Enabling Efficient Verification of Dynamic Data Possession and Batch Updating in Cloud Storage

Yining Qi[1], Xin Tang[1] and Yongfeng Huang[1]
[1] Department of Electronic Engineering, Tsinghua University
Beijing, 100084, China
[e-mail: qyn15@mails.tsinghua.edu.cn]
*Corresponding author: Yining Qi

Received January 12, 2017; revised July 13, 2017; accepted January 2, 2018;
published June 30, 2018

Abstract

Dynamic data possession verification is a common requirement in cloud storage systems. After the client outsources its data to the cloud, it needs to not only check the integrity of its data but also verify whether the update is executed correctly. Previous researches have proposed various schemes based on Merkle Hash Tree (MHT) and implemented some initial improvements to prevent the tree imbalance. This paper tries to take one step further: Is there still any problems remained for optimization? In this paper, we study how to raise the efficiency of data dynamics by improving the parts of query and rebalancing, using a new data structure called Rank-Based Merkle AVL Tree (RB-MAT). Furthermore, we fill the gap of verifying multiple update operations at the same time, which is the novel batch updating scheme. The experimental results show that our efficient scheme has better efficiency than those of existing methods.

Keywords: Cloud/Grid Computing, Distributed Computing, Network Security

This work is supported by the National Natural Science Foundation of China under Grant No. U1405254, No. U1536201, No. 612713922 and No. 61472092.

http://doi.org/10.3837/tiis.2018.06.001 **ISSN : 1976-7277**

1. Introduction

Accompanied by the trend of cloud computing, cloud storage has gone through an explosive development. This novel outsourced data storage equips its clients with massive capacity under the “pay-as-you-go” model. With the help of cloud storage, clients upload huge amounts of datasets to remote servers which are run by the cloud service providers (CSP), instead of maintaining local data centers by themselves. One of the most common clients of cloud storage is enterprise with big data application, who usually needs to access and update its different parts of outsourced data frequently and simultaneously, often through distributed and parallel processing. However, since clients are deprived of the physical control over their data, they cannot know directly whether the data is intact or the update is executed correctly. Considering the huge amounts of data, an efficient dynamic data possession verification scheme is in desperate need.

Provable data possession (PDP) [12] is an elementary solution for the integrity verification problem, which works in a “block-tag” mode. In a typical PDP scheme, the client segments its data into blocks and generates their corresponding homomorphic tags. Once checking the integrity of outsourced data, the client sends challenge message to audit some randomly chosen blocks. In response to the challenge, a proof for required blocks is generated by CSP, which will be verified by the client. A significant drawback of such a scheme is that it does not support verification of data dynamics. The major reason lies in the difficulty of maintaining indices once data blocks are updated. For ensuring that the returned proof is generated from the required data blocks without replacement, the PDP scheme chooses to introduce indices of blocks into the computation of tags. Taking an instance, when the client tries to insert a data block at a certain position, all the tags of subsequent blocks have to be recomputed by the client, which leads to high computational and communication overhead. To get rid of the re-computation, some works [14][16][17][18] try to construct a linear mapping table called the index hash table used for querying and updating the block indices. However, the nature of these linear structures is doomed to its failure for adapting to updating multiple blocks concurrently, which impairs their potential of practical application.

Another idea to solve the problem is introducing the classical authentication data structure Merkle Hash Tree (MHT) into the verification scheme [20]. In MHT, every leaf stores hash value of a data block and every internal node stores the hash value of its two children. In this method, the indices of blocks are contained in the order of leaves, rather than in tags or using mapping tables as previous schemes. Given a leaf node in MHT, as well as all the sibling nodes on its path to the root, the client can easily re-compute the hash value of root node and compare it with the original one, in order to check the correctness of the hash value and its index. So the problem of update verification is transformed into verifying the insertion or deletion of tree nodes, which is much easier to maintain than that in previous works. In addition, the tree structure has long been used in traditional file system and is promising to be extended to the distributed cloud environment.

Although the MHT-based scheme is a potential solution, it still leaves some problems unsolved. First of all, the original MHT lacks an explicit method for querying the leaf nodes. Since the hash value cannot be directly used as search key, there are two possible methods to find a designated leaf. One of them is to search every leaf node and compare the hash values. The other is maintaining an extra pointer array, each element pointing to a leaf node. However, both of their average complexities are O(n). Moreover, neither of these two methods can prevent the attack of replacing the required node with another one, because there is no information to ensure the correctness of tree shape in original MHT.

Secondly, due to the property of tree structure, the original MHT also suffers from degeneration problem inevitably. That is to say, MHT may become seriously imbalanced after multiple update operations, which means the increase of overhead for both integrity verification and data dynamics. To solve this problem, some schemes [23][24] try to introduce rebalancing method from classical self-balancing tree structure such as B+ tree. Nevertheless, this kind of scheme leaves many problems to be further studied. One of the concerns is the frequency of rebalancing. Although binding the rebalancing process to every time of update can keep the tree always balanced, it may be costly and unnecessary. Since the update can be seen as random, the imbalance of tree may be counteracted by later update. Decreasing the frequency of rebalancing can reduce the amortizing overhead of maintaining the tree.

Besides, an important problem to improve the efficiency of MHT-based scheme still remains a vacuum, that is to support batch update verification. In practical application scenarios, the client often needs to update multiple blocks at one time. However, in previous schemes, if the client tries to do a series of update operations, it has to verify the MHT repeatedly, each time for one block. This is obviously inefficient. In fact, there has been no work supports verifying batch update at one time in MHT-based schemes. The most crucial reasons for the absence of batch update is that the previous MHTs lack a mechanism to eliminate the conflicts when both the children of one node try to update their common parent. On one hand, to avoid repeatedly handling the parent node, we need method to know whether both the children have completed their own operations and how to merge the two processes into one. On the other hand, if the process of one child rebalances the parent node, the position of its sibling may be changed, leading to more imbalance in another updating process. Thus simply introducing updating and rebalancing method from single case possibly results in disorder among adjacent nodes, calling for an effective method to segregate the unnecessary mutual interferences of branches in the tree. Additionally, updating multiple branches of MHT at the same time may cause accumulated imbalance, which breaks the rebalancing conditions of most standard self-balancing trees. In order to improve the efficiency of traditional MHT scheme, designing a new scheme enabling batch update verification with all the aforementioned problems solved is imperative.

In this paper, we target a scheme that not only able to avoid the degeneration of MHT with adjustable frequency for rebalancing but also supports batch update. The main contribution of this paper can be summarized as follows:

1. We propose an efficient dynamic provable data possession scheme based on a novel self-balancing MHT. Our new scheme optimizes the efficiency of querying leaf nodes, provides rebalancing with flexible frequency and takes the first to support batch updating verification.
2. We propose a new data structure called Rank-Based Merkle AVL Tree (RB-MAT), using ranks to achieve efficient query and enhanced security of checking block indices. The RB-MAT is a generalized AVL tree, which can adjust the frequency of rebalancing operations via relaxed balance. Besides, the RB-MAT also offers necessary support for lock-based batch updating method via maintaining status for every node.
3. We design a new batch update scheme based on update lock for RB-MAT, which eliminates the conflicts in updating and rebalancing adjacent branches of the tree. Our implementation fills in the blank of verifying multiple update at one time, reducing the communication and computational overhead for clients.

2. Related Work

Different from traditional data centers, the cloud storage has unique advantages and brings in new challenges for data security and privacy. As A. Razaque et al. has studied in [1], in addition to the property of security and privacy-preserving, the efficiency and data dynamics are the most pressing concerns of cloud data stakeholders.

