of OST_1's core multi-map encryption scheme Ω_P;

(b) a detailed description of OST_1's range multi-map encryption scheme Ω_R;

(c) *a detailed description of* OST_1 *after storage-level emulation (see section 7)* ;

Version 2 of this document will include:

(a) a security analysis of Ω_P, Ω_R and OST_1;

(b) a detailed description of OST_1 before emulation;

1 Introduction

In this document we describe and analyze the OST_1 construction. OST_1 is a (document) database encryption scheme scheme designed with the following goals in mind: (1) snapshot security; (2) support for multiple clients; (3) efficient support for concurrent operations; and (4) resilience to client failures.

Building blocks. OST_1 is based on two new multi-map encryption schemes Ω_P and Ω_R that achieve all the properties above. Ω_R is a range multi-map encryption scheme itself based on Ω_P and Ω_P is based on four multi-map encryption schemes that each achieve different characteristics and are used for different purposes.

Snapshot security. A (memory-level) snapshot adversary has access to the entire memory and disk of the server at a particular point in time. This means that at that instant, it can access the entire database, any keys stored in memory, all the caches and all the logs. Snapshot-secure structured encryption was introduced in [AKM19] together with a scheme called DLS that is both zero-leakage against snapshot adversaries and forward-private against persistent adversaries. While DLS is relatively efficient, it is very complex and does not support the properties above. Ω_P and Ω_R, on the other hand, are more efficient than DLS but provide weaker security guarantees against persistent adversaries.

The multi-client setting. In practice, databases are accessed by multiple clients so any underlying STE scheme needs to work in the multi-writer multi-reader (MWMR) setting [KL10]. In a multi-writer setting, clients can issue put operations at the same time which can cause contention and reduce write throughput. Designing for the multi-writer setting is much more challenging than the standard single-writer single-reader setting for which we have many constructions. In fact, as far as we know, our new constructions are the first multi-writer multi-reader structured encryption schemes.

State. One of the biggest challenges in designing MWMR schemes is dealing with state. All modern dynamic multi-map encryption schemes require the client to keep state which becomes difficult to manage in a multi-client setting because clients need to maintain a consistent view of it. Another important consideration in our setting is that that clients can crash at any time and cause state information to be lost. In addition, it is important that any crash recovery protocol be efficient. For all these reasons, our main technical goal is to design schemes that are stateless.

2 Preliminaries

Notation. The set of all binary strings of length n is denoted as $\{0,1\}^n$, and the set of all finite binary strings as $\{0,1\}^*$. $[n]$ is the set of integers $\{1,\ldots,n\}$. The output y of a probabilistic algorithm $\mathcal{A}$ on input x is denoted by $y \leftarrow \mathcal{A}(x)$. The output y of a deterministic algorithm $\mathcal{A}$ on input x is denoted by $y := \mathcal{A}(x)$. If S is a set then $x \xleftarrow{\$} S$ denotes sampling from S uniformly at random. Given a sequence $\mathbf{s}$ of n elements, we refer to its ith element as s_i. If S is a set then $\#S$ refers to its cardinality. Throughout, k will denote the security parameter.

Dictionaries & multi-maps. A dictionary DX with capacity n is a collection of n label/value pairs $\{(\ell_i, v_i)\}_{i\leq n}$ and supports Get and Put operations. We write $v_i := \mathsf{DX}[\ell_i]$ to denote getting the value associated with label ℓ_i and $\mathsf{DX}[\ell_i] := v_i$ to denote the operation of putting the value v_i in DX with label ℓ_i.

A multi-map MM with capacity n is a collection of n label/tuple pairs $\{(\ell_i, \mathbf{v}_i)_i\}_{i\leq n}$ that supports Get and Put operations. We write $\mathbf{v}_i := \mathsf{MM}[\ell_i]$ to denote getting the tuple associated with label ℓ_i and $\mathsf{MM}[\ell_i] := \mathbf{v}_i$ to denote operation of putting the tuple $\mathbf{v}_i$ to label ℓ_i. Multi-maps are the abstract data type instantiated by an inverted index. We define a *range multi-map* as a multi-map RMM that supports—in addition to Get and Put operations—range queries: given a range $[a, b] \subseteq \mathbb{Z}^2$, return the set of values $V = \bigcup_{\ell\in[a,b]} \mathsf{RMM}[\ell]$. We write $V = \mathsf{RMM}[[a, b]]$ to denote getting the values associated with the range $[a, b]$.

Document databases. A document database DDB of size n holds n documents $\{\mathbf{D}_1, \ldots, \mathbf{D}_n\}$ each of which is a set of field/value pairs. For ease of exposition and without loss of generality, we will assume throughout that all documents in a database have the same number of field/value pairs. More precisely, for all $1 \leq i \leq n$, we have $\mathbf{D}_i = \big((f_1, v_1), \ldots, (f_m, v_m)\big)$. Here, we consider document databases with fields that support the following queries. An *exact search* query takes as input a field/value pair (f, v) and returns the documents in DDB that include the field f with value v. A *range search* query takes as input a range $[a, b]$ and returns the documents in DDB that include the field f with values between a and b.

In this work, we are concerned with designing MongoDB-friendly schemes and we assume familiarity with mongo shell query and update operations [mon].

Basic cryptographic primitives. A symmetric-key encryption scheme is a set of three polynomial-time algorithms $\mathsf{SKE} = (\mathsf{Gen}, \mathsf{Enc}, \mathsf{Dec})$ such that Gen is a probabilistic algorithm that takes a security parameter k and returns a secret key K; Enc is a probabilistic algorithm that takes a key K and a message m and returns a ciphertext c; Dec is a deterministic algorithm that takes a key K and a ciphertext c and returns m if K was the key under which c was produced. Informally, a private-key encryption scheme is secure against chosen-plaintext attacks (CPA) if the ciphertexts it outputs do not reveal any partial information about the plaintext even to an adversary that can adaptively query an encryption oracle. We say a scheme is random-ciphertext-secure against chosen-plaintext attacks (RCPA) if the ciphertexts it outputs are computationally indistinguishable from random even to an adversary that can adaptively query an encryption oracle.[1] In addition to encryption schemes, we also

[1]RCPA-secure encryption can be instantiated practically using either the standard PRF-based private-key encryption scheme or, e.g., AES in counter mode.

make use of pseudo-random functions (PRF), which are polynomial-time computable functions that cannot be distinguished from random functions by any probabilistic polynomial-time adversary. We usually denote the evaluation of a pseudo-random function F with a key K on an input x as $F_K(x)$ but sometimes as $F(K, x)$ for visual clarity. We refer the reader to [KL08] for formal security definitions.

Hypergraphs. A hypergraph $H = (V, \mathbf{E})$ consists of a set of n vertices $V = \{v_1, \ldots, v_n\}$ and a collection of m non-empty edges $\mathbf{E} = \{e_1, \ldots, e_m\}$ such that, for all $i \in [m]$, $e_i \subseteq V$. The degree of a vertex $v \in V$ is the number of edges in $\mathbf{E}$ that contain v and is denoted by $\mathsf{deg}(v)$. We define a *range hypergraph* to be a hypergraph $H = (V, \mathbf{E})$ such that V is a total order and such that for all ranges $r \in \mathcal{R}(V)$, there exists a subset $\mathbf{C}_r \subseteq \mathbf{E}$ such that $\bigcup_{e \in \mathbf{C}_r} e = r$. We refer to such a set as a *cover* of the range r. The *min-cover* of a range $r \subseteq V$ is the set

$$\mathbf{C}_r = \mathrm{argmin}_{\mathbf{C} \subseteq \mathbf{E}} \left\{ \#\mathbf{C} \; : \; \bigcup_{e \in \mathbf{C}} e = r \right\}.$$

To make use of a hypergraph H in our constructions, we will need efficient algorithms to create and manipulate it. In particular, we will need two efficient algorithms: Edges_H and $\mathsf{Mincover}_H$. Edges_H takes as input a vertex v and outputs the subset of edges $\mathbf{E}_v \subseteq \mathbf{E}$ that include v. Finally, $\mathsf{Mincover}$ takes as input a range $r \in \mathcal{R}(V)$ and outputs its min-cover $\mathbf{C}_r$.

3 Building Blocks

Our stateless multi-map encryption scheme Ω_P makes use of four multi-map encryption schemes as building blocks; each one with different characteristics and used for a different purpose. At a high level, the first scheme, Σ_M, is used to encrypt the input multi-map which results in what we call the main encrypted multi-map EMM_M. The second scheme, Σ_C, is used to encrypt metadata about the main encrypted multi-map that is needed to avoid overwriting items in EMM_M. The third scheme, Σ_D, is used to store information about items deleted in the main encrypted multi-map in order to speed up queries on EMM_M. The last scheme, Σ_P, is used to store information that is needed to compact the auxiliary structures; that is, to reduce their space consumption. We now describe each scheme individually before describing Ω_P in Section 4.

3.1 Two-Dimensional Addressable Multi-Maps

As mentioned above, Ω_P will use the first scheme Σ_M to encrypt the input multi-map MM, resulting in the "main" encrypted multi-map EMM_M. Σ_M is a π_{dyn}-style construction [CJJ+14] but with the following differences. First, it is what we refer to as a two-dimensional multi-map encryption scheme which we will explain in more detail below. Second, it achieves statelessness at the cost of correctness in the sense that the values associated to a label can be overwritten. To better capture this behavior, Σ_M is defined as supporting read, write and erase operations instead of get, put and delete operations. More precisely, these operations work as follows:

- **write**: takes as input a label ℓ, a tuple $\mathbf{v}$ and a sequence of addresses $\mathbf{a}$ and stores the pair $(\ell, \mathbf{v}')$ such that for all $1 \leq i \leq \#\mathbf{v}$, v_i is stored at index a_i of $\mathbf{v}'$. Note that $\#\mathbf{v}' \geq \#\mathbf{v}$.

- **read**: takes as input a label ℓ and a sequence of addresses $\mathbf{a}$ and returns the values in ℓ's tuple $\mathbf{v}'$ indexed by $\mathbf{a}$.

- **erase**: takes as input a label ℓ and a sequence of addresses $\mathbf{a}$ and removes the values indexed by $\mathbf{a}$ from ℓ's tuple $\mathbf{v}'$.

We refer to this kind of multi-map as an *addressable multi-map*.

Concurrency via two-dimensionality. The encrypted multi-map EMM_M will be used by Ω_P to store the tuple associated with a label ℓ. The way we use this structure, however, will result in contention when multiple clients are writing to the same label which will, in turn, slow down Ω_P's write throughput under parallel put operations.

We address this in the following way. Instead of using a standard multi-map, EMM_M will be a 2-dimensional (encrypted) multi-map by which we mean that it holds label/tuple pairs with labels of the form $\ell = (\ell_x, \ell_y)$. Given a high contention label ℓ, Ω_P will treat it as a two-dimensional label $\ell' = (\ell, u)$, where u is a value sampled uniformly at random from $\{1, \ldots, p\}$, and store the pair $((\ell, u), \mathbf{v})$ in EMM_M. The high-level idea is that if n clients try to write to the same high-contention label ℓ then, in expectation, only n/p writes will be executed on the same two-dimensional label $\ell' = (\ell, u)$ in EMM_M. Note that a possible optimization here would be to use two-choice allocation instead of just sampling u at random.

To make this idea work in practice, we will need the two-dimensional encrypted multi-map to support—in addition to read, write and erase operations–read operations on a single dimension. To see why, note that under our approach, n write $(\ell, \mathbf{v})$ operations for EMM_M will be transformed to n write of the form $((\ell, u), \mathbf{v})$ for $1 \leq u \leq p$. This does not cause any issue during write operations but it does create a problem for reads since a read for ℓ now needs to return the values associated with every two-dimensional label $(\ell, u)_{1 \leq u \leq p}$. Handling this in the naive way would require the client to compute and send p read tokens to the server; one for each $u \in \{1, \ldots, p\}$. Instead, we will design our scheme to support two additional algorithms, $\mathsf{ReadXToken}$ and $\mathsf{ReadXYToken}$, that work as follows. The first algorithm, $\mathsf{ReadXToken}$, is used by the client to generate a read token for the x-component of a label $\ell = (\ell_x, \ell_y)$. The second algorithm, $\mathsf{ReadXYToken}$, is used by the server to generate a read token for $\ell = (\ell_x, \ell_y)$ given a read token for ℓ_x and the y-component ℓ_y. Returning to our concurrency problem, when querying for a label ℓ, the client can now send to the server a read token for ℓ and the server can use that to generate read tokens for the two-dimensional labels $(\ell, 1), \ldots, (\ell, p)$.

We now provide the syntax of addressable two-dimensional multi-map encryption schemes.

Definition 3.1. *A response-hiding stateless addressable two-dimensional multi-map encryption encryption scheme is a structured encryption scheme* $\Sigma_M = (\mathsf{Init}, \mathsf{WriteToken}, \mathsf{Write}, \mathsf{ReadToken}, \mathsf{ReadXToken}, \mathsf{ReadXYToken}, \mathsf{Read}, \mathsf{EraseToken}, \mathsf{Erase}, \mathsf{Resolve})$ *that consists of ten polynomial-time algorithms that work as follows:*

- $(K, \mathsf{EMM}) \leftarrow \mathsf{Init}(1^k)$*: takes as input a security parameter* k *and outputs a secret key* K *and an encrypted multi-map* EMM*;*
- $\mathsf{wtk} \leftarrow \mathsf{WriteToken}(K, (\ell_x, \ell_y), \mathbf{v})$*: takes as input a key* K*, a two-dimensional label* (ℓ_x, ℓ_y) *and a tuple of values* $\mathbf{v}$ *and outputs a write token* wtk*;*
- $\mathsf{EMM}' \leftarrow \mathsf{Write}(\mathsf{EMM}, \mathsf{wtk}, \mathbf{a})$*: takes as input an encrypted multi-map* EMM*, a write token* wtk *and a sequence of addresses* $\mathbf{a}$ *and outputs an updated encrypted multi-map* EMM'*;*
- $\mathsf{rtk} \leftarrow \mathsf{ReadToken}(K, (\ell_x, \ell_y))$*: takes as input a key* K*, a two-dimensional label* (ℓ_x, ℓ_y) *and outputs a read token* rtk*;*
- $\mathsf{rxtk} \leftarrow \mathsf{ReadXToken}(K, \ell_x)$*: takes as input a key* K*, the x-component* ℓ_x *of a two-dimensional label and outputs a read-x token* rxtk*;*
- $\mathsf{rtk} \leftarrow \mathsf{ReadXYToken}(\mathsf{rxtk}, \ell_y)$*: takes as input a read-x token, the y-component* ℓ_y *of a two-dimensional label and outputs a read token* rtk*;*
- $\mathbf{ct} \leftarrow \mathsf{Read}(\mathsf{EMM}, \mathsf{rtk}, \mathbf{a})$*: takes as input an encrypted multi-map* EMM*, a read token* rtk *and a sequence of addresses* $\mathbf{a}$ *and outputs a sequence of ciphertexts* $\mathbf{ct}$*;*
- $\mathsf{etk} \leftarrow \mathsf{EraseToken}(K, (\ell_x, \ell_y))$*: takes as input a key* K*, a two-dimensional label* (ℓ_x, ℓ_y) *and outputs an erase token* etk*;*
- $\mathsf{EMM}' \leftarrow \mathsf{Erase}(\mathsf{EMM}, \mathsf{etk}, a)$*: takes as input an encrypted multi-map* EMM*, an erase token* etk *and an address* a *and outputs an updated encrypted multi-map* EMM'*;*
- $\mathbf{v} \leftarrow \mathsf{Resolve}(K, \mathbf{ct})$*: takes as input a key* K *and a sequence of ciphertexts* $\mathbf{ct}$ *and outputs a sequence of values* $\mathbf{v}$*.*

We now describe Σ_M, our practical *stateless* encryption scheme for addressable two-dimensional multi-maps.

Overview. The scheme is described in detail in Figure 1 and works as follows. It makes use of a pseudo-random function F and of a symmetric encryption scheme SKE. Init samples a k-bit key K_t for F, generates a key K_e for SKE and initializes an empty dictionary DX that will represent the encrypted multi-map EMM. The $\mathsf{WriteToken}$ algorithm produces a write token wtk that consists of a key $K_\ell := F(F_{K_t}(\ell_x), \ell_y)$ and encryptions of each value in $\mathbf{v}$ under the key K_e. The Write algorithm stores pairs of the form (t_i, ct_i) in the dictionary DX, where $t_i := F_{K_\ell}(a_i)$ and ct_i is the encryption of v_i. The $\mathsf{ReadToken}$ algorithm simply returns the key $K_\ell := F(F_{K_t}(\ell_x), \ell_y)$ as the read token rtk and Read returns the ciphertexts in DX associated to the labels $F_{K_\ell}(a_i)$, for all $a_i \in \mathbf{a}$. The $\mathsf{ReadXToken}$ algorithm simply returns $K_x := F_{K_t}(\ell_x)$ as its read-x token and $\mathsf{ReadXYToken}$ returns $F_{K_x}(\ell_y)$ as the read token. $\mathsf{EraseToken}$ outputs $K_\ell := F(F_{K_t}(\ell_x), \ell_y)$ as the erase token etk and Erase sets $\mathsf{DX}[F_{K_\ell}(a)]$ to $\perp$. Finally, $\mathsf{Resolve}$ recovers $\mathbf{v}$ by decrypting the sequence of ciphertexts $\mathbf{ct}$ using K_e.

Remark on correctness. Note that since the scheme is addressable, it does not inherently guarantee correctness since tuple values can be overwritten if writes for two different values are made to the same address. In the next section, we will see how to use another scheme to encrypt an auxiliary structure that will provide "overwrite protection" for EMM_M.

Efficiency analysis. Σ_M is optimal with respect to communication complexity: write tokens are $O(\#\mathbf{v})$, read and erase tokens are $O(1)$ and read responses are $O(\#\mathbf{a})$. The scheme is also optimal with respect to server-side computation since writes and reads are $O(\#\mathbf{a})$ and erase operations are $O(1)$. Finally, client-side operations are also optimal since computing write tokens is $O(\#\mathbf{a})$, computing read and erase tokens is $O(1)$ and resolving is $O(\#\mathbf{ct})$.

3.2 Two-Dimensional Immutable Dictionaries

Our second building block, Σ_C, is a dictionary encryption scheme that achieves statelesness and correctness at the cost of limited query functionality and (in some cases) a slight decrease in query efficiency. It is the most complex of our building blocks because it needs to satisfy several non-standard properties which we will now discuss.

Overwrite protection. As explained above, Σ_M achieves statelessness by giving up on correctness and, specifically, by not guaranteeing that values cannot be overwritten. To address this limitation Ω_P will use an auxiliary encrypted structure EDX_C produced with a dictionary encryption scheme Σ_C to store information that will help prevent overwrites in EMM_M. Σ_C, however, has to be designed in such a way that it is both stateless and correct, in the sense that it does not allow overwrites.

The simplest way to achieve this is to associate a counter count_ℓ with every label ℓ in the main encrypted multi-map EMM_M, store the pairs $(\ell, \mathsf{count}_\ell)$ in a dictionary DX, encrypt DX using a response-revealing dictionary encryption scheme and store the resulting encrypted dictionary EDX_C with the main encrypted multi-map EMM_M. To add a label/tuple pair $(\ell, \mathbf{v})$ to EMM_M, the client sends encryptions of $\mathbf{v}$ and a Σ_C get token gtk_C for ℓ so that the server can query EDX_C, recover count_ℓ and store the ciphertexts $\mathbf{ct}$ in EMM_M at addresses $\mathbf{a} = (\mathsf{count}_\ell + 1, \ldots, \mathsf{count}_\ell + \#\mathbf{ct})$. The server then updates the pair $(\ell, \mathsf{count}_\ell)$ in EDX_C to $(\ell, \mathsf{count}_\ell + \#\mathbf{ct})$.

