Fast Range Query Processing with Strong Privacy Protection for Cloud Computing

Mohammed Jasam, Pooja Kancherla, Ali Allami

Abstract

Cloud computing services have been increasing rapidly due to the low costs as opposed to traditional infrastructures. However, with lower cost, higher reliability, better performance and faster deployment, the cloud-based services pose challenges when it comes to the privacy of the users. Hence, this paper concerns the problem of privacy preserving range query processing on clouds. The paper utilises SSE the secure searchable encryption technique to achieve the privacy preserving. In this project we will implementing the exact solutions of the authors and emphasise its advantages and disadvantages. In addition, we will suggest some future solutions based on the techniques that we learned in the secure data analysis class that could solve some of the issues raised by this paper.

I. Introduction

Cloud computing has introduced a new view in service providing, where it provides hardware and software resources on demand to a remote client on demand. Unfortunately, there are many privacy issues related to cloud which makes it impossible to fully trust the cloud. There are many problems that cause these trust issues, some of which include corrupted employees who fail to follow the data privacy policies, vulnerability to external malicious attacks and various other data integrity problems. This paper deals with the condition according to the following paradigm: the data owner and multiple data users query the data. To have a better understanding, a real-life example can be considered as following: In a hospital, the cloud stores all the data files related to the patients in a cloud. Multiple doctors access the data in the cloud. Hence, privacy is an important factor here, as the identity of the patients and their health record have to be kept confidential. The problem concerning such privacy is described in the following sections below.

II. Problem Definition

Cloud computing model consists of three main components: the data owner, the cloud and the data user. In this model, we consider that both data owner and data user are fully trusted while the cloud is semi honest which means that the cloud will follow the protocols, but can be curious and use the results to learn more about the data and even the query. Below is an abstract definition of range query processing for encrypted data based on the cloud computing model. 1 For a set of records which have the

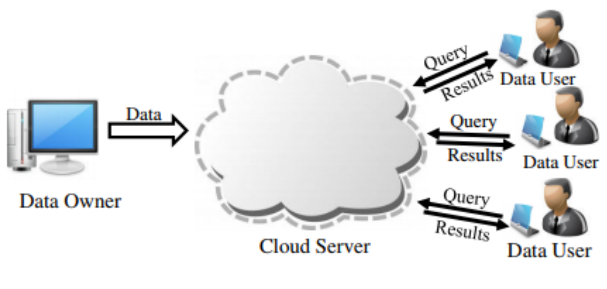


Figure 1: Computational Cloud Model

same attribute A, which has numerical values or can be represented as numerical values, given a range query specified by an interval $[a, b]$, the query result is the set of records whose A attribute falls into the interval. Given data items $d_1, ..., d_n$ the data owner encrypts this data using a symmetric key K, which is shared between the data owner and data users. The data owner generates an index and sends both the encrypted data denoted $(d_1)_k, , (d_n)_k$ and the index to the cloud. Given a range query $q[a, b]$, the data user generates a trapdoor and then sends it to the cloud. The index and the trapdoor should allow the cloud to determine which data items satisfy the query. During this process the cloud should not be able to infer useful information about the data and the queries [1].

III. Methodology

This section introduces the main building blocks to achieve the SSE of the cloud computing model.

A. Prefix Encoding

The key idea of this step is to convert the checking of whether a data item falls into a range to the testing of whether two sets have common elements, where the basic step is testing whether two numbers are equal. This could be achieved by adopting the prefix membership verification scheme as following: Given a number x, which is made of w bits with a binary representation of $b_1b_2...b_w$, its prefix family denoted as $F(x)$ is defined as the set of $w + 1$ prefixes $b_1b_2...b_w, b_1b_2...b_{w-1}*, ..., b_1 * ...*, *...*$, where the i-th prefix is $b_1b_2...b_{w-i+1} * ...*$. For example, the prefix family of number 6 of 5 bits is:
$F(6) = F(00110) = \{00110, 0011*, 001 * *, 00 * **, 0 * * * *, * * * * *\}$.