On the basis of some preliminary works [2][3], Juels and Kaliski [4] propose the concept of "proof of retrievability" (POR), using spot-checking and error-correcting codes to find and fix the possible data damages. Particularly, the POR scheme precomputes some "sentinels" for error detection based on designated blocks randomly embedded in file, which, however, hamper the implementation of data dynamics. The following works [5][6][7][8][9][10][11] discuss the security and encoding performance of POR schemes, leaving the problem of data dynamics unsolved. Wang et al. [12] try to break this limitation and meanwhile keep the advantage of encoding-based data recovery. Nevertheless, they only realize a partial dynamic scheme which fails to support insertion. Recently, a few POR works such as [28][29][30] exploring some emerging fields has been proposed, applying new techniques.

In the same year as [4], the first provable data possession (PDP) scheme is proposed by Ateniese et al. [13] to verify the data integrity of files on untrusted storages, providing solution in a different thought. They make use of RSA-based homomorphic tags, which contain the block indices to prevent replacement. Similar as the POR scheme, PDP also confronts with the problem of how to extend the scheme to the case of data dynamics. Thus Ateniese et al. [14] take the lead to propose a dynamic version of their prior work, which, however, also fails to support block insertion due to the problem of maintaining indices contained in hash values of tags.

In order to overcome this difficulty, Zhu et al. [15] construct an index-hash table to support dynamic data operations in PDP scheme and extend the structure to multi-cloud scenario in [16]. The index-hash table is a linear linked record list using for querying and maintaining the indices of blocks, each record for a block. The inserted block is attached to the serial number of old blocks, distinguished by different version numbers. Following their work, Yang et al. [17] add time stamps to the records of index table in order to enhance the security. Jin et al. [18] apply a simplified version of index table called "index switcher", to implement arbitration for dynamic cloud data auditing. Tian et al. [19] alter the index table to a two-dimension version to raise the efficiency of querying and maintaining the table. However, the efficiency of index table is still relatively low compared with tree structure. On the other hand, the schemes aforementioned only provide method to verify the correctness of block tags, without checking the index table itself.

On the other hand, Erway et al. [20] propose a complete dynamic data possession auditing scheme, taking advantage of authentication skip list. Wang et al. [21] constructs their dynamic public auditing scheme based on the classical authentication structure Merkle Hash Tree (MHT), which supports auditing the correctness of update operations. To extend the application scenarios of MHT, Liu et al. [22] and Tang et al. [23] design auditing schemes specific to multiple replica and multiple clouds respectively. Meanwhile, the drawbacks of original MHT in [20] begins to draw attention of researchers. The most prominent problem is that it lacks an explicit query method for efficient querying leaf nodes and suffers from the problem of tree degeneration. Mo et al. [24] try to design a novel structure called Coordinate Merkle Hash Tree (CMHT) to implement a balanced MHT with the property of non-repudiation. To achieve this goal, they assign a coordinate c for each leaf node in CMHT to encode the path from the root to the leaf and keep the CMHT a complete binary tree at the cost of the leaf nodes arranged out of order. However, the disordered leaf nodes make the querying and management a hard work. An extra index table should be constructed and maintained in both cloud side and client side simultaneously. Moreover, the deletion of CMHT needs a process of fixing the tree shape, which makes the scheme suffers from heavy extra overhead. In their another work [25], the typical self-balancing data structure B+ tree is combined with MHT to make use of the rebalancing operations merging and splitting. They also propose a different kind of balanced MHT originated from red-black tree [26], which is designed for key management. Generally speaking, the proposed balanced MHTs are still simple combination of MHT and traditional self-balancing tree structure, requiring further study. Besides, the state of the art remains blank in the aspect of enabling batch updating.

3. Rank-Based Merkle AVL-Tree

In order to prepare for our efficient provable data possession scheme, we introduce the new data structure as preliminary.

3.1 Tree Construction

Our Rank-Based Merkle AVL-Tree (RB-MAT) is a novel authenticated data structure with efficient query method, authentication of indices and adjustable balance factor, providing necessary support for batch updating. An instance of RB-MAT is shown in **Fig. 1**. The leaves of the tree are denoted as $w_1, w_2, \dots, w_n$ from left to right, each one representing a corresponding data block. Internal nodes are denoted by letter subscribes, such as w_a and w_b. The characteristics of the tree are as following:

For an arbitrary node w_x, a rank value $r(w_x)$ is assigned to it to denote the number of leaves in the subtree rooted at w_x. In other words, the rank value means the maximum number of leaves that can be reached from the current node. Apparently, the rank of a leaf node is always 1. Rank value is the basis of efficient query method of RB-MAT and also play a crucial part in ensuring the correctness of queried block indices.

Different from the original Merkle Hash Tree, the hash value of each node w_x in RB-MAT is defined as below, denoted as label $l(w_x)$,

$$l(w_x) = \begin{cases} H(m_x), \text{if } w_x \text{ is a leaf node of } m_x \\ H\left(l(w_{left}) \parallel l(w_{right}) \parallel r(w_x)\right), \text{if } w_x \text{ is an internal node} \end{cases} \tag{1}$$

where $H(\cdot)$ is a pre-defined cryptographic hash function. Obviously, the auxiliary authentication information (AAI) should be also altered to the format of tuple $\left(r(w_x), l(w_x)\right)$. When checking the AAI, the rank values will be used to recompute the labels. Containing the rank in hash value is used for preventing the forgery of block indices, whose detailed discussion will be presented in section 3.2.

Since the RB-MAT is constructed based on the AVL tree, there must be a height value *h* stored in every node, which is the maximum number of edges on the path from a node to the leaves of subtree rooted at it. Some AVL trees also maintain balance factor *b* in every node. It must be noted that since the balance factor of a node is the difference between the heights of its two children, we can choose to maintain only the height in the node, in order to reduce the size of RB-MAT. For clarity of presentation, we still display balance factors of every node in **Fig. 1**.

In order to prepare for the lock-based batch update scheme, we assigns 2 bits to each node to maintain the status value *st*. Each bit of *st* represents whether the node is locked by its left or right child respectively. In the process of batch updating, the status value is used to indicate whether an internal node shared by multiple branches is ready to be updated. The details of status value used for batch updating will be presented in Section 4.3.

Once the client constructs a RB-MAT, it will sign the root with its private key and send it as well as the whole tree to the cloud storage. Then the client can delete its local copy.

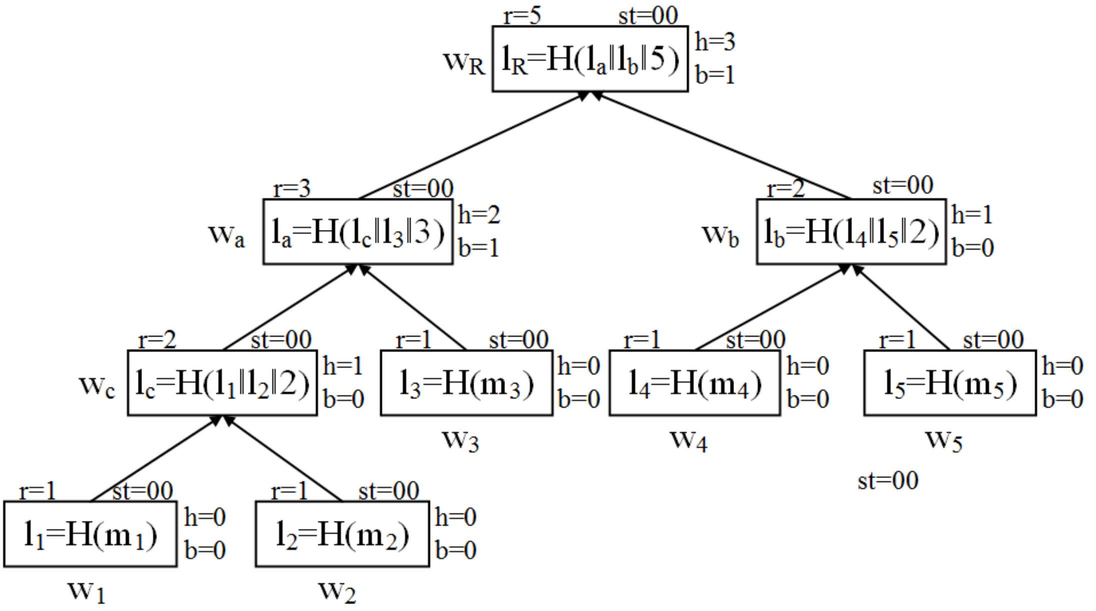


Fig. 1. A typical RB-MAT

3.2 Use of Rank

Considering that rank is the number of leaves which can be reached from a node, it can reflect the shape of the tree. Taking this advantage, we will show how to use ranks to query the leaves of tree and ensure the correctness of the block indices in this part, as well as the advantage of updating ranks to maintain the block indices. Applying ranks in the case of batch query is also discussed.

