Clone-based Data Index in Cloud Storage Systems

Jing HE[1,2], Yue WU[1] , Yang FU[2] and Wei ZHOU[2,a]

[1] *School of Computer Science and Engineering, University of Electronic Science and Technology of China, 611731 Chengdu, P.R.China*
[2] *School of Software, Yunnan University, 650091 Kunming, P.R.China*

Abstract. The storage systems have been challenged by the development of cloud computing. The traditional data index cannot satisfy the requirements of cloud computing because of the huge index volumes and quick response time. Meanwhile, because of the increasing size of data index and its dynamic characteristics, the previous ways, which rebuilding the index or fully backup the index before the data has changed, cannot satisfy the need of today`s big data index. To solve these problems, we propose a double-layer index structure that overcomes the throughput limitation of single point server. Then, a clone based B+ tree structure is proposed to achieve high performance and adapt dynamic environment. The experimental results show that our clone-based solution has high efficiency.

1 Introduction

With rapid development of information technology, huge amount digital data brings great challenges to data storage and data retrieve. Recently, the cloud storage techniques were proposed to satisfy the huge storage requirements. These cloud storage systems can provide higher scalability and fault tolerance than existing relational database system, because they adopt distributed file systems to manage large scale datasets. Users share a "black-box" which known as Cloud that consists of a large number of interconnected data nodes. They can tailor the storage resources for their own purposes from the infinite amount of resources that Cloud systems can provide.

However, due to storage schema is changed in cloud environment, the traditional data index structures are unable to directly be transplanted to the current cloud storage systems. For example, the B+-tree has been successfully used in many disk file index systems, such as XFS[1] and JFS (Journal File System)[2]. Paper [3] proposed a distributed B+-tree structure to construct data index for cloud storage systems. Although this structure is not only easy to implement but also has high scalability and fault tolerance, it consume too much memory space with the amount of data increasing, besides, the enormous index is difficult to maintenance. Therefore it is not suitable to directly use traditional B+ tree in cloud environment.

Meanwhile, under the dynamic cloud environment, users can only see the current value of the data now. In practice, some specific data needs to be traced for its value history. For example, the products' price adjusting history is important to sales decision. It is a useful method that duplicated the different versions of the same records on the disk. But the versions' management is so complex. Even the disk space cost is unbearable for big data.

To address these problems, this paper proposes a double-layer index framework firstly. Then, a clone based B+ tree structure is designed, which can be dynamic modified during processing. Historical value of records can be traced and even to be modified according to specific requirements. Finally, the experimental results show that our clone-based solution has high performance.

2 Related works

The existing cloud storage systems mainly include Yahoo's PNUTS [4], Amazon's Dynamo [5], Facebook's Cassandra [6], Google's Bigtabe [7] and its open-source variant HBase [8], which are designed to carry out the large-scale distributed storage. They generally adopt pairs of key-value to organize data. It is efficiently when using the decided keyword to retrieve data in these systems, while it cannot satisfy users' time requirement when users send out non-key filed or multiple attributes query requests. Therefore, many research works have focused on construct data index for cloud storage systems.

Sai Wu et.al [9] proposed a double layer cloud index. In this framework, the cloud index is consists of two components: local index and global index. The local index is used to organize data on each individual data nodes, while the global index can locate data's storage location among the cluster.

Paper [10] presents an improved B+ tree index. This solution builds a local B+-tree index for each data node that only indexes data on that node. With an adaptive algorithm, a proportion of the local B+-tree nodes are

[a] Corresponding author: zwei@ynu.edu.cn

published to the BATON [11] overlay. It is efficiently for single attribute query.

An R tree and CAN (Content Addressable Network) based multi-dimensional index schema called RT-CAN is proposed in paper [12]. In RT-CAN, a CAN [13] overlay is construct on top of local R-tree index. Furthermore, a dynamic index nodes selection algorithm and a cost model are proposed for RT-CAN. This solution provides high performance for multi-attributes query.

Paper [14] proposed another efficiently double layers index that support multi-dimensional query. Its local index using the improved quadtree, and global index adopts Chord overlay. Based on quadtree's coding, this index structure makes local index to the global index node release strategy is more concise.

