Proof of Storage in Cloud

VIPUL MAHESHWARI
MCA 2nd Year
SCIS, University of Hyderabad
EMAIL: vipul2519@gmail.com

Project Guide
G. R. Gangadharan
Assistant Professor
IDRBT, Hyderabad
This is to certify that Mr. Vipul Maheshwari, pursuing MCA course at University of Hyderabad in the School of Computer and Information Sciences (SCIS) has undertaken a project as an intern at IDRBT, Hyderabad from May 3, 2013 to July 3, 2013.

He was assigned the project “Proof of Storage in Cloud” under my guidance. During the course of the project he has undertaken a study of Proof of Storage and also has done excellent work.

I wish him all the best for all his endeavours.

Dr. G.R. Gangadharan  
(Project Guide)  
Assistant Professor  
IDRBT, Hyderabad
Acknowledgment

I express my deep sense of gratitude to my Guide Dr. G.R. Gangadharan Assistant Professor, IDRBT for giving me an opportunity to do this project in the Institute for development and research in Banking Technology and providing all the support and guidance needed which made me complete the project on time.

I am thankful to Mr. Deep Narayan Tiwari who took keen interest in my project work and guided me all along till the completion of my project work by providing me all the necessary information.

I am also thankful to SCIS, University of Hyderabad for giving me this golden opportunity to work in a high-end research institute like IDRBT.

Vipul Maheshwari
MCA 2\textsuperscript{nd} Year
SCIS, University of Hyderabad
Contents

1. Introduction

2. Related Work

3. Proposed Work
   3.1 Notations
   3.2 General Idea
   3.3 Set-up Phase
   3.4 Verification Phase

4. Block Operations
   4.1 Insertion
   4.2 Search
   4.3 Update
   4.4 Delete

5. Complexity Analysis

6. Conclusion

7. References
1. Introduction

Cloud computing is a model that provides access to shared pool of resources that can be rapidly provisioned and released with minimum management effort. There are three service models present namely SaaS (Software as a Service), PaaS (Platform as a service), IaaS (Infrastructure as a service). SaaS also referred to as Software on demand is a software delivery model in which software is centrally hosted on cloud. PaaS is a model in which platform is provided to develop applications. IaaS is a cloud computing service model in which computing, storage facilities are provided on demand basis. Cloud computing has certain characteristic such as Pay per use, on demand self-service, broad network accessibility, rapid elasticity. But as all the cloud services require the data to be outsourced at an un-trusted third party which poses serious security issues. By outsourcing it is meant that the data owner moves the data to a third party who is supposed to faithfully store the data and make it available on demand. To make sure that the third party (Service provider) is not cheating on the client, a method has to be developed which could provide a proof that the data is safely stored at the third party and has not been altered in any way.

In the recent years a number of security related issues in the data outsourcing have been studied. Initially the major focus is kept at data authentication and integrity which means that how to efficiently and securely ensure that the server returns correct and complete result in response to its client’s queries. But later the problem diverted to PDP (Provable data possession) and POR (Proof of Retrivrability). Provable data possession is used to efficiently, securely and frequently verify that the server is possess the client data and has not tampered it. PDP works by generating some metadata about the data that is to be stored at the server and is used to verify by using random challenges. PDP is a public key based technique which allows any verifier to query the server and obtain the proof of possession. Proof of retrievability uses special blocks called sentinels hidden among other blocks in the data. In POR random challenges are made against these sentinels to check whether they are intact or not. In POR sentinels should be indistinguishable so from other regular blocks so the blocks in POR must be in encrypted form which makes it unusable for public databases. Moreover the number of sentinels are limited that makes the numbers of queries limited.

PDP schemes are valid for static data storage i.e. a file that is outsourced never changes. The static model does not fit in all scenarios. To manage client updates on the outsourced data by
inserting, modifying or deleting stored blocks while maintaining the proof of possession of data, a dynamic PDP scheme is essential in practical cloud computing systems for file storage.