Snapshot security via immutability. While this approach may seem reasonable, it has a subtle security flaw if implemented naively. The problem is with the last step where the server

Let $F : \{0,1\}^k \times \{0,1\}^* \to \{0,1\}^*$ be a pseudo-random function, $\mathsf{SKE} = (\mathsf{Gen}, \mathsf{Enc}, \mathsf{Dec})$ be a symmetric encryption scheme. Consider the response-revealing stateless addressable two-dimensional multi-map encryption scheme $\Sigma_M = (\mathsf{Init}, \mathsf{WriteToken}, \mathsf{Write}, \mathsf{ReadToken}, \mathsf{ReadXToken}, \mathsf{ReadXYToken}, \mathsf{Read}, \mathsf{EraseToken}, \mathsf{Erase}, \mathsf{Resolve})$ defined as follows:

- $\mathsf{Init}\big(1^k\big)$:
  1. sample a key $K_t \xleftarrow{\$} \{0,1\}^k$ and compute $K_e \leftarrow \mathsf{SKE.Gen}(1^k)$;
  2. initialize an empty dictionary DX;
  3. output $K := (K_t, K_e)$ and $\mathsf{EMM} := \mathsf{DX}$;
- $\mathsf{WriteToken}\big(K, (\ell_x, \ell_y), \mathbf{v}\big)$:
  1. parse K as (K_t, K_e);
  2. compute $K_\ell := F(F_{K_t}(\ell_x), \ell_y)$;
  3. for all $1 \leq i \leq \#\mathbf{v}$, compute $\mathrm{ct}_i \leftarrow \mathsf{SKE.Enc}(K_e, v_i)$;
  4. set $\mathbf{ct} := (\mathrm{ct}_1, \ldots, \mathrm{ct}_{\#\mathbf{v}})$;
  5. output $\mathsf{wtk} := (K_\ell, \mathbf{ct})$;
- $\mathsf{Write}\big(\mathsf{EMM}, \mathsf{wtk}, \mathbf{a}\big)$:
  1. parse EMM as DX, wtk as $(K_\ell, \mathbf{ct})$;
  2. for all $1 \leq i \leq \#\mathbf{a}$, set $\mathsf{DX}[F_{K_\ell}(a_i)] := \mathrm{ct}_i$;
  3. output $\mathsf{EMM} := \mathsf{DX}$;
- $\mathsf{ReadToken}\big(K, (\ell_x, \ell_y)\big)$:
  1. parse K as (K_t, K_e);
  2. compute $K_\ell := F(F_{K_t}(\ell_x), \ell_y)$;
  3. output $\mathsf{rtk} := K_\ell$;
- $\mathsf{ReadXToken}\big(K, \ell_x\big)$: parse K as (K_t, K_e) and output $\mathsf{rxtk} := F_{K_t}(\ell_x)$.
- $\mathsf{ReadXYToken}\big(\mathsf{rxtk}, \ell_y\big)$: output $\mathsf{rtk} := F_{\mathsf{rxtk}}(\ell_y)$.
- $\mathsf{Read}\big(\mathsf{EMM}, \mathsf{rtk}, \mathbf{a}\big)$:
  1. parse EMM as DX and rtk as K_ℓ;
  2. initialize an empty sequence $\mathbf{ct}$;
  3. for all $1 \leq i \leq \#\mathbf{a}$, compute $\mathrm{ct}_i := \mathsf{DX}\big[F_{K_\ell}(a_i)\big]$ and set $\mathbf{ct} := (\mathbf{ct}, \mathrm{ct}_i)$;
  4. output $\mathbf{ct}$;
- $\mathsf{EraseToken}\big(K, (\ell_x, \ell_y)\big)$:
  1. parse K as (K_t, K_e);
  2. compute $K_\ell := F_{K_t}(\ell)$;
  3. output $\mathsf{etk} := K_\ell$;
- $\mathsf{Erase}\big(\mathsf{EMM}, \mathsf{etk}, a\big)$:
  1. parse EMM as DX and etk as K_ℓ;
  2. set $\mathsf{DX}[F_{K_\ell}(a)] := \perp$;
  3. output $\mathsf{EMM} := \mathsf{DX}$;
- $\mathsf{Resolve}\big(K, \mathbf{ct}\big)$:
  1. parse K as (K_1, K_2);
  2. initialize an empty sequence $\mathbf{v}$;
  3. for all $1 \leq i \leq \#\mathbf{ct}$, compute $v_i := \mathsf{Dec}_{K_2}(\mathrm{ct}_i)$ and set $\mathbf{v} := (\mathbf{v}, v_i)$;
  4. output $\mathbf{v}$.

Figure 1: Σ_M: a stateless addressable two-dimensional multi-map encryption scheme.

updates EDX_C with the new counter value. If this is done in-place, then a snapshot adversary will be able to correlate EDX_C put operations—and therefore EMM_M write operations—since every put for a label ℓ results in changes at a specific location of EDX.[2] To handle this, we need to design Σ_C in such a way that it supports edits in an immutable manner so that correlations are not revealed. One way to do this is to implement the encrypted dictionary using an encrypted multi-map and to implement dictionary edit operations with multi-map append operations; for example, changing a pair (ℓ, v) in the encrypted dictionary to (ℓ, v') is implemented by appending the new value v' to ℓ's tuple in an encrypted multi-map. A dictionary get operation for ℓ can then be implemented by returning the last value of ℓ's tuple in the underlying multi-map. Note that because an EDX_C-level edit is implemented as an encrypted multi-map append, a snapshot adversary cannot correlate between edit operations.

(Efficient) Immutability via completeness. Recall that any STE scheme we use as a building block for Ω_P must be stateless; including the encrypted dictionary EDX_C and its underlying encrypted multi-map. This may seem contradictory, however, since the problem we are trying to solve in the first place is to design a stateless encrypted multi-map. Fortunately, the way we use EDX_C's underlying EMM will guarantee that the EMM has a special property which will allow us to design a stateless and correct scheme. Specifically, the underlying multi-map will always be *complete*, in the sense that for all labels ℓ, if ℓ's tuple $\mathbf{v}$ includes m values then there does not exist an index $1 \leq i \leq m$ such that $v_i = \perp$.

This guarantee of completeness will allow us to support *get tail* operations on the underlying encrypted multi-map efficiently, where the tail of a label/tuple pair is the last element of the label's tuple. More precisely, we do this using the following variant of binary search. Consider a sequence $S = (v_1, \cdots, v_n, \perp_{n+1}, \cdots, \perp_N)$. Given S, we would like to find the address a such that $v_a \neq \perp$ but $v_{a+1} = \perp$. This problem can be solved in $O(N)$ time with linear scanning but also in $O(\log N)$ time as follows: given S, check if the element at address $N/2$ is $\perp$; if so we recur on the "left half" of S otherwise recur on the "right half" of S. The base case occurs when the set holds a single element. Note that this algorithm can only work if S is complete. The algorithm is described in detail in Figure 6

Concurrency via two-dimensionality. Other characteristics of Σ_C is that, like Σ_M, it is two-dimensional in order to provide support for concurrent Ω_P operations.

Definition 3.2. *A response-revealing stateless immutable two-dimensional multi-map encryption scheme is a structured encryption scheme* $\Sigma_C = (\mathsf{Init}, \mathsf{PutKey}, \mathsf{PutToken}, \mathsf{Put}, \mathsf{GetToken}, \mathsf{GetXToken}, \mathsf{GetXYToken}, \mathsf{Get}, \mathsf{DeleteToken}, \mathsf{Delete})$ *consists of ten polynomial-time algorithms that work as follows:*

- $(K, \mathsf{EDX}) \leftarrow \mathsf{Init}(1^k)$*: takes as input a security parameter k and outputs a secret key K and an encrypted dictionary* EDX*;*

[2]Even if the location of the pairs in EDX_C's underlying structured are randomized, there would still be a consistent string associated to the pair that could be used to correlate.

- $\mathsf{pk} \leftarrow \mathsf{PutKey}(K, (\ell_x, \ell_y))$*: takes as input a key* K*, a two-dimensional label* (ℓ_x, ℓ_y) *and outputs a put key* pk*;*

- $\mathsf{ptk} \leftarrow \mathsf{PutToken}(\mathsf{pk}, v)$*: takes as input a put key* pk*, a value* v *and outputs a put token* ptk*;*

- $\mathsf{EDX}' \leftarrow \mathsf{Put}(\mathsf{EDX}, \mathsf{ptk})$*: takes as input an encrypted dictionary* EDX*, a put token* ptk *and outputs an updated encrypted dictionary* EDX'*;*

- $\mathsf{gtk} \leftarrow \mathsf{GetToken}(K, (\ell_x, \ell_y))$*: takes as input a key* K*, a two-dimensional label* (ℓ_x, ℓ_y) *and outputs a get token* gtk*;*

- $\mathsf{gxtk} \leftarrow \mathsf{GetXToken}(K, \ell_x)$*: takes as input a key* K*, the x-component* ℓ_x *of a two-dimensional label and outputs a get-x token* gxtk*;*

- $\mathsf{gtk} \leftarrow \mathsf{GetXYToken}(\mathsf{gxtk}, \ell_y)$*: takes as input a get-x token, the y-component of a label* ℓ_y *and outputs a get token* gtk*;*

- $v \leftarrow \mathsf{Get}(\mathsf{EDX}, \mathsf{gtk})$*: takes as input an encrypted dictionary* EDX*, a get token* gtk *and outputs a value* v*;*

- $\mathsf{dtk} \leftarrow \mathsf{DeleteToken}(K, (\ell_x, \ell_y))$*: takes as input a key* K*, a two-dimensional label* (ℓ_x, ℓ_y) *and outputs a delete token* dtk*;*

- $\mathsf{EDX}' \leftarrow \mathsf{Delete}(\mathsf{EDX}, \mathsf{dtk})$*: takes as input an encrypted dictionary* EDX*, a delete token* dtk *and outputs an updated encrypted dictionary* EDX'*;*

The scheme is described in detail in Figure 2.

Efficiency analysis. Σ_C is optimal with respect to communication complexity: all tokens and responses are $O(1)$. All its algorithms are also $O(1)$ with the exception of Put and Get which are $O(\log \#\mathsf{MM}_C)$ and Delete which is $O(\#\mathsf{MM}_C[\ell])$.

3.3 Two-Dimensional Append Multi-Maps

So far we have seen that Ω_P encrypts the input multi-map MM with our stateless addressable scheme Σ_M to produce a main encrypted multi-map EMM_M and then encrypts a dictionary that will be used to avoid overwrites with a stateless (two-dimensional) immutable dictionary encryption scheme Σ_C. This design would be enough to achieve a stateless snapshot-secure semi-dynamic scheme but we also wish to support deletes. Augmenting the scheme to support deletes is not particularly challenging if all we require is correctness but handling deletes without affecting the scheme's query complexity is challenging. Roughly speaking, the problem is that deleting label/value pairs from the main encrypted multi-map EMM_M does not affect query efficiency. So for example, if the multi-map originally stored a pair $(\ell, \mathbf{v})$, where $\#\mathbf{v} = m$, and then values $(v_1, \ldots, v_{m-1})$ are deleted, querying the structure for ℓ would still be $O(m)$.

Let $F : \{0,1\}^k \times \{0,1\}^* \to \{0,1\}^*$ be a pseudo-random function, $\mathsf{SKE} = (\mathsf{Gen}, \mathsf{Enc}, \mathsf{Dec})$ be a symmetric encryption scheme. Consider the stateless response-revealing two-dimensional dictionary encryption scheme $\Sigma_C = (\mathsf{Init}, \mathsf{PutKey}, \mathsf{PutToken}, \mathsf{Put}, \mathsf{GetToken}, \mathsf{GetXToken}, \mathsf{GetXYToken}, \mathsf{Get}, \mathsf{DeleteToken}, \mathsf{Delete})$ defined as follows:

- $\mathsf{Init}\big(1^k\big)$:
  1. sample a key $K \xleftarrow{\$} \{0,1\}^k$;
  2. initialize a dictionary DX;
  3. output K and $\mathsf{EDX} := \mathsf{DX}$;
- $\mathsf{PutKey}\big(K, (\ell_x, \ell_y)\big)$:
  1. compute $K_x := F_K(\ell_x)$ and $K_\ell := F_{K_x}(\ell_y)$;
  2. output $\mathsf{pk} := K_\ell$;
- $\mathsf{PutToken}\big(\mathsf{pk}, v\big)$:
  1. parse pk as K_ℓ;
  2. compute $K_t := F_{K_\ell}(1)$ and $K_e := F_{K_\ell}(2)$;
  3. compute $\mathrm{ct} \leftarrow \mathsf{SKE.Enc}_{K_e}(v)$;
  4. output $\mathsf{ptk} := (K_t, \mathrm{ct})$.
- $\mathsf{Put}\big(\mathsf{EDX}, \mathsf{ptk}\big)$:
  1. parse EDX as DX and wtk as (K_t, ct);
  2. compute $a := \mathsf{Binary}(K_t, \mathsf{DX})$ and set $\mathsf{DX}[F_{K_t}(a+1)] := \mathrm{ct}$;
  3. output $\mathsf{EDX} := \mathsf{DX}$;
- $\mathsf{GetToken}\big(K, (\ell_x, \ell_y)\big)$: compute $K_x := F_K(\ell_x)$ and output $\mathsf{gtk} := F_{K_x}(\ell_y)$;
- $\mathsf{GetXToken}\big(K, \ell_x\big)$: output $\mathsf{gxtk} = F_K(\ell_x)$;
- $\mathsf{GetXYToken}\big(\mathsf{gxtk}, \ell_y\big)$:
  1. parse gxtk as K_x;
  2. output $\mathsf{gtk} := F_{K_x}(\ell_y)$;
- $\mathsf{Get}\big(\mathsf{EDX}, \mathsf{gtk}\big)$:
  1. parse EDX as DX and gtk as K_ℓ;
  2. compute $K_t := F_{K_\ell}(1)$ and $K_e := F_{K_\ell}(2)$;
  3. compute $a := \mathsf{Binary}(K_t, \mathsf{DX})$;
  4. if $a \neq 0$,
     (a) compute $\mathrm{ct} := \mathsf{DX}[F_{K_t}(a)]$;
     (b) set $v := \mathsf{SKE.Dec}(K_e, \mathrm{ct})$;
  5. else set $v := \perp$;
  6. output v;
- $\mathsf{DeleteToken}\big(K, (\ell_x, \ell_y)\big)$: compute $K_x := F_K(\ell_x)$ and output $\mathsf{dtk} := F_{K_x}(\ell_y)$;
- $\mathsf{Delete}\big(\mathsf{EDX}, \mathsf{dtk}\big)$:
  1. parse EDX as DX and dtk as K_ℓ;
  2. compute $K_t := F_{K_\ell}(1)$;
  3. set $a := 1$;
  4. while $\mathsf{DX}[F_{K_t}(a)] \neq \perp$,
     (a) set $\mathsf{DX}[F_{K_t}(a)] := \perp$
     (b) set $a := a + 1$;
  5. output $\mathsf{EDX} := \mathsf{DX}$.

Figure 2: Σ_C: a stateless two-dimensional dictionary encryption scheme.

To address this, Ω_P includes, in addition to EMM_M and EMM_C, an encrypted multi-map EMM_D that stores, for every label ℓ in EMM_M, the gaps/holes in ℓ's tuple $\mathbf{v}$. When the server executes a get for ℓ, it first queries EMM_D to retrieve ℓ's gaps $\mathbf{g}_\ell$ and uses that to only read from the existing locations in ℓ's tuple.

Other characteristics of Σ_D is that, like Σ_C, it is two-dimensional in order to provide support for concurrent Ω_P operations. It also supports two kinds of insert operations, append and put which work as follows:

- **append**: takes as input a two-dimensional label $\ell = (\ell_x, \ell_y)$ and a value v and appends v to ℓ's tuple $\mathbf{v}$.

- **put**: takes as input a two-dimensional label $\ell = (\ell_x, \ell_y)$ and a tuple of values $\mathbf{v}$ and inserts the pair $(\ell, \mathbf{v})$ into the multi-map if it does not exist already.

The reason Σ_D supports two kinds of inserts is because Ω_P needs to make different kinds of insertions at different times: it needs to append gaps to ℓ's tuple in EMM_D when deletes on ℓ are made; and it needs to put entire label/tuple pairs in EMM_D during compaction which we will discuss in the next section. In the following definition, we provide the syntax of Σ_D.

Definition 3.3. *A response-revealing stateless two-dimensional multi-map encryption scheme is a structured encryption scheme* $\Sigma_D = (\mathsf{Init}, \mathsf{AppendKey}, \mathsf{AppendToken}, \mathsf{Append}, \mathsf{PutKey}, \mathsf{PutToken}, \mathsf{Put}, \mathsf{GetToken}, \mathsf{GetXToken}, \mathsf{GetXYToken}, \mathsf{Get}, \mathsf{DeleteToken}, \mathsf{Delete})$ *consists of thirteen polynomial-time algorithms that work as follows:*

- $(K, \mathsf{EDX}) \leftarrow \mathsf{Init}(1^k)$*: takes as input a security parameter k and outputs a secret key K and an encrypted dictionary* EDX*;*

- $\mathsf{ak} \leftarrow \mathsf{AppendKey}(K, (\ell_x, \ell_y))$*: takes as input a key K, a two-dimensional label (ℓ_x, ℓ_y) and outputs an append key* ak*;*

- $\mathsf{atk} \leftarrow \mathsf{AppendToken}(\mathsf{ak}, v)$*: takes as input an append key* ak*, a value v and outputs an append token* atk*;*

- $\mathsf{EMM} \leftarrow \mathsf{Append}(\mathsf{EMM}, \mathsf{atk})$*: takes as input an encrypted multi-map* EMM*, an append token* atk *and outputs an updated encrypted multi-map* EMM'*;*

- $\mathsf{pk} \leftarrow \mathsf{PutKey}(K, (\ell_x, \ell_y))$*: takes as input a key K, a two-dimensional label (ℓ_x, ℓ_y) and outputs a put key* pk*;*

- $\mathsf{ptk} \leftarrow \mathsf{PutToken}(\mathsf{pk}, \mathbf{v})$*: takes as input a put key* pk*, a tuple of values $\mathbf{v}$ and outputs a put token* ptk*;*

- $\mathsf{EMM}' \leftarrow \mathsf{Put}(\mathsf{EMM}, \mathsf{ptk})$*: takes as input an encrypted multi-map* EMM*, a put token* ptk *and outputs an updated encrypted multi-map* EMM'*;*

- $\mathsf{gtk} \leftarrow \mathsf{GetToken}(K, (\ell_x, \ell_y))$: *takes as input a key K, a two-dimensional label (ℓ_x, ℓ_y) and outputs a get token* gtk*;*

- $\mathsf{gxtk} \leftarrow \mathsf{GetXToken}(K, \ell_x)$: *takes as input a key K, the x-component of a two-dimensional label ℓ_x and outputs a get-x token* gxtk*;*

- $\mathsf{gtk} \leftarrow \mathsf{GetXYToken}(\mathsf{gxtk}, \ell_y)$: *takes as input a get-x token* gxtk*, the y-component ℓ_y of a two-dimensional label and outputs a get token* gtk*;*

- $v \leftarrow \mathsf{Get}(\mathsf{EMM}, \mathsf{gtk})$: *takes as input an encrypted multi-map* EMM*, a get token* gtk *and outputs a tuple* $\mathbf{v}$*;*

- $\mathsf{dtk} \leftarrow \mathsf{DeleteToken}(K, (\ell_x, \ell_y))$: *takes as input a key K, a two-dimensional label (ℓ_x, ℓ_y) and outputs a delete token* dtk*;*

- $\mathsf{EDX}' \leftarrow \mathsf{Delete}(\mathsf{EMM}, \mathsf{dtk})$: *takes as input an encrypted multi-map* EMM*, a delete token* dtk *and outputs an updated encrypted multi-map* EMM'*;*

Design. The design of Σ_D is described in detail in Figures 3 and 4. Since it is similar to Σ_C we do not give a high-level overview.

