Design Overview of OST-1

Version 2

**Differences between v2 and v1***. This document is the second version of the* [*Design Overview of OST-1 version 1*](https://docs.google.com/document/u/0/d/1zSo7C_1TflgkST-6l7HI3Tzl_x_YpeB0qL_8qRm7b9M/edit)*.*

*We have made the following design changes as a response to the questions and concerns raised in the first iteration of this document:*

1. *We changed the design to perform all the updates to the encrypted cache collection (ECC) during the execution of the delete operations instead of during the execution of the read operations.*
2. *We modified the binary search algorithm so that it does not depend on the size of the encrypted state collection (ESC).*
3. *We modified the scheme so that concurrent writes can be executed simultaneously for high-contention fields.*

*Changes 1 and 2 have no impact on the efficiency of the scheme. Change 3 has a small effect on the efficiency of high-contention fields only.*

We also added the following details and examples:

1. *an array field to the examples we use*
2. *a high-level description of the compaction process*

*Please keep in mind that this document is not meant to explain the entire scheme and every design decision. The only purpose is for us to provide a quick overview so we can get quick feedback on any red flags. A more detailed document that explains the entire scheme, the intuition behind how it works and its analysis will be provided.*

In this document, we describe the design of the OST-1 construction using an example. We discuss the following aspects of the construction:

1. The encrypted collection
2. Document insertions
3. Point queries
4. Range queries
5. Conjunctive queries
6. Disjunctive queries
7. Document deletions
8. Document updates
9. Post deletion/update search queries
10. Compaction process

**Disclaimer**. This document is purposefully kept high level and only describes how OST-1 works through an example. It should not be used as a reference to implement the scheme. There are several details that are omitted in order to simplify the exposition. Furthermore, the construction’s security has not been analyzed yet and several changes could occur after its formal analysis has been completed.

# Preliminaries

Let F be a pseudo-random function (PRF) and SKE = (Gen, Enc, Dec) be a symmetric-key encryption scheme composed of a key generation, an encryption and a decryption algorithm. We will use || to denote the concatenation of two bitstrings. Throughout, we consider three clients (C1, C2, C3) connecting to the same MongoDB server.

# The Encrypted Collection

A plaintext collection will be represented and stored using four collections:

* **Encrypted Data Collection (EDC)**: this collection will store the original documents in encrypted form together with some cryptographic metadata. Every document in the plaintext collection is stored in the EDC with the following changes: (1) the values of the encrypted fields are encrypted using a standard symmetric-key encryption scheme; and (2) a new field called **\_safeContent** of type array is added to the document. The **\_safeContent** field is used to store cryptographic metadata and is indexed. Note also that the encrypted fields will now have binary type independently of their original types.
* **Encrypted State Collection (ESC)**: this collection stores state information that is needed for the underlying structured encryption scheme to operate. Note that this collection does not contain any data from the plaintext collection and it may be hidden from the user. The ESC holds two types of documents:
  + *Counter documents*: for every document update on the original plaintext collection---whether it is an insertion or modification---one (or possibly multiple) counter document with two fields, **\_id** and **value**, is inserted in the ESC. Counter documents store counter information that correspond to encrypted field/value pairs in the EDC. These counters are necessary to generate the tags that are stored in the **\_safeContent** field of EDC documents as well as to speedup search operations.
  + *Range documents*: to speed up range searches on encrypted numerical data, cryptographic metadata is inserted in the ESC in the form of documents which have one field, **\_id**, that stores a “range tag”. We sometimes call these documents encrypted range documents (ERDs)
* **Encrypted Cache Collection (ECC)**: this collection is used by the server to store information about field/value pairs that have been deleted for the purpose of speeding up future queries. Like the ESC, this collection does not store any of the original plaintext data and can be hidden from the end user/application.
* **Encrypted Compaction Collection (ECoC)**: this collection is used by the server to store information needed to shrink the size of the ESC and the ECC. In particular it has one type of documents:
  + *Compaction documents*: the ESC and the ECC can grow large after a period of time so we store cryptographic objects to help the server shrink its size once it reaches a certain threshold. We refer to this metadata as compaction ciphertexts and they are stored in compaction documents which have two fields, **\_id** and **value**. The compaction ciphertexts are stored in the **value** field which is of type array. We sometimes call these documents encrypted compaction documents.

# Document Insertions

We now describe how document insertions are made. For our example, we consider a scenario that includes three clients C1, C2 and C3 and three documents:

D1 : { “\_id” : 1,

“first”: “Bob”,

“ssn”: “111-11-1111”,

“phone”: [401-111-1111, 401-222-2222],

“age”: 33

}

D2 : { “\_id” : 2,

“first”: “Alice”,

“ssn”: “222-22-2222”,

“phone”: [917-111-1111],

“age”: 38

}

D3 : { “\_id” : 3,

“first”: “Bob”,

“ssn”: “333-33-3333”,

“phone”: [212-111-1111, 212-222-2222],

“age”: 55

}

We assume that each client inserts a single document and, specifically, that C1 inserts D1, C2 inserts D2 and C3 inserts D3. Moreover, the clients want to encrypt the **first**, **ssn** , **phone** and **age** fields and want to perform point queries on all four and range queries on **age**. Furthermore, **age** has a fixed domain {0, …, 255}.

We also assume that the field **first** is highly-contentious.

In this construction, all clients share a 128-bit key K.