In this report we have given a method which can deal with dynamic updates using the PDP scheme. The performance of PDP is bound by the Disk I/O operations not by the Cryptographic computations so to reduce that we have used B+ trees to store the data using composite keys to organize the data. The composite key used consists of the index value for the block and the timestamp for that block. B+ is a self-balancing tree structure which helps in maintaining the height of the tree and will make the computations faster. The file to be stored at the server is divided into blocks and each block is pre-processed to generate metadata and the thus generated metadata is stored at the server along with the file. Random oracle model is used to generate the challenges which are sent to the server for verification.
2. Related Work

Ateniese et al [1] were the first to have included public audit-ability in their scheme “Provable Data Possession” for ensuring the possession of files at the un-trusted sources. They have utilized RSA based homomorphic tags for auditing outsourced data, thus public audit-ability is achieved. Atniese did not consider the case for dynamic data storage it works just for the static case. Using this scheme for dynamic scenario is insecure due to replay attack.

Ateniese et al [2] also proposed a dynamic and scalable version of the previous PDP scheme. But in this scheme they have estimated all the future challenges stored the pre-computed answers as metadata because of which one can perform a fix number of updates and challenges. Moreover the insertions cannot be performed anywhere only append-type insertions are possible.

Juels et al [3] described proof of retrievability which focused on static data storage. This scheme pre-processes the data, in which sentinels blocks are randomly inserted into the file to detect corruption and the file is encrypted to hide these sentinels. This scheme also included error-correcting codes to make the model error resistant. But this model has some drawbacks as it does not support dynamic update, the number of queries that could be performed are also limited because of the sentinels.

Shacham et al [4] designed an improved PoR scheme called compact PoR with full proofs of security. They use publicly verifiable homomorphic authenticators built by BLS signatures because of which public retrievability is achieved. But their solution is also static.
3. Proposed Work

In this section we describe the proposed work, by showing how the set-up is done, how tokens are generated for verification and then how the verification is done and at the end we will show how the dynamic updates and deletions are done.

3.1 Notations

F – File that has to be stored at the server, which consist of n blocks f[1], f[2], f[3] …f[n]

OWN – owner of the data.

SRV – Entity that stores the outsourced data.

t – Is a variable used to store the time-stamps of the blocks.

ctr – is a variable which keeps tracks of the index that have been assigned to the blocks.

H(.) – is a cryptographic hash function.

sk,pk – is the pair of private and public key, private key to be stored just along with the owner of the data while the public key is made available to everyone.

A_key - is an authenticated encryption scheme, which will be used to encrypt the generated tags.

A_key^{-1} Decryption operation for the scheme introduced.

3.2 General Idea

In the scheme proposed file F is divided into a number of blocks and these blocks are pre-processed before being outsourced to the server where they will be stored. Tokens are produced for each block. These tokens will be used to verify the possession of the data. These computed tokens are then encrypted by using the private key of the owner and are sent along with the file to be stored at the server where the file is stored block by block into the B+ tree
using the index number of the blocks as at the initial stage indexes will be unique, variable ctr

is used which will keep track of the index numbers that have been used.

In the B+ tree the data entry will be done at the leaf node (each leaf node will have the
pointer to block it is pointing). All the internal nodes of the tree will have the value of the
index along with the time stamp which together will be used to locate a block.

At the time of verification challenge is generated by the verifier and is sent to the server,
server performs some computation and then returns computed value along with the metadata
that was previously computed and stored. Owner then verifies whether the data is intact or
not.

3.3 Set-up Phase

Given a file F divided into ‘n’ number of blocks , f [1], f [2], f [3]…. f[n] and t_i be the time
stamp of each block. Blocks will be stored into the server using their index values.

**Algorithm 1:**

Begin

\[ \text{ctr} = 0 \]

While (ctr<=n)

\begin{align*}
\text{Begin} \\
\text{Compute } v_i = H(t_i, f[i]) \\
\text{End }
\end{align*}

\[ v_i = A_{pk}(v_i) \]
\[ ctr = ctr + 1 \]

Send to SRV \((f[i], t_i, v'_i)\)

End

End

Using this algorithm the initial set-up is done by storing the data into the server and the tags are stored after being signed.