However, these index frameworks are difficultly be applied to existing cloud storage systems because they are mostly adopt a P2P overlay as their global index while current cloud storage systems are master-slave architecture. Moreover, above cloud index systems do not support dynamic changing tracing and specific historical records modifying. That means, although we have a mount of huge volumes storages space, we can only search data block through current index systems, but cannot access historical data block by index systems.

3. Double-layer index framework

In distributed storage systems, a large-scale data set is usually divided into many small data sheets called data shard through horizontal partitioning method. Then, these small data shards are distributed stored on many different data nodes with specific load balance algorithm. For improve the query performance, it is a simple method that built a global distributed data index for the whole data set. However, this kind of global distributed data index is difficult to maintenance, and it will take up a lot of memory space on each data node. Once appeared, double layer index framework has gained much attention in the field of cloud storage.

In this paper, a B+ tree based double layer index framework is showed below.

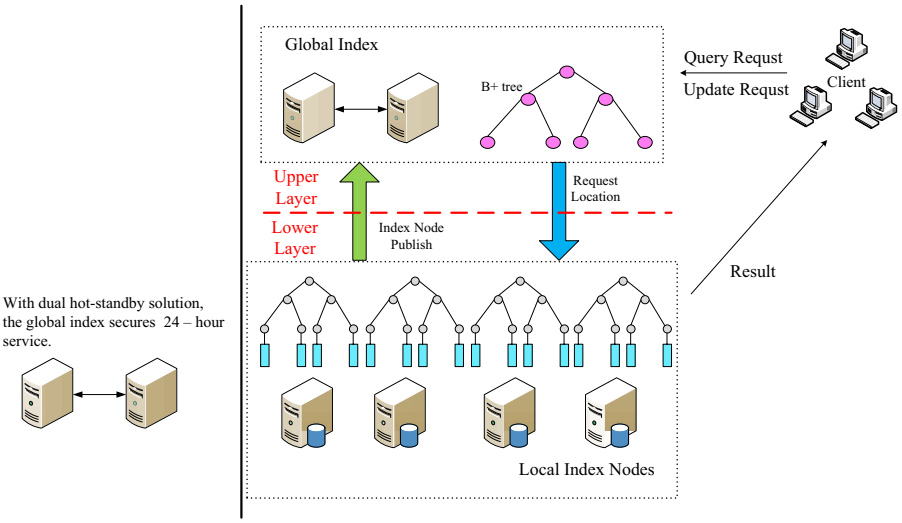

Figure 1. The B+ tree based double-layer index framework.

As shown in the Figure1, there are two layers (upper layer and lower layer). In the upper layer, it is a global index. That index store in a cloud server by the structure of B+ tree. This server stores the index nodes without the data. It uses the RPC (Remote Procedure Call) protocol to communicate between the cloud index server and local server. In the lower layer, the circles represent the index nodes of the local data server. The rounded rectangles represent final data blocks. Every local data server maintains a local index that is built by B+ tree.

When user proposed a query request, it is send to the global index server which performs index retrieve on global B+ tree to find the local data servers that may contain the query results. Then, the query request is redirected to the corresponding local data servers. Finally, each selected local data server start to retrieve the data on its own local indexes, and return the query results to the end user.

Compared with unitary key-value pair storage schema, our cloud index can help to cope with range queries. However, since the B + structure is adopted to the entire index, if a problem arises in any index node, such as, index information error and node information loss, the other nodes will be affected. The nearer the necrotic index node to the root node, the more data node will be affected. Therefore, the fast backup and recovery scheme of the index structure deployment is greatly important. It is necessary to study deeply in efficient clone-base B-tree in cloud environments.

4. Clone-based B+ tree Structure

Previous cloud index systems can only record current index information, which is just a one version index. However, due to huge storages, the cloud storage system can even store all data block update information. It requires that the cloud index system provide multi-version index function, which is used to trace historical records.

4.1. Background of clone

Clone and snapshot are two important techniques which can be used to replicate the system in certain time. Compare with clone technique, the weakness of snapshot is that it cannot be modified. Any attempt to change the snapshot mirror is a kind of disaster, because the modification to the snapshot may cause the damage to the original data. Instead clone can solve this problem. Paper [15] proposed a basic clone model.