Efficiency. Σ_D is optimal with communication complexity: all tokens are $O(1)$ and responses are $O(\mathsf{MM}_D[\ell])$. All of its algorithms are also optimal with the exception of Append which is $O(\log \#\mathsf{MM}_D)$.

3.4 Enumerable Sets

As discussed above, Ω_P encrypts the input multi-map with a stateless addressable multi-map encryption scheme Σ_M which results in a main encrypted multi-map EMM_M. Overwrite protection is then achieved by encrypting a dictionary that stores counters with a stateless two-dimensional dictionary encryption scheme Σ_C which results in an auxiliary structure EDX_C. Information about deletions is stored in encrypted multi-map EMM_D using a two-dimensional scheme Σ_D. This information is then used to speed up query operations. The design described so far achieves statelessness and correctness but has one major limitation: it is not space efficient. In fact, the space complexity of the three structures described so far is $O(\Sigma_\ell \#\mathsf{MM}[\ell] + \#\mathsf{puts} + \#\mathsf{deletes})$, where $\Sigma_\ell \#\mathsf{MM}[\ell]$ is the size of the input multi-map and $\#\mathsf{puts}$ and $\#\mathsf{deletes}$ are the the total number of put and erase operations made on the input multi-map. Note that this depends on the total number of puts and deletes *ever made* and not on the size of the input multi-map. To address this, Ω_P uses a process called *compaction* to remove stale data from EMM_C and EMM_D and bring the size down to $O(\Sigma_\ell \#\mathsf{MM}[\ell])$.

The compaction process is executed by the server which means it needs access to information stored in both EMM_C and EMM_D. More precisely, it needs the ability to query these structures, to delete certain pairs and to add new ones. To enable this, the client generates get, put and delete tokens for EMM_C and EMM_D whenever it executes a put or erase for Ω_P.

Let $F : \{0,1\}^k \times \{0,1\}^* \rightarrow \{0,1\}^*$ be a pseudo-random function, $\mathsf{SKE} = (\mathsf{Gen}, \mathsf{Enc}, \mathsf{Dec})$ be a symmetric encryption scheme. Consider the stateless response-revealing two-dimensional multi-map encryption scheme $\Sigma_D = (\mathsf{Init}, \mathsf{AppendKey}, \mathsf{AppendToken}, \mathsf{Append}, \mathsf{PutKey}, \mathsf{PutToken}, \mathsf{Put}, \mathsf{GetToken}, \mathsf{GetXToken}, \mathsf{GetXYToken}, \mathsf{Get}, \mathsf{DeleteToken}, \mathsf{Delete})$ defined as follows:

- $\mathsf{Init}\big(1^k\big)$:
  1. sample a key $K \stackrel{\$}{\leftarrow} \{0,1\}^k$;
  2. initialize an empty dictionary DX;
  3. output K and $\mathsf{EMM} := \mathsf{DX}$;
- $\mathsf{AppendKey}\big(K, (\ell_x, \ell_y)\big)$: compute $K_x := F_K(\ell_x)$ and output $\mathsf{ak} := F_{K_x}(\ell_y)$;
- $\mathsf{AppendToken}\big(\mathsf{ak}, v\big)$:
  1. parse ak as K_ℓ;
  2. compute $K_t := F_{K_\ell}(1)$ and $K_e := F_{K_\ell}(2)$;
  3. compute $\mathrm{ct} \leftarrow \mathsf{SKE.Enc}(K_e, v)$;
  4. output $\mathsf{atk} := (K_t, \mathrm{ct})$;
- $\mathsf{Append}\big(\mathsf{EMM}, \mathsf{atk}\big)$:
  1. parse EMM as DX and atk as (K_t, ct);
  2. compute $a \leftarrow \mathsf{Binary}(K_t, \mathsf{DX})$;
  3. set $\mathsf{DX}[F_{K_t}(a+1)] := \mathrm{ct}$;
  4. output $\mathsf{EMM} := \mathsf{DX}$;
- $\mathsf{PutKey}\big(K, (\ell_x, \ell_y)\big)$:
  1. compute $K_x := F_K(\ell_x)$ and $K_\ell := F_{K_x}(\ell_y)$;
  2. output $\mathsf{pk} := K_\ell$;
- $\mathsf{PutToken}\big(\mathsf{pk}, \mathbf{v}\big)$:
  1. parse pk as K_ℓ;
  2. compute $K_t := F_{K_\ell}(1)$ and $K_e := F_{K_\ell}(2)$;
  3. for all $1 \leq i \leq m$, compute $\mathrm{ct}_i \leftarrow \mathsf{SKE.Enc}(K_e, v_i)$;
  4. set $\mathbf{ct} := (\mathrm{ct}_1, \ldots, \mathrm{ct}_m)$;
  5. output $\mathsf{ptk} := (K_t, \mathbf{ct})$;
- $\mathsf{Put}\big(\mathsf{EMM}, \mathsf{ptk}\big)$:
  1. parse EMM as DX and ptk as $(K_t, \mathbf{ct})$;
  2. for all $1 \leq i \leq m$, set $\mathsf{DX}[F_{K_t}(i)] := \mathrm{ct}_i$;
  3. output $\mathsf{EMM} := \mathsf{DX}$;

Figure 3: Σ_D: a stateless two-dimensional multi-map encryption scheme (part 1).

- $\mathsf{GetToken}\big(K, (\ell_x, \ell_y)\big)$: compute $K_x := F_K(\ell_x)$ and output $\mathsf{gtk} := F_{K_x}(\ell_y)$;
- $\mathsf{GetXToken}\big(K, \ell_x\big)$: output $\mathsf{gxtk} := F_K(\ell_x)$;
- $\mathsf{GetXYToken}\big(\mathsf{gxtk}, \ell_y\big)$:
  1. parse gxtk as K_x;
  2. output $\mathsf{gtk} := F_{K_x}(\ell_y)$;
- $\mathsf{Get}\big(\mathsf{EMM}, \mathsf{gtk}\big)$:
  1. parse EMM as DX and gtk as K_ℓ;
  2. compute $K_t := F_{K_\ell}(1)$ and $K_e := F_{K_\ell}(2)$;
  3. initialize an empty sequence $\mathbf{v}$ and set $i := 1$;
  4. while $\mathsf{DX}[F_{K_t}(i)] \neq \perp$,
     (a) compute $\mathrm{ct} := \mathsf{DX}\big[F_{K_t}(i)\big]$ and $v \leftarrow \mathsf{SKE.Dec}(K_e, \mathrm{ct})$
     (b) set $\mathbf{v} := (\mathbf{v}, v)$ and $i = i + 1$;
  5. output $\mathbf{v}$.
- $\mathsf{DeleteToken}\big(K, (\ell_x, \ell_y)\big)$: compute $K_x := F_K(\ell_x)$ and output $\mathsf{dtk} = F_{K_x}(\ell_y)$;
- $\mathsf{Delete}\big(\mathsf{EMM}, \mathsf{dtk}\big)$:
  1. parse EMM as DX and dtk as K_ℓ;
  2. compute $K_t = F_{K_\ell}(1)$ and set $i := 1$;
  3. while $\mathsf{DX}[F_{K_t}(i)] \neq \perp$, set $\mathsf{DX}[F_{K_t}(i)]) := \perp$ and $i := i + 1$;
  4. output $\mathsf{EMM} = \mathsf{DX}$.

Figure 4: Σ_D: a stateless two-dimensional multi-map encryption scheme (part 2).

These tokens are stored in an auxiliary encrypted set structure EST_P and used at compaction time. The encrypted set structure is the simplest of our auxiliary structures and supports the following operations:

- **insert**: takes as input an element and stores it in the set;
- **enum**: enumerates all the elements in the set.

We now describe the syntax of Σ_P.

Definition 3.4. *A response-revealing stateless set encryption scheme is a structured encryption scheme* $\Sigma_P = (\mathsf{Init}, \mathsf{InsertToken}, \mathsf{Insert}, \mathsf{Enum})$ *consists of four polynomial-time algorithms that work as follows:*

- $(K, \mathsf{EST}) \leftarrow \mathsf{Init}(1^k)$*: takes as input a security parameter* k *and outputs a secret key* K *and an encrypted set* EST*;*
- $\mathsf{itk} \leftarrow \mathsf{InsertToken}(K, e)$*: takes as input a key* K*, an element* e *and outputs an insert token* itk*;*
- $\mathsf{EST}' \leftarrow \mathsf{Insert}(\mathsf{EST}, \mathsf{itk})$*: takes as input an encrypted set* EST *and an insert token* itk *and outputs an updated encrypted set* EST'*;*
- $P \leftarrow \mathsf{Enum}(K, \mathsf{EST})$*: takes as input a key* K*, an encrypted set* EST *and outputs a set* P*.*

Let $\mathsf{SKE} = (\mathsf{Gen}, \mathsf{Enc}, \mathsf{Dec})$ be a symmetric-key encryption scheme. Consider the stateless response-revealing set encryption scheme $\Sigma_P = (\mathsf{Init}, \mathsf{InsertToken}, \mathsf{Insert}, \mathsf{Enum})$ defined as follows:

- $\mathsf{Init}(1^k)$:
  1. sample a key $K \xleftarrow{\$} \{0,1\}^k$;
  2. initialize an empty set SET;
  3. output K and $\mathsf{EST} = \mathsf{SET}$.
- $\mathsf{InsertToken}(K, v)$:
  1. compute $\mathrm{ct} = \mathsf{Enc}_K(v)$;
  2. output $\mathsf{itk} = \mathrm{ct}$.
- $\mathsf{Insert}(\mathsf{EST}, \mathsf{itk})$:
  1. parse EST as SET;
  2. add itk to SET;
  3. output $\mathsf{EST}' = \mathsf{SET}$.
- $\mathsf{Enum}(K, \mathsf{EST})$:
  1. parse EST as $\mathsf{SET}^{\mathsf{old}}$;
  2. initialize an empty set SET and set EST as SET;
  3. initialize a set Result;
  4. for all $\mathrm{ct} \in \mathsf{SET}$, add $\mathsf{Dec}_K(\mathrm{ct})$ to Result;
  5. output Result.

Figure 5: Σ_P: a stateless enumerable encrypted set scheme.

Overview. The scheme Σ_P is described in detail in Figure 5. The scheme is simple. An encrypted set consists of symmetrically-encrypted elements, an insert token consists of the encryption of the inserted element and enumeration consists of decrypting all the ciphertexts in the encrypted set and listing the plaintexts.

4 A Stateless Multi-Map Encryption Scheme

The high level structure of Ω_P was described in the previous sub-sections to motivate the design of its building blocks so here we mainly provide a high-level overview. Recall that the scheme makes use of an addressable multi-map encryption scheme Σ_M, an immutable two-dimensional dictionary encryption scheme Σ_C, a two-dimensional append multi-map encryption scheme and an enumerable set encryption scheme Σ_P. It consists of ten algorithms $\Omega_P = (\mathsf{Init}, \mathsf{PutToken}, \mathsf{Put}, \mathsf{GetToken}, \mathsf{Get}, \mathsf{DeleteToken}, \mathsf{Delete}, \mathsf{CompactionToken}, \mathsf{Compaction}, \mathsf{Resolve})$ which are all described in Figures 8, 9 and 10. Init initializes the main encrypted multi-map EMM_M together with three auxiliary structures EDX_C, EMM_D and EST_P.

Put operations. $\mathsf{PutToken}$ for a label ℓ and tuple $\mathbf{v}$ first determines if ℓ is a high contention label. If so, it creates a two-dimensional label $\ell' = (\ell, u)$, where $u \xleftarrow{\$} \{1, \ldots, p\}$. If not, it creates a two-dimensional label $\ell' = (\ell, 0)$. It then creates a put token ptk which consists of: (1) an EMM_M write token wtk_M for $(\ell', \mathbf{v})$; (2) an EDX_C get token gtk_C for ℓ'; (3) an EDX_C

- $\mathsf{BinSearch}(K, \mathsf{DX})$:
  1. let ρ to the size estimate of DX;
  2. while $\mathsf{DX}[F_K(\rho)] \neq \perp$, set $\rho := 2\rho$;
  3. set $i := 0$, $\mathsf{median} := 0$, $\mathsf{min} := 1$ and $\mathsf{max} := \rho$;
  4. for $i = 1$ to $\lceil \log(\rho) \rceil$,
     (a) set $\mathsf{median} := \lceil (\mathsf{max} - \mathsf{min})/2 \rceil + \mathsf{min}$;
     (b) compute $\mathsf{tag} := F_K(\mathsf{median})$;
     (c) if $\mathsf{DX}[\mathsf{tag}] \neq \perp$,
        i. set $\mathsf{min} := \mathsf{median}$;
        ii. if $i = \lceil \log(\rho) \rceil$, then set $i := \mathsf{min}$;
     (d) otherwise if $\mathsf{DX}[\mathsf{tag}] = \perp$,
        i. set $\mathsf{max} := \mathsf{median}$;
        ii. if $i = \lceil \log(\rho) \rceil$ and $\mathsf{DX}[F_K(\mathsf{min})] \neq \perp$, then set $i := \mathsf{min}$;
  5. output i.

Figure 6: The binary search subroutine.

- $\mathsf{Merge}(\mathbf{g})$:
  1. parse $\mathbf{g}$ as $((a_1, b_1), \cdots, (a_n, b_n), d_1, \cdots, d_m)$;
  2. set $S_1 := \{(a_1, b_1), \cdots, (a_n, b_n)\}$ and $S_2 = \{d_1, \cdots, d_m\}$;
  3. initialize flag to `true`;
  4. while flag is `true`,
     (a) set $\mathsf{count} := \#S_2$;
     (b) for $i = 1$ to $\#S_2$,
        i. if there exists $j \in \{1, \cdots, \#S_1\}$ such that $d_i = b_j + 1$,
           A. set $b_j := d_i$;
           B. if $a_{j+1} = b_j$, then remove $[a_j, b_j]$ and $[a_{j+1}, b_{j+1}]$ from S_2, and add $[a_j, b_{j+1}]$ to S_1;
           C. remove d_i from S_2;
     (c) if $\#S_2 = \mathsf{count}$, set flag to false;
  5. add S_1 and S_2 to $\mathbf{g}$;
  6. output $\mathbf{g}$.

Figure 7: The merge subroutine.

put key for ℓ'; (4) an EST_P insert token itk_P; and (5) the size of $\mathbf{v}$. The EST_P insert token itk_P is for an element that is the concatenation of EDX_C get and delete tokens for ℓ', a put key for ℓ' and EMM_D get and delete tokens for ℓ'. These elements will be stored in EST_P and used later during compaction.

Given a put token $\mathsf{ptk} = (\mathsf{wtk}_M, \mathsf{gtk}_C, \mathrm{pk}_C, \mathsf{itk}_P, m)$, the Put algorithm uses gtk_C to retrieve a counter count_ℓ from EDX_C that represents the number of previously used addresses in the tuple of ℓ'. The server uses this counter, together with the write token wtk, to write to EMM_M without overwriting. Specifically, it executes $\Sigma_M.\mathsf{Write}$ with wtk_M and addresses $\mathbf{a} = (\mathsf{count}_\ell, \ldots, \mathsf{count}_\ell + m - 1)$. The server then updates the counter of ℓ' in EDX_C by generating a put token ptk_C with the put key pk_C and value $\mathsf{count} + m$ and applying ptk_C to EDX_C. Finally, it updates the encrypted set EST_P with itk_P.

Get operations. $\mathsf{GetToken}$ produces a get token gtk for a label ℓ that consists of: (1) a read x-token rxtk_M for ℓ; (2) a get-x token gxtk_C for ℓ; (3) a get-x token gxtk_D for ℓ; and (4) a flag that describes whether the label is a high contention label or not. Given a get token $\mathsf{gtk} = (\mathsf{rxtk}_M, \mathsf{gxtk}_C, \mathsf{gxtk}_D, \mathsf{cont})$, the Get algorithm first uses the flag to determine if the label is a high contention label. If so, the server uses gxtk_C with values $\{1, \ldots, p\}$ to generate p get tokens $(\mathsf{gtk}_{C,1}, \ldots, \mathsf{gtk}_{C,p})$, where $\mathsf{gtk}_{C,i}$ is for the two-dimensional label (ℓ, i). It then queries EDX_C with these tokens to retrieve p counters $(\mathsf{count}_1, \ldots, \mathsf{count}_p)$ from EDX_C for the two-dimensional labels $((\ell, 1), \ldots, (\ell, p))$. Similarly, for all $1 \leq i \leq p$, if $\mathsf{count}_i > 0$, it uses gxtk_D with $\{i\}$ to generate a get token $\mathsf{gtk}_{D,i}$ and uses it to recover the gaps $\mathbf{g}_i$ for the two-dimensional label (ℓ, i). In addition, it uses rxtk_M to generate a read token $\mathsf{rtk}_{M,i}$ for the two-dimensional label (ℓ, i). It then uses the counters and gaps to generate the sequence of used addresses it needs to read from EMM_M. If the label is not a high contention label, the server does the above with a single two-dimensional label $(\ell, 0)$.

Erase operations. $\mathsf{EraseToken}$ produces an erase token etk for a two-dimensional label (ℓ, u) and address a that consists of: (1) an erase token etk_M for $\ell \| u$; (2) a get token gtk_D for (ℓ, u); (3) an append token atk_D for (ℓ, u); (4) the address a to erase; and (5) an insert token itk_P for a set of compaction-time tokens, i.e., a set of EDX_C and EMM_D tokens that will be needed during compaction. The Erase algorithm uses itk_P to insert the compaction-time tokens in the encrypted set EST_P and uses etk_M to erase the element at address a from ℓ's tuple in EMM_M.

Compaction. $\mathsf{CompactionToken}$ simply outputs the key K_P as a compaction token. At a high level, for every ℓ in EMM_M, the compaction algorithm first retrieves ℓ's counter from EDX_C and ℓ's gaps from EMM_D. It then deletes everything related to ℓ from both EDX_C and EMM_D which includes "stale" data like old counter values in EDX_D. Note that this is the step that reclaims wasted space. It then re-inserts ℓ's counter in EDX_C, merges ℓ's gaps and re-inserts them in EMM_D. By *merging* here, we mean that ℓ's gaps are re-encoded into a more compact representation. For example, if ℓ's gaps includes four holes $i, i+1, i+2, i+3$

then this gets encoded as a single gap $[i, 3]$. A detailed description of this merge process is given in Figure 7.

More precisely, the compaction algorithm enumerates EST_P which returns a set P of elements of the form $\mathsf{gtk}_C \| \mathsf{dtk}_C \| \mathrm{pk}_C \| \mathsf{gtk}_D \| \mathsf{dtk}_D$. Each one of these elements encodes a set of tokens needed to compact EDX_C and EMM_D for some label ℓ. For each of the elements in P, the algorithm does the following. It uses gtk_C to retrieve ℓ's counter from EDX_C and gtk_D to retrieve ℓ' gaps $\mathbf{g}$ from EMM_D. It then merges $\mathbf{g}$ into a new sequence $\mathbf{g}'$. ℓ is then deleted from EDX_C and EMM_D using dtk_C and dtk_D, respectively. If $\mathbf{g}' = \{1, \ldots, \mathsf{count}\}$ then every element of ℓ's tuple has been erased and nothing else needs to be done. If $\mathbf{g}' \neq \{1, \ldots, \mathsf{count}\}$, however, the algorithm: (1) uses pk_C to generate a put token for ℓ's counter and inserts it into EDX_C; and (2) uses pk_D to generate a put token for $\mathbf{g}'$ and inserts it into EMM_D.