The process to query the *i*th leaf can be designed as follows: Initialize a temporary variable k and set its value to i. Take the root of RB-MAT as the start node. Denote r_{left} and r_{right} as the ranks of the children nodes. If k is larger than r_{left}, go to the right child and subtract r_{left} from k. Otherwise go to the left child and do nothing to k. Once reaching a new node, the cloud does the same thing as previous until the new node is a leaf node. If k=1, the node is the required one and all the siblings on the traversed path constitute the auxiliary authentication information (AAI) for leaf w_i. The cloud returns AAI and the signed root to the client as the proof for query. The complexity of rank-based query is $O(\log n)$.

On the other hand, ranks ensure the correctness of the tree shape, which means the query result cannot be forged by replacing the requested node with another one. The client is able to verify the query proof as following: firstly, reconstruct the root label and compare it with the signed one; secondly, invert the process of query to check the correctness of index. If both the verifications are passed, output *TRUE*. Otherwise, output *FALSE*. The key point for preventing the forgery of replacing block is containing the ranks in the labels. Even if the cloud has the ability to tamper the ranks to pass the verification of index, the check of root label is sure to fail because the false ranks will be used to recomputed the labels, which certainly leads to an inconsistent result.

In dynamic scenarios, the usage of ranks is efficient to maintain the indices for leaves. When we insert a leaf w_* in RB-MAT, the ranks of nodes on the path to root will be increased by one, which automatically increases the indices of all the leaves on the right of w_* by one. And it is also true another way around. The complexity of maintaining ranks is $\mathrm{O}(\log n)$. In the case without ranks, however, the overhead of maintaining an index table and modifying the indices in sequence is $\mathrm{O}(n)$, much higher than using ranks.

In the case of batch update, using ranks has special advantages in querying and maintaining indices. When the client wants to query m leaf nodes to the CSP, all the indices of required leaves should be sorted in ascending order as a query set $Q = \{i_1, i_2, \dots, i_m\}$. The batch query method is slightly different to that of the single case. Once reaching a node, the rank value of its left child r_{left} is compared with the indices in query set Q which will be divided into two subsets, $Q_1 = \left\{i_k \middle| i_k \le r_{left}, i_k \in Q\right\}$ and $Q_2 = \left\{i_k \middle| i_k > r_{left}, i_k \in Q\right\}$. The indices in Q_1 will go to the left child, while the others go to the right one. When Q_1 and Q_2 reach their new nodes, they will also be divided and the subsets will be directed to different branches. All the processes on multiple branches can be executed concurrently to raise the query efficiency. Finally, after several times of division and direction, all the query processes will find their corresponding leaf nodes on different branches of the tree. On the other hand, considering that using ranks to maintain index for a leaf only involves its ancestor nodes, processes on different branches of tree only need to increase or decrease the ranks of nodes in their paths by 1, which are independent to each other and easy to be executed concurrently. By contrast, updating an index in linear table must consider whether there are blocks in prior position inserted or deleted, which is the bottleneck to raise its batch processing efficiency.

3.3 Relaxed Balance

As mentioned before, we borrow the definitions of height and balance factor from AVL tree. The balance factor of a node describes the degree of imbalance for the node. In a strictly balanced tree, the value range of balance factor should only be [-1,1], which is the balance criterion of the original AVL tree.

If the imbalance criterion is changed from [-1,1] to [-δ,δ](δ>1), we can obtain a generalized AVL tree [27] with the relaxed balance. Relaxed balance is different from the approximate balance such as Red-Black tree, because RB-MAT will be strictly balanced again after the rebalancing operation. That is to say, relaxed balance means that the RB-MAT is allowed to postpone the process of rebalancing until the imbalance accumulates to a certain degree. The main motivation for relaxed balance is to decrease the frequency of rebalancing operations. Taking the extra computation and communication for rebalancing into account, lower frequency can decrease the amortizing overhead of maintaining RB-MAT.

A typical rebalancing method in AVL tree is rotation. It appeared very early and is easy to understand. However, when extended to relaxed balance, the rotation-based method become too complicated because all the different patterns of tree shape have to be analyzed respectively and the complexity will explode in exponential growth. We choose another rebalancing method based on adjustment which is more suitable to rebalance the node with accumulated imbalance. The adjustment-based method will be introduced in next section.

4. Efficient Data Possession Verification Scheme

In this section, we will show how to enable our efficient dynamic data possession verification scheme to achieve the aforementioned design issues, with the help of RB-MAT. Except the necessary Setup and Default Verification phase, we focus on the efficient data dynamics scheme, starting with the single update case then presenting the implementation of batch update support.

4.1 Setup

The client invokes *KeyGen* to generate its public and private keys, then preprocesses the file and constructs RB-MAT by running *SigGen* before outsourcing data to the cloud.

$KeyGen(1^k) \rightarrow (pk, sk)$. For a bilinear map $\mathbb{G}_1 \times \mathbb{G}_2 \rightarrow \mathbb{G}_T$, the client chooses $x \leftarrow \mathbb{Z}_p$, a random element $u \leftarrow \mathbb{G}_1$ and the generator g of $\mathbb{G}_2$. Then the client computes $v \leftarrow g^x$. So the private key is $sk = (x)$ and the public one is $pk = (u, g, v)$.

$SigGen(sk, F) \rightarrow \left(\Phi, T, sig_{sk}\left(l(w_R)\right)\right)$. The client firstly divides the file F into n blocks $\{m_1, m_2, \dots, m_n\}$, and then computes tag for each block m_i as $\sigma_i \leftarrow (H(m_i) \cdot u^{m_i})^x$. The set of tags is denoted as Φ. The client also constructs a RB-MAT T for the blocks. The root label of the tree is signed with the private key as $sig_{sk}\left(l(w_R)\right) \leftarrow \left(l(w_R)\right)^x$. At last, $\left(F, \Phi, T, sig_{sk}\left(l(w_R)\right)\right)$ is sent to the cloud and deleted from the client's local storage.

4.2 Default Verification

With the algorithms *GenChal*, *GenProof* and *VerifyProof*, the integrity of outsourced file can be checked by challenging the cloud. In response to the request, a proof will be generated by the cloud to prove the data possession. RB-MAT has the ability to ensure that the returned proof includes the correct challenged blocks. Our scheme is able to support both public and private auditing. Since the public auditability is not the core issue of this paper, here we call the client or TPA collectively the "verifier".

$GenChal(n) \rightarrow chal$. The verifier picks a c-element subset $I = \{i_1, i_2, \dots, i_c\}$ of set $[1, n]$. For each $i \in I$, it chooses a random element $\lambda_i \leftarrow \mathbb{Z}_p$ to construct the challenge $chal = \{(i, \lambda_i)\}_{i \in I}$. The challenge is sent to the cloud to launch the integrity verification.

$GenProof(F, \Phi, chal) \rightarrow P$. Once receiving the challenge, the cloud constructs its possession proof as follows: it firstly queries the leaves $\{w_i\}_{i \in I}$ as well as auxiliary authentication information (AAI) $\{\Omega_i\}_{i \in I}$ corresponding to the challenged blocks in the RB-MAT, using the efficient method in section 3.2. Then it computes $\mu = \sum_{i \in I} \lambda_i m_i$, $\sigma = \prod_{i \in I} \sigma_i^{\lambda_i}$. The signed root label $sig_{sk}\left(l(w_R)\right)$ is also appended. The cloud sends the proof $P = \left(\mu, \sigma, \{w_i, \Omega_i\}, sig_{sk}\left(l(w_R)\right)\right)$ to the verifier.