B. Equal Size Prefix Family Partition

Partitioning a node into sub nodes left and right is critical for the performance of query processing on the PBtree because querying the common prefixes that both left and right nodes share will lead to the traversal of both subtrees. The authors define the problem of construction of the PBTree (Equal Size Prefix Family Partition problem) as following: Given a nonterminal node v with prefix family set $S\{F_1() \cup ... \cup F_2())\}$, partition S into two child nodes S_1, S_2 so that $0 \leq |s_1| - |s_2| \leq 1$ and get the minimized set of $\{F_i \cap F_j | F_i \in S_1, F_j \in S_2\}$, which is the maximum number of prefixes in the intersection of two prefix families in which one is from S_1 and the other is from S_2. To present the details of these two phases, we need to define two concepts: longest common prefix and child prefixes.

C. The longest common prefix

Given a set of prefix families $S = \{F_1, F_2, ..., F_n\}$, the longest prefix in $F_1 \cap F_2 \cap ... \cap F_n$ is denoted by $LCP(S)$. For example, the longest common prefix of $\{F(1101), F(1100)\}$ is $110*$. Note that for any set of prefix families, there is only one longest common prefix because no prefix family consists of two prefixes of the same length.

D. Child prefixes

For any prefix $b_1b_2...b_{w-i+1} * ...*$, it has two child prefixes $b_1b_2...b_{w-i+10} * ...*$ and $b_1b_2...b_{w-i+11} * ...*$, which are obtained by replacing the first * by 0 and 1 and called child-0 and child-1 prefixes, respectively. For example, prefix 11* has two child prefixes 110* and 111*. For any prefix p, use p^0 and p^1 to denote p's child-0 and child-1 prefixes respectively.

E. Partition phase

Consider a set of prefix families $S = \{F_1, F_2, ..., F_n\}$. First compute $LCP(S)$, the longest common prefix of S. Next,partition S into two subsets, one subset whose each prefix family contains $LCP(S)^0$ and one subset whose each prefix family contains $LCP(S)^1$. If any of the two subsets has a larger size than $\lceil n/2 \rceil$, then recursively apply the above two partition processes to that subset. This process terminates when all subsets have a size smaller than $\lceil n/2 \rceil$. Finally, for any two subsets whose union has a size smaller than $\lceil n/2 \rceil$, we merge them. This process terminates when no two subsets are mergeable. Thus, the result in either two subsets or three subsets. It is impossible to result in four subsets or more because otherwise there are at least two subsets that can be merged.

F. The re-organization phase

This phase chooses one subset to split into multiple smaller subsets, and then union these new subsets with the two other subsets to obtain two subsets S1 and S2 that satisfy the condition of $0 \leqslant ||S_1| - |S_2|| \leqslant 1$. Let S_1, S_2, and S_3 be the three subsets. Let the subset whose longest common prefix is the smallest, that is, the subset whose prefix families share the least number of prefixes. Let S_3 be the subset that we choose. First compute its longest common prefix $LCP(S_3)$. Note that for any prefix family in S_3, it contains either $LCP(S_3)^0$ or $LCP(S_3)^1$. Second, split S_3 into two subsets S_{31}, whose each prefix family contains $LCP(S_3)^0$, and S_{32}, whose each prefix family contains $LCP(S_3)^1$. Either S_1 or S_2 can be merged with S_{31}. Otherwise, if both $|S_1| + |S_{31}| > \lceil n/2 \rceil$ and $|S_2| + |S_{31}| > \lceil n/2 \rceil$ are true, then $|S_1|+|S_2|+|S_3| = |S_1|+|S_2|+|S_{31}|+|S_{32}| \geqslant |S_1|+|S_2|+|S_{31}|+|S_{31}| = (|S_1|+|S_{31}|)+(|S_2|+|S_{31}|) > \lceil n/2 \rceil + \lceil n/2 \rceil \geqslant n$. Again, suppose $|s_1| \geq |s_2|$ and S_1 can be merged with S_{31}, merge S_1 with S_{31}. After merging S_1 with S_{31}, check whether S_2 can be merged with S_{32}. If they can, merge them and output the partition result $S_1 \cup S_{31}$ and $S_2 \cup S_{32}$. If S_2 and S_{32} can not be merged, further split S_{32}, and repeat the above process. If S_1 can not be merged with S_{31}, merge S_{31} with S_2 and split S_{32}, and then repeat the above process. The pseudo code is shown in Algorithm 1.