### Insertion of Document D1

We describe how client C1 inserts document D1, explaining first the client-side steps followed by the server-side steps. The same steps will hold for the insertion of the other documents by the other clients.

#### Client Side

Client C1 generates a derived key, Ke:

Ke := F(K, 1),

and encrypts the fields in D1 using the symmetric-key encryption scheme. In practice, this can be instantiated with AES with CTR mode for confidentiality only, or with AES-GCM for both confidentiality and integrity of the value. This results in an encrypted document:

D1 : { “\_id” : 1,  
 “first”: SKE.Enc(Ke, “Bob”),

“ssn”: SKE.Enc(Ke, “111-11-1111”),

“phone”: [SKE.Enc(Ke, “401-111-111”), SKE.Enc(Ke, “401-222-2222”)],

“age”: SKE.Enc(Ke, 33),

“\_safeContent”: [ ]

}

Note that the **\_safeContent** field was added to the document. The client then generates a *put token*, ptk, that enables the server to generate the tags that will be stored in the **\_safeContent** field. The put token consists of six sub-tokens:

ptk := (ptk1, ptk2, ptk3, ptk4, ptk5, ptk6),

where

ptk1 := F(Kf1, first||Bob) and ptk2 := F(Kf1, ssn||111-11-1111), and ptk3 := F(K,2)

with

Kf1 := F(K, 3).

We now describe how the last three sub-tokens, ptk4 , ptk5 and ptk6, are generated. The fourth sub-token is defined as

ptk4 := (ptk41, ptk42),  
  
where ptk41:= F(Kf1, phone||401-111-111) and ptk42:= F(Kf1, phone||401-222-2222).

The fifth sub-token, ptk5, is defined as

ptk5 := (ptk51, RTags),

where

ptk51 := F(F(....F(Kage, b1).., b7), b8),

with Kage := F(Kf2, age), Kf2 := F(K, 4) and (b1, …,b8) is the binary encoding of the age value “33”.

RTags is a list of range tags:

RTags := (RTag1, …, RTag8)

where

RTag1 := F(F(Kage, b1), \*) and RTag2 := F(F(F(Kage, b1), b2), \*) etc,

and \* is a special character different from 0 and 1.

The sixth sub-token, ptk6, is itself composed of four subparts:

ptk6 := (ptk61, ptk62, ptk63, ptk64),

where ptk61 := F(K, 5 || first), ptk62 := F(K, 5 || ssn), ptk63 := F(K, 5|| phone), and ptk64 := F(K, 5|| age).

Finally, the client C1 sends the encrypted document D1 along with the put token ptk to the server. The client also needs to communicate to the server which fields are contentious and to what degree. In our example, the client would have to communicate that the field **first** is contentious and set a contention parameter p = 20 (we describe how contention is handled and how p affects it later). The client could communicate this information to the server in different ways, e.g., via the schema, by adding a field to the documents, or by sending an out of band message. We’re not sure what the tradeoffs are between these different approaches so we leave it open for now.

**Analysis**. The cost of doing a document insertion in terms of client-side computation and document size expansion is:

* Computation: the client needs to perform O(m log N+s) cryptographic operations (i.e., either PRF or encryption operations), where m is the number of encrypted values across all range fields, N the domain size of the range field, and s the number of encrypted values across all fields in a document. In our example m = 1, N = 2^8 and s = 5. Given that all PRF and encryption operations are symmetric-key operations, the execution time is in the order of microseconds on commodity machines.
* Size expansion: the size of the encrypted document is roughly equal to the size of the original plaintext document (the additional expansion is due to encryption). Note that the client also sends the put token which is O(s+m log N), where m is the number of values across all encrypted range fields (which is equal to 1 in our case), N is the domain size of the largest encrypted numerical field, and s the number of values across all encrypted fields.

#### Server Side

Once the server receives the encrypted document D1 along with the put token ptk it needs to execute the following steps:

1. Retrieve and update the relevant counters in the encrypted state collection,
2. Use these counters to generate the cryptographic metadata that will be inserted into the encrypted document,
3. Insert the range tags from the put token into encrypted range document(s),
4. Create the encrypted compaction document(s)

##### Retrieving and Updating Counters

Since there are four encrypted fields with a total of 5 values in this document, the server will need to retrieve and update five counters from the ESC. To do this, it first uses ptk1 to retrieve and update the counter associated with **first||Bob**, then ptk2 to retrieve the counter associated with **ssn||111-11-1111**, then ptk4 to retrieve the counters associated with **phone||401-111-1111** and **phone||401-222-2222**, and ptk51 to retrieve the counter associated with **age||33**. In the following we only show how to retrieve the first counter since the other four can be retrieved in the same way.

The server updates the ESC as follows:

1. Given that first is marked as a contentious field, the server samples a value r, called partition identifier, uniformly at random from {1, …, p},
2. It computes two derived keys K’ and K’’:  
     
    K’ := F(ptk1, 1 || r) and K’’ := F(ptk1, 2)
3. It retrieves the document D\* from the ESC with \_id = F(K’, i\*). The value i\* is, roughly speaking, the number of times the field/value pair first:Bob is inserted or modified since the last compaction. It finds i\* by performing a binary search within the range [1, #UP-ESC], where #UP-ESC is an upper bound on the number of documents in the ESC. The upper bound can be calculated as follows. The server will test for \_id = F(K’, **fastCount**) where **fastCount** is the server’s estimate of the size of the ESC. If there is no document with this \_id, then #UP-ESC := **fastCount**, otherwise the server tests if a document with \_id = F(K’, 2x**fastCount**) exists and so on and so forth until convergence. The worst-case computation is O(log#ESC) where #ESC is the exact number of documents in the ESC.   
     