### 3.4 Verification Phase

In this phase verification is done, that the data is properly stored into the server. This is done by randomly choosing an index value and then sending it to the server along with the value of its time stamp, then the server computes the required tag and sends the new generated tag along with the tag stored in the metadata is also sent as a proof to the verifier as the proof. It is then verified by the verifier.

Algorithm 2:

**Begin**

OWN sends challenge to SRV \(\{i, t_i\}\)

SRV locates the index and computes

\[ z = H(t_i, f[i]) \]

SRV retrieves \(v'_i\) and sends back to the verifier the proof as \(z, v'_i\)

Verifier then checks the proof

\[ v'_i = A^{-1}_{pk}(v'_i) \]

If \((z = v_i)\) then accept

else reject

**End**
4. Block Operations

Now consider a B+ tree with order $n$, which means that each node in the tree can have $n$ key values and $n+1$ pointer’s. Each pointer is represented by $P_i$, points to a node depending upon the key values. All the leaf nodes contain a pointer which points to page containing information and these leaf nodes are linked to each other. Now we will present the algorithms used to insert, modify, search, and delete elements in the B+ tree.

4.1 Insertion

Insertions in a B+ trees is performed by searching for the leaf node where the entry belongs then inserting the entry at that location. The idea behind this algorithm is that we recursively insert the entry by calling the insert algorithm, on the appropriate child node. This procedure is performed by going down to the leaf node where it belongs and then returning all the way up to the root node. If the node where the insertion is to be done is full, it must split. When a split is made an entry pointing to the node created by the split must be inserted into the parent, this entry is pointed by a pointer variable $new$.

If the root node is split, height of tree increases by one.

The location of the leaf node is found by using both the index value and timestamp of the new block which is to be inserted.

The given algorithm takes a pointer to the root node, $k,t_i$ and a pointer $new$ which is null initially as input. Pointer new remains null until a split occurs. Here $k,t_i$ are the index and the time stamp of the new block. Assume that the order of the tree is $d$.

Algorithm 3: Insert (node pointer, $k,t_i$, new)

Begin

Let $N =$ node pointer

If (node pointer is a non-leaf node)

Find $i$ such that $k_i < k < k_{i+1}$ or $k_i = k$ and $(k < k_{i+1})$ and $(t_k \leq t_{k_i})$

Insert $P_i, k, t_i, new$

If (new is null)

return
else
    If (N has space)
        Put new on N
        new = null
        return
    Else
        Split N to create a new node
        new = pointer to the smallest key value of new node formed
        if (N is the root)
            Create new root
            return

If (node pointer is a leaf node)
    L = node pointer
    If (L has space)
        Put entry on it, set new to null
        return
    else
        Split L
        Create the link between the new leaf nodes formed

End

The above algorithm for insertion works well for inserting an item anywhere in the tree. Insertion by using this algorithm will never require the restructuring of the Tree. As when an index value repeats its timestamp will be different.

In the case of bulk loading in a tree performing insertion by calling the insertion algorithm for each block would be lot of time consuming to improve that we can sort the data as per their key blocks i.e. arrange the blocks in the ascending order of their indexes. Insertion begins by adding the elements to the root and when the root fills up it is split up. By doing this new entry is always added to the right most index page and when it fills up it is split. Storing
blocks using this method cause less I/O operations which are time consuming so this saves time.

The time complexity for this algorithm is $O(\log_b n)$

### 4.2 Search

To perform search SRV receives a request which and as per the request received a search is performed. The search operation in the tree is performed by using the index value and the time stamp. The given algorithm searches for the leaf node which contains the required block. Algorithm to perform search operation is given below:

**Algorithm: Search**  node pointer, $k,t_i$

**Begin**

If (node pointer is a leaf node)
   return node pointer
else
   If ($k < k_1$)
      return search $(p_0,k,t_k)$
   else
      If ($k > k_m$)
         return search $(p_m,k,t_k)$
      else
         Search for $i$ such that $k_i < k < k_{i+1}$ or
          $[(k_i = k)and(k < k_{i+1}) and (t_k <= t_i)]$
         return search $(p_i,k,t_k)$

**End**

The above algorithm is used to find the leaf node which contains the required block. This is used when just a block is required to be check.
If a range of blocks are to be searched then a property of B+ tree is used as we know that the data resides in the leaf nodes of the B+ trees which are linked to each other. So now suppose we need to search in the range \( a, b \) to search for the elements within the range we can just search for the staring element in the range and the rest of the elements can be retrieved sequentially which saves a lot of effort as we need not traverse the complete tree again and again which reduces the number of I/O operations considerably thus reducing the access time and making the search faster.