G. Node Randomization Using Bloom Filters

After constructing the tree based on the algorithm described in sec 3.2, the data owner randomizes each node using a bloom filter as following:

1) For each prefix p_i, the data owner uses the r secret keys to compute r hashes:
   $HMAC(k_1, p_i), ..., HMAC(k_r, p_i)$.
2) For node v, the data owner generates a random number $v.R$, which has the same number of bits as a secret key.
   Use $v.R$ to compute r hashes: $HMAC(v.R, HMAC(k_1, p_i)), ..., HMAC(v.R, HMAC(k_r, p_i))$.
3) For each prefix p_i and for each $1 \leq j \leq r$
   let $v.B[HMAC(v.R, HMAC(k_j, p_i)) mod M] := 1$.

The following figure 2 shows the above hashing process for Bloom filters.

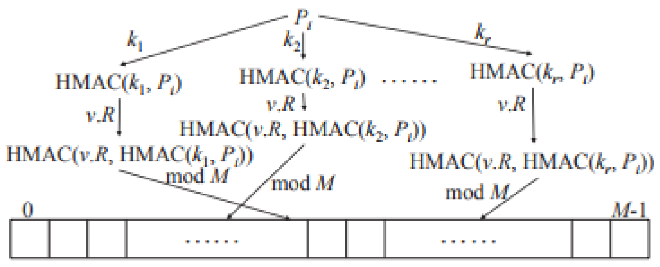


Figure 2: hash

H. Trapdoor Computation

The data user builds the trapdoor for his/her range query q [a, b] as following:

1) The data user computes the minimum prefix set that is sufficient to represent the range in $q[a, b]$ we call this minimum set $S([a, b])$. It consists of z prefixes $p_1, ..., p_z$, for each prefix p_i, $1 \leq i \leq z$
2) The data user computes r hashes: $HMAC(k_1, p_i), ..., HMAC(k_r, p_i)$.
3) The trapdoor for range query $[a, b]$, denoted as $M\,[a, b]$, is a matrix of $z * r$ hashes:

Algorithm 1 Equal size prefix family partition

```
1: **Input** $S = \{F_1, F_2, ..., F_n\}$
2: **Output** $S_1$ and $S_2$ where $S_1 \subset S$, $S_2 \subset S$, and $0 \leq ||S_1| - |S_2|| \leq 1$
3: Initiate an empty partition subset list L.
4: **while** $(|S > \lceil n/2 \rceil|)$ **do**
5:     Compute $P^s$ for $S$. Partition $S$ into:
6:     $S_1 = \{\forall F_i \mid LCP(S)^0 \in F_i, F_i \in S)\}$, and
7:     $S_2 = \{\forall F_i \mid LCP(S)^1 \in F_i, F_i \in S)\}$
8:         IF $(|S_1| \geqslant |S_2|)$
9:             Insert $S_2$ into $L$, $S := S_1$
10:        Else
11:            Insert $S_1$ into $L$, $S := S_2$
12:        EndIF
13: insert $S$ into $L$.
14: **while** subsets $S_i$ and $S_j$ are mergable in $L$ **do**
15:     Merge $S_i$ and $S_j$ into one subset $S_{ij}$.
16:     Replace $S_i$ and $S_j$ with $S_{ij}$ in $L$.
17: IF $L$ contains only two suvset $S_1$ and $S_2$
18:     Return $S_1$ and $S_2$
19: Else
20:    Let $S_3$ has $|\bigcap F_i \in S_3| \leqslant |\bigcap F_i \in S_1|, and |\bigcap F_i \in S_3| \leqslant |\bigcap F_i \in S_2|$ holds.
21: **while** $L$ has 3 subsets denoted by $S_1$, $S_2$, and $S_3$ **do**
22:        Remove $S_3$ from $L$. Split $S_3$ into $S_{31}$,and $S_{32}$.
23:        Let $|S_1| \geq |S_2|, and |S_{31}| \leq |S_{32}|$.
24:        IF $(|S_1| + |S_{31}| \leq \lceil n/2 \rceil)$
25:            Merge $S_{31}$ with $S_1$.
26:        IF $(|S_2| + |S_{32}| \leq \lceil n/2 \rceil)$
27:            Merge $S_{32}$ with $S_2$.
28:        Else
29:            $S_3 := S_{32}$. Insert $S_3$ into $L$.
30:        Else
31:            Merge $S_{31}$ with $S_2$.$S_3 := S_{32}$. Insert $S_3$ into $L$.
32: Return Labels of the two subsets in $L$.
```