Note that during compaction, if the data related to a particular label ℓ is being compacted then get, put and delete operations can still occur simultaneously on any label $\ell' \neq \ell$.

Resolve. The $\mathsf{Resolve}$ algorithm simply executes Σ's resolve algorithm and returns its output.

5 A Stateless Range Multi-Map Encryption Scheme

In this section, we describe the range multi-map encryption scheme $\Omega_R = (\mathsf{Init}, \mathsf{PutToken}, \mathsf{Put}, \mathsf{RangeToken}, \mathsf{Range}, \mathsf{EraseToken}, \mathsf{Erase}, \mathsf{CompactionToken}, \mathsf{Compaction}, \mathsf{Resolve})$ used by OST_1. The scheme is a result of the ERX framework from [KKM21] which makes use of a multi-map encryption scheme Σ and a range hypergraph H equipped with efficient algorithms Edges_H and $\mathsf{Mincover}_H$. Here, we instantiate Σ with our stateless multi-map encryption scheme Ω_P and H with a new hypergraph we refer to as the *sparse partition hypergraph*. The details of our construction are provided in Figure 11 and the sparse partition hypergraph is described in Appendix A. The scheme works as follows.

Init. The Init algorithm takes as input a security parameter k. It uses $\Omega_P.\mathsf{Init}$ to output a key K and an encrypted multi-map EMM. It outputs the key K as its own key, and the encrypted multi-map as the encrypted *range* multi-map ERMM.

Range token. The $\mathsf{RangeToken}$ algorithm takes as input a secret key K and a range query $r = [a, b]$. It uses $\mathsf{Mincover}_H$ to compute the minimum cover $\mathbf{C}_r$ of the range query and, for each edge $e \in \mathbf{C}_r$, computes a get token gtk_e using $\Omega_P.\mathsf{GetToken}$. It then outputs a range token $\mathsf{rtk} = (\mathsf{gtk}_e)_{e \in \mathbf{C}_r}$.

Ranges. The Range algorithm takes as input an encrypted range multi-map $\mathsf{ERMM} = \mathsf{EMM}$ and a range token rtk parsed as $(\mathsf{tk}_e)_{e \in \mathbf{C}_r}$. It then uses $\Omega_P.\mathsf{Get}$ to query EMM on each of the sub-tokens in rtk and outputs the union of the results.

Let $\Sigma_M = (\mathsf{Init}, \mathsf{WriteToken}, \mathsf{Write}, \mathsf{ReadToken}, \mathsf{Read}, \mathsf{EraseToken}, \mathsf{Erase}, \mathsf{Resolve})$, $\Sigma_C = (\mathsf{Init}, \mathsf{PutToken}, \mathsf{Put}, \mathsf{GetToken}, \mathsf{GetXToken}, \mathsf{GetXYToken}, \mathsf{Get}, \mathsf{DeleteToken}, \mathsf{Delete})$ and $\Sigma_D = (\mathsf{Init}, \mathsf{AppendToken}, \mathsf{Append}, \mathsf{PutKey}, \mathsf{PutToken}, \mathsf{Put}, \mathsf{GetToken}, \mathsf{Get}, \mathsf{DeleteToken}, \mathsf{Delete})$ be the schemes described in Figures 2, 3 and 4, and 5, respectively. Let $\mathbb{L}_C$ denote the set of labels in in the multi-map with a high degree of contention and let $p \in \mathbb{N}$ be the maximum number of partitions. Consider the stateless response-hiding multi-map encryption scheme $\Omega_P = (\mathsf{Init}, \mathsf{PutToken}, \mathsf{Put}, \mathsf{GetToken}, \mathsf{Get}, \mathsf{DeleteToken}, \mathsf{Delete}, \mathsf{CompactionToken}, \mathsf{Compaction}, \mathsf{Resolve})$ defined as follows:

- $\mathsf{Init}\big(1^k\big)$:
  1. compute $(K_M, \mathsf{EMM}_M) \leftarrow \Sigma_M.\mathsf{Init}(1^k)$;
  2. compute $(K_C, \mathsf{EDX}_C) \leftarrow \Sigma_C.\mathsf{Init}(1^k)$;
  3. compute $(K_D, \mathsf{EMM}_D) \leftarrow \Sigma_D.\mathsf{Init}(1^k)$;
  4. compute $(K_P, \mathsf{EST}_P) \leftarrow \Sigma_P.\mathsf{Init}(1^k)$;
  5. output $K := (K_M, K_C, K_D, K_P)$ and $\mathsf{EMM} = (\mathsf{EMM}_M, \mathsf{EDX}_C, \mathsf{EMM}_D, \mathsf{EST}_P)$;
- $\mathsf{PutToken}\big(K, \ell, \mathbf{v}\big)$:
  1. parse K as (K_M, K_C, K_D);
  2. if $\ell \in \mathbb{L}_C$, sample $u \xleftarrow{\$} \{1, \cdots, p\}$, otherwise set $u := 0$;
  3. compute $\mathsf{wtk}_M \leftarrow \Sigma_M.\mathsf{WriteToken}(K_M, (\ell, u), \mathbf{v})$;
  4. compute $\mathsf{gtk}_C \leftarrow \Sigma_C.\mathsf{GetToken}(K_C, (\ell, u))$;
  5. compute $\mathsf{dtk}_C \leftarrow \Sigma_C.\mathsf{DeleteToken}(K_C, (\ell, u))$;
  6. compute $\mathsf{pk}_C \leftarrow \Sigma_C.\mathsf{PutKey}(K_C, (\ell, u))$;
  7. compute $\mathsf{gtk}_D \leftarrow \Sigma_D.\mathsf{GetToken}(K_D, (\ell, u))$;
  8. compute $\mathsf{dtk}_D \leftarrow \Sigma_D.\mathsf{DeleteToken}(K_D, (\ell, u))$;
  9. compute $\mathsf{itk}_P \leftarrow \Sigma_P.\mathsf{InsertToken}(K_P, \mathsf{gtk}_C \| \mathsf{dtk}_C \| \mathsf{pk}_C \| \mathsf{gtk}_D \| \mathsf{dtk}_D)$;
  10. output $\mathsf{ptk} := (\mathsf{wtk}_M, \mathsf{gtk}_C, \mathsf{pk}_C, \mathsf{itk}_P, \#\mathbf{v})$;
- $\mathsf{Put}\big(\mathsf{EMM}, \mathsf{ptk}\big)$:
  1. parse EMM as $(\mathsf{EMM}_M, \mathsf{EDX}_C, \mathsf{EMM}_D, \mathsf{EST}_P)$;
  2. parse ptk as $(\mathsf{wtk}_M, \mathsf{gtk}_C, \mathsf{pk}_C, \mathsf{itk}_P, m)$;
  3. compute $c \leftarrow \Sigma_C.\mathsf{Get}(\mathsf{EDX}_C, \mathsf{gtk}_C)$;
  4. if $c \neq \perp$ set $\mathsf{count} := c + 1$ else set $\mathsf{count} := 1$;
  5. compute $\mathsf{ptk}_C \leftarrow \Sigma_C.\mathsf{PutToken}(\mathsf{pk}_C, \mathsf{count} + m)$;
  6. set $\mathsf{EDX}_C \leftarrow \Sigma_C.\mathsf{Put}(\mathsf{EDX}_C, \mathsf{ptk}_C)$;
  7. compute $\mathsf{EMM}_M \leftarrow \Sigma_M.\mathsf{Write}(\mathsf{EMM}_M, \mathsf{wtk}_M, \{\mathsf{count}, \ldots, \mathsf{count} + m - 1\})$;
  8. set $\mathsf{EST}_P \leftarrow \Sigma_P.\mathsf{Append}(\mathsf{EST}_P, \mathsf{itk}_P)$;
  9. output $\mathsf{EMM} := (\mathsf{EMM}_M, \mathsf{EDX}_C, \mathsf{EMM}_D, \mathsf{EST}_P)$;
- $\mathsf{GetToken}\big(K, \ell\big)$:
  1. parse K as (K_M, K_C, K_D);
  2. compute $\mathsf{rtk}_M \leftarrow \Sigma_M.\mathsf{ReadXToken}(K_M, \ell)$;
  3. compute $\mathsf{gxtk}_C \leftarrow \Sigma_C.\mathsf{GetXToken}(K_C, \ell)$;
  4. compute $\mathsf{gxtk}_D \leftarrow \Sigma_D.\mathsf{GetXToken}(K_D, \ell)$;
  5. if $\ell \in \mathbb{L}_C$, set $\mathsf{cont} := \mathtt{true}$ else set $\mathsf{cont} := \mathsf{false}$;;
  6. output $\mathsf{gtk} = \big(\mathsf{rxtk}_M, \mathsf{gxtk}_C, \mathsf{gxtk}_D, \mathsf{cont}\big)$;

Figure 8: Ω_P: a stateless multi-map encryption scheme (part 1).

- $\mathsf{Get}\big(\mathsf{EMM}, \mathsf{gtk}\big)$:
  1. parse EMM as $(\mathsf{EMM}_M, \mathsf{EDX}_C, \mathsf{EMM}_D, \mathsf{EST}_P)$ and gtk as $(\mathsf{rtk}_M, \mathsf{gxtk}_C, \mathsf{gxtk}_D, \mathsf{cont})$;
  2. initialize an empty sequence $\mathbf{ct}$;
  3. if $\mathsf{cont} = \mathtt{true}$ set $U := \{1, \ldots, p\}$ else set $U := \{0\}$;
  4. for all $u \in U$,
     (a) compute $\mathsf{gtk}_C \leftarrow \Sigma_C.\mathsf{GetXYToken}(\mathsf{gxtk}_C, u)$;
     (b) compute $\mathsf{count}_u \leftarrow \Sigma_C.\mathsf{Get}(\mathsf{EDX}_C, \mathsf{gtk}_C)$;
     (c) if $\mathsf{count}_u \neq \perp$,
        i. compute $\mathsf{gtk}_D \leftarrow \Sigma_D.\mathsf{GetXYToken}(\mathsf{gxtk}_D, u)$;
        ii. compute $\mathbf{g}_u \leftarrow \Sigma_D.\mathsf{Get}(\mathsf{EMM}_D, \mathsf{gtk}_D)$;
        iii. compute $\mathsf{rtk}_M \leftarrow \Sigma_M.\mathsf{ReadXYToken}(\mathsf{rxtk}_M, u)$;
        iv. compute $\mathbf{ct}_u \leftarrow \Sigma_M.\mathsf{Read}(\mathsf{EMM}_M, \mathsf{rtk}_M, \{1, \ldots, \mathsf{count}_u\} \setminus \mathbf{g}_u)$;
        v. set $\mathbf{ct} = (\mathbf{ct}, \mathbf{ct}_u)$;
  5. output $\mathbf{ct}$;
- $\mathsf{EraseToken}\big(K, \ell, u, a\big)$:
  1. parse K as (K_M, K_C, K_D);
  2. compute $\mathsf{etk}_M \leftarrow \Sigma_M.\mathsf{EraseToken}(K_M, (\ell, u))$;
  3. compute $\mathsf{gtk}_D \leftarrow \Sigma_D.\mathsf{GetToken}(K_D, (\ell, u))$;
  4. compute $\mathsf{atk}_D \leftarrow \Sigma_D.\mathsf{AppendToken}(K_D, (\ell, u), a)$;
  5. compute $\mathsf{dtk}_D \leftarrow \Sigma_D.\mathsf{DeleteToken}(K_D, (\ell, u))$;
  6. compute $\mathsf{gtk}_C \leftarrow \Sigma_C.\mathsf{GetToken}(K_C, (\ell, u))$;
  7. compute $\mathsf{dtk}_C \leftarrow \Sigma_C.\mathsf{DeleteToken}(K_C, (\ell, u))$;
  8. compute $\mathsf{pk}_C \leftarrow \Sigma_C.\mathsf{PutKey}(K_C, (\ell, u))$;
  9. compute $\mathsf{itk}_P \leftarrow \Sigma_P.\mathsf{InsertToken}(K_P, \mathsf{gtk}_C \| \mathsf{dtk}_C \| \mathsf{pk}_C \| \mathsf{gtk}_D \| \mathsf{dtk}_D)$
  10. output $\mathsf{etk} := (\mathsf{etk}_M, \mathsf{gtk}_D, \mathsf{atk}_D, a, \mathsf{itk}_P)$;
- $\mathsf{Erase}\big(\mathsf{EMM}, \mathsf{etk}\big)$:
  1. parse EMM as $(\mathsf{EMM}_M, \mathsf{EDX}_C, \mathsf{EDX}_C, \mathsf{EST}_P)$;
  2. parse etk as $(\mathsf{etk}_M, \mathsf{gtk}_D, \mathsf{atk}_D, a, \mathsf{itk}_P)$;
  3. compute $\mathsf{EST}_P \leftarrow \Sigma_P.\mathsf{Append}(\mathsf{EST}_P, \mathsf{itk}_P)$;
  4. compute $\mathsf{EMM}_M \leftarrow \Sigma_M.\mathsf{Erase}(\mathsf{EMM}_M, \mathsf{etk}_M, a)$;
  5. compute $\mathsf{EMM}_D \leftarrow \Sigma_D.\mathsf{Append}(\mathsf{EMM}_D, \mathsf{atk}_D)$;
  6. output EMM_D;

Figure 9: Ω_P: a stateless multi-map encryption scheme (part 2).

- $\mathsf{CompactionToken}(K)$:
  1. parse K as (K_M, K_C, K_D, K_P);
  2. output $\mathsf{ctk} = K_P$;
- $\mathsf{Compaction}(\mathsf{EMM}, \mathsf{ctk})$:
  1. parse EMM as $(\mathsf{EMM}_M, \mathsf{EDX}_C, \mathsf{EMM}_D, \mathsf{EST}_P)$;
  2. parse ctk as K_P;
  3. compute $P \leftarrow \Sigma_P.\mathsf{Enum}(K_P, \mathsf{EST}_P)$,
  4. for all $(\mathsf{gtk}_C \| \mathsf{dtk}_C \| \mathsf{pk}_C \| \mathsf{gtk}_D \| \mathsf{dtk}_D) \in P$;
     (a) compute $\mathsf{count} \leftarrow \Sigma_C.\mathsf{Get}(\mathsf{EDX}_C, \mathsf{gtk}_C)$;
     (b) compute $\mathbf{g} \leftarrow \Sigma_D.\mathsf{Get}(\mathsf{EMM}_D, \mathsf{gtk}_D)$;
     (c) compute $\mathbf{g}' \leftarrow \mathsf{Merge}(\mathbf{g})$;
     (d) compute $\mathsf{EDX}_C \leftarrow \Sigma_C.\mathsf{Delete}(\mathsf{EDX}_C, \mathsf{dtk}_C)$;
     (e) if $\mathbf{g}' \neq \mathbf{g}$, compute $\mathsf{EMM}_D \leftarrow \Sigma_D.\mathsf{Delete}(\mathsf{EMM}_D, \mathsf{dtk}_D)$;
     (f) if $\mathbf{g}' \neq \{1, \ldots, \mathsf{count}\}$
        i. compute $\mathsf{ptk}_C \leftarrow \Sigma_C.\mathsf{PutToken}(\mathsf{pk}_C, \mathsf{count})$;
        ii. compute $\mathsf{EDX}_C \leftarrow \Sigma_C.\mathsf{Put}(\mathsf{EDX}_C, \mathsf{ptk}_C)$;
        iii. compute $\mathsf{ptk}_D \leftarrow \Sigma_D.\mathsf{PutToken}(\mathsf{pk}_D, \mathbf{g}')$;
        iv. if $\mathbf{g}' \neq \mathbf{g}$, compute $\mathsf{EMM}_D \leftarrow \Sigma_D.\mathsf{Put}(\mathsf{EMM}_D, \mathsf{ptk}_D)$;
  5. output $\mathsf{EMM} = (\mathsf{EMM}_M, \mathsf{EDX}_C, \mathsf{EMM}_D, \mathsf{EST}_P)$;
- $\mathsf{Resolve}(K, \mathbf{ct})$: parse K as (K_M, K_C, K_D, K_P) and output $\mathbf{v} \leftarrow \Sigma_M.\mathsf{Resolve}(K_M, \mathbf{ct})$.

Figure 10: Ω_P: a stateless multi-map encryption scheme (part 3).

Put token. The $\mathsf{PutToken}$ algorithm takes as input a secret key K and a new label/tuple pair $(\ell, \mathbf{v})$. It first uses Edges_H to find the set of edges $\mathbf{E}_\ell$ in H that contain ℓ. For all $e \in \mathbf{E}_\ell$, it uses $\Omega_P.\mathsf{PutToken}$ to create a put token ptk'_e. It then outputs a put token $\mathsf{ptk} = (\mathsf{ptk}'_e)_{e \in \mathbf{E}_\ell}$.

Put. The Put algorithm takes as input the encrypted range multi-map $\mathsf{ERMM} = \mathsf{EMM}$ and a put token ptk. It first parses the put token as a tuple of sub-tokens $(\mathsf{ptk}'_e)_{e \in \mathbf{E}_\ell}$. It then uses $\Omega_P.\mathsf{Put}$ to apply each of the sub-tokens to the encrypted multi-map. Finally, it outputs the updated encrypted multi-map.

Erase token. The $\mathsf{EraseToken}$ algorithm takes as input a secret key K, a label ℓ, and a set of counters $(c_1, \cdots, c_n)$. It first uses Edges_H to find the set of edges $\mathbf{E}_\ell = \{e_1, \cdots, e_n\}$ in H that contain ℓ. For all $i \in [n]$, it uses $\Omega_P.\mathsf{EraseToken}$ to create an erase token etk_i. It then outputs an erase token $\mathsf{etk} = (\mathsf{etk}_i)_{i \in [n]}$.

Erase. The Erase algorithm takes as input the encrypted range multi-map $\mathsf{ERMM} = \mathsf{EMM}$ and the erase token etk. It first parses the erase token as a tuple of sub-tokens $(\mathsf{etk}_i)_{i \in [n]}$. It then uses $\Omega_P.\mathsf{Erase}$ to apply each of the sub-tokens to the encrypted multi-map. Finally, it outputs the updated encrypted multi-map.

Compaction token. The $\mathsf{CompactionToken}$ takes as input a secret key K. It simply outputs the compaction token ctk by using $\Omega_P.\mathsf{CompactionToken}$.

Compaction. The $\mathsf{Compaction}$ algorithm takes as input the encrypted range multi-map $\mathsf{ERMM} = \mathsf{EMM}$. It then uses $\Omega_P.\mathsf{Compaction}$ to apply the compaction token ctk to the encrypted multi-map. Finally, it outputs the updated encrypted multi-map.

6 The OST_1 Database Encryption Scheme

To be written

7 Storage-Level Emulation of OST_1

The main limitation of STE is its use of non-standard data structures and query algorithms which limits its applicability since it requires re-architecting existing database systems. In fact, this lack of "legacy-friendliness" is widely considered to be the main reason practical encrypted search deployments use PPE-based designs despite their undesirable leakage profiles. Recently, Zhao, Kamara, Moataz and Zdonik showed that the common belief that STE is not legacy-friendly is not true by introducing a new technique called *emulation* that makes STE schemes legacy-friendly [ZKMZ21, ZKMZ20].

Legacy-friendly STE. The reason traditional STE schemes are not legacy-friendly is because they make two implicit assumptions about the server: (1) that it can store arbitrary data structures; and (2) that it can execute arbitrary algorithms. A *legacy-friendly* scheme does not make these assumptions and is designed to work with servers that can only store a fixed kind of data structure and execute a fixed set of operations. For example, a SQL-friendly STE scheme is a scheme that produces encrypted structures that can be stored as relational databases and that has query and update algorithms that can be executed as standard SQL operations. Similarly, a MongoDB-friendly STE scheme is a scheme that produces encrypted structures that can be stored as document databases and that have query and update algorithms that can be executed using standard MongoDB operations.