   Once the upper bound identifier, the binary search will test for a logarithmic number of \_id’s before identifying the last counter. In our case given that this is the first insertion, i\*=0 and counter = 0.
4. It retrieves the counter from D\* by computing counter1 := SKE.Dec(K’’,**value**)
5. It updates the counter by inserting a new document D’ in ESC  
     
   D’ : { “\_id” : F(K’, i\*+1),  
    “value” : SKE.Enc(K”,counter+1)  
    }

It is important that the server does not overwrite an existing document but creates a new one. This is important for security.

**Remark**. For all other fields that are not marked as contentious, Step 1 in the process above does not exist.

##### Generating the Cryptographic Metadata

At this stage the server holds the counters (counter1, counter2, counter31, counter32, counter4) for the five encrypted field/value pairs of the new document. We now describe how the server generates the cryptographic metadata for **first||Bob** but the same steps apply to the other field/value pairs:

1. It computes a new key:  
      
    K’ := F(ptk1, 3)
2. It computes the following  
   ct11 = SKE.Enc(ptk61, (counter1, r) || ptk1) and tag21 = F(K’, counter1 || r)

Once all the tags are generated, the server inserts the following final encrypted document into the encrypted data collection:

D1 :{ “\_id” : 1,  
 “first”: SKE.Enc(Ke, “Bob”) || ct11,

“ssn”: SKE.Enc(Ke, “111-11-1111”) || ct12,

“phone”: [SKE.Enc(Ke, “401-111-111”)||ct131, SKE.Enc(Ke, “401-222-2222”)||ct132],

“age”: SKE.Enc(Ke, 33) || ct14,

“\_safeContent”: [ tag21, tag22, tag231, tag232, tag24]

}

Note that if the field is not contentious then the cryptographic metadata (i.e., the keys, the tags and the ciphertexts) will not be a function of a partition number.

##### Range Tags

During this step, the server inserts each range tag rtag found in ptk into a new document with **\_id** := rtag. If a document with the same **\_id** is already stored in the ESC the server does not insert it.

##### Compaction Tags

The server creates a new compaction document with a unique **\_id** and with a **value** equal to SKE.Enc(ptk3, (ptk1, r) || ptk2 || ptk4 || ptk51) where r is the random value that determines the partition number assigned to **first||Bob**.

**Analysis**. The server-side computation and storage overhead resulting from the insertion of a single document is:

* Computation: the server performs O(s log #ESC) cryptographic operations (i.e., either PRF evaluations or encryption and decryption operations), where s is the number of values across all encrypted fields and #ESC is the size of the encrypted state collection.
* Storage overhead: the size of the encrypted document increases as a function of s. In our example s = 5 and the number of cryptographic metadata objects (i.e., tags and ciphertexts) added is 10. Note that the size of the ESC increases by 5 documents and that the number of ERDs created is at most logarithmic. Once enough numerical values are added to the encrypted data collection, the number of ERDs becomes almost constant with a maximum number equal to O(2N-1). We will provide more details in the full version of this writeup. The number of new ECDs increases linearly with the number of encrypted fields in the document.

### Insertion of the Remaining Documents

The other clients will insert their documents in a similar way. After all the insertions, the server will hold the following documents:

Encrypted Data Collection:

D1 :{ “\_id” : 1,  
 “first”: SKE.Enc(Ke, “Bob”) || ct11,

“ssn”: SKE.Enc(Ke, “111-11-1111”) || ct12,

“phone”: [SKE.Enc(Ke, “401-111-111”) || ct131, SKE.Enc(Ke, “401-222-2222”) || ct132],

“age”: SKE.Enc(Ke, 33) || ct14,

“\_safeContent”: [ tag21, tag22, tag231, tag232, tag24]

}

D2 :{ “\_id” : 1,  
 “first”: SKE.Enc(Ke, “Alice”) || ct31,

“ssn”: SKE.Enc(Ke, “222-22-2222”) || ct32,

“phone”: [SKE.Enc(Ke, “917-111-11111”) || ct33],

“age”: SKE.Enc(Ke, 38) || ct34,

“\_safeContent”: [ tag41, tag42, tag43, tag44]

}

D3 :{ “\_id” : 1,  
 “first”: SKE.Enc(Ke, “Bob”) || ct51,

“ssn”: SKE.Enc(Ke, “333-33-3333”) || ct52,

“phone”: [SKE.Enc(Ke, “212-111-1111”) || ct531, SKE.Enc(Ke, “212-222-2222”) || ct532],

“age”: SKE.Enc(Ke, 55) || ct54,

“\_safeContent”: [ tag61, tag62, tag631, tag632, tag64]

}

Encrypted State Collection:

D1: { “\_id” : F(...),

“value”: SKE.Enc(...)}

…

D14: { “\_id” : F(...),

“value”: SKE.Enc(...)}

ERD1: { “\_id” : rtag1}

…

ERD16: { “\_id” : rtag16}

Note that the ESC is composed of 14 documents representing counter values along with the encrypted range documents. There are 14 counter documents because there are 14 values to be encrypted in total, 5 values in D1, 4 values in D2, and 5 values in D3. Also, the number of ERDs is 16. The reason will be clearer in the formal writeup of the construction but essentially this is because the values 33 and 38 have the first 5 bits in common, whereas the values 33/38 and 55 have the first 3 bits in common.