$VerifyProof(pk, chal, P) \rightarrow (TRUE, FALSE)$. First, the verifier checks the integrity and correctness of $\{w_i, \Omega_i\}$ by using the method in section 3.2. It re-computes the root label $l(w_R)$ and checks $e\left(sig_{sk}\left(l(w_R)\right), g\right) = e\left(l(w_R), v\right)$. Also, the verifier confirms the correctness of indices using ranks, preventing the attack of replacing the returned blocks with other ones, which threatens the security of the original MHT-based scheme. If success, the verifier continues to check $e(\sigma, g) = e\left(\prod_{i \in I} \left(l(w_i)\right)^{\lambda_i} \cdot u^{\mu}, v\right)$ and returns $TRUE$ or $FALSE$ according to the verification result.

4.2 Data Updating with Rebalancing Operations

In this part we will show how our scheme effectively fixes the imbalance of RB-MAT caused by block insertion or deletion, especially the case of relaxed balance. During the updating process, the client invokes *UpdateRequest* to generate request for asking the cloud to perform data dynamics. The cloud runs *ExecuteUpdate* to update the file with the RB-MAT, involving possible rebalancing operations. Finally, the client uses *VerifyUpdate* to check whether the update is executed correctly. Note that the case we discuss here is single updating, preparing for the batch operation introduced in the next part.

$UpdateRequest() \rightarrow request$. When the client wants to update a certain block m_i, it sends a request to the cloud. In the case of insertion or modification, the client computes hash value and tag for the new block and generates request as $(insert, i, m_*, \sigma_*)$ or $(modify, i, m_i', \sigma_i')$, preserving the hash value for verification. The request for deletion is simply $(delete, i)$ which does not need preprocessing and additional information. Then the update request is sent to the cloud.

$ExecuteUpdate(F, \Phi, T, request) \rightarrow P_u$. Once receiving the request, the cloud starts with querying the leaf w_i and AAI Ω_i following the method in section 3.2, and reserve the result for generating proof later. Once finding the required position, the cloud executes updating operation in the same way as original MHT-based scheme [21]. Then along the path from the required position to the root, every traversed node will be checked whether it needs to be rebalanced before recomputing its rank and hash value. One of the most significant differences of our scheme from previous works is that the cloud fixes the imbalanced node using the adjustment-based method to support relaxed balance, which has been mentioned in section 3.3.

The principle of this method is shown in **Fig. 2**. For simplicity, we only present the tree structure in the figure, without data stored in nodes. The internal nodes are denoted as circles while the leaves are represented by solid rectangles. Taking a RB-MAT with relaxed balance $\delta = 3$ for instance, when node w_R is imbalanced, we first find out all its subtrees which will be the minimum units to constitute the balanced tree. Comparing the children of w_R, the right one has the lower height, whose subtree is s_6, circled by dashed rectangle in **Fig. 2**. Then we search the branch from the other child to find all the remaining subtrees with the same height as s_6, which are $s_1, s_2, \dots, s_5$. The root node of every such subtree is denoted as a concentric circle in the figure, to be distinguished from other internal nodes. Such a search process can be easily implemented by preorder traversal and the permutation of result will not change the original order of those subtrees. Finally, all the subtrees are reorganized into a complete binary tree, which is balanced and filled from left to right. Obviously, the adjustment-based rebalancing method can also adapt to the case with strict balance.

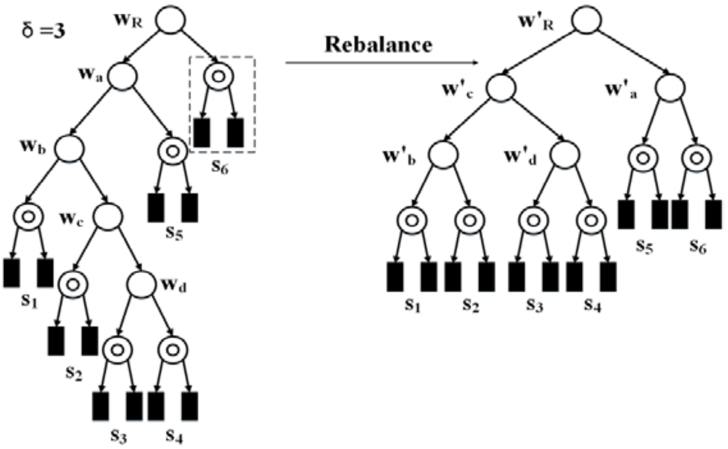


Fig. 2. Adjustment of Tree with Relaxed Balance

Once a node on the path is balanced, its labels and ranks will be consequently updated. This process continues until the root is updated to w_R'. At last, a proof for update as well as possible rebalancing is generated as $P_u = \left(\{w_i, \Omega_i\}, w_R', sig_{sk}(l(w_R))\right)$ and sent to the client. The algorithm of *ExecuteUpdate* is shown in **Algorithm 1**.

Algorithm 1. ExecuteUpdate

```
Algorithm $ExecuteUpdate(F, \Phi, T, request) \rightarrow P_u$
$w_i, \Omega_i \leftarrow Query(T, i)$
Perform the update of block at $w_i$
$w \leftarrow updated\ node$
while ($w$ is not the new root $w_R'$)
   update $h(w)$
   $b(w) \leftarrow h(w.leftchild) - h(w.rightchild)$
   if ($|b(w)| > \delta$) //Adjustment-based rebalancing
     $reOrg \leftarrow \emptyset$
     Add the child with lower height $ch_0$ to $reOrg$
     $h_0$ ←height of $ch_0$
     $ch_1$ ←the sibling of $ch_0$
     define function $SearchSubtree(ch, h, set)$
        if ($ch.height == h$)
           Add $ch$ to $set$
        else
           $SearchSubtree(ch.leftchild, h, set)$
           $SearchSubtree(ch.rightchild, h, set)$
        end
     end
     $SearchSubtree(ch_1, h_0, reOrg)$
     Reorganize nodes in $reOrg$
     $w$ ←root of the reorganized subtree
   end
   update $r(w)$ and $l(w)$
   $w \leftarrow w.parent$
end
$P_u = \left(\{w_i, \Omega_i\}, w_R', sig_{sk}(l(w_R))\right)$
```

$VerifyUpdate(P_u) \rightarrow (TRUE, FALSE)$. The client verifies the correctness of update by checking P_u using the algorithm in **Algorithm 2**. Once receiving the proof, the client firstly verifies w_i and Ω_i to make sure whether the update is executed at the correct position. If the proof passes the first step, the client continues to check whether the cloud performs update correctly by computing the new root w_{Rnew}. In this process, necessary rebalancing operations will be performed using the tree reconstructed from w_i and Ω_i. The client manipulates this process following the same adjustment-based method as the cloud does. Finally, w_{Rnew} is compared with w'_R. The algorithm returns $TRUE$ or $FALSE$ according to the verification result.

Algorithm 2. VerifyUpdate

```
Algorithm $VerifyUpdate(P_u) \rightarrow (TRUE, FALSE)$
$w \leftarrow w_i, s \leftarrow 0$
for every $w_x$ in $\Omega_i$
    reconstruct the parent $w_{parent}$ of $w$ and $w_x$
    if ($w_x$ is the left child of its parent)
        $s \leftarrow s + r(w_x)$
    end
    $w \leftarrow w_{parent}$
end
if ($e\big(sig_{sk}(l(w_R)), g\big) \neq e(l(w), v)$ $or$ $s \neq i - 1$)
    output $FALSE$ and return
end
perform the update of block with the reconstructed tree.
$w \leftarrow updated\ node$
while ($w$ is not the new root $w_{Rnew}$)
    update $h(w)$ and $b(w)$
    if ($|b(w)| > 1$)
        adjust $w$
    end
    update $r(w)$ and $l(w)$
    $w \leftarrow$parent of $w$
end
if ($w_{Rnew} \neq w'_R$)
    output $FALSE$ and return
else
    output $TRUE$ and sign $w'_R$
end
```

4.3 Batch Updating

Now we are going to present the implementation of support to batch updating in detail, making use of the properties of RB-MAT to construct our novel lock-based method. Similar to single update case, batch update scheme also consists of three algorithms: *BatchUpdateRequest*, *ExecuteBatchUpdate* and *VerifyBatchUpdate*. In the following discussion, we will focus on how to solve the difficulties lying in the unique scenario of updating multiple data blocks at the same time.