4) The data user organizes these $z * r$ hashes in a matrix M.
5) The data user sends $M[a, b]$ to the cloud.

I. *Query Processing*

The cloud uses the trapdoor, which is received from the data user, to search over the PBtree.

1) The cloud checks whether there exists a row i $(1 \leq i \leq z)$ in matrix $M[a, b]$.
   So, that for every j $(1 \leq j \leq r)$ it has: $v.B[HMAC(v.R, HMAC(k_j, p_i))modM] = 1$
2) For a row i in $M[a, b]$
   - if there exists j $(1 \leq j \leq r)$ so that $v.B[HMAC(v.R, HMAC(k_j, p_i))modM] = 0$ then $\bigcup(v) \cap p_i = \varnothing$.
   - If $\bigcup(v) \cap p_i = \varnothing$ then

for any descendent node v' of node v, it has $\bigcup(v) \cap p_i = \varnothing$ because $\bigcup(v') \subset \bigcup(v)$. so, remove ith from $M\,[a,b]$.

3) Terminate when M[a,b] becomes empty or when it finishes searching the terminal nodes.

IV. 4- Analysis

This section discuss the issues and the possible solution to fix such problems. Giving the fact that the proposed technique is secure based on the analyses that the paper gives through satisfying the following two properties (a) the contents of the data items are not revealed from the structure of the Bloom filter in which they are stored or from the contents of other Bloom filters, and (b) given any two Bloom filters, with different number of data items, they are indistinguishable to an adversary. However, it suffers from many drawbacks as shown bellow:

1) It works only for numerical data items which limits its usefulness. Moreover, through the course of this project we found that it can not handel large numbers easily since it would generate a large sets of prefixes especially when it try to generate the minimum prefix subset of the data user range query. It is worth to mention that the number of prefixes in the minimum prefixes subset $S([a,b])$ is at most $2w-2$ where w is the number of bits. Also, the bloom filter to handle the huge data will be growing extensively for huge numbers. The bloom filter size is $10*n$ where n is the number of prefixes in each node.
2) The same trapdoor is generated every time for the same query which would leak statistical information about data user search patterns.
3) The data structure that they use (bloom filter) is already having a problem by including false positive results. The reason is that when it sets $v.B[HMAC(v.R, HMAC(kj, pi)) mod M] := 1$ that could set the same index by different data items which cause correlation and some data items that are not part of the query will appear in the result.
4) The optimization that they propose to build the binary tree is NP-hard which make it hard to have the best partitioning of the data at each node based on the prefix family verification technique.
5) The optimization of the depth search having a false negative which will exclude some of the results.
6) The technique is not benign for the basic database operations such as deletion, insertion,and update. Since it needs to regenerate the tree from the beginning once the data owner updated his data even if he updated a single instance. That would cause a problem of error synchronization when the data is not reflected exactly on the BFtree during the time of running the data user query. Therefore, it would not be applicable for real time use.

The disadvantages that mentioned above can be solved using the following techniques:

1) additive homomorphic encryption: it allows the data owner from encrypting his data and sent it to the cloud and also would allow the end user from encrypting his own query. The cloud can run the search over the encrypted data items since the additive homomorphic would allow doing the operations over encrypted data without being able to break into the original data. The cloud then sends the result of the search to the end user which have the right to decrypt it and tell the clod which encrypted data item to bring as a final result of the query process. item Garbled Circuite: it allows the data owner, cloud, and the data user to work over encrypted data without leaking any information of both the data items or the user query.