Encrypted Compaction Collection:

ECD1: { “\_id” : ...,

“value”: comptag1}

ECD2: { “\_id” : ...,

“value”: comptag2}

ECD3: { “\_id” : ... ,

“value”: comptag3}

Encrypted Cache Collection:

This collection is still empty at this stage.

### Insertion Contention

Note that both clients C1 and C3 insert documents with the same **first** value equal to **Bob**. Assuming C1 and C3 do their insertions at the same time, we show how the probability of contention will be reduced. Recall that **first** was assigned a partition size of p = 20.

For each client, the server samples a value r from {1, …, 20} uniformly at random. The value r is then used in the tag generation process in a way that guarantees that tags for the same field/value pair but with different r’s will have different tags. The goal is to assign two concurrent inserts different partition values and therefore different tags. Since the partition values r are chosen uniformly at random over {1, …, p}, the probability that two inserts for the same field/value pairs lead to the same tag is 1/p.

# Point Search

In this section we describe how a client performs a point query. We use the following example:

*C1 searches for all documents where ssn: 111-11-1111.*

**Note**. For this section, we assume that no document has been deleted from the database. We will describe how point queries are made post-deletion in a later section.

## Client Side

Given the key K, the client computes a search token stk:

stk = F(Kf1, ssn||111-11-1111),

where Kf1 = F(K, 3).

The client sends stk to the server.

## Server Side

Given the search token stk the server performs the following three steps:

* It retrieves the counter from the ESC,
* It generates the necessary tags
* It finds and returns the documents that store the tags

To retrieve the counter from the ESC, the server:

* computes two keys K’ and K’’ such that  
    
   K’ := F(stk, 1) and K’’ := F(stk, 2)
* Similar to document insertion, it retrieves the document D\* from the ESC where \_id = F(K’, i\*) by performing a binary search in the range [1, #UP-ESC]. The binary search will test for a logarithmic number of \_id’s before identifying the last counter. In our example, i\*=0 since there is a single document with ssn:111-11-1111 and that field/value pair was inserted or modified only once.
* retrieves the counter from D\* by computing counter := SKE.Dec(K’’,value).

**Remark 2**. Note that for the case of high-contention fields like first, the above computation would be performed p times (e.g., in our case p=20) so that each partition is “searched”. Note that the partitions can be searched in parallel.

The server then:

1. computes K’’’ = F(stk, 3),
2. from i=0 to counter-1, it computes  
     
    tagi := F(K’’’, i)

and finds the documents in the encrypted data collection whose **\_safeContent** field includes tagi

Finally, the server sends all the documents back to the client who decrypts the encrypted fields using Ke = F(K,1) and removes the **\_safeContent** field and the cryptographic metadata.

**Remark 2**. Note that Step 2 above can be parallelized.

**Analysis**. The client-side computation is optimal. It includes the time needed to decrypt the matching documents and two PRF evaluations. The server-side computation is O(log#ESC + #docs) where #ESC is the number of documents in the encrypted state collection and #docs is the number of matching documents. Whenever counter >= log#ESC, the search overhead is asymptotically optimal.

For high-contention fields, the server-side computation is O(p log#ESC +#docs). Whenever #docs >= p log#ESC, the search overhead is asymptotically optimal.

# Range Search

In this section we describe range searches. We use the following example:

*C1 searches for all documents where age is at least 32 and less than 52.*

**Note**. Again, we assume no deletions have been performed.

## Client Side

Given the key K, the client generates a search token as follows:

1. It generates the minimum cover for the range r = [32, 51] over a binary tree built over the domain [0,255]. In our example, the minimum cover includes two nodes [[1]](#footnote-0): node1 = (00100) and node2 = (001100).
2. It “encrypts” the nodes in the minimum cover:

enode1 := F(F(F(F(F(Kage,0),0),1),0),0)

enode2 := F(F(F(F(F(F(Kage,0),0),1),1),0),0),

where Kage := F(Kf2, age) and Kf2 := F(K,4).

1. It sends stk := ((enode1, 5), (enode2,6)) to the server, where 5 and 6 are the depth of node1 and node2 in the binary tree, respectively.

## Server Side

Once the server receives the search token stk it uses it together with the ERDs to identify the values in the range r = [32, 51] that exist in the database. More precisely, it uses information in the search token to derive a “tree of keys”. Since this tree can be very large it uses the information encoded in the ERDs to avoid generating sub-trees that are not relevant to the current range query. At a high level, the server does the following:

1. For each (enode, level) pair in the search token,
   1. it initializes a queue Q that holds enode,
   2. While Q is non-empty:
      1. it dequeues an encrypted node enode’
      2. it computes rtag := F(enode’,\*),
      3. if rtag is in the **\_id** field of an ERD and if level < log N,
         1. it enqueues enode1 := F(enode,1) and enode2 := F(enode,0) to Q

Note that, for each (enode, level) pair, the server performs a pre-order traversal of a “derived” tree of keys but uses the ERDs to avoid sub-trees that are not relevant. By “derived” here we mean that the tree is never completely instantiated. Instead, its nodes are derived on the fly as the tree is traversed. At the end of this process, the server will identify a subset of all the possible leaves. These leaves are related to the values of the domain that are in the query range. More precisely, these leaves can be used to generate the tags for the values in the domain that are within the queried range. In our example, there will be two leaves leaf1 and leaf2 such that F(leaf1, \*) and F(leaf2, \*) are the **\_id** of two ERDs. One would be for the value 38 while the other for value 33.