$BatchUpdateRequest() \rightarrow batchreq$. When the client wants to update *m* blocks at one time, it firstly generates a series of single update requests following the same method as *UpdateRequest*. For the convenience of applying batch querying method, these requests will be arranged according to their indices of updated positions in ascending order, then packed into $batchreq = \{request_j, j = 1, \ldots, m\}$.

$ExecuteBatchUpdate(F, \Phi, T, batchreq) \rightarrow P_{bu}$. Once receiving the batch update request, the cloud invokes this algorithm to launch the process. An instance for lock-based batch updating is presented in **Fig. 3**, which demonstrates the process of deleting the 2nd block, modifying the 3rd block and inserting a block behind the 5th position at the meantime.

As in the single case, batch updating starts with querying the nodes to be handled. Revisiting the batch query method introduced section 3.2, in order to apply the lock-based method, we need to lock all the nodes on the path from the root to the queried leaves, for example, w_R, w_a, w_b and w_c in **Fig. 3**. To “lock” a node means modifying its status value. For a locked node, if there is a path leading to its left child, the higher bit of status value should be set to 1, while the lower bit will be set to 1 in the case of a path to the right child. Particularly, if a node is shared by two different paths (its left and right children), there should be double-lock, just as w_a. Then at every queried leaf node, update operations are respectively executed in the same way as the single case. However, what is different is that when updating the nodes on the path from the leaf node to the root, once reaching a node, before updating and rebalancing, the update process firstly unlocks the lock by modifying the corresponding bit in status value to 0. If there is no remained lock for the node, then the updating can be continued. Otherwise, it should be terminated. The node will wait for process from another child to unlock and continue the update. The lock, especially double-lock, avoids that the nodes shared by different query path for different update to be updated repeatedly. **Fig. 3** shows unlocking in three processes. For better presentation of double-lock unlocking, these operations are displayed one by one in the figure. In practice, they can be run at the same time, involving concurrent processing. The update lock can be used to prevent chaos in concurrent update. Finally, a proof for batch update $P_{bu} = \left(\{w_j, \Omega_j\}, w_R', sig_{sk}\left(l(w_R)\right)\right)$ is sent to the client. The algorithm of *ExecuteBatchUpdate* is shown in **Algorithm 3**.

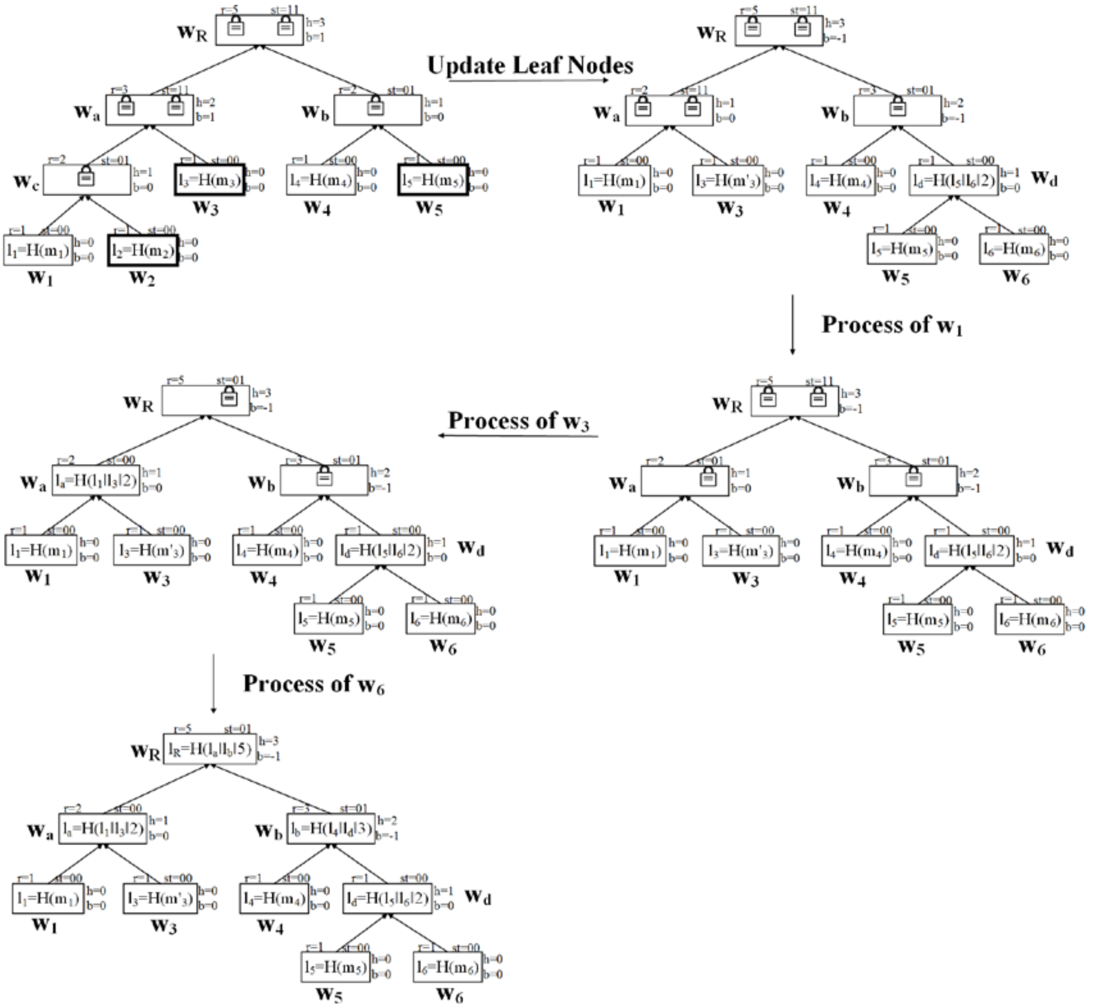


Fig. 3. Batch Update with Lock

$VerifyBatchUpdate(P_{bu}) \rightarrow (TRUE, FALSE)$. The verification of batch update is almost the same as single update. The only difference is that the client should reconstruct the RB-MAT following the same method in *ExecuteBatchUpdate*.

Algorithm 3. ExecuteBatchUpdate

```
Algorithm $ExecuteBatchUpdate(F, \Phi, T, batchreq) \rightarrow P_{bu}$
$queryset \leftarrow \{i_j = request_j.i\}$
define function $BatchQuery(w, set, result)$
    if (there is only one element $i$ in $set$)
        if ($w.rank == 1$ & $i == 1$)
            Add $w$ and its AAI to $result$
        end
    else
        $r_{left} \leftarrow r(w.leftchild)$
        $lset \leftarrow \{i | i \leq r_{left}, i \in set\}$
        $rset \leftarrow \{i - r_{left} | i > r_{left}, i \in set\}$
        if ($lset \neq \emptyset$)
            set the higher bit of $w.st$ to 1
            $BatchQuery(w.leftchild, lset, result)$
        end
        if ($rset \neq \emptyset$)
            set the lower bit of $w.st$ to 1
            $BatchQuery(w.rightchild, rset, result)$
        end
    end
end
$result \leftarrow \emptyset$
$BatchQuery(T.root, queryset, result)$
for each $leaf_j$ in $result$
    perform update operation required by $request_j$ in $batchreq$
    $w \leftarrow leaf_j$
    while ($w$ is not the new root $w'_R$)
        set the corresponding bit of $w.st$ to 0
        if ($w.st \neq '00'$)
            break
        end
        updating and rebalancing $w$
        $w \leftarrow w.parent$
    end
end
$P_{bu} = \left(\{w_j, \Omega_j\}, w'_R, sig_{sk}\left(l(w_R)\right)\right)$
```

5. Security Analysis

Theorem 1: If there exists a collision-resistant hash function which is used in the construction of RB-MAT, then the proposed verification scheme for data dynamics is secure.

