`Design Overview of OST-1

[superseded by [version 2](https://docs.google.com/document/d/1njJ-Og6zy1bjDGJC7vtIap9LQiLnWnN-dchhjAjLI1U/edit#) of this document]

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. Document updates
4. Point queries
5. Range queries
6. Conjunctive queries
7. Disjunctive queries
8. Post-deletion queries

**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.

**Note**. This document describes the OST-1 construction at a high-level and we are also going to share a second document that details a second construction called OST-2. Both these constructions are candidates for FLE 2.0.   
  
OST-1 and OST-2 offer different efficiency vs. security trade-offs in that OST-2 provides better security guarantees against more powerful adversaries at the cost of being (slightly) less efficient than OST-1. OST-2 is also technically more complex than OST-1as it involves more structures and techniques than OST-1.   
  
While this is not the final design document, we would like the team to send us as much feedback as possible. In particular, it is important to let us know about any engineering challenges or red flags.

# 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 three 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 types 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 three 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)
  + Compaction documents: the ESC 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 this document the encrypted compaction document (ECD).
* **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.

# 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”,

“age”: 33

}

D2 : { “\_id” : 2,

“first”: “Alice”,

“ssn”: “222-22-2222”,

“age”: 38

}

D3 : { “\_id” : 3,

“first”: “Bob”,

“ssn”: “333-33-3333”,

“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** and **age** fields and want to perform point queries on all three and range queries on **age**. Furthermore, **age** has a fixed domain {0, …, 255}.

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[[1]](#footnote-0). This results in an encrypted document:

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

“ssn”: SKE.Enc(Ke, “111-11-1111”),

“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 five sub-tokens:

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

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 two sub-tokens, ptk4 and ptk5, are generated. The fourth sub-token, ptk4, is defined as

ptk4 := (ptk41, RTags),

where

ptk41 := 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 fifth sub-token, ptk5, is itself composed of three subparts:

Ptk5 := (ptk51, ptk52, ptk53),

where ptk51 := F(K, 5 || first), ptk52 := F(K, 5 || ssn) and ptk53 := F(K, 5|| age).

Finally, the client C1 sends the encrypted document D1 along with the put token ptk to the server.

**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 range fields, N the domain size of the range field, and s the number of encrypted fields in a document. In our example m = 1, N = 2^8 and s = 3. 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 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 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 three encrypted fields in this document, the server will need to retrieve and update three 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**, and ptk41 to retrieve the counter associated with **age||33**. In the following we only show how to retrieve the first counter since the other two can be retrieved in the same way.

The server updates the ESC as follows:

1. It computes two derived keys K’ and K’’:  
     
    K’ := F(ptk1, 1) and K’’ := F(ptk1, 2)
2. It retrieves the document D\* from the ESC with \_id = F(K’, i\*). The value i\* is, roughly speaking, the number of times first=Bob is inserted or modified since the last compaction. It finds i\* by performing a binary search within the range [1, #ESC], where #ESC is the number of documents in the ECS. 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.
3. It retrieves the counter from D\* by computing counter1 := SKE.Dec(K’’,**value**)
4. It updates the counter by inserting a new document D’ in ECS[[2]](#footnote-1)  
     
   D’ : { “\_id” : F(K’, i\*+1),  
    “value” : SKE.Enc(K”,counter+1)  
    }

##### Generating the Cryptographic Metadata

At this stage the server holds the counters (counter1, counter2, counter3) for the three 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(ptk51, counter1 || tag21) and tag21 = F(K’, counter1)

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,

“age”: SKE.Enc(Ke, 33) || ct13,

“\_safeContent”: [ tag21, tag22, tag23 ]

}

##### 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 || ptk2 || ptk41).

**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 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, the number of encrypted fields. In our example s = 3 and the number of cryptographic metadata objects (i.e., tags and ciphertexts) added is 6. Note that the size of the ESC increases by 3 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,

“age”: SKE.Enc(Ke, 33) || ct13,

“\_safeContent”: [ tag21, tag22, tag23 ]

}

D2 : { “\_id” : 2,  
 “first”: SKE.Enc(Ke, “Alice”) || ct31,

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

“age”: SKE.Enc(Ke, 38) || ct33,

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

}

D3 : { “\_id” : 3,  
 “first”: SKE.Enc(Ke, “Bob”) || ct51,

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

“age”: SKE.Enc(Ke, 55) || ct53,

“\_safeContent”: [ tag61, tag62, tag63 ]

}

Encrypted State Collection:

D1: { “\_id” : F(...),

“Value”: SKE.Enc(...)}

…

D9: { “\_id” : F(...),

“Value”: SKE.Enc(...)}

ERD1: { “\_id” : rtag1}

…

ERD16: { “\_id” : rtag16}

ECD1: { “\_id” : ...,

“\_value”: comptag1}

ECD2: { “\_id” : ...,

“\_value”: comptag2}

ECD3: { “\_id” : ... ,

“\_value”: comptag3}

Note that the ESC is composed of 9 documents representing counter values along with the encrypted range documents and the encrypted compaction documents. There are 9 counter documents because there are 3 encrypted fields for each document inserted. 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 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. If these insertions occur at the same time the server will need to increment the counter associated with **first||Bob** twice sequentially.