The complexity for the above search algorithm is \( O(\log_b n) \) and the complexity for bulk search changes to \( O(\log_b n + \frac{t}{b}) \), here \( t \) is the number of blocks that we are looking for and \( b \) is number of blocks that are there in one page.

### 4.3 Update

To update a block we can overwrite the existing block. But while updating a block we need to modify the tokens associated with it. In Block update we send a request to the server, server searches for the block and the token associated with it and sends it back, where the block is replaced and new token is generated and then sent back to the server. Server then stores the block and token. Let the new file block be

**Algorithm:**

**Begin**

Send request to SRV: Update \( k,t_k \)

At the SRV:

Search \( k,t_k \)

Send to OWN \((f[k],v'_k,t'_k)\)

At the OWN:

\[ f[k] = f[k]' \]

\[ t_k = t_k' \]

Compute new tag \[ v_k = H(f[k],k,t_k) \]
The complexity of the above search algorithm is $O(\log_b n)$.

### 4.4 Delete

Delete operation is quite essential as it is possible that the client might want to delete some records which are unused this could be achieved in two ways i.e. either the block which has to deleted is just disconnected from the structure but in this the block will still be occupying some memory so the other option is we delete the block and restructure the tree. In the proposed method we have used the latter option, to do so client will send a request to the server to delete a block. Server then looks for the block then removes it and checks whether the minimum element condition for the B+ tree is satisfied or not. The condition is that the number of elements in the node must be at least half of its order. If the condition becomes false then nodes of the tree are merged together.

The algorithm takes root node, $k,t_k$, as the input and deleted the entry corresponding to $k,t_k$.

**Algorithm: (node pointer, $k,t_k$)**

**Begin**

Let N= node pointer

If (node pointer is a non leaf node)

Find $i$ such that $(k_i < k < k_{i+1})$ or $[(k_i = k) \text{and}(k < k_{i+1})] \text{and}(t_k = t_{ki})$

Delete $(P_i,k,t_i,\text{old})$

If (old is null)

return

else

remove old from N

If (N has entry $> \frac{n}{2}$)

Set old to null
else
    get a sibling S of N
    If (S has extra entries)
        redistribute entries between S and N
    else
        merge(S,N)

if (node pointer is null)
    L = node pointer
    If (L has entry > $\frac{n}{2}$)
        remove entry
        old = Null
        return
    else
        get a sibling S of L
        if (S has extra entries)
            redistribute entries between S and L
        else
            merge(L,S)
End

The complexity for the above algorithm is $O(\log_b n)$. 
5. Complexity Analysis

In this section we will show the complexity of the various phases.

Let us assume the following notations:

Number of blocks = n
Size of each block = k bits
Size of time-stamp = m bits
Size of the encrypted tag = p bits
Size of the hashed token = q bits

<table>
<thead>
<tr>
<th>Phase</th>
<th>Time Complexity</th>
<th>Space Complexity</th>
<th>Communication Complexity</th>
</tr>
</thead>
<tbody>
<tr>
<td>Set-up Phase</td>
<td>$O(n)$</td>
<td>$O(n(k + m + p))$</td>
<td>$k + m + p$ bits</td>
</tr>
<tr>
<td>Verification</td>
<td>$O(\log_b n)$</td>
<td>$O(k + m)$</td>
<td>$m + q$ bits</td>
</tr>
</tbody>
</table>

6. Conclusion

In this study, we consider the problem of proof of storage on the cloud computing. We have presented the design of an efficient algorithm to minimize the computational overhead for the proof of storage on the cloud storage server by considering the advantages of B+ tree based implementation with dynamic data possession. Our scheme was developed to reduce the computational and storage overhead of the client. To reduce the network bandwidth consumption the size of the proof of data integrity is minimized.
7. References