Emulation. At a high level, the idea behind emulation is to take an encrypted data structure (e.g., an encrypted multi-map) and find a way to represent it as another data structure (e.g., a graph) without any additional storage or query overhead. Intuitively, emulation is a more sophisticated version of the classic data structure problem of simulating a stack with two queues. Designing storage- and query-efficient emulators can be challenging depending on the encrypted structure being emulated and the target structure (i.e., the structure we wish to emulate on top of). The benefits of emulation are twofold: (1) a low-overhead emulator essentially makes an STE scheme legacy-friendly; and (2) it preserves the STE scheme's security.

Storage-level emulation of OST_1. Here we will not describe a fully-emulated version of OST_1 but only a *storage*-level emulation. The difference between full and storage-level

Let $\Omega_P = (\mathsf{Init}, \mathsf{GetToken}, \mathsf{Get}, \mathsf{PutToken}, \mathsf{Put}, \mathsf{EraseToken}, \mathsf{Erase}, \mathsf{CompactionToken}, \mathsf{Compaction}, \mathsf{Resolve})$ be a stateless, response-hiding dynamic multi-map encryption scheme and $\Gamma_H = (\mathsf{Edges}_H, \mathsf{Mincover}_H)$ be a hypergraph scheme. Consider the stateless range multi-map encryption scheme $\Omega_R = (\mathsf{Init}, \mathsf{PutToken}, \mathsf{Put}, \mathsf{RangeToken}, \mathsf{Range}, \mathsf{EraseToken}, \mathsf{Erase}, \mathsf{CompactionToken}, \mathsf{Compaction}, \mathsf{Resolve})$ defined as follows:

- $\mathsf{Init}(1^k)$:
  1. compute $(K, \mathsf{EMM}) \leftarrow \Omega_P.\mathsf{Init}(1^k)$;
  2. output K and $\mathsf{ERMM} := \mathsf{EMM}$.
- $\mathsf{PutToken}(K, \ell, \mathbf{v})$:
  1. compute $\mathbf{E}_\ell := \Gamma.\mathsf{Edges}_H(\ell)$;
  2. for all $e \in \mathbf{E}_\ell$, set $\mathsf{ptk}_e \leftarrow \Omega_P.\mathsf{PutToken}(K, e, \mathbf{v})$;
  3. output $\mathsf{ptk} = (\mathsf{ptk}_e)_{e \in \mathbf{E}_\ell}$.
- $\mathsf{Put}(\mathsf{ERMM}, \mathsf{ptk})$:
  1. parse ERMM as EMM_1 and ptk as $(\mathsf{ptk}_1, \ldots, \mathsf{ptk}_n)$;
  2. for all $1 \le i \le n$, compute $\mathsf{EMM}_{i+1} \leftarrow \Omega_P.\mathsf{Put}(\mathsf{EMM}_i, \mathsf{ptk}_i)$;
  3. output EMM_n.
- $\mathsf{RangeToken}(K, r)$:
  1. compute $\mathbf{C}_r := \mathsf{Mincover}_H(r)$;
  2. for all $e \in \mathbf{C}_r$, compute $\mathsf{gtk}_e \leftarrow \Omega_P.\mathsf{GetToken}(K, e)$;
  3. output $\mathsf{rtk} = (\mathsf{gtk}_e)_{e \in \mathbf{C}_r}$.
- $\mathsf{Range}(\mathsf{ERMM}, \mathsf{rtk})$:
  1. parse ERMM as EMM and rtk as $(\mathsf{gtk}_1, \ldots, \mathsf{gtk}_n)$;
  2. initialize an empty sequences $\mathbf{ct}$;
  3. for all $1 \le i \le n$,
     (a) compute $\mathbf{ct}_i \leftarrow \Omega_P.\mathsf{Get}(\mathsf{EMM}, \mathsf{gtk}_i)$;
     (b) set $\mathbf{ct} := (\mathbf{ct}, \mathbf{ct}_i)$;
  4. output $\mathbf{ct}$;
- $\mathsf{EraseToken}(K, \ell, c_1, \ldots, c_n)$:
  1. compute $(e_1, \cdots, e_n) := \Gamma.\mathsf{Edges}_H(\ell)$;
  2. for all $1 \le i \le n$, set $\mathsf{etk}_j \leftarrow \Omega_P.\mathsf{EraseToken}(K, e_i, c_i)$;
  3. output $\mathsf{etk} = (\mathsf{etk}_1, \ldots, \mathsf{etk}_n)$;
- $\mathsf{Erase}(\mathsf{ERMM}, \mathsf{etk})$:
  1. parse ERMM as EMM_1 and etk as $(\mathsf{etk}_1, \ldots, \mathsf{etk}_n)$;
  2. for all $1 \le i \le n$, compute $\mathsf{EMM}_{i+1} \leftarrow \Omega_P.\mathsf{Delete}(\mathsf{EMM}_i, \mathsf{etk}_i)$;
  3. output EMM_n.
- $\mathsf{CompactionToken}(K)$: output $\mathsf{ctk} \leftarrow \Omega_P.\mathsf{CompactionToken}(K)$.
- $\mathsf{Compaction}(\mathsf{EMM}, \mathsf{ctk}))$:
  1. parse ERMM as EMM;
  2. output $\mathsf{EMM} := \Omega_P.\mathsf{Compaction}(\mathsf{EMM}, \mathsf{ctk})$.
- $\mathsf{Resolve}(K, \mathbf{ct})$: output $\mathbf{v} := \Omega_P.\mathsf{Resolve}(K, \mathbf{ct})$.

Figure 11: Ω_R: a stateless range multi-map encryption scheme.

emulation is that the latter only emulates the data structures of the scheme but not its query and update algorithms. In other words, our emulated OST_1 scheme requires no modifications to the server's storage system but does require the server to implement new query algorithms. We note that it is possible to fully emulate OST_1 but our storage-level emulation results in a more communication-efficient scheme.

Our storage-level emulation of OST_1 is described in Figures 12 through 28 and includes the following (encrypted) operations: *collection creation*, *document insertion*, *document update*, *exact search*, *range search*, *conjunctive search*, *document deletion* and *compaction*. Some of these operations make use of subroutines which are detailed in Figures 29, 30 and 31.

Schema. We assume that the server stores a schema that indicates, for every encrypted field f of the database, its:

- **query type**: whether the field supports exact or range queries;
- **numerical type**: $\perp$ for fields that support exact queries and a tuple of the form $(\mathsf{precision}, \mathsf{lBound}, \mathsf{uBound}, \mathsf{sparsity})$ for fields that support range queries;
- **contention level**: an integer $p \in \mathbb{N}_{\geq 1}$ that determines the field's level of contention. If $p = 0$, the field is not considered to be a high contention field.

We denote the set of all encrypted fields in the database as $\mathbf{F}$, the set of encrypted fields that support exact queries as $\mathbf{EF} \subseteq \mathbf{F}$, the set of encrypted fields that support range queries as $\mathbf{RF} \subseteq \mathbf{F}$ and the set of encrypted fields that are high contention as $\mathbf{HC} \subseteq \mathbf{F}$. Note that we use a field f to denote the "absolute" path of the field, i.e., $\mathsf{db.collection.f}$ if the field f is not nested, or $\mathsf{db.collection.\{field\}.f}$ if it is nested. This guarantees that every field in the database is unique.

Customer-chosen keys. OST_1 is designed to be used in different modes, one where the data encryption keys are chosen/derived by the scheme itself and another where they are chosen by the customer. The two modes are determined by how the scheme is initialized so we provide two different database creation algorithms defined in Figures 12 and 13.

References

[AKM19] Ghous Amjad, Seny Kamara, and Tarik Moataz. Breach-resistant structured encryption. In *Proceedings on Privacy Enhancing Technologies (Po/PETS '19)*, 2019.

[CJJ+14] David Cash, Joseph Jaeger, Stanislaw Jarecki, Charanjit Jutla, Hugo Krawczyk, Marcel Rosu, and Michael Steiner. Dynamic searchable encryption in very-large databases: Data structures and implementation. In *Network and Distributed System Security Symposium (NDSS '14)*, 2014.

Let $F : \{0,1\}^k \times \{0,1\}^* \rightarrow \{0,1\}^*$ be a pseudo-random function and $\mathsf{SKE} = (\mathsf{Gen}, \mathsf{Enc}, \mathsf{Dec})$ be a symmetric encryption scheme.

- $\mathsf{db.createCollection}(1^k, \mathsf{schema}, \mathsf{coll})$:
  1. compute $\mathsf{db.createCollection}(\text{“}\mathtt{edc}_{\mathsf{coll}}\text{”})$;
  2. compute $\mathsf{db.createCollection}(\text{“}\mathtt{esc}_{\mathsf{coll}}\text{”})$;
  3. compute $\mathsf{db.createCollection}(\text{“}\mathtt{ecc}_{\mathsf{coll}}\text{”})$;
  4. compute $\mathsf{db.createCollection}(\text{“}\mathtt{ecoc}_{\mathsf{coll}}\text{”})$.
  5. derive from schema the set of all encrypted fields $\mathbf{F}$;
  6. for all $f \in \mathbf{F}$, compute $K_f \stackrel{\$}{\leftarrow} \{0,1\}^k$;
  7. output $K = (K_f)_{f \in \mathbf{F}}$.

Figure 12: Emulated OST_1: $\mathsf{createCollection}$.

Let $F : \{0,1\}^k \times \{0,1\}^* \rightarrow \{0,1\}^*$ be a pseudo-random function and $\mathsf{SKE} = (\mathsf{Gen}, \mathsf{Enc}, \mathsf{Dec})$ be a symmetric encryption scheme.

- $\mathsf{db.createCollection}((K_f)_{f \in \mathbf{F}}, \mathsf{coll})$:
  1. compute $\mathsf{db.createCollection}(\text{“}\mathtt{edc}_{\mathsf{coll}}\text{”})$;
  2. compute $\mathsf{db.createCollection}(\text{“}\mathtt{esc}_{\mathsf{coll}}\text{”})$;
  3. compute $\mathsf{db.createCollection}(\text{“}\mathtt{ecc}_{\mathsf{coll}}\text{”})$;
  4. compute $\mathsf{db.createCollection}(\text{“}\mathtt{ecoc}_{\mathsf{coll}}\text{”})$.
  5. for all $f \in \mathbf{F}$, compute $K_f^\star := F_{K_f}(f)$;
  6. output $K = (K_f^\star)_{f \in \mathbf{F}}$.

Figure 13: Emulated OST_1: $\mathsf{CreateCollection}$ with customer-chosen keys.

- $\mathsf{db.collection.insert}(K, \mathbf{D})$:
  - **Client**:
    1. parse $\mathbf{D}$ as $\left\{\left(f : v\right)_{f \in \mathbf{EF}}, \left(f : v\right)_{f \in \mathbf{RF}}\right\}$;
    2. parse K as $\left(K_f\right)_{f \in \mathbf{F}}$;
    3. for all $f \in \mathbf{EF}$,
       (a) compute $K_{f,1} := F_{K_f}(1)$, $K_{f,2} := F_{K_f}(2)$ and $K_{f,3} := F_{K_f}(3)$;
       (b) compute

$$K_f^{\mathsf{edc}} := F_{K_{f,1}}(1) \qquad K_f^{\mathsf{esc}} := F_{K_{f,1}}(2) \qquad K_f^{\mathsf{ecc}} := F_{K_{f,1}}(3) \quad \text{and} \quad K_f^{\mathsf{ecoc}} := F_{K_{f,1}}(4)$$

       (c) if $f \in \mathbf{HC}$, sample $u \stackrel{\$}{\leftarrow} \{1, \cdots, p\}$, otherwise set $u := 0$;
       (d) compute $K_{f,v}^{\mathsf{edc}} := F\left(F_{K_f^{\mathsf{edc}}}(v), u\right)$;
       (e) compute $K_{f,v}^{\mathsf{esc}} := F\left(F_{K_f^{\mathsf{esc}}}(v), u\right)$;
       (f) compute $K_{f,v}^{\mathsf{ecc}} := F\left(F_{K_f^{\mathsf{ecc}}}(v), u\right)$;
       (g) compute $\mathrm{ct}_{f,v}^{\mathsf{cp}} := \mathsf{SKE.Enc}\big(K_f^{\mathsf{ecoc}}, K_{f,v}^{\mathsf{esc}} \| K_{f,v}^{\mathsf{ecc}}\big)$;
       (h) compute $\mathrm{ct}_{f,v} := \mathsf{SKE.Enc}\big(K_{f,2}, v\big)$;
    4. for all $f \in \mathbf{RF}$,
       (a) compute $K_{f,1} := F_{K_f}(1)$, $K_{f,2} := F_{K_f}(2)$ and $K_{f,3} := F_{K_f}(3)$;
       (b) compute

$$K_f^{\mathsf{edc}} := F_{K_{f,1}}(1) \qquad K_f^{\mathsf{esc}} := F_{K_{f,1}}(2) \qquad K_f^{\mathsf{ecc}} := F_{K_{f,1}}(3) \quad \text{and} \quad K_f^{\mathsf{ecoc}} := F_{K_{f,1}}(4)$$

       (c) Let ntype_f be the precision, lower and upper bounds and sparsity of the domain of f;
       (d) compute $\mathbf{E}_{f,v} := \mathsf{Edges}_{\mathsf{SPH}}(v, \mathsf{ntype}_f)$;
       (e) if $f \in \mathbf{HC}$, sample $u \stackrel{\$}{\leftarrow} \{1, \cdots, p\}$, otherwise set $u := 0$;
       (f) for all $e \in \mathbf{E}_{f,v}$,
          i. compute $K_{f,e}^{\mathsf{edc}} := F\left(F_{K_f^{\mathsf{edc}}}(e), u\right)$;
          ii. compute $K_{f,e}^{\mathsf{esc}} := F\left(F_{K_f^{\mathsf{esc}}}(e), u\right)$;
          iii. compute $K_{f,e}^{\mathsf{ecc}} := F\left(F_{K_f^{\mathsf{ecc}}}(e), u\right)$;
          iv. compute $\mathrm{ct}_{f,e}^{\mathsf{cp}} := \mathsf{SKE.Enc}\left(K_f^{\mathsf{ecoc}}, K_{f,e}^{\mathsf{esc}} \| K_{f,e}^{\mathsf{ecc}}\right)$;
       (g) compute $\mathrm{ct}_{f,v} := \mathsf{SKE.Enc}\left(K_{f,2}, v\right)$;
    5. set $\mathbf{D} := \left\{\left(f : \mathrm{ct}_{f,v}\right)_{f \in \mathbf{EF}}, \left(f : \mathrm{ct}_{f,v}\right)_{f \in \mathbf{RF}}, \left(\mathsf{safeContent} : []\right)\right\}$;
    6. send $\mathbf{D}$, $\big((K_{f,v}^{\mathsf{edc}}, K_{f,v}^{\mathsf{esc}}, K_{f,v}^{\mathsf{ecc}}, \mathrm{ct}_{f,v}^{\mathsf{cp}})\big)_{f \in \mathbf{EF}}$,

$$\left(\left(\left(K_{f,e}^{\mathsf{edc}}, K_{f,e}^{\mathsf{esc}}, K_{f,e}^{\mathsf{ecc}}, \mathrm{ct}_{f,e}^{\mathsf{cp}}\right)\right)_{e \in \mathbf{E}_{f,v}}\right)_{f \in \mathbf{RF}}$$

       and $\left(K_{f,3}\right)_{f \in \mathbf{F}}$ to the server.

Figure 14: Emulated OST_1: Insert (part 1).

- $\mathsf{db.collection.insert}(K, \mathbf{D})$:
  - **Server**:
    1. initialize an empty array Tags;
    2. for all $f \in \mathbf{EF}$,
       (a) compute $K^1_{f,v} := F_{K^{\mathsf{esc}}_{f,v}}(1)$ and $K^2_{f,v} := F_{K^{\mathsf{esc}}_{f,v}}(2)$;
       (b) compute $a \leftarrow \mathtt{EmuBinary}(K^1_{f,v}, \mathtt{esc})$;
       (c) if $a := 0$, set $\mathsf{count} := 1$, otherwise
          i. compute
          $$\mathbf{r} := \mathtt{db.esc.find}\left(\left\{\mathsf{\_id} : F_{K^1_{f,v}}(a)\right\}\right)$$
          ii. parse $\mathbf{r}$ as $\left\{\mathsf{\_id} : F_{K^1_{f,v}}(a), \mathsf{value} : \mathrm{ct}\right\}$;
          iii. compute $\mathsf{count} := \mathsf{SKE.Dec}\left(K^2_{f,v}, \mathrm{ct}\right) + 1$;
       (d) compute $\mathsf{tag}^{\mathsf{esc}}_{f,v} := F_{K^1_{f,v}}(a+1)$;
       (e) compute $\mathrm{ct}^{\mathsf{esc}}_{f,v} := \mathsf{SKE.Enc}\left(K^2_{f,v}, \mathsf{count}\right)$
       (f) compute
       $$\mathtt{db.esc.insert}\left(\left\{\mathsf{\_id} : \mathsf{tag}^{\mathsf{esc}}_{f,v}, \mathsf{value} : \mathrm{ct}^{\mathsf{esc}}_{f,v}\right\}\right)$$
       (g) compute $K^3_{f,v} := F_{K^{\mathsf{edc}}_{f,v}}(1)$;
       (h) compute $\mathsf{tag}^{\mathsf{edc}}_{f,v} := F_{K^3_{f,v}}(\mathsf{count})$;
       (i) add $\mathsf{tag}^{\mathsf{edc}}_{f,v}$ to Tags;
       (j) set $\mathrm{ct}^{\mathsf{edc}}_{f,v} := \mathsf{SKE.Enc}\left(K_{f,3}, \mathrm{ct}_{f,v} \| \mathsf{count} \| K^{\mathsf{edc}}_{f,v} \| K^{\mathsf{esc}}_{f,v} \| K^{\mathsf{ecc}}_{f,v}\right)$;
       (k) sample a random value $r \overset{\$}{\leftarrow} \{0,1\}^k$ and compute
       $$\mathtt{db.ecoc.insert}\left(\left\{\mathsf{\_id} : r, \mathsf{field} : f, \mathsf{value} : \mathrm{ct}^{\mathsf{cp}}_{f,v}\right\}\right)$$

Figure 15: Emulated OST_1: Insert (part 2).