Once the server identifies these matching leaves, it executes a point search on each one using the leaf as a search token. More precisely, for each matching leaf it does the following:

1. It computes two keys K’ and K’’ such that  
     
    K’ = F(leaf1, 1) and K’’= F(leaf1, 2)
2. Similar to document insertion, it retrieves the document D\* from the ESC with \_id = F(K’, i\*) by performing a binary search within the range [1, #UP-ESC]. The binary search will test for a logarithmic number of **\_id**’s before identifying the last counter. In our case, i\* = 0 since there is only one document with age = 38 and that the age/38 pair was inserted or modified just once.
3. It retrieves the counter from D\* by computing counter1 := SKE.Dec(K’’,value).

To generate the tags, the server:

1. computes K’’’ := F(leaf1, 3),
2. from i = 0 to counter1-1, it computes  
     
    tagi := F(K’’’, i),

and finds the documents in the encrypted data collection that store tagi in their **\_safeContent** field.

Finally, the server sends all the matching documents back to the client who decrypts the encrypted fields using Ke = F(K,1) and removes the **\_safeContent** field and any padding used.

**Remark 1**. Like point searches, range searches can be parallelized both during the tree traversal and the search tag generation.

**Analysis**. The computation and search token size overhead are as follows:

* Computation: the client performs O(log^2(N)) cryptographic operations (i.e., PRF evaluations) since, in the worst case, the size of the minimum cover is logarithmic and the number of PRF evaluations per node in the minimum cover is also logarithmic. In our example, the minimum cover included two nodes and the client evaluated 5 PRFs for the first one and 6 for the second one.[[2]](#footnote-1) The server computation is O(m logN + m log#ESC +#docs) where m is the number of numerical values that exist in the database and the range query, N is the domain size, #ESC the number of documents in the encrypted state collection and #docs is the number of matching documents. In our example, m = 2, #docs = 2 and #ESC = 14.
* Search token size: the size of the search token is logarithmic in the worst case.

# Conjunctive Search

In this section we describe how to perform conjunctive queries. There are two possible forms of conjunctions: (1) encrypted conjunctions; and (2) mixed conjunctions. The former are when all fields in the query are encrypted whereas the latter are when the conjunction includes both encrypted and plaintext fields. We start with encrypted conjunctions and show how mixed conjunctions can be viewed as the former.

We consider the following example:

*C1 wants to find all documents where first:Bob and ssn:111-11-1111*

## Client Side

Given the key K, the client computes a search token stk as follows:

stk := (stk1, stk2),

where

stk1 := (stk11, stk12) = (F(Kf1, ssn||111-11-1111), F(K, 5||ssn))

and

stk2 := (stk21, stk22) = (F(Kf1, first||Bob), F(K, 5||first))

with Kf1 =F(K,3).

The client sends the search token to the server. Recall that the field “first” is highly contentious.

## Server Side

Once the server receives the search token stk it performs the following three operations:

* it retrieves the counters associated with the search terms. Using these counters, the server can figure out which of the two terms is the least frequent.[[3]](#footnote-2) This matters because if the server starts by searching for the less frequent term, the computational cost of the entire query execution will be less.
* it uses the counter of the less frequent term to find the documents that match it
* It then uses another part of the search token to remove the documents that do not match the second term

The server first retrieves the counters from the ESC:

1. It computes two keys K’ and K’’:  
     
    K’ := F(stk11, 1) and K’’ := F(stk11, 2)
2. It retrieves the document D from the ESC with **\_id** = F(K’, i1) by performing a binary search within the range [1, #UP-ESC].
3. It retrieves the counter from D by computing counter1 := SKE.Dec(K’’,value).
4. For i in {1, …, p=20},
   1. It computes two keys K’ and K’’:  
        
       K’’’ := F(stk21, 1 || i) and K’’’’ := F(stk21, 2)
   2. It retrieves the document D from ESC where **\_id** = F(K’’’, i2) by performing a binary search within the range [1, #UP-ESC].
   3. It retrieves the counter from D by computing counter2i := SKE.Dec(K’’’’,value).
5. It orders the counters in increasing order such that (counter1, \sum\_i(counter2i)). Note that we have inserted two documents where “first”:”Bob” and only one document with “ssn”: 111-11-1111. Therefore \sum\_i (counter2i) = 2 and counter1 = 1.

Given counter1, the server retrieves all the documents that match the first search token. In particular, it performs the following steps:

1. It computes K’ := F(stk11, 3),
2. from i = 0 to counter1-1, it computes  
    tagi := F(K’, i)

and finds the document whose **\_safeContent** field holds tagi.

Note that D1 will be the only document that matches the first term so let’s recall its structure:

D1 :{ “\_id” : 1,  
 “first”: SKE.Enc(Ke, “Bob”) || ct11,

“ssn”: SKE.Enc(Ke, “111-11-1111”) || ct12,

“phone”: [SKE.Enc(Ke, “401-111-111”)||ct131 , SKE.Enc(Ke, “401-222-2222”)||ct132] ,

“age”: SKE.Enc(Ke, 33) || ct14,

“\_safeContent”: [ tag21, tag22, tag231, tag232, tag24]

}

The server does the following for all the encrypted documents that result from step 2 above (in our example just D1):

1. It decrypts ct12   
    (counter\*, r)||ptk := SKE.Dec(sk12, ct12)
2. It computes K’ := F(sk21,3),
3. It computes tag\* := F(K’, counter\* || r)
4. If tag\* does not belong to **\_safeContent**, it removes the document from the result

In our case, tag\* does belong to **\_safeContent** so document D1 will be kept and sent to the client.