Proof: The theorem will be proved in two steps: First, we show that the received auxiliary authentication information from the cloud including ranks of nodes is correct since the probability for the cloud to forge hash values of nodes on RB-MAT to pass the verification is negligibly small. Second, data dynamics could be verified by checking the root of RB-MAT.

The auxiliary authentication information returned from the cloud mainly consists of two parts: the labels and ranks of node siblings on the path from the leaf w_i to the root. According to our scheme, once the block corresponding to w_i is updated, the client receives the auxiliary authentication information as well as the old signed root of the RB-MAT. It reconstructs nodes on the path from w_i to the root of RB-MAT in local based on the labels and ranks of nodes received. If the cloud tampers either the returned ranks or labels, the root label computed with them will be incorrect due to the collision-resistant property of hash function. So once the verification of the signed root is passed, both the labels and ranks received must be authentic. Therefore, the difficulty of the cloud to provide false auxiliary authentication information without being detected is the same as breaking the security of the hash function used in RB-MAT.

The shape of a RB-MAT is maintained by the cloud. Once RB-MAT is imbalanced, the cloud rebalances it by adjusting nodes in the path from the updated leaf node to the root. Then the root is computed and sent to the client. At the same time, the client adjusts the shape of

RB-MAT locally. The cloud and the client will maintain the same shape of RB-MAT which is determined by the same rebalancing algorithm. Since the correctness of nodal information as well as the shape of RB-MAT before rotation is proved previously, the probability of a forge root generated by the cloud to pass the verification of clients is negligibly small.

6. Evaluation

To demonstrate the efficiency of our scheme, we conduct experiments using C++ on a system with Intel Core i5-4590 CPU @ 3.30GHz, 8GB RAM and a 1TB hard drive. Algorithms are implemented with the help of using the Pairing-Based Cryptography (PBC) library version 0.5.14 and the crypto library of OpenSSL version 1.0.2h. In this section, we compare the efficiency of our rank-based query method to that of the linear index table. The performance of our RB-MAT in basic data updating operations is also tested. We vary the balance criterion δ to different values and compare the average overhead of updating under relaxed balance. Finally, we conduct experiments to see the improvement by our batch updating scheme in contrast to the traditional single method.

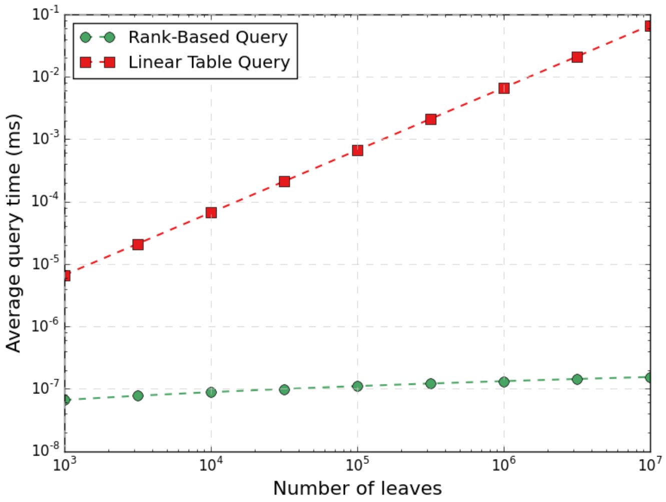


Fig. 4. Comparison of rank-based query and linear index table

First of all, we compare our rank-based query method to the one with linear index table. The total number of blocks used in the experiment varies from 10^3 to 10^7, which means the scale of RB-MAT and index table to be constructed. For each kind of scale, we randomly choose 1000 blocks to query using both of these two methods respectively. The average time of querying one block is shown in **Fig. 4**. For better presentation, both the X and Y axes are drawn in logarithmic coordinates. No matter which kind of method, the average time increase along with the total number of data blocks, because both the depth of RB-MAT and the length of linear index table are growing. However, the rank-based query method shows significant advantage that its speed of growth is much lower than that of linear structure. In theory, the complexity of rank-based query is $O(\log n)$, in contrast to $O(n)$ of index table. In fact, when the total number of blocks increases up to 10^7, the average time of query a block using index table exceeds 10^5 times as long as that of rank-based method.

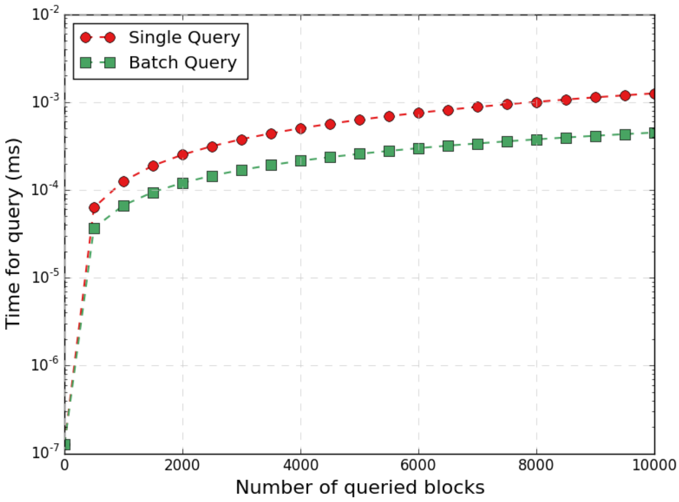
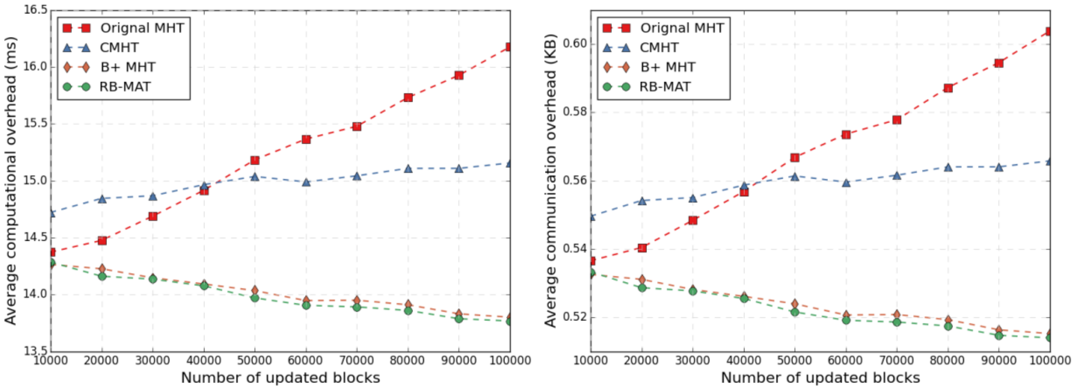


Fig. 5. Comparison of batch and single query

We also compare our batch query method to the single one in the case of searching multiple blocks. Here we fix the total number of blocks to 10^6 and change the number of required blocks from 10^0 to 10^4. We query all the randomly selected blocks at one time via the batch method, then repeatedly invoke the single query to complete the same task. The query time for both these two methods is presented in **Fig. 5**, where Y axis is drawn as logarithmic coordinate. The result indicates that in the case of searching multiple blocks, the query time of using batch method is obviously lower than that of repeatedly conducting single query. Even though 10^4 blocks are only 1% of the total data, batch querying can save nearly 2/3 of the query time than before. The main reason for this phenomenon is due to that batch query reduces repeated visiting to nodes in the first few levels of RB-MAT. Considering the case of integrity verification which needs frequent querying large amount of blocks, batch query will undoubtedly play an efficient role in such applications.