- $\mathsf{db.collection.insert}(K, \mathbf{D})$:
  - **Server**:
    3. for all $f \in \mathbf{RF}$,
       (a) for all $e \in \mathbf{E}_{f,v}$,
          i. compute $K^1_{f,e} := F_{K^{\mathsf{esc}}_{f,e}}(1)$ and $K^2_{f,e} := F_{K^{\mathsf{esc}}_{f,e}}(2)$;
          ii. compute $a \leftarrow \mathtt{EmuBinary}(K^1_{f,e}, \mathtt{esc})$;
          iii. if $a := 0$ set $\mathsf{count} := 1$, otherwise
             A. compute
             $$\mathbf{r} := \mathtt{db.esc.find}\left(\left\{\mathsf{\_id} : F_{K^1_{f,e}}(a)\right\}\right)$$
             B. parse $\mathbf{r}$ as $\left\{\mathsf{\_id} : F_{K^1_{f,e}}(a), \mathsf{value} : \mathrm{ct}\right\}$;
             C. compute $\mathsf{count}_e := \mathsf{SKE.Dec}\left(K^2_{f,e}, \mathrm{ct}\right) + 1$;
          iv. compute $\mathsf{tag}^{\mathsf{esc}}_{f,e} := F_{K^1_{f,e}}(a+1)$ and $\mathrm{ct}^{\mathsf{esc}}_{f,e} := \mathsf{Enc}_{K^2_{f,e}}(\mathsf{count}_e)$
          v. compute
          $$\mathtt{db.esc.insert}\left(\{\mathsf{\_id} : \mathsf{tag}^{\mathsf{esc}}_{f,e}, \mathsf{value} : \mathrm{ct}^{\mathsf{esc}}_{f,e}\}\right)$$
          vi. compute $K^3_{f,e} := F_{K^{\mathsf{edc}}_{f,e}}(1)$;
          vii. compute $\mathsf{tag}^{\mathsf{edc}}_{f,e} := F_{K^3_{f,e}}(\mathsf{count}_e)$;
          viii. add $\mathsf{tag}^{\mathsf{edc}}_{f,e}$ to Tags;
          ix. sample a random value $r \xleftarrow{\$} \{0,1\}^k$ and compute
          $$\mathtt{db.ecoc.insert}\left(\left\{\mathsf{\_id} : r, \mathsf{field} : f, \mathsf{value} : \mathrm{ct}^{\mathsf{cp}}_{f,e}\right\}\right)$$
       (b) set
       $$\mathrm{ct}^{\mathsf{edc}}_{f,v} := \mathsf{SKE.Enc}\left(K_{f,3}, \mathrm{ct}_{f,v} \| \mathsf{count}_{e_1} \| K^{\mathsf{edc}}_{f,e_1} \| K^{\mathsf{esc}}_{f,e_1} \| K^{\mathsf{ecc}}_{f,e_1} \| \cdots \| \mathsf{count}_{e_n} \| K^{\mathsf{edc}}_{f,e_n} \| K^{\mathsf{esc}}_{f,e_n} \| K^{\mathsf{ecc}}_{f,e_n}\right)$$
       where $\mathbf{E}_{f,v} := \{e_1, \cdots, e_n\}$;
    4. set $\mathbf{D} := \left(\left(f : \mathrm{ct}^{\mathsf{edc}}_{f,v}\right)_{f \in \mathbf{EF}}, \left(f : \mathrm{ct}^{\mathsf{edc}}_{f,v}\right)_{f \in \mathbf{RF}}, (\mathsf{safeContent} : \mathsf{Tags}) \right)$;
    5. compute $\mathsf{db.edc.insert}(\mathbf{D})$.

Figure 16: Emulated OST_1: Insert (part 3).

- $\mathsf{db.collection.find}(K, \{f : v\})$:
  - **Client** (Part 1):
    1. parse K as $\left(K_f\right)_{f \in \mathbf{F}}$;
    2. compute $K_{f,1} := F_{K_f}(1)$;
    3. compute
    $$K^{\mathsf{edc}}_{f,v} := F\left(F_{K_{f,1}}(1), v\right) \qquad K^{\mathsf{esc}}_{f,v} := F\left(F_{K_{f,1}}(2), v\right) \quad \text{and} \quad K^{\mathsf{ecc}}_{f,v} := F\left(F_{K_{f,1}}(3), v\right)$$
    4. send $\left(K^{\mathsf{edc}}_{f,v}, K^{\mathsf{esc}}_{f,v}, K^{\mathsf{ecc}}_{f,v}\right)$ to the server;
  - **Server**:
    1. initialize two sets U and Result;
    2. if $f \in \mathbf{HC}$ set $U := \{1, \cdots, p\}$, otherwise set $U := \{0\}$;
    3. for all $u \in U$,
       (a) compute
       $$K^1_{f,v} := F\left(F_{K^{\mathsf{esc}}_{f,v}}(u), 1\right) \qquad K^2_{f,v} := F\left(F_{K^{\mathsf{esc}}_{f,v}}(u), 2\right) \qquad K^3_{f,v} := F\left(F_{K^{\mathsf{edc}}_{f,v}}(u), 1\right)$$
       $$K^4_{f,v} := F\left(F_{K^{\mathsf{ecc}}_{f,v}}(u), 1\right) \qquad K^5_{f,v} := F\left(F_{K^{\mathsf{ecc}}_{f,v}}(u), 2\right)$$
       (b) compute $a \leftarrow \texttt{EmuBinary}(K^1_{f,v}, \text{“}\texttt{esc}\text{”})$;
       (c) if $a \neq 0$,
          i. initialize a set $\mathbf{g}$;
          ii. compute $\mathbf{r} := \texttt{db.esc.find}\left(\left\{\mathsf{\_id} : F_{K^1_{f,v}}(a)\right\}\right)$;
          iii. parse $\mathbf{r}$ as $\left\{\mathsf{\_id} : F_{K^1_{f,v}}(a), \mathsf{value} : \mathrm{ct}\right\}$;
          iv. compute $\mathsf{count} := \mathsf{Dec}\left(K^2_{f,v}, \mathrm{ct}\right)$;
          v. set $\mathsf{flag} := \texttt{true}$ and $j := 1$;
          vi. while $\mathsf{flag} = \texttt{true}$,
             A. compute $\mathbf{r} := \texttt{db.ecc.find}\left(\left\{\mathsf{\_id} : F_{K^4_{f,v}}(j)\right\}\right)$;
             B. if $\mathbf{r} \neq \perp$,
                * parse $\mathbf{r}$ as $\left\{\mathsf{\_id} : F_{K^4_{f,v}}(j), \mathsf{value} : \mathrm{ct}\right\}$;
                * add $\mathsf{SKE.Dec}\left(K^5_{f,v}, \mathrm{ct}\right)$ to $\mathbf{g}$;
                * set $j := j + 1$;
             C. otherwise set $\mathsf{flag} := \mathsf{false}$;
          vii. for all $i \in \{1, \cdots, \mathsf{count}\} \setminus \mathbf{g}$, add
          $$\mathbf{r} := \texttt{db.edc.find}\left(\left\{\mathsf{safeContent} : F_{K^3_{f,v}}(i)\right\}\right)$$
          to Result.
    4. send Result to client.

Figure 17: Emulated OST_1: Find with exact search (part 1).

- $\mathsf{db.collection.find}(K, \{f : v\})$:
  - **Client** (Part 2):
    1. for every $\mathbf{D}$ in Result,
       (a) parse $\mathbf{D}$ as
       $$\Big(\big(f : \mathrm{ct}^{\mathsf{edc}}_{f,v}\big)_{f \in \mathbf{EF}}, \big(f : \mathrm{ct}^{\mathsf{edc}}_{f,v}\big)_{f \in \mathbf{RF}}, (\mathsf{safeContent} : \mathsf{Tags}) \Big);$$
       (b) for every $f \in \mathbf{EF}$,
          i. compute $K_{f,2} := F_{K_f}(2)$ and $K_{f,3} := F_{K_f}(3)$;
          ii. compute $\mathrm{ct} := \mathsf{SKE.Dec}\big(K_{f,3}, \mathrm{ct}^{\mathsf{edc}}_{f,v}\big)$ and parse ct as $\mathrm{ct}_{f,v} \| x$;
          iii. compute $v := \mathsf{SKE.Dec}\left(K_{f,2}, \mathrm{ct}_{f,v}\right)$;
       (c) for every $f \in \mathbf{RF}$,
          i. compute $K_{f,2} := F_{K_f}(2)$ and $K_{f,3} := F_{K_f}(3)$;
          ii. compute $\mathrm{ct} := \mathsf{SKE.Dec}\big(K_{f,3}, \mathrm{ct}^{\mathsf{edc}}_{f,v}\big)$ and parse ct as $\mathrm{ct}_{f,v} \| x_1 \| \cdots \| x_n$;
          iii. compute $v := \mathsf{SKE.Dec}\left(K_{f,2}, \mathrm{ct}_{f,v}\right)$;
       (d) set $\mathbf{D} := \Big((f : v)_{f \in \mathbf{EF}}, (f : v)_{f \in \mathbf{RF}} \Big)$;
    2. output Result.

Figure 18: Emulated OST_1: Find with exact search (part 2).

[KKM21] Kasemsan Kongsala, Seny Kamara, and Tarik Moataz. Encrypted range search via range hypergraphs. Technical report, 2021.

[KL08] J. Katz and Y. Lindell. *Introduction to Modern Cryptography.* Chapman & Hall/CRC, 2008.

[KL10] Seny Kamara and Kristin Lauter. Cryptographic cloud storage. In *Workshop on Real-Life Cryptographic Protocols and Standardization*, pages 136–149. Springer, 2010.

[mon] Mongodb crud operations. https://docs.mongodb.com/manual/crud/.

[ZKMZ20] Zeguang Zhao, Seny Kamara, Tarik Moataz, and Stan Zdonik. Kafedb: a structurally-encrypted relational database management system. Technical report, 2020.

[ZKMZ21] Zheguang Zhao, Seny Kamara, Tarik Moataz, and Stan Zdonik. Encrypted databases: From theory to systems. In *Conference on Innovative Data Systems Research (CIDR '21)*, 2021.

A The Sparse Partition Hypergraph

The encrypted range scheme used by OST_1 results from the ERX framework of [KKM21] instantiated with the stateless multi-map encryption scheme Ω from section 4 and a new

- $\mathsf{db.collection.find}(K, \{f : \{\mathsf{\$gte} : v_1, \mathsf{\$lte} : v_2\}\})$:
  - **Client** (part 1):
    1. parse K as $\left(K_f\right)_{f\in\mathbf{F}}$;
    2. compute $K_{f,1} := F_{K_f}(1)$;
    3. compute $\mathbf{C} := \mathsf{Mincover}_{\mathsf{SPH}}([v_1, v_2], \mathsf{ntype}_f)$;
    4. for all $e \in \mathbf{C}$,
       (a) compute $K^{\mathsf{edc}}_{f,e} := F\left(F_{K_{f,1}}(1), e\right)$;
       (b) compute $K^{\mathsf{esc}}_{f,e} := F\left(F_{K_{f,1}}(2), e\right)$;
       (c) compute $K^{\mathsf{ecc}}_{f,e} := F\left(F_{K_{f,1}}(3), e\right)$;
    5. send $\left(\left(K^{\mathsf{edc}}_{f,e}, K^{\mathsf{esc}}_{f,e}, K^{\mathsf{ecc}}_{f,e}\right)\right)_{e\in\mathbf{C}}$ to the server.
  - **Server**:
    1. initialize two sets U and Result;
    2. if $f \in \mathbf{HC}$, set $U := \{1, \cdots, p\}$, otherwise set $U := \{0\}$;
    3. for all $e \in \mathbf{C}$ and $u \in U$,
       (a) compute

$$K^1_{f,e} := F\left(F_{K^{\mathsf{esc}}_{f,e}}(u), 1\right) \qquad K^2_{f,e} := F\left(F_{K^{\mathsf{esc}}_{f,e}}(u), 2\right) \qquad K^3_{f,e} := F\left(F_{K^{\mathsf{edc}}_{f,e}}(u), 1\right)$$

$$K^4_{f,e} := F\left(F_{K^{\mathsf{ecc}}_{f,e}}(u), 1\right) \quad \text{and} \quad K^5_{f,e} := F\left(F_{K^{\mathsf{ecc}}_{f,e}}(u), 2\right)$$

       (b) compute $a \leftarrow \mathtt{EmuBinary}(K^1_{f,e}, \text{“}\mathtt{esc}\text{”})$;
       (c) if $a \neq 0$,
          i. initialize a set $\mathbf{g}$;
          ii. compute $\mathbf{r} := \mathtt{db.esc.find}\left(\left\{\mathsf{\_id} : F_{K^1_{f,e}}(a)\right\}\right)$;
          iii. parse $\mathbf{r}$ as $\left\{\mathsf{\_id} : F_{K^1_{f,e}}(a), \mathsf{value} : \mathrm{ct}\right\}$;
          iv. compute $\mathsf{count} := \mathsf{SKE.Dec}\left(K^2_{f,e}, \mathrm{ct}\right)$;
          v. set $\mathsf{flag} := \mathtt{true}$ and $j := 1$;
          vi. while $\mathsf{flag} = \mathtt{true}$,
             A. compute $\mathbf{r} := \mathtt{db.ecc.find}\left(\left\{\mathsf{\_id} : F_{K^4_{f,e}}(j)\right\}\right)$;
             B. if $\mathbf{r} \neq \perp$,
                A. parse $\mathbf{r}$ as $\left\{\mathsf{\_id} : F_{K^4_{f,e}}(j), \mathsf{value} : \mathrm{ct}\right\}$;
                B. add $\mathsf{SKE.Dec}\left(K^5_{f,e}, \mathrm{ct}\right)$ to $\mathbf{g}$;
                C. set $j := j + 1$;
             D. otherwise set $\mathsf{flag} := \mathsf{false}$;
          vii. for all $i \in \{1, \cdots, \mathsf{count}\} \setminus \mathbf{g}$,
             A. compute $\mathbf{r} := \mathtt{db.edc.find}\left(\left\{\mathsf{safeContent} : F_{K^3_{f,e}}(i)\right\}\right)$;
             B. add $\mathbf{r}$ to Result;
    4. send Result to client.
  - **Client** (Part 2):
    1. similar to Client (Part 2) of Figure 17.

Figure 19: Emulated OST_1: Find with range search.

- $\mathsf{db.collection.update}(K, \{f_q : v_q\}, \{\$\mathsf{set} : \{f : v\}\})$:
  - **Client**:
    1. parse K as $\left(K_f\right)_{f \in \mathbf{F}}$;
    2. compute $K_{f,1} := F_{K_f}(1)$, $K_{f,2} := F_{K_f}(2)$ and $K_{f,3} := F_{K_f}(3)$;
    3. compute

$$K_f^{\mathsf{edc}} := F_{K_{f,1}}(1) \qquad K_f^{\mathsf{esc}} := F_{K_{f,1}}(2) \qquad K_f^{\mathsf{ecc}} := F_{K_{f,1}}(3) \quad \text{and} \quad K_f^{\mathsf{ecoc}} := F_{K_{f,1}}(4)$$

    4. if $f \in \mathbf{EF}$,
       (a) if $f \in \mathbf{HC}$ sample $u \xleftarrow{\$} \{1, \cdots, p\}$, otherwise set $u := 0$;
       (b) compute

$$K_{f,v}^{\mathsf{edc}} := F\left(F_{K_f^{\mathsf{edc}}}(v), u\right) \qquad K_{f,v}^{\mathsf{edc}} := F\left(F_{K_f^{\mathsf{esc}}}(v), u\right) \quad \text{and} \quad K_{f,v}^{\mathsf{ecc}} := F\left(F_{K_f^{\mathsf{ecc}}}(v), u\right)$$

       (c) compute $\mathrm{ct}_{f,v}^{\mathsf{cp}} := \mathsf{SKE.Enc}\left(K_f^{\mathsf{ecoc}}, K_{f,v}^{\mathsf{esc}} \| K_{f,v}^{\mathsf{ecc}}\right)$;
       (d) compute $\mathrm{ct}_{f,v} := \mathsf{SKE.Enc}\left(K_{f,2}, v\right)$;
    5. otherwise if $f \in \mathbf{RF}$,
       (a) compute $\mathbf{E}_{f,v} := \mathsf{Edges}(v, \mathsf{ntype}_f)$;
       (b) for all $e \in \mathbf{E}_{f,v}$,
          i. if $f \in \mathbf{HC}$ sample $u \xleftarrow{\$} \{1, \cdots, p\}$, otherwise set $u := 0$;
          ii. compute

$$K_{f,e}^{\mathsf{edc}} := F\left(F_{K_f^{\mathsf{edc}}}(e), u\right) \qquad K_{f,e}^{\mathsf{esc}} := F\left(F_{K_f^{\mathsf{esc}}}(e), u\right) \quad \text{and} \quad K_{f,e}^{\mathsf{ecc}} := F\left(F_{K_f^{\mathsf{ecc}}}(e), u\right)$$

          iii. compute $\mathrm{ct}_{f,e}^{\mathsf{cp}} := \mathsf{SKE.Enc}\left(K_f^{\mathsf{ecoc}}, K_{f,e}^{\mathsf{esc}} \| K_{f,e}^{\mathsf{ecc}}\right)$;
       (c) compute $\mathrm{ct}_{f,v} := \mathsf{SKE.Enc}\left(K_{f,2}, v\right)$;
    6. if $f_q \in \mathbf{EF}$, then send the output of Client (part 1) of Figure 17 to the server;
    7. if $f_q \in \mathbf{RF}$, then send the output of Client (part 1) of Figure 19 to the server;
    8. send $\mathrm{ct}_{f,v}$, K_f^{ecoc}, $\left(K_{f,v}^{\mathsf{edc}}, K_{f,v}^{\mathsf{esc}}, K_{f,v}^{\mathsf{ecc}}, \mathrm{ct}_{f,v}^{\mathsf{cp}}\right)$ or $\left(\left(K_{f,e}^{\mathsf{edc}}, K_{f,e}^{\mathsf{esc}}, K_{f,e}^{\mathsf{ecc}}, \mathrm{ct}_{f,e}^{\mathsf{cp}}\right)\right)_{e \in \mathbf{E}_{f,v}}$, and $K_{f,3}$ to the server.

Figure 20: Emulated OST_1: Update a single field (part 1).

- $\mathsf{db.collection.update}(K, \{f_q : v_q\}, \{\$\mathsf{set} : \{f : v\}\})$:
  - **Server**:
    1. if $f_q \in \mathbf{EF}$, let Result be the output of Server in Figure 17;
    2. if $f_q \in \mathbf{RF}$, let Result be the output of Server in Figure 19;
    3. for all $\mathbf{D} \in \mathsf{Result}$,
       (a) let i be the identifier $\mathsf{\_id}$ of the document $\mathbf{D}$;
       (b) parse $\mathbf{D}$ as

       $$\Big(\left(f' : \mathrm{ct}^{\mathsf{edc}}_{f,v}\right)_{f' \in \mathbf{EF}}, \left(f' : \mathrm{ct}^{\mathsf{edc}}_{f,v}\right)_{f' \in \mathbf{RF}}, (\mathsf{safeContent} : \mathsf{Tags}) \Big);$$

       (c) if $f \in \mathbf{EF}$,
          i. parse $\mathsf{SKE.Dec}\left(K_{f,3}, \mathrm{ct}^{\mathsf{edc}}_{f,v}\right)$ as $\mathrm{ct}_{f,v} \| x$;
          ii. parse x as $\mathsf{count} \| K^{\mathsf{edc}} \| K^{\mathsf{esc}} \| K^{\mathsf{ecc}}$;
          iii. compute $K^1 := F_{K^{\mathsf{ecc}}}(1)$ and $K^2 := F_{K^{\mathsf{ecc}}}(2)$;
          iv. compute $a \leftarrow \mathtt{EmuBinary}(K^1, \text{“}\mathtt{ecc}\text{”})$;
          v. compute $\mathsf{tag} := F_{K^1}(a+1)$, $\mathrm{ct} := \mathsf{Enc}_{K^2}(\mathsf{count})$ and compute

          $$\mathsf{db.ecc.insert}\left(\{\mathsf{\_id} : \mathsf{tag}, \mathsf{value} : \mathrm{ct}\}\right)$$

          vi. sample $r \xleftarrow{\$} \{0,1\}^k$ and compute

          $$\mathsf{db.ecoc.insert}\left(\left\{\mathsf{\_id} : r, \mathsf{field} : f, \mathsf{value} : \mathsf{SKE.Enc}\left(K^{\mathsf{ecoc}}_f, K^{\mathsf{esc}} \| K^{\mathsf{ecc}}\right)\right\}\right)$$

          vii. compute $K^3 := F_{K^{\mathsf{edc}}}(1)$;
          viii. compute

          $$\mathsf{db.edc.update}\left(\{\mathsf{\_id} : i\}, \{\$\mathsf{pull} : \{\mathsf{safeContent} : F_{K^3}(\mathsf{count})\}\}\right)$$

          ix. compute $K^1_{f,v} := F_{K^{\mathsf{esc}}_{f,v}}(1)$ and $K^2_{f,v} := F_{K^{\mathsf{esc}}_{f,v}}(2)$;
          x. compute $a \leftarrow \mathtt{EmuBinary}(K^1_{f,v}, \mathsf{esc})$;
          xi. if $a := 0$, then set count to 1, otherwise
             A. compute $\mathbf{r} := \mathsf{db.esc.}\mathtt{find}\left(\left\{\mathsf{\_id} : F_{K^1_{f,v}}(a)\right\}\right)$;
             B. parse $\mathbf{r}$ as $\left\{\mathsf{\_id} : F_{K^1_{f,v}}(a), \mathsf{value} : \mathrm{ct}\right\}$;
             C. compute $\mathsf{count} := \mathsf{SKE.Dec}\left(K^2_{f,v}, \mathrm{ct}\right) + 1$;
          xii. compute

          $$\mathsf{db.esc.insert}\left(\left\{\mathsf{\_id} : F_{K^1_{f,v}}(a+1), \mathsf{value} : \mathsf{SKE.Enc}(K^2_{f,v}, \mathsf{count})\right\}\right);$$

          xiii. compute $K^3_{f,v} := F_{K^{\mathsf{edc}}_{f,v}}(1)$;
          xiv. compute

          $$\mathsf{db.edc.update}(\{\mathsf{\_id} : i\}, \{\$\mathsf{push} : \{\mathsf{safeContent} : F_{K^3_{f,v}}(\mathsf{count})\}\})$$

          xv. set $\mathrm{ct}^{\mathsf{edc}}_{f,v} := \mathsf{SKE.Enc}\left(K_{f,3}, \mathrm{ct}_{f,v} \| \mathsf{count} \| K^{\mathsf{edc}}_{f,v} \| K^{\mathsf{esc}}_{f,v} \| K^{\mathsf{ecc}}_{f,v}\right)$;
          xvi. compute $\mathsf{db.edc.update}\left(\{\mathsf{\_id} : i\}, \left\{\$\mathsf{set} : \left\{f : \mathrm{ct}^{\mathsf{edc}}_{f,v}\right\}\right\}\right)$
          xvii. sample a random value $r \xleftarrow{\$} \{0,1\}^k$ and compute

          $$\mathsf{db.ecoc.insert}\left(\left\{\mathsf{\_id} : r, \mathsf{field} : f, \mathsf{value} : \mathrm{ct}^{\mathsf{cp}}_{f,v}\right\}\right)$$

Figure 21: Emulated OST_1: Update a single field (part 2).