# 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) [[3]](#footnote-2)

### 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. 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\*|| tag\* := SKE.Dec(utk4, ct32).

The server then finds and deletes tag\* from the **\_safeContent** field. The counter counter\* is not used at this stage; it is only needed for conjunctive queries which we describe below.

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 || tag2) 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.

### 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.

# 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:

1. computes two keys K’ and K’’ such that  
     
    K’ := F(stk, 1) and K’’ := F(stk, 2)
2. retrieves the document D\* from the ECS where \_id = F(K’, i\*) by performing a binary search in the range [1, #ECS], where #ECS is the number of documents in the ECS and “i\*” is the last document inserted for label **ssn||111-11-1111**. 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.
3. retrieves the counter from D\* by computing counter := SKE.Dec(K’’,value).

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

**Analysis**. Ignoring the time needed to decrypt the matching,[[4]](#footnote-3) the client-side computation is optimal since it simply requires two PRF evaluations which is independent of the size of the result. 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.

# 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 [[5]](#footnote-4): 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 to identify the values in the range r = [32, 51] that exist in the database. And in order to perform this operation efficiently, it traverses the binary tree (stored in the encrypted range documents (ERDs)) searching for existing paths. If there is a path from a cover to a leaf, it means that this leaf (or numerical value) exists in the database (under the field age in our example). Note that the server does not need to traverse the entire path in order to know that an element does not exist. It can stop way before depending on the sparsity of the binary tree. We will provide more details about this algorithm in the formal version of this writeup but the high level idea works as follows:

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 [[6]](#footnote-5) but uses the ERDs to avoid sub-trees that are not relevant. 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. It retrieves the document D\* from the ESC with \_id = F(K’, i\*) by performing a binary search within the range [1, #ESC], where #ESC is the number of documents in the ESC and “i\*” is the last document inserted for **age||38** . 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.[[7]](#footnote-6) 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 = 10 (10 counter documents when accounting for the update operation in the previous section).
* 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.

## 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.[[8]](#footnote-7) 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 counter 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, #ESC], where #ESC is the number of documents in the ESC and “i1” is the last document inserted for label **ssn||111-11-1111**.
3. It retrieves the counter from D by computing counter1 := SKE.Dec(K’’,value).
4. It computes two keys K’ and K’’:  
     
    K’’’ := F(stk21, 1) and K’’’’ := F(stk21, 2)
5. It retrieves the document D from ESC where **\_id** = F(K’’’, i2) by performing a binary search within the range [1, #ESC], where #ESC is the number of documents in the ESC and “i2” is the last document inserted for label **first||Bob**.
6. It retrieves the counter from D by computing counter2 := SKE.Dec(K’’’’,value).
7. It orders the counters in an increasing order such that (counter1, counter2). Note that we have inserted two documents where “first”:”Bob” and only one document with “ssn”: 111-11-1111. Therefore counter2 = 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,

“age”: SKE.Enc(Ke, 33) || ct13,

“\_safeContent”: [ tag21, tag22, tag23 ]

}

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\*||tag := SKE.Dec(sk12, ct12)
2. It computes K’ := F(sk21,3),
3. It computes tag\* := F(K’, counter\*)
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.

**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.
* 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.[[9]](#footnote-8)
* 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).

# Post-Deletion Search

In this section, describe how to do deletions and the way it affects future search queries, i.e., post-deletion queries. This matters for the following reason. Suppose we delete a document that includes field/value pair **first: Bob** and then search twice for all documents where **first: Bob**. In this construction, the second post-delete search will be faster than the first. We describe all this in more detail with the following example:

*C1 deletes all documents where ssn:111-11-1111*

As a result of this query D1 will be deleted.

## Deletions

Deletes are simple in OST-1. They consist of a search query followed by a deletion of the results. In our example, the server does a point search for all documents that have **ssn** value 111-11-1111 and deletes them. It is important to note that, at this stage, the server does not modify the encrypted state collection.