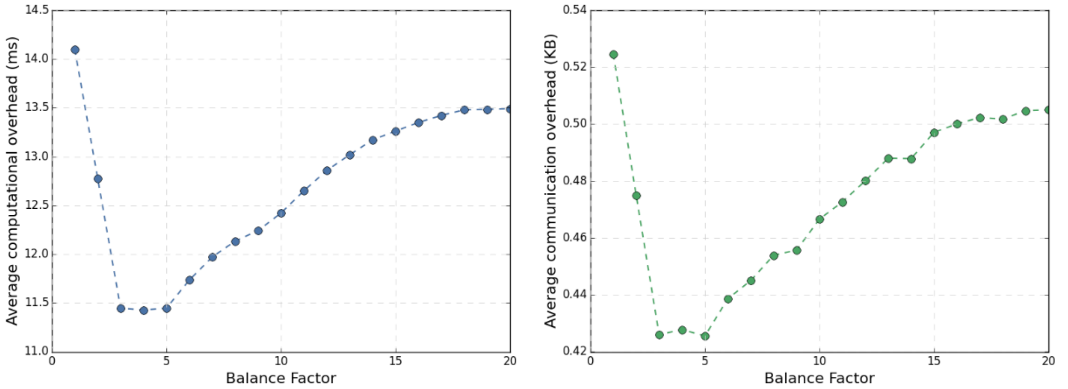
The efficiency of basic updating operations in our scheme is compared with the ones using basic MHT, CMHT and B+ MHT. Here we set the balance criterion $\delta = 1$, which means the strict balance. We segment a file into 10^6 blocks and update these blocks for 10^5 times, each time one block, randomly choosing insertion, deletion or modification. The experimental results are shown in **Fig. 6**. The average computational and communication overheads of our scheme and the B+ MHT one are significantly lower than the other two schemes. The original MHT suffers from the imbalance problem while the low efficiency of CMHT is due to its heavy computational and communication overhead in deletion case. It should be noted that the overhead of B+ MHT is very similar to the RB-MAT with $\delta = 1$ because they are both strictly balanced tree structure. However, along with relaxing the balance criterion, RB-MAT shows more advantages due to the decrease of rebalancing frequency, which we will analyze in next part.



(a) Computational overhead (b) Communication overhead

Fig. 6. Comparing the overhead of basic updating in different schemes

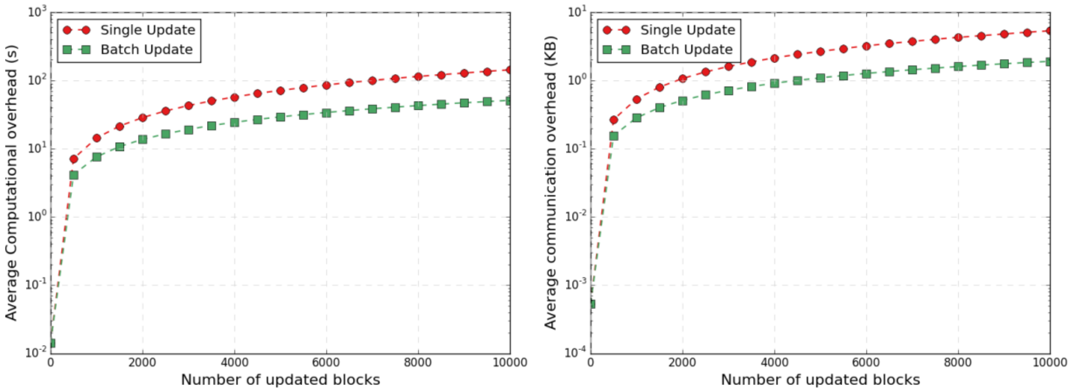
To evaluate the influence of relaxed balance, we vary the balance criterion from 1 to 20. In every case, we also perform random data updating operations as before and compare their average computational and communication overhead. In **Fig. 7**, we can easily find that the best performance appears in the range of $\delta \in [3,5]$. Actually, the average ratio of rebalancing occurrence in updating processes under $\delta = 1$ is up to 34.58%, and fall rapidly to only 3.24% when $\delta = 3$. When $\delta > 13$, the rebalancing ratio is even less than 0.1%. However, larger δ does not mean definitely lower overhead. Along with the accumulation of imbalance, the number of nodes involved in rebalancing will also grow rapidly, leading to more computation and communication. That is the reason why the average overhead increases when $\delta > 5$.



(a) Computational overhead (b) Communication overhead

Fig. 7. The overhead of data updating under different balance criterion

We also show the improvement in computational and communication overhead for update verification made by the batch update. We divide the file into 10^6 blocks and choose different numbers of them to be updated, ranging from 10^0 to 10^4. Since relaxed balance is not concerned in this part, the balance criterion of RB-MAT is set to $\delta = 1$. The experimental result is shown in **Fig. 8** and logarithmic coordinate is used for Y axis. As the number of updated blocks growing, the improvement by our batch update scheme becomes more significant. In the case of updating 10^4 blocks, the batch updating method reduces about 65% overhead of the single one. One reason for this phenomenon is that the more blocks are updated, the more shared nodes avoid to be repeatedly updated. In addition, rebalancing in batch update reduces some unnecessary rebalancing operations, which also contributes to the increase of efficiency.



(a) Computational overhead (b) Communication overhead

Fig. 8. The overhead improvement for update verification by batch update

7. Conclusion

In this paper, we study how to raise efficiency for the existing dynamic provable data possession schemes. To solve their problems, we optimize the query and rebalancing method in data dynamics scheme, and fill in the blank of batch updating scheme. In order to achieve these targets, we propose a novel data structure named Rank-Based Merkle AVL Tree which make use of ranks to describe the block indices and has update locks designed for processing in multiple branches. Our scheme is compared with previous work through experiments and the result shows that the proposed scheme has better performance. In future work, we will try to find out how to determine the best relaxed balance criterion for RB-MAT.

References

[1] Abdul Razaque and Syed S. Rizvib, "Triangular data privacy-preserving model for authenticating all key stakeholders in a cloud environment," *Computers & Security*, vol.62, pp. 328-347, September, 2016. Article (CrossRef Link)

[2] Moni Naor and Guy N. Rothblum, "The Complexity of Online Memory Checking, " *Journal of the ACM*, vol.56, no.1, pp. 1-46, January, 2009. Article (CrossRef Link)

[3] Alina Oprea , Michael K. Reiter and Ke Yang, "Space-Efficient Block Storage Integrity," in *Proc. of 12th Annual Network and Distributed System Security Symposium (NDSS)*, 2005. Article (CrossRef Link)

[4] A. Juels and B.S. Kaliski Jr., "Pors: Proofs of Retrievability for Large Files, " in *Proc. of 14th ACM Conference. Computer and Communication Security (CCS '07)*, pp. 584-597, October 29-November 2, 2007.Article (CrossRef Link)

[5] H. Shacham and B. Waters, "Compact Proofs of Retrievability, " in *Proc. of 14th Internationl Conference on Theory and Application of Cryptology and Information Security (ASIACRYPT '08)*, pp. 90-107, December 7-11, 2008. Article (CrossRef Link)

[6] K. D. Bowers, A. Juels and A. Oprea, "Proofs of retrievability: Theory and implementation, " in *Proc. of the 2009 ACM workshop on Cloud computing security*, pp. 43-54, November 9-13, 2009. Article (CrossRef Link)

[7] E.C. Chang and J. Xu, "Remote Integrity Check with Dishonest Storage Server, " in *Proc. of 13th European Symp. Research in Computer Security (ESORICS '08)*, pp. 223-237, October 6-8, 2008. Article (CrossRef Link)