   The advantages of using the two techniques mentioned above are summarised bellow:

   a) It works for different type of data items.
   b) I allows the cloud to work over the encrypted data which offers more benefits for both the data owners and the data users.
   c) It is completely benign for updating the data items which required the data owner to update only the exact instants without need to run the protocol over the whole data.

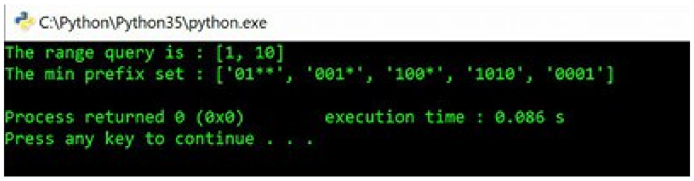
d) It does not have leak any statistical information since it generates different queries for the same query which would make it impossible for the cloud to track any pattern at the end.

works efficiently for small data sets which leads to limit its use. While the additive homomorphic encryption is able to handle large data sets. They can be used interchangeably based on the data set size.

V. Results

This section shows some results of the important methods as following

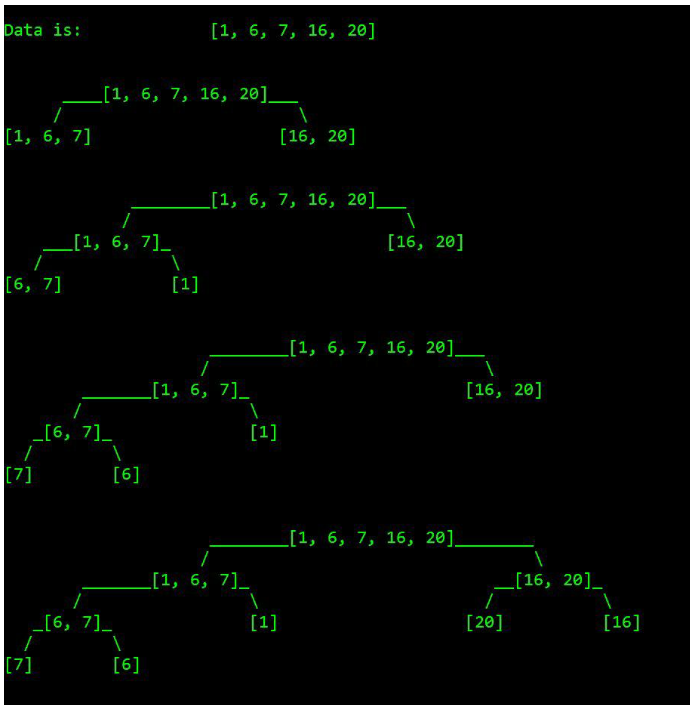
1) Figure 3 shows the minimum subset of prefixes $s\,[a,b]$. Giving a query $q\,[a,b]$ where $a = 0$, and $b = 10$ the data user needs to generate the minimum subset of prefixes that would be sufficient to represent the entire query as mentioned in the section III-H.



```
C:\Python\Python35\python.exe
The range query is : [1, 10]
The min prefix set : ['01**', '001*', '100*', '1010', '0001']

Process returned 0 (0x0)       execution time : 0.086 s
Press any key to continue . . .
```