**Remark**. In the example above, **first** is a contentious field which is why the partition number r is included in the PRF evaluation in Step 3. If the field wasn’t contentious, then PRF evaluation in Step 3 would not include a partition number.

**Mixed conjunctions**. Mixed conjunctions work the same way as encrypted conjunctions except that the server loses its ability to order the terms. In particular, the server can no longer obtain a total order on the selectivity of the terms. The reason is that there is no plaintext counter information maintained by the server. However, the server can always have a partial order where the the counters of the encrypted fields are ranked.

As a result, the server can arbitrarily pick a plaintext term to start with or the encrypted term with the least-selectivity (in case there is more than one encrypted field in the conjunction).

**Limitations**. It is possible to include an encrypted range term in the conjunction but, in this case, the server has to start with the range term independently of whether the counters are smaller or larger than the counters of the other terms. So, in this case, the server no longer needs to retrieve any counter information to identify the least frequent term.

If the client queries for a conjunction of two range terms, the server will need to perform two separate range queries and then perform an intersection of the results.

If the client queries for an encrypted conjunction composed of two or more encrypted range terms and multiple point terms, then the server will need to perform two or more separate range queries and then perform the intersection of the results. The server will then filter out this intersection similar to an encrypted conjunction with point terms only.

**Analysis**. The client and server computation and the size of the search token for encrypted conjunctions is:

* Computation without range terms: the client performs a number of cryptographic operations (i.e., PRF evaluations) that is equal to the number of terms in the conjunction. The server performs O(s log#ESC + s #LeastSel) cryptographic operations, where s is the number of encrypted term, #ESC the size of the encrypted state collection and #LeastSel is the number of documents that match the least selective term. In the case of mixed conjunctions, the cost is O(s #Sel) where #Sel denotes the number of documents that match a randomly selected term in the conjunction. Note that we assume that none of the encrypted fields is highly-contentious.
* Computation with *one* range term: the client performs O(log^2 N +s1) cryptographic operations, where s1 is the number of encrypted point terms in the conjunction. The server cost is the cost of a range query (see Range Search section) plus O(s), where s is the number of terms in the conjunction. There are many different forms of conjunctions (e.g., encrypted, mixed, with or without ranges, with one range term or multiple range terms, with or without contention) which we don’t cover here but will in the more detailed writeup.
* Size overhead: the size of the token is linear in the number of encrypted terms. Depending on whether the term is an encrypted point or range term, the size will vary.

# Disjunctive Search

To handle disjunctions, we execute each term (e.g., a point or a range query) separately and the server returns the union of the results (removing any duplicates).

# Document Deletions

In this section, we describe how to do deletions. We detail the process using the following example:

*C1 deletes document D1.*

## Client Side

Given the key K, the client computes a delete token dtk:

dtk := (dtk1, dtk2, dtk3, dtk4, dtk5),

where dtk1 := F(K, 5 || first), dtk2 := F(K, 5 || ssn), dtk3 := F(K, 5 || phone), dtk4 := F(K, 5|| age) and dtk5 = F(K,2).

where Kf1 = F(K, 3).

The client sends dtk to the server.

## Server Side

Given the delete token dtk the server performs the following four steps:

* It retrieves the document to be deleted,
* It parses all the encrypted values and extract the counter, the put token and, in the case of a high-contention field, the partition identifier,
* It updates the ECC,
* It adds compaction documents,
* It deletes the document.

More precisely, the server performs the following computation:

* It retrieves D1. We recall its structure below:

D1 :{ “\_id” : 1,  
 “first”: SKE.Enc(Ke, “Bob”) || ct11,

“ssn”: SKE.Enc(Ke, “111-11-1111”) || ct12,

“phone”: [SKE.Enc(Ke, “401-111-111”) || ct131, SKE.Enc(Ke, “401-222-2222”) || ct132],

“age”: SKE.Enc(Ke, 33) || ct14,

“\_safeContent”: [ tag21, tag22, tag231, tag232, tag24]

}

* For each encrypted field, the server decrypts the ciphertext and performs the following operations. Here we focus on **first** but the same operations are performed on the other fields:
  + It computes   
     (counter1, r1) || ptk1:= SKE.Dec(dtk1, ct11)  
    where counter1 is equal to 0 here since it was the first document to be inserted for the partition r1.
  + It adds a compaction document to the ECoC where the value field is equal to SKE.Enc(dtk5, (ptk1,r)).
  + It updates the ECC. In particular, it first checks if a document has already been created for the field/value pair first:Bob in the ECC. More precisely it:
    - computes K’ := F(dtk,4 || r1) and K’’ := F(dtk,5),
    - retrieves the document D\*i\* from the ECC with \_id = F(K’, i\*) with a binary search similar to the way we access the ESC. Note that in our case, there are no documents yet created in the ECC for **first||Bob**.
    - It creates and inserts the following document into the ECC  
        
      D : {   
       “\_id” : F(K’, 0),  
       “value”: SKE.Enc(K’’, [0,1])  
       }
* It deletes D1.