- $\mathsf{db.collection.update}(K, \{f_q : v_q\}, \{\$\mathsf{set} : \{f : v\}\})$:
  - **Server**:
    (d) if $f \in \mathbf{RF}$,
      i. parse $\mathsf{SKE.Dec}(K_{f,3}, \mathrm{ct}^{\mathsf{edc}}_{f,v})$ as $\mathrm{ct}_{f,v} \| x_1 \| \cdots \| x_n$;
      ii. for $1 \leq j \leq n$,
        A. parse x_j as $\mathsf{count}_j \| K^{\mathsf{edc}}_j \| K^{\mathsf{esc}}_j \| K^{\mathsf{ecc}}_j$;
        B. compute $K^1 := F_{K^{\mathsf{ecc}}_j}(1)$ and $K^2 := F_{K^{\mathsf{ecc}}_j}(2)$;
        C. compute $a \leftarrow \mathtt{EmuBinary}(K^1, \text{“}\mathtt{ecc}\text{”})$;
        D. compute $\mathsf{tag} := F_{K^1}(a+1)$, $\mathrm{ct} := \mathsf{Enc}_{K^2}(\mathsf{count}_j)$ and compute
$$\mathsf{db.ecc.insert}\left(\{\_\mathsf{id} : \mathsf{tag}, \mathsf{value} : \mathrm{ct}\}\right)$$
        E. sample $r \xleftarrow{\$} \{0,1\}^k$ and compute
$$\mathsf{db.ecoc.insert}(\{\_\mathsf{id} : r, \mathsf{field} : f, \mathsf{value} : \mathsf{SKE.Enc}(K^{\mathsf{ecoc}}_f, K^{\mathsf{esc}}_j \| K^{\mathsf{ecc}}_j)\})$$
        F. compute $K^3 := F_{K^{\mathsf{edc}}_j}(1)$;
        G. compute
$$\mathsf{db.edc.update}(\{\_\mathsf{id} : i\}, \{\$\mathsf{pull} : \{\mathsf{safeContent} : F_{K^3}(\mathsf{count}_j)\}\})$$
      iii. for all $e \in \mathbf{E}_{f,v}$,
        A. compute $K^1_{f,e} := F_{K^{\mathsf{esc}}_{f,e}}(1)$ and $K^2_{f,e} := F_{K^{\mathsf{esc}}_{f,e}}(2)$;
        B. compute $a \leftarrow \mathtt{EmuBinary}(K^1_{f,e}, \text{“}\mathtt{esc}\text{”})$;
        C. if $a := 0$, then set count_e to 1, otherwise
          A. compute
$$\mathbf{r} := \mathsf{db.esc}.\mathtt{find}\left(\left\{\_\mathsf{id} : F_{K^1_{f,e}}(a)\right\}\right)$$
          B. parse $\mathbf{r}$ as $\{\_\mathsf{id} : F_{K^1_{f,e}}(a), \mathsf{value} : \mathrm{ct}\}$;
          C. compute $\mathsf{count}_e := \mathsf{SKE.Dec}\left(K^2_{f,e}, \mathrm{ct}\right) + 1$;
        D. compute
$$\mathsf{db.esc.insert}\left(\left\{\_\mathsf{id} : F_{K^1_{f,e}}(a+1), \mathsf{value} : \mathsf{SKE.Enc}(K^2_{f,e}, \mathsf{count}_e)\right\}\right)$$
        E. compute $K^3_{f,e} := F_{K^{\mathsf{edc}}_{f,e}}(1)$;
        F. compute
$$\mathsf{db.edc.update}(\{\_\mathsf{id} : i\}, \{\$\mathsf{push} : \{\mathsf{safeContent} : F_{K^3_{f,e}}(\mathsf{count}_e)\}\})$$
        G. sample a random value $r \xleftarrow{\$} \{0,1\}^k$ and compute
$$\mathsf{db.ecoc.insert}\left(\left\{\_\mathsf{id} : r, \mathsf{field} : f, \mathsf{value} : \mathrm{ct}^{\mathsf{cp}}_{f,e}\right\}\right)$$
      iv. set
$$\mathrm{ct}^{\mathsf{edc}}_{f,v} := \mathsf{SKE.Enc}\left(K_{f,3}, \mathrm{ct}_{f,v} \| \mathsf{count}_{e_1} \| K^{\mathsf{edc}}_{f,e_1} \| K^{\mathsf{esc}}_{f,e_1} \| K^{\mathsf{ecc}}_{f,e_1} \cdots \mathsf{count}_{e_n} \| K^{\mathsf{edc}}_{f,e_n} \| K^{\mathsf{esc}}_{f,e_n} \| K^{\mathsf{ecc}}_{f,e_n}\right)$$
        where $\mathbf{E}_{f,v} := \{e_1, \cdots e_n\}$;
      v. compute
$$\mathsf{db.edc.update}(\{\_\mathsf{id} : i\}, \{\$\mathsf{set} : \{f : \mathrm{ct}^{\mathsf{edc}}_{f,v}\}\})$$

Figure 22: Emulated OST_1: Update a single field (part 3).

- $\mathsf{db.collection.delete}(K, \{f_q : v_q\})$:
  - **Client**:
    1. parse K as $\big(K_f\big)_{f\in\mathbf{F}}$;
    2. for all $f \in \mathbf{F}$, compute $K_{f,1} := F_{K_f}(1)$, $K_{f,3} := F_{K_f}(3)$, and $K_f^{\mathsf{ecoc}} := F_{K_{f,1}}(4)$;
    3. if $f_q \in \mathbf{EF}$, then send the output of Client (Part 1) in Figure 17;
    4. if $f_q \in \mathbf{RF}$, then send the output of Client (Part 1) in Figure 19;
    5. send to the server $(K_f^{\mathsf{ecoc}})_{f\in\mathbf{F}}$ and $(K_{f,3})_{f\in\mathbf{F}}$.
  - **Server**:
    1. if $f_q \in \mathbf{EF}$, let Result be the output of Server in Figure 17 to the server;
    2. if $f_q \in \mathbf{RF}$, let Result be the output of Server in Figure 19 to the server;
    3. for all $\mathbf{D} \in \mathsf{Result}$,
       (a) let i be the identifier $\mathsf{\_id}$ of document $\mathbf{D}$;
       (b) parse $\mathbf{D}$ as
       $$\Big(\big(f : \mathrm{ct}_{f,v}^{\mathsf{edc}}\big)_{f\in\mathbf{EF}}, \big(f : \mathrm{ct}_{f,v}^{\mathsf{edc}}\big)_{f\in\mathbf{RF}}, (\mathsf{safeContent} : \mathsf{Tags})\Big);$$
       (c) for all $f \in \mathbf{F}$,
         i. if $f \in \mathbf{EF}$,
           A. compute $\mathrm{ct} := \mathsf{SKE.Dec}\big(K_{f,3}, \mathrm{ct}_{f,v}^{\mathsf{edc}}\big)$ and parse ct as $\mathrm{ct}_{f,v}\|x$;
           B. parse x as $\mathsf{count}\|K^{\mathsf{edc}}\|K^{\mathsf{esc}}\|K^{\mathsf{ecc}}$;
           C. compute $K^1 := F_{K^{\mathsf{ecc}}}(1)$ and $K^2 := F_{K^{\mathsf{ecc}}}(2)$;
           D. compute $a \leftarrow \mathtt{EmuBinary}(K^1, \text{“}\mathtt{ecc}\text{”})$;
           E. compute $\mathsf{tag} := F_{K^1}(a+1)$ and $\mathrm{ct} := \mathsf{Enc}_{K^2}(\mathsf{count})$;
           $$\mathsf{db.ecc.insert}(\{\mathsf{\_id} : \mathsf{tag}, \mathsf{value} : \mathrm{ct}\})$$
           F. sample $r \xleftarrow{\$} \{0,1\}^k$ and compute
           $$\mathsf{db.ecoc.insert}\left(\{\mathsf{\_id} : r, \mathsf{field} : f, \mathsf{value} : \mathsf{SKE.Enc}(K_f^{\mathsf{ecoc}}, K^{\mathsf{esc}}\|K^{\mathsf{ecc}})\right\}$$
         ii. if $f \in \mathbf{RF}$,
           A. compute $\mathrm{ct} := \mathsf{SKE.Dec}\big(K_{f,3}, \mathrm{ct}_{f,v}^{\mathsf{edc}}\big)$ and parse ct as $\mathrm{ct}_{f,v}\|x_1\|\cdots\|x_n$;
           B. for $1 \leq j \leq n$,
             A. parse x_j as $\mathsf{count}_j\|K_j^{\mathsf{edc}}\|K_j^{\mathsf{esc}}\|K_j^{\mathsf{ecc}}$;
             B. compute $K^1 := F_{K_j^{\mathsf{ecc}}}(1)$ and $K^2 := F_{K_j^{\mathsf{ecc}}}(2)$;
             C. compute $a \leftarrow \mathtt{EmuBinary}(K^1, \text{“}\mathtt{ecc}\text{”})$;
             D. compute $\mathsf{tag} := F_{K^1}(a+1)$ and $\mathrm{ct} := \mathsf{Enc}_{K^2}(\mathsf{count}_j)$;
             $$\mathsf{db.ecc.insert}(\{\mathsf{\_id} : \mathsf{tag}, \mathsf{value} : \mathrm{ct}\})$$
             E. sample $r \xleftarrow{\$} \{0,1\}^k$ and compute
             $$\mathsf{db.ecoc.insert}(\{\mathsf{\_id} : r, \mathsf{field} : f, \mathsf{value} : \mathsf{SKE.Enc}(K_f^{\mathsf{ecoc}}, K_j^{\mathsf{esc}}\|K_j^{\mathsf{ecc}})\})$$
       (d) compute $\mathsf{db.edc.delete}(\{\mathsf{\_id} : i\})$

Figure 23: Emulated OST_1: Delete.

- $\mathsf{db.collection.find}(K, \{\$\mathsf{and} : [\{f_1, v_1\}, \cdots, \{f_n, v_n\}]\})$:
  - **Client** (Part 1):
    1. parse K as $\left(K_f\right)_{f\in\mathbf{F}}$;
    2. set $\mathbf{F}^\star = \{f_1, \cdots, f_n\}$;
    3. for all $f \in \mathbf{F}^\star$,
       (a) if $f \in \mathbf{EF}$, send the output of Client (part 1) of Figure 17 to the server;
       (b) if $f \in \mathbf{RF}$, send the output of Client (part 1) of Figure 19 to the server;
       (c) compute $K_{f,2} := F_{K_f}(2)$ and $K_{f,3} := F_{K_f}(3)$;
    4. send to the server $(K_{f,3})_{f\in\mathbf{F}^\star}$.
  - **Server**:
    1. initialize count_m to 0;
    2. for all $f \in \mathbf{EF}$,
       (a) if $f \in \mathbf{HC}$, then set $U := \{1, \cdots, p\}$, otherwise set $U := \{0\}$;
       (b) for all $u \in U$,
          i. compute
          $$K^1_{f,v} := F\left(F_{K^{\mathsf{esc}}_{f,v}}(u), 1\right) \qquad K^2_{f,v} := F\left(F_{K^{\mathsf{esc}}_{f,v}}(u), 2\right)$$
          ii. compute $a \leftarrow \mathtt{EmuBinary}(K^1_{f,v}, \text{“}\mathtt{esc}\text{”})$;
          iii. if $a = 0$, set $\mathsf{count}_{f,u} = 0$, otherwise,
             A. compute $\mathsf{tag} := F_{K^1_{f,v}}(a)$;
             B. compute
             $$\mathbf{r} := \mathsf{db.esc.}\mathtt{find}(\{\mathsf{\_id} : \mathsf{tag}\})$$
             C. parse $\mathbf{r}$ as $\{\mathsf{\_id} : \mathsf{tag}, \mathsf{value} : \mathrm{ct}\}$;
             D. compute $\mathsf{count}_{f,u} := \mathsf{SKE.Dec}\left(K^2_{f,v}, \mathrm{ct}\right)$;
          iv. otherwise set $\mathsf{count}_{f,u}$ to 0;
       (c) if $\mathsf{count}_m > \sum_{u\in U} \mathsf{count}_{f,u}$, then set $\mathsf{count}_m := \sum_{u\in U} \mathsf{count}_{f,u}$;
    3. for all $f \in \mathbf{RF}$,
       (a) if $f \in \mathbf{HC}$, then set $U := \{1, \cdots, p\}$, otherwise set $U := \{0\}$;
       (b) for all $e \in \mathbf{C}$ and $u \in U$,
          i. compute
          $$K^1_{f,e} := F\left(F_{K^{\mathsf{esc}}_{f,e}}(u), 1\right) \qquad K^2_{f,e} := F\left(F_{K^{\mathsf{esc}}_{f,e}}(u), 2\right)$$
          ii. compute $a \leftarrow \mathtt{EmuBinary}\left(K^1_{f,e}, \text{“}\mathtt{esc}\text{”}\right)$;
          iii. if $a = 0$, set $\mathsf{count}_{f,e,u} = 0$, otherwise,
             A. compute
             $$\mathbf{r} := \mathsf{db.esc.}\mathtt{find}(\{\mathsf{\_id} : F_{K^1_{f,e}}(a)\})$$
             B. parse $\mathbf{r}$ as $\{\mathsf{\_id} : F_{K^1_{f,e}}(a), \mathsf{value} : \mathrm{ct}\}$;
             C. compute $\mathsf{count}_{f,e,u} := \mathsf{SKE.Dec}\left(K^2_{f,e}, \mathrm{ct}\right)$;
       (c) if $\mathsf{count}_m > \sum_{u\in U, e\in\mathbf{E}_{f,v}} \mathsf{count}_{f,e,u}$, then set $\mathsf{count}_m := \sum_{u\in U, e\in\mathbf{E}_{f,v}} \mathsf{count}_{f,e,u}$;

Figure 24: Emulated OST_1: Find with conjunctive search (part 1).

- $\textsf{db.collection.find}(K, \{\$\textsf{and} : [\{f_1, v_1\}, \cdots, \{f_n, v_n\}]\})$:
  - **Server**:
    4. let $f^\star$ be the field with the smallest $\textsf{count}_m$;
    5. if $\textsf{count}_m > 0$,
       (a) if $f^\star \in \mathbf{EF}$,
          i. for all $u \in U$,
             A. initialize a set $\mathbf{g}$;
             B. compute

$$K^3_{f^\star,v} := F\left(F_{K^{\textsf{edc}}_{f^\star,v}}(u), 1\right) \qquad K^4_{f^\star,v} := F\left(F_{K^{\textsf{ecc}}_{f^\star,v}}(u), 1\right) \quad \text{and} \quad K^5_{f^\star,v} := F\left(F_{K^{\textsf{ecc}}_{f^\star,v}}(u), 2\right);$$

             C. set $\textsf{flag} := \texttt{true}$ and $j := 1$;
             D. while $\textsf{flag} = \texttt{true}$,
                A. compute $\mathbf{r} := \texttt{db.ecc.find}\left(\left\{\textsf{\_id} : F_{K^4_{f^\star,v}}(j)\right\}\right)$
                B. if $\mathbf{r} \neq \perp$,
                   A. parse $\mathbf{r}$ as $\left\{\textsf{\_id} : F_{K^4_{f^\star,v}}(j), \textsf{value} : \mathrm{ct}\right\}$;
                   B. add $\textsf{SKE.Dec}\left(K^5_{f^\star,v}, \mathrm{ct}\right)$ to $\mathbf{g}$;
                   C. set $j := j + 1$;
                D. otherwise set $\textsf{flag} := \textsf{false}$;
             E. for all $i \in \{1, \cdots, \textsf{count}_{f^\star,u}\} \setminus \mathbf{g}$, add to $\textsf{Result}$

$$\mathbf{r} := \texttt{db.edc.find}\left(\left\{\textsf{safeContent} : F_{K^3_{f^\star,v}}(i)\right\}\right)$$

       (b) if $f^\star \in \mathbf{RF}$,
          i. for all $e \in \mathbf{E}_{f^\star,v}$ and $u \in U$,
             A. initialize a set $\mathbf{g}$;
             B. compute

$$K^3_{f^\star,e} := F\left(F_{K^{\textsf{edc}}_{f^\star,e}}(u), 1\right) \qquad K^4_{f^\star,e} := F\left(F_{K^{\textsf{ecc}}_{f^\star,e}}(u), 1\right) \quad \text{and} \quad K^5_{f^\star,e} := F\left(F_{K^{\textsf{ecc}}_{f^\star,e}}(u), 2\right);$$

             C. set $\textsf{flag} := \texttt{true}$ and $j := 1$;
             D. while $\textsf{flag} = \texttt{true}$,
                A. compute $\mathbf{r} := \texttt{db.ecc.find}\left(\left\{\textsf{\_id} : F_{K^4_{f^\star,e}}(j)\right\}\right)$
                B. if $\mathbf{r} \neq \perp$,
                   A. parse $\mathbf{r}$ as $\left\{\textsf{\_id} : F_{K^4_{f^\star,e}}(j), \textsf{value} : \mathrm{ct}\right\}$;
                   B. add $\textsf{SKE.Dec}\left(K^5_{f^\star,e}, \mathrm{ct}\right)$ to $\mathbf{g}$;
                   C. set $j := j + 1$;
                D. otherwise set $\textsf{flag} := \textsf{false}$;
             E. for all $i \in \{1, \cdots, \textsf{count}_{f^\star,e,u}\} \setminus \mathbf{g}$, add to $\textsf{Result}$

$$\mathbf{r} := \texttt{db.edc.find}\left(\left\{\textsf{safeContent} : F_{K^3_{f^\star,e}}(i)\right\}\right)$$

Figure 25: Emulated $\textsf{OST}_1$: $\textsf{Find}$ with conjunctive search (part 2).