[8] M. Naor and G. N. Rothblum, "The complexity of online memory checking, " in *Proc. of 46th Annual IEEE Symposium on Foundations of Computer Science (FOCS'05)*, pp. 573-582, October 23-25, 2005. Article (CrossRef Link)

[9] N. Cao, S. Yu, Z. Yang, W. Lou and Y. T. Hou, "LT codes-based secure and reliable cloud storage service," in *Proc. of The 31st Annual IEEE International Conference on Computer Communications (INFOCOM 2012)*, pp. 693-701, March 25-30, 2012. Article (CrossRef Link)

[10] J. Li and B. Li, "Cooperative repair with minimum-storage regenerating codes for distributed storage," in *Proc. of The 33rd Annual IEEE International Conference on Computer Communications (INFOCOM 2014)*, pp. 316-324, April 27-May 2, 2014. Article (CrossRef Link)

[11] X. Tang, Y. Qi and Y. Huang, "Fragile Watermarking Based Proofs of Retrievability for Archival Cloud Data, " in *Proc. of The 15th International Workshop on Digital-forensics and Watermarking (IWDW)*, pp. 296-311, September 17-19, 2016. Article (CrossRef Link)

[12] C. Wang, Q. Wang and K. Ren, "Ensuring data storage security in cloud computing, " in *Proc. of International Workshop on Quality of Service*, pp. 1-9, July 13-15, 2009. Article (CrossRef Link)

[13] G. Ateniese, R. Burns, R. Curtmola, J.Herring, L. Kissner, Z. Peterson and D. Song, "Provable Data Possession at Untrusted Stores," in *Proc. of 14th ACM Conference. Computer and Communication Security (CCS '07)*, pp. 598-609, October 29-November 2, 2007. Article (CrossRef Link)

[14] G. Ateniese, R.D. Pietro, L.V. Mancini and G. Tsudik, "Scalable and Efficient Provable Data Possession, " in *Proc. of 4th International Conference on Security and Privacy in Communication Networks (SecureComm '08)*, pp. 1-10, September 22 – 25, 2008. Article (CrossRef Link)

[15] Y. Zhu, H. Wang, Z. Hu, G. J. Ahn, H. Hu, and S. S. Yau, "Dynamic audit services for integrity verification of outsourced storages in clouds, " in *Proc. of ACM Symposium on Applied Computing (SAC 11)*, pp. 1550–1557, March 21 - 24, 2011. Article (CrossRef Link)

[16] Yan Zhu, Hongxin Hu, Gail-Joon Ahn and Mengyang Yu, "Cooperative Provable Data Possession for Integrity Verification in Multi-Cloud Storage, " *IEEE Transactions on Parallel and Distributed Systems*, vol.23, no.12, pp. 2231–2244, December, 2012. Article (CrossRef Link)

[17] Kan Yang and Xiaohua Jia, "An Efficient and Secure Dynamic Auditing Protocol for Data Storage in Cloud Computing, " *IEEE Transactions on Parallel and Distributed Systems*, vol. 24, no. 9, pp.1717-1726, September, 2013. Article (CrossRef Link)

[18] Hao Jin, Hong Jiang and Ke Zhou, "Dynamic and Public Auditing with Fair Arbitration for Cloud Data," *IEEE Transactions on Cloud Computing*, vol.PP, no.99, pp. 1-1, September, 2016. Article (CrossRef Link)

[19] Hui Tian, Yuxiang Chenm, Chin-Chen Chang, Hong Jiang, Yongfeng Huang, Yonghong Chen and Jin Liu, "Dynamic-Hash-Table Based Public Auditing for Secure Cloud Storage," *IEEE Transactions on Service Computing*, vol. 10, no. 5, pp. 701-714, September-October, 2017. Article (CrossRef Link)

[20] C. Erway, A. Kupcu, C. Papamanthou and R. Tamassia, "Dynamic Provable Data Possession, " in *Proc. of 16th ACM Conference. Computer and Communication Security (CCS '09)*, pp. 213-222, November 9-13, 2009. Article (CrossRef Link)

[21] Qian Wang, Cong Wang, Kui Ren, Wenjing Lou and Jin Li, "Enabling Public Auditability and Data Dynamics for Storage Security in Cloud Computing, " *IEEE Transactions on Parallel and Distributed Systems*, vol. 22, no. 5, May, 2011. Article (CrossRef Link)

[22] Chang Liu, Rajiv Ranjan, Chi Yang, Xuyun Zhang, Lizhe Wang and Jinjun Chen, "MuR-DPA: Top-down Levelled Multi-replica Merkle Hash Tree Based Secure Public Auditing for Dynamic Big Data Storage on Cloud," *IEEE Transactions on Computers*, vol. 64, no. 9, pp. 2609-2622, September, 2015. Article (CrossRef Link)

[23] X. Tang, Y. Qi and Y. Huang, "Reputation Audit in Multi-Cloud Storage through Integrity Verification and Data Dynamics, " in *Proc. of 2016 IEEE 9th International Conference on Cloud Computing (CLOUD)*, pp. 624-631, June 27-July 2, 2016. Article (CrossRef Link)

[24] Z. Mo, Y. A. Zhou, S. G. Chen and C. Z. Xu, "Enabling Non-repudiable Data Possession Verification in Cloud Storage Systems, " in *Proc. of 2014 IEEE 7th International Conference on Cloud Computing (CLOUD)*, pp. 232-239, June 27-July 2, 2014. Article (CrossRef Link)

[25] Z. Mo, Y. A. Zhou and S. Chen, "A Dynamic Proof of Retrievability (POR) Scheme with O(log n) Complexity," in *Proc. of 2012 IEEE International Conference on Communications (ICC)*, pp. 912-916, June 10-15, 2012. Article (CrossRef Link)

[26] Z. Mo, Q. J. Xiao, Y. A. Zhou and S. G. Chen, "On Deletion of Outsourced Data in Cloud Computing," in *Proc. of 2014 IEEE 7th International Conference on Cloud Computing (CLOUD)*, pp. 344-351, June 27-July 2, 2014. Article (CrossRef Link)

[27] CC. Foster, "A Generalization of AVL Trees," *Communications of the ACM*, vol. 16, issue. 8, pp. 513-517, 1973. Article (CrossRef Link)

[28] E. Stefanov, M. Van Dijk, A. Juels and A. Oprea, "Iris: a scalable cloud file system with efficient integrity checks," in *Proc. of the 28th Annual Computer Security Applications Conference*, pp. 229-238, December 3-7, 2012. Article (CrossRef Link)

[29] David Cash, Alptekin Küpçü, and Daniel Wichs. "Dynamic Proofs of Retrievability Via Oblivious RAM," *Journal of Cryptology*, vol. 30, no. 1, pp. 22-57, January, 2017. Article (CrossRef Link)

[30] N. Chandran, B. Kanukurthi and R. Ostrovsky, "Locally updatable and locally decodable codes," in *Proc. of Theory of Cryptography Conference*, pp. 489-514, February 24-26, 2014. Article (CrossRef Link)

Yining Qi received her bachelor's degree in Electronic Engineering from Tsinghua University, Beijing, China. She is currently working towards the master degree in Tsinghua National Laboratory for Information Science and Technology, Beijing, China. Her major research interests include data security in cloud computing, parallel and distributed system, reliability in cloud computing, cloud service level agreement and digital watermarking.

Xin Tang received the PhD degree in Computer Science from Beijing University of Posts and Telecommunications, Beijing, China in 2015. He is currently working as a post-doctoral researcher in Tsinghua National Laboratory for Information Science and Technology, Beijing, China. His major research interests include cloud data auditing, scalable distributed data storage, reliability in cloud computing, cloud service level agreement and digital watermarking.

Yongfeng Huang is currently a professor at the Department of Electronic Engineering, Tsinghua University, Bejing, China. He has been working on information hiding, digital watermarking, digital forensics, cloud security, data mining and natural language processing. He is a senior member of the IEEE.