Note that Step 2 above can be parallelized.

**Remark on gap encodings.** Updating the ECC consists of adding a new encrypted document that stores information about the counters of the deleted tags. When a tag is deleted, “gaps” are created in the counter space that we use to generate tags. For example, before any deletion occurs, we might use counters 1, …, 100 to generate tags for a field/value pair first:bob (assuming there are 100 people with first name Bob in the collection). Now suppose we deleted the 5th, 17th, 18th, 19th and 53rd Bobs from the collection. It follows then that the tags generated with counter values 5, 17, 18, 19 and 53 are no longer present/used. To avoid generating and searching for these tags at query time, we will store information in the ECC that tells us which counters to avoid. We view these counters as gaps to avoid in the counter space, e.g., 5 is a gap of size 1 whereas {17, 18, 19} is a gap of size 3. We encode these gaps as [a,b] where a is the start of the gap and b is the size of the gap.

Below we describe how the ECC is updated. Note that updating the ECC occurs when a client deletes a document or when a client modifies an existing document.

**Remark**. Note that the documents in the ECC may have different sizes. This will depend on the pattern of deletes for the particular field/value pair of that document. In practice, it is possible to make all the ECC documents the same size, but in this case one field/value pair will have several documents associated with it. We will provide more details in the full version of this writeup.

**Analysis**. The client-side computation is optimal since it simply requires four PRF evaluations which are independent of the size of the deleted document. The server-side computation is O(s log#ECC ) where #ECC is the number of documents in the encrypted cache collection. For the sake of simplicity, we have assumed in this calculation that none of the fields is a high-contention field.

# Document Updates

In this section we describe how a client updates a value in an existing document. In the following, we will consider the following update:

*Client C1 updates the ssn value of document D2 to 444-44-4444.*

As in the previous section, we describe the client-side and server-side work needed to execute this update.

## Client Side

Given the key K, the client generates an update token utk which is composed of four parts:

utk = (utk1, utk2, utk3, utk4),

where

utk1 := SKE.Enc(Ke, 444-44-4444),

utk2 := F(Kf1, ssn||444-44-4444),

utk3 := F(K,2),

utk4 := F(K, 5||ssn),

and Ke := F(K,1) and Kf1 := F(K,3).

The client sends the update token to the server.

## Server Side

Once the server receives the update token utk it performs similar operations to those performed at insertion time, namely:

1. It retrieves and updates the counters in the encrypted state collection,
2. It generates new tags and uses them to replace old tags in the encrypted document,
3. It creates new encrypted compaction document(s)
4. It updates the encrypted cache collection[[4]](#footnote-3)

### Updating the ESC

This is done exactly as in the Document Insertion section. The only difference is “notational” in the sense that ptk1 is replaced by utk2. Note that since the field “ssn” is not a high-contention field like the field “first”, we don’t need the first step in that process. The result of this phase is a new document with an updated counter that is inserted in the ESC.

### Generating and Finding Tags

The purpose of this step is to generate the tags for the updated value while also finding and removing the tags for the old value. In our example, the old ssn value for document D2 was 222-22-2222 but the challenge here is that neither the client nor the server know this value at this stage because the client is stateless (modulo the secret key K) and the server does not have access to the plaintext data.

We enable the server to find the old tag by allowing it to decrypt the ciphertext ct32 that was appended to the value of the **ssn** field. Recall that the **ssn** value in D2 has the following format:

“ssn”: SKE.Enc(Ke, “222-22-2222”) || ct32.

So the server will decrypt the appended ciphertext to reveal a counter/tag pair:

counter\*|| ptk\* := SKE.Dec(utk4, ct32).

The server then finds and deletes F(F(ptk\*,3),counter\*) from the **\_safeContent** field.

The server generates a new tag and ciphertext to replace the old tag and ciphertext as follows:

1. It computes a new key:  
      
    K’ := F(utk2, 3),
2. and a new tag and ciphertext:  
   ct1 = SKE.Enc(utk4, counter || utk2) and tag2 = F(K’, counter),

where counter is the value obtained by the server after updating the ESC.

Finally, the server overwrites the old **ssn** value with utk1|| ct1 and adds tag2 to the **\_safeContent** field.

### Updating the ECC

This is done exactly as in the Document Deletion section. The only difference is that instead of updating the ECC for all the field/values, the update only occurs for the field value ssn||222-22-2222.

### Compaction Ciphertexts

The details of this step are similar to those presented in the previous section. In particular, the server simply creates a new encrypted compaction document with the **value** field equal to SKE.Enc(utk3, utk2).

**Remark1**. The steps outlined above can be used for any document update; even the ones that require a search operation. For example, if client C1 wants to update the ssn of any document where “first”:”Bob” the update steps of this operation will be the same as the ones described above.

**Analysis**. The computation and storage overhead incurred by updating a single document are:

* Computation: the client does O(s1 log(N)+s2) cryptographic operations (i.e., PRF evaluations or encryptions), where s1 is the number of updated numerical fields, N is the maximum domain size over all the encrypted numerical fields, and s2 is the number of the remaining fields. In our example, s1 = 0 and s2 = 1. The number of cryptographic operations performed by the server is O((s1+s2) log(#ESC)), where #ESC is the number of documents in the encrypted state collection.
* Storage overhead: the size of the update token utk is O(s1 logN + s2) while the added storage at the server is between O(s1 + s2) and O(s1 logN + s2) depending on the number of previously-stored numerical values in the database.