- $\mathsf{db.collection.find}(K, \{\$\mathsf{and} : [\{f_1, v_1\}, \cdots, \{f_n, v_n\}]\})$:
  - **Server**:
    6. for all $\mathbf{D} \in \mathsf{Result}$,
       (a) let Tags be the array of the $\mathsf{safeContent}$ field of document $\mathbf{D}$;
       (b) for all $f \in \mathbf{EF} \setminus \{f^\star\}$
          i. compute $\mathrm{ct} := \mathsf{SKE.Dec}\left(K_{f,3}, \mathrm{ct}^{\mathsf{edc}}_{f,v}\right)$ and parse ct as $\mathrm{ct}_{f,v} \| x$;
          ii. parse x as $\mathsf{count} \| K^{\mathsf{edc}} \| K^{\mathsf{esc}} \| K^{\mathsf{ecc}}$;
          iii. if $f \in \mathbf{HC}$, then set $U := \{1, \cdots, p\}$, otherwise set $U := \{0\}$;
          iv. set $\mathsf{flag} := \mathsf{false}$;
          v. for all $u \in U$,
             A. compute $K^3_{f,v} := F\left(F_{K^{\mathsf{edc}}_{f,v}}(u), 1\right)$;
             B. if $F_{K^3_{f,v}}(\mathsf{count}) \in \mathsf{Tags}$, set $\mathsf{flag} := \mathtt{true}$ and exit the loop;
          vi. if $\mathsf{flag} = \mathsf{false}$ remove $\mathbf{D}$ from Result;
       (c) for all $f \in \mathbf{RF} \setminus \{f^\star\}$
          i. compute $\mathrm{ct} := \mathsf{SKE.Dec}\left(K_{f,3}, \mathrm{ct}^{\mathsf{edc}}_{f,v}\right)$ and parse ct as $\mathrm{ct}_{f,v} \| x_1 \| \cdots \| x_n$;
          ii. for $1 \le i \le n$, parse x_i as $\mathsf{count}_i \| K^{\mathsf{edc}}_i \| K^{\mathsf{esc}}_i \| K^{\mathsf{ecc}}_i$;
          iii. if $f \in \mathbf{HC}$, then set $U := \{1, \cdots, p\}$, otherwise set $U := \{0\}$;
          iv. set $\mathsf{flag} := \mathsf{false}$;
          v. for all $e \in \mathbf{E}_{f,v}$ and $u \in U$,
             A. compute $K^3_{f,e} := F\left(F_{K^{\mathsf{edc}}_{f,e}}(u), 1\right)$;
             B. if there exists $i \in [n]$ such that $F_{K^3_{f,e}}(\mathsf{count}_i) \in \mathsf{Tags}$, set $\mathsf{flag} := \mathtt{true}$ and exit the loop;
          vi. if $\mathsf{flag} = \mathsf{false}$ remove $\mathbf{D}$ from Result;
    7. send Result to client.
  - **Client** (Part 2):
    1. similar to Client (Part 2) of Figure 18.

Figure 26: Emulated OST_1: Find with conjunctive search (Part 3).

- $\mathsf{db.collection.compact}(\mathbf{F})$:
  - **Client**:
    1. parse K as $\left(K_f\right)_{f\in\mathbf{F}}$;
    2. for all $f \in \mathbf{F}$, compute $K_f^{\mathsf{ecoc}} := F(F_{K_f}(1), 4)$;
    3. send $(K_f^{\mathsf{ecoc}})_{f\in\mathbf{F}}$ to the server.
  - **Server**:
    1. compute $\mathtt{db.ecoc.renameCollection}(\text{“}\mathtt{ecoc}^\star\text{”})$;
    2. compute $\mathtt{db.createCollection}(\text{“}\mathtt{ecoc}\text{”})$;
    3. initialize a set C and compute $\mathsf{Result} := \mathtt{db.ecoc}^\star\mathtt{.find}()$;
    4. for all $\mathbf{D} \in \mathsf{Result}$,
       (a) parse $\mathbf{D}$ as $\{\mathsf{\_id} : r, \mathsf{field} : f, \mathsf{value} : \mathrm{ct}\}$;
       (b) compute $K^{\mathsf{esc}} \| K^{\mathsf{ecc}} := \mathsf{SKE.Dec}\left(K_f^{\mathsf{ecoc}}, \mathrm{ct}\right)$;
       (c) if $K^{\mathsf{esc}} \| K^{\mathsf{ecc}} \notin C$,
          i. add $K^{\mathsf{esc}} \| K^{\mathsf{ecc}}$ to C and compute

          $$K^1 := F_{K^{\mathsf{esc}}}(1) \qquad K^2 := F_{K^{\mathsf{esc}}}(2) \qquad K^3 := F_{K^{\mathsf{ecc}}}(1) \quad \text{and} \quad K^4 := F_{K^{\mathsf{ecc}}}(2)$$

          ii. set $\mathsf{flag}_1 := \mathtt{true}$, $\mathsf{count} := 0$, and $i := 2$;
          iii. while $\mathsf{flag}_1 = \mathtt{true}$,
             A. compute $\mathsf{tag} := F_{K^1}(i)$;
             B. compute $\mathbf{r} := \mathtt{db.esc.find}(\{\mathsf{\_id} : \mathsf{tag}\})$
             C. if $\mathbf{r} \neq \perp$,
                A. parse $\mathbf{r}$ as $\{\mathsf{\_id} : \mathsf{tag}, \mathsf{value} : \mathrm{ct}\}$,
                B. set $t := \mathrm{ct}$ and set $i := i + 1$;
                C. compute $\mathtt{db.esc.delete}(\{\mathsf{\_id} : \mathsf{tag}\})$
             D. otherwise set $\mathsf{flag}_1 := \mathsf{false}$;
          iv. if $t \neq \perp$, compute $\mathsf{count} := \mathsf{SKE.Dec}\left(K^2, t\right)$;

Figure 27: Emulated OST_1: Compaction (part 1).

- db.collection.compact($\mathbf{F}$):
  - **Server**:

    v. if count > 0,
      A. set $\mathsf{flag}_2 :=$ `true`, initialize a set $\mathbf{g}$, and set $j := 1$;
      B. while $\mathsf{flag}_2 =$ `true`,
        A. compute tag $:= F_{K^3}(j)$
        B. compute $\mathbf{r} :=$ db.ecc.`find`({_id : tag})
        C. if $\mathbf{r} \neq \perp$,
          A. parse $\mathbf{r}$ as {_id : tag, value : ct},
          B. add SKE.Dec$\left(K^4, \mathrm{ct}\right)$ to $\mathbf{g}$ and set $j := j + 1$;
        C. otherwise set flag := false;
      D. compute $\mathbf{g}' \leftarrow$ Merge($\mathbf{g}$);
      E. if $\mathbf{g}' \neq \mathbf{g}$,
        A. compute for all $k \in \{1, \cdots, j - 1\}$

$$\mathsf{db.ecc.delete}\left(\{\_\mathsf{id} : F_{K^3}(k)\}\right)$$

      B. if $\{1, \cdots, \mathsf{count}\} \setminus \mathbf{g}' \neq \emptyset$,
        A. if $\mathbf{g}' \neq \mathbf{g}$, compute for all $k \in [\#\mathbf{g}']$,

$$\mathsf{db.ecc.insert}\left(\left\{\_\mathsf{id} : F_{K^3}(k), \mathsf{value} : \mathsf{SKE.Enc}\left(K^4, g_k\right)\right\}\right)$$

        B. compute db.edc.update({_id : $F_{K^1}(1)$}, {\$set : {f : Enc_{K^2}(count)}})
      C. otherwise, compute db.esc.delete({_id : $F_{K^1}(1)$})

    5. compute db.ecoc*.drop().

Figure 28: Emulated OST_1: Compaction (part 2).

- EmuBinary(K, "coll"):
  1. compute $\mathbf{r} :=$ db.coll.count();
  2. parse $\mathbf{r}$ as {"n" : ρ};
  3. set flag := `true`;
  4. while flag = `true`,
     (a) compute $\mathbf{r} :=$ db.coll.`find`({_id : $F_K(\rho)$});
     (b) if $\mathbf{r} \neq \perp$, then set $\rho := 2\rho$, otherwise set flag := false;
  5. set $i := 0$, median := 0, min := 1 and max := ρ;
  6. for $1 \leq i \leq \lceil \log(\rho) \rceil$,
     (a) set median := $\lceil$(max − min)/2$\rceil$ + min;
     (b) compute $\mathbf{r} :=$ db.coll.`find`({_id : F_K(median)});
     (c) if $\mathbf{r} \neq \perp$,
        i. set min := median;
        ii. if $i := \lceil \log(\rho) \rceil$, then set $i :=$ min;
     (d) otherwise if $\mathbf{r} := \perp$,
        i. set max := median;
        ii. if $i := \lceil \log(\rho) \rceil$,
           A. compute $\mathbf{r} :=$ db.coll.`find`({_id : F_K(min)})
           B. if $\mathbf{r} \neq \perp$, then set $i :=$ min;
  7. output i.

Figure 29: The emulated binary search subroutine.

- $\mathsf{Mincover}_{\mathsf{SPH}}([a,b], \mathsf{ntype})$:
  1. compute $a_0 \cdots a_n \leftarrow \mathsf{BitRep}(a, \mathsf{ntype})$;
  2. compute $b_0 \cdots b_n \leftarrow \mathsf{BitRep}(b, \mathsf{ntype})$;
  3. set $N := 2^n$;
  4. parse ntype as $(\mathsf{precision}, \mathsf{lBound}, \mathsf{uBound}, \theta)$;
  5. output $\mathsf{Mincover}_{N,\theta}([a_0 \cdots a_n, b_0 \cdots b_n])$.
- $\mathsf{Mincover}_{N,\theta}([a_0 \cdots a_i, b_0 \cdots b_i])$:
  1. find $j \in \{0, \cdots, i\}$ s.t. $a_0 \cdots a_j := b_0 \cdots b_j$ and set $x := a_0 \cdots a_j$;
  2. if $[x\mathbf{0}^{\log N - |x|}, x\mathbf{1}^{\log N - |x|}] \subseteq [a_0 \cdots a_i, b_0 \cdots b_i]$,
     (a) find λ s.t. $2^{\lambda \cdot \theta} \leq 2^{\log N - |x|} < 2^{(\lambda+1)\theta}$;
     (b) set $y := \log N - |x| - \lambda \cdot \theta$;
     (c) if $y \neq 0$, for all $p_1 \cdots p_y \in \{0,1\}^y$, add $xp_1 \cdots p_y$ to $\mathbf{C}$, otherwise add x to $\mathbf{C}$;
  3. otherwise if $|x| < \log N$,
     (a) compute $\mathsf{Mincover}_{N,\theta}([a_0 \cdots a_i, x0\mathbf{1}^{\log N - |x| - 1}])$;
     (b) compute $\mathsf{Mincover}_{N,\theta}([x1\mathbf{0}^{\log N - |x| - 1}, b_0 \cdots b_i])$.

Figure 30: The minimum cover algorithm.

- $\mathsf{Edges}_{\mathsf{SPH}}(v, \mathsf{ntype})$:
  1. compute $a_0 \cdots a_n \leftarrow \mathsf{BitRep}(v, \mathsf{ntype})$
  2. initialize an empty set $\mathbf{C}$ and set $N := 2^n$;
  3. parse ntype as $(\mathsf{precision}, \mathsf{lBound}, \mathsf{uBound}, \theta)$;
  4. for $i := 0$ to $\log N$,
     (a) if $\log N - i \in \{i \cdot \theta\}^{\lfloor \theta^{-1} \cdot \log N \rfloor}$, then add $a_0 \cdots a_i$ to $\mathbf{C}$;
  5. output $\mathbf{C}$.

Figure 31: The edges extraction algorithm.

- $\mathsf{BitRep}(v, \mathsf{ntype})$:
  1. parse ntype as $(\mathsf{precision}, \mathsf{lBound}, \mathsf{uBound}, \theta)$;
  2. compute $x := v \cdot 10^{\mathsf{precision}} - \mathsf{lBound} \cdot 10^{\mathsf{precision}}$;
  3. compute $N := \mathsf{uBound} \cdot 10^{\mathsf{precision}} - \mathsf{lBound} \cdot 10^{\mathsf{precision}}$;
  4. let $b_1 \cdots b_{\log N}$ be the binary representation of x s.t. $x := \sum_{i:=1}^{\log N} b_i \cdot 2^{\log N - i}$;
  5. set $b_0 := \perp$;
  6. output $b_0 \cdots b_{\log N}$.

Figure 32: Bit representation algorithm.

hypergraph we refer to as the sparse partition hypergraph. Before describing the SPH we recall the binary partition hypergraph from [KKM21] and its limitations.

Binary partition hypergraph. The binary partition hypergraph $\mathrm{H}_{\mathrm{BP}} = (\mathbb{D}, \mathcal{B}(\mathbb{D}))$ is a hypergraph defined as a collection of subsets $\mathcal{B}(\mathbb{D})$ over a vertex set $\mathbb{D}$. Let $e_{a,w}$ be the set of elements $\{a, a+1, \ldots, a+w-1\}$; that is, the range of width w starting at a. $\mathcal{B}(\mathbb{D})$ is then defined as the collection

$$\mathcal{B}(\mathbb{D}) = \left\{ e_{(k-1)w+1,w} \subseteq \mathbb{D} : w \in \{2^i\}_{i=0}^{\log N} \text{ and } k \in \left\{1, \cdots, \frac{N}{w}\right\}\right\},$$

Throughout, we consider N to be a power of 2. The binary partition hypergraph has $\log N+1$ levels and a total of $2N-1$ edges so when N is large the resulting encrypted range scheme will have large storage overhead.

Sparse partition hypergraph. To address this, the sparse parition hypergraph is designed to only have a θ fraction of the binary partition hypergraph' s levels. We call θ its *sparsity factor* and define four levels of sparsity: (1) *no* sparsity where we set $\theta := 1$; (2) *low* sparsity where we set $\theta := 2$; (3) *medium* sparsity where we set $\theta := 4$; and (4) *high* sparsity where we set $\theta := 8$. The θ-sparse parition hypergraph is formally defined as $\mathrm{H}_{\mathrm{SP}_\theta} = (\mathbb{D}, \mathcal{B}_\theta(\mathbb{D}))$, where:

$$\mathcal{B}_\theta(\mathbb{D}) = \left\{ e_{(k-1)w+1,w} \subseteq \mathbb{D} : w \in \left\{2^{i \cdot \theta}\right\}_{i=0}^{\lfloor \theta^{-1} \cdot \log N \rfloor} \text{ and } k \in \left\{1, \cdots, \frac{N}{w}\right\}\right\}.$$

Note that sparsity could be defined in many ways and that our choice is just one possibility. One could also choose sparsity levels as a function of the data or its distribution. Different variations could lead to an even lower number of levels and edges without increasing the communication and computation complexity of ERX.

Minimum cover. Recall that, given a range r, the minimum cover algorithm $\mathsf{Mincover}_H$ identifies the *minimum* set of edges that cover r. Our minimum cover algorithm works as follows: given a range $r = [a, b]$ such that $0 \le b - a \le N - 1$, it does the following:

1. it computes the binary tree hypergraph minimum cover $\mathbf{C}_r \leftarrow \mathsf{Mincover}_{\mathrm{BT}}$ from [KKM21];
2. for all edges $e \in \mathbf{C}_r$ such that $\#e \notin \left\{2^{i \cdot \theta}\right\}_{i=0}^{\lfloor \theta^{-1} \cdot \log N \rfloor}$, it performs the following
   (a) it removes e from $\mathbf{C}_r$;
   (b) it identifies the smallest width w^* such that $w^* \le \#e$;
   (c) it calculates the ratio $\gamma = \#e/w^*$ and parse e as $(e_1, \cdots, e_\gamma)$ where e_i is the ith block in e with a window equal to w^*;
   (d) it adds $e_1, \cdots, e_\gamma$ to $\mathbf{C}_r$.

$N = 2^8$	No sparsity	Low sparsity	Medium sparsity	High sparsity
Storage	9×	5×	3×	1×
Token size	[1, 16]	[1,32]	[1,64]	[1,128]

Table 1: Costs on domain of size $N = 2^8$ as a function of sparsity.

$N = 2^{32}$	No sparsity	Low sparsity	Medium sparsity	High sparsity
Storage	33×	17×	9×	5×
Token size	[1, 64]	[1,128]	[1,256]	[1,512]

Table 2: Costs on domain of size $N = 2^{32}$ as a function of sparsity.

Computing edges. Recall that given a value v, the Edges_H algorithm identifies all the edges $e \in \mathcal{B}(\mathbb{D})$ such that $v \in e$. For the SPH hypergraph $\mathrm{H}_{\mathrm{SP}_\theta} = (\mathbb{D}, \mathcal{B}_\theta(\mathbb{D}))$ the Edges algorithm outputs

$$\mathbf{E}(v) = \left\{ e_{\lfloor \frac{v}{w} \rfloor + 1, w} \in \mathcal{B}(\mathbb{D}) : w \in \left\{ 2^{i \cdot \theta} \right\}_{i=0}^{\lfloor \theta^{-1} \cdot \log N \rfloor} \right\}.$$

Compactness. We now analyze the compactness of a sparse and bounded-width binary partition hypergraph.

Theorem A.1. *The binary partition hypergraph* $\mathrm{H}_{\mathrm{BP}}^{\alpha,\beta,\theta} = (\mathbb{D}, \mathcal{B}(\mathbb{D}))$ *is*

$$\left(2^{\theta+1} \cdot \log N, 1, 1 + \lfloor \frac{\log N}{\theta} \rfloor, 1 + \lfloor \frac{\log N}{\theta} \rfloor \right)\text{-}compact.$$

Practical considerations. We now consider how the sparsity factor θ impacts the storage overhead and token size of the resulting encrypted range scheme. Tables 1, 2, 3 and 4 summarize the multiplicative storage overhead and the size of the range tokens for $N = 2^{32}$, $N = 2^{64}$ and $N = 2^{128}$ which are the sizes of the numerical data types `int`, `long`, `double` and `decimal` supported by MongoDB. Note that the size of the range token can vary from 1 to $2^{\theta+1} \log N$ depending on the size of the minimum cover.

We stress that the values in the table are worst-case and do not always reflect the real cost if the client is aware of the exact boundaries of the domain. For example, consider the case a field `age` and assume the client knows that the maximum value it can have is 255. While ages will be stored as `int`s the real domain of the field is $\{0, \cdots, 2^8 - 1\}$ which is a non-trivial difference. In Ω_R and, therefore, in OST_1, if the client can provide the *real* domain when creating a collection (in the form of lower and upper bounds in `ntype`) the storage overhead of Ω_R can be significantly reduced from what is described in Table 1.

$N = 2^{64}$	No sparsity	Low sparsity	Medium sparsity	High sparsity
Storage	65×	33×	17×	9×
Token size	[1,128]	[1,256]	[1,512]	[1,1024]

Table 3: Costs on domain of size $N = 2^{64}$ as a function of sparsity.

$N = 2^{128}$	No sparsity	Low sparsity	Medium sparsity	High sparsity
Storage	129×	65×	33×	17×
Token size	[1,256]	[1,512]	[1,1024]	[1,2048]

Table 4: Costs on domain of size $N = 2^{128}$ as a function of sparsity.