Figure 3: Minimum Subset of Prefixes

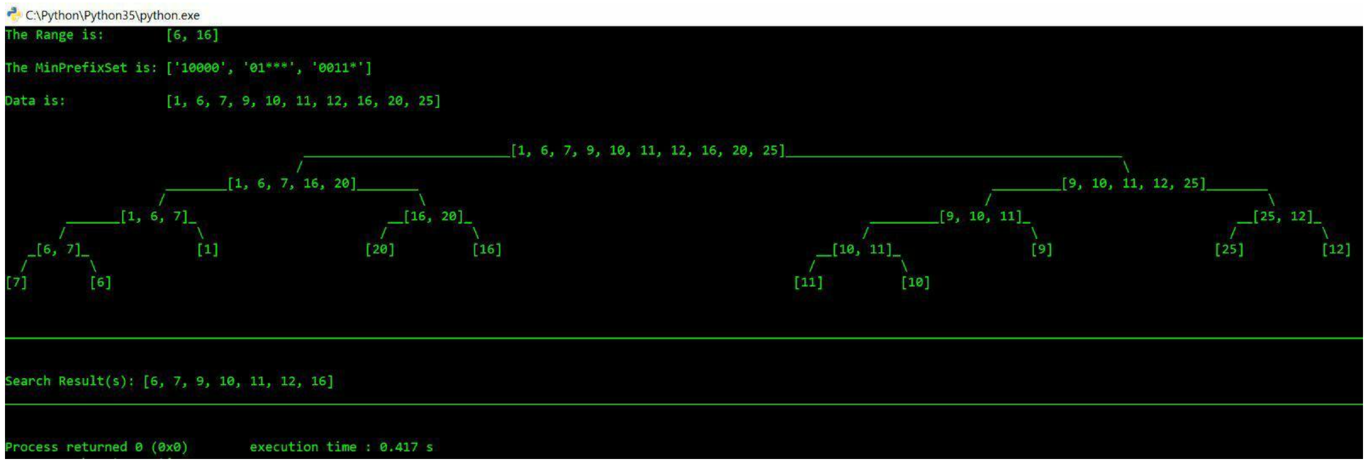
2) Figure 4 shows the building of the Binary tree mentioned in section III-F based on algorithm 1for data items $d = \{1, 6, 7, 16, 20\}$.



```
Data is:             [1, 6, 7, 16, 20]

      ____[1, 6, 7, 16, 20]___
     /                        \
[1, 6, 7]                   [16, 20]

             ________[1, 6, 7, 16, 20]___
            /                            \
    ___[1, 6, 7]_                      [16, 20]
   /             \
[6, 7]           [1]

                     ________[1, 6, 7, 16, 20]___
                    /                            \
        _______[1, 6, 7]_                      [16, 20]
       /                 \
   _[6, 7]_              [1]
  /        \
[7]        [6]

                     ________[1, 6, 7, 16, 20]________
                    /                                 \
        _______[1, 6, 7]_                         __[16, 20]_
       /                 \                       /           \
   _[6, 7]_              [1]                   [20]          [16]
  /        \
[7]        [6]
```

Figure 4: Tree building

3) Figure 5 shows the results of running the query $q\,[6, 16]$ over the Binary tree that represent the data items $d = \{1, 6, 7, 9, 10, 11, 12, 16, 20, 25\}$ based on the method mentioned in section III-I.



```
C:\Python\Python35\python.exe
The Range is:        [6, 16]

The MinPrefixSet is: ['10000', '01***', '0011*']

Data is:             [1, 6, 7, 9, 10, 11, 12, 16, 20, 25]

                                       ____________________________[1, 6, 7, 9, 10, 11, 12, 16, 20, 25]____________________________________________
                                      /                                                                                                            \
                     ________[1, 6, 7, 16, 20]________                                                                            _________[9, 10, 11, 12, 25]________
                    /                                  \                                                                         /                                    \
        _______[1, 6, 7]_                         __[16, 20]_                                                     _________[9, 10, 11]_                           __[25, 12]_
       /                 \                        /           \                                                  /                     \                         /           \
   _[6, 7]_              [1]                   [20]          [16]                                          __[10, 11]_                [9]                    [25]           [12]
  /        \                                                                                              /           \
[7]        [6]                                                                                          [11]          [10]


Search Result(s): [6, 7, 9, 10, 11, 12, 16]


Process returned 0 (0x0)        execution time : 0.417 s
```

Figure 5: Tree building

VI. Conclusion

The project implements the proposed method of the cited paper. The paper takes care of range query processing for numerical data. During the course of the implementation we show the drawbacks of the proposed data structure as well as some of the used techniques disadvantages. We suggest two well known techniques to solve the issues of this paper. Also, we provide some of the implementation results.

References

[1] R. Li, A. X. Liu, A. L. Wang, and B. Bruhadeshwar, “Fast range query processing with strong privacy protection for cloud computing,” *Proceedings of the VLDB Endowment*, vol. 7, no. 14, pp. 1953–1964, 2014.