## First Post-Deletion Search

Consider the case where client C2 performs the following point search query:

*C2 searches for all documents where ssn:111-11-1111.*

Since this is the first time a client executes a query that matches the deleted document D1, there will be some extra cost for the server compared to the next time. Below is a high-level overview of the search process.

1. The client and server execute an encrypted point search,
2. The server notices that one of the tags that was generated as part of the search does not match any document. Note that in our example, only one tag is generated and it will not match any document.
3. The client receives the results. Note that in our example, it will receive an empty set,
4. The server *possibly* updates the encrypted cache structure. The update to the ECC does not always occur; it occurs under certain conditions which we describe below. The purpose of this update will be to speed up future search queries for this field/value pair by storing information about the tags that no longer need to be queried since their corresponding documents were deleted.

## Encrypted Cache Collection

### Gap and Encoding

The encrypted cache collection is parametrized by a gap parameter G. At a high level, this parameter determines when the server updates the ECC after a search operation. In particular, if the tags of the deleted documents were generated based on counter values that are contiguous (e.g., tag1 was generated using counter = 5, tag2 using counter2 = 6, ...), and if the contiguous counters form a segment of size at least G, the server will update the ECC, otherwise it does nothing.

Updating the ECC consists of adding a new encrypted document that stores information about the counters of the deleted tags. This information is encoded in a compact way. If the gap parameter G is set to 1, the plaintext variant of the encoding is:

{ssn:111-11-1111 : [0,1]},

which basically means that the tag based on counter 0 for ssn:111-11-1111 has been deleted.

If, after a search operation, the server detects that 4 documents were deleted with tags based on the counters 0,1,2 and 7, the encoding would have the following form:

{ssn:111-11-1111 : [0,3], [7,1]}

In general, the encoding has the following form:

{label : [a1, b1], [a2, b2], ...},

where a1, a2, … are the first counters at which a gap starts and b1, b2, … are the widths of the gap that are a multiple of the gap value G.

### Updating the ECC

To update the ECC, the server does the following:

* It first checks if a document has already been created for the field/value pair ssn:111-11-1111 in the ECC. More precisely it:
  + computes K’ := F(stk,4) and K’’ := F(stk,5),
  + retrieves the document D\* from the ECC with \_id = F(K’, i\*) by performing a binary search in the range [1, #ECC], where #ECC is the number of documents in the ECC and “i\*” is the last document inserted for label **ssn||111-11-1111**. 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 no document with ssn:111-11-1111.
* creates and insert the document  
    
  D : {   
   “\_id” : F(K’, 0),  
   “value”: SKE.Enc(K’’, [0,1])  
   }

**Remark 1**. 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.

**Remark 2**. Since the client has no state, it cannot communicate to the server whether a document with a given field/value pair has been deleted in the past. Because of this, the server will always need to search the ECC to find the latest document for the field/value pair. This search is logarithmic but can be done in parallel with searching for the counter in the ESC.

**Remark 3**. Note that updating the ECC can be thought of as a separate process independent of the search query. It is true that this process is triggered by a search but it does not need to be viewed as part of the search operation itself.[[10]](#footnote-9)

**Remark 4**. Updating the ECC in the case of a range search or of a conjunctive search is similar except that the server may need to update multiple field/value pairs after the search.

**Remark 5**. Picking the right value for G is challenging as different workloads may have different delete patterns. In the example above, we picked G = 1 to simplify the exposition. In practice, however, a larger value of G should probably be selected.

1. In practice, this encryption can be performed using AES with CTR mode for confidentiality only, or with AES-GCM for both confidentiality and integrity of the value. [↑](#footnote-ref-0)
2. It is important to note that the server does not overwrite the existing document. This is crucial to reduce leakage to a snapshot adversary. We will provide more details in the formal version of this writeup. [↑](#footnote-ref-1)
3. 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-2)
4. The time needed to decrypt the matching documents is linear in the number of returned documents and the number of encrypted fields and is optimal. [↑](#footnote-ref-3)
5. 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-4)
6. By “derived” we mean that the tree is never completely instantiated. Instead, its nodes are derived on the fly as the tree is traversed. [↑](#footnote-ref-5)
7. 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-6)
8. 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-7)
9. There are many different forms of conjunctions (e.g., encrypted, mixed, with or without ranges, with one range term or multiple range terms) which we don’t cover here but will in the more detailed writeup. [↑](#footnote-ref-8)
10. We added this remark to highlight the fact that clients can still perform search operations on MongoDB secondary nodes. However, only the primary can update the ECC. [↑](#footnote-ref-9)