# Post-Update/Delete Search

A search that occurs after a deletion or an update is exactly the same as the one described in the Point/Range Search sections except that the server needs to retrieve the deleted counters from the ECC as well. We detail below the process of a point search. For this, we use the following example:

*C1 searches for all documents where ssn: 111-11-1111.*

Note that we have deleted document D1 and therefore the search shouldn’t return any document.

## Client Side

Given the key K, the client computes a search token stk:

stk = F(Kf1, ssn||111-11-1111),

where Kf1 = F(K, 3).

The client sends stk to the server.

## Server Side

Given the search token stk the server performs the following four steps:

* It retrieves the counter from the ESC,
* It retrieves the delete encoding from the ECC
* It generates the necessary tags
* It finds and returns the documents that store the tags

To retrieve the counter from the ESC, the server:

* computes two keys K’ and K’’ such that  
    
   K’ := F(stk, 1) and K’’ := F(stk, 2)
* retrieves the document D\* from the ESC where \_id = F(K’, i\*) by performing a binary search in the range [1, #UP-ESC].
* retrieves the counter from D\* by computing counter := SKE.Dec(K’’,value).

The server then retrieves the delete encoding from the ECC as follows:

* computes two keys K’ and K’’ such that  
    
   K’ := F(stk, 4) and K’’ := F(stk, 5)
* retrieves the document D\* from the ECC where \_id = F(K’, i\*) by performing a binary search in the range [1, #UP-ECC].
* retrieves the delete encoding from D\* by computing deletedCounters := SKE.Dec(K’’,value).

The server then:

1. computes K’ = F(stk, 3),
2. For all i in {0,...,counter -1} \ {deletedCounters}, it computes  
     
    tagi := F(K’’’, i)

and finds the documents in the encrypted data collection whose **\_safeContent** field includes tagi. Note however that in our example {0, …, counter-1} \ {deletedCounters} is an empty set.

# Compaction

Compaction is a process that shrinks the size of the ESC and the ECC. More precisely, it removes any lingering information in the ESC and ECC about field/value pairs that have been deleted from the EDC. This is illustrated in the Document Delete section where we deleted documents with ssn:111-11-1111. After this operation, the EDC no longer contained documents with that field/value pair but the ESC and the ECC still had information about the insertion and deletion of these documents, respectively.

We describe the compaction process below for fields that support point queries and that are not contentions. Note that, as described here, the compaction process will block queries and insertions to the collection. In the formal document, we will describe how to execute compaction without blocking queries and insertions. We will do this by creating a temporary version of the structure/collection to hold new pairs while the original structure is being compacted. There are some additional tricks needed to make sure the original structure is still queryable but that’s too long to describe here.

## Client Side

Given the key K, to initiate a compaction, the client sends the following compaction token to the server:

ctk := F(K,2).

## Server Side

Once the server receives the compaction token, it performs the following steps:

* For every document in the encrypted compaction collection the (ECoC), the server performs the following (while skipping redundant put tokens):
  + It decrypts the value field and retrieves a vector of put tokens   
     (ptk1|| …|| ptkm):= SKE.Dec(ctk, value)
  + For i=1 to m, it:
    - computes two keys K’ and K’’ such that  
        
       K’ := F(ptki, 1) and K’’ := F(ptki, 2)
    - retrieves the document D\*1 from the ESC where \_id = F(K’, i\*1) by performing a binary search in the range [1, #UP-ESC].
    - retrieves the counter counter:=SKE.Dec(K’’, value)
    - computes two keys K’’’ and K’’’’ such that  
       K’’’ := F(ptki, 4) and K’’’’ := F(ptki, 5)
    - retrieves the document D\*2 from the ECC where \_id = F(K’, i\*2) by performing a binary search in the range [1, #UP-ECC].
    - computes the delete encoding such that deletedCounters:=SKE.Dec(K’’’’, value) where deletedCounters corresponds to the set of *deleted* counters.
    - If {0, …, counter-1} \ {deletedCounters} is non-empty,
      * Create a new document D\*\*1 (in the temporary ESC) such that  
          
        D\*\*1: { \_id : F(K’, \bot),  
         value: SKE.Enc(K’’, counter) }
      * Create a new document D\*\*2 (in the temporary ECC) such that  
          
        D\*\*2: { \_id : F(K’’’, \bot),  
         value: SKE.Enc(K’’’’, deletedCounter) }
    - Otherwise if {0, …, counter-1} \ {deletedCounters} is empty, do nothing
* Delete all compaction documents in the ECoC
* Delete the old versions of the ESC and the ECC

The compaction process for contentious fields and range fields will be described in the formal document.

1. We will provide more details on how minimum covers are computed in the full version of the writeup. Note that the size of the minimum cover can range from 1 node to log N nodes, where N is the size of the domain. [↑](#footnote-ref-0)
2. There is a way to reduce this computational overhead when the nodes in the minimum cover share a common ancestor. In our example, the two nodes share the first three bits which means that we can reduce the number of PRF evaluations by 3. [↑](#footnote-ref-1)
3. Note that the counter information might not be sufficient to identify which of the terms is the least frequent. This is the case when accounting for deletions. [↑](#footnote-ref-2)
4. If we change this example to include an update to a range field, then we would have another step to update the encrypted range document which would also require the client to send range tags. [↑](#footnote-ref-3)