Let Tp is a B+ tree and Tq is a clone mirror of Tp, a tree-based structure's clone model should have following properties:

(1) Space usage rate: Tp and Tq should share the common node as much as possible.

(2) Time usage rate: creating Tq from Tp should take little time.

(3) Number of clones: Tp can be cloned many times.

(4) Multiple-clone: Tq is also can be cloned.

Typically, fully backup the whole tree is the easiest way to clone which can store a complete clone mirror for all over data at a certain time. However, this method is time inefficiency and the fully backup version will occupy much more storage spaces.

In this section, an improved clone model is designed to overcome above weaknesses. In which, an indicator named as reference count is added to every node for a B+ tree. Through check the value of this reference count, the index system determines a node is cited by how many index versions. Then, there is only partial index nodes should be replicated according to specific rules, while other nodes can be shared by different index versions.

4.2 Clone-based B+ tree Creation

An original B+ tree index is shown in figure 2(a). It is no double that each node is belong to one tree, so the reference count on each node is 1.

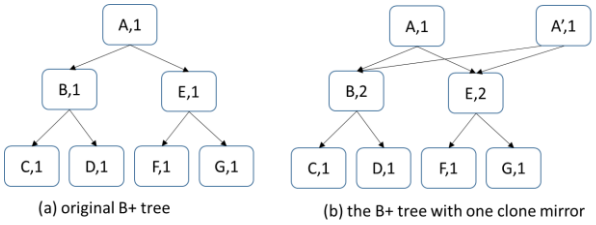

Figure 2. Create a clone-based B+ tree.

In figure 2(b), a clone tree *A'* is created from original B+ tree *A*. Firstly, there is only one index node that is the root node *A* needs to be replicated as *A'*. Then, the reference count of index node *B* and *E* is increasing 1 respectively. Finally, a newly cloned B+ tree is produced. Our B+ tree clone model meets all previous described properties. By the same clone method, the original B+ tree *A* can be cloned again, and the clone mirror *A'* also can be cloned. Due to few index nodes need to be duplicated, this clone model has constant time efficiency and space efficiency. With multiple-version index function, users can retrieve or update data from any index tree of them.

4.3 Clone-based B+ tree Maintenance

When data modification (inserting or deleting) happens to the index tree, in order to ensure the isolation between each tree, the key ideas are described below:

(1) If the reference count of the index node which would be modified is 1, it is indicated that this node is belong to only one tree. The modification to this node cannot affect other index trees, so the traditional B+ tree update process is suitable in this situation.

(2) Otherwise, if the reference count of the index node which would be modified is larger than one, it is indicated that this node is belong to more than one tree. In this case, this index node needs to be duplicated before updated.

A typical data updating procedure is show as figure 3.

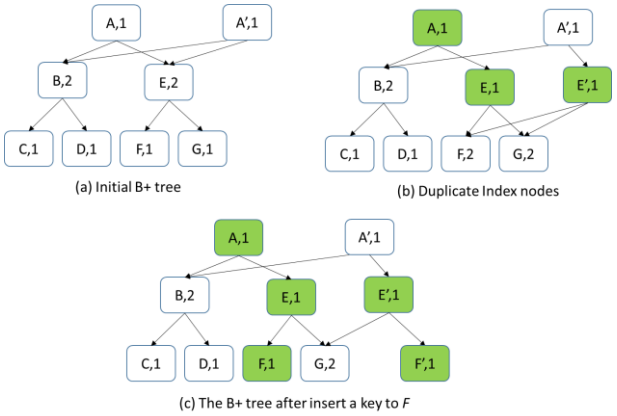

Figure 3. The procedure of data update

In figure 3(a), there are two B+ trees which are represented as original B+ tree *A* and its cloned B+ tree *A'*. Due to the index nodes *B* and *E* are belong to both index tree *A* and *A'*, their reference count are 2. Assume a new keyword will be inserted into index node *F* in B+ tree *A* now, the CSD (Clone Status Detect) function checks the reference count of all nodes in the path from root to *F* will be triggered. Firstly, since the reference count of index node *E* is 2, the node *E* is replicated as *E'* which cause the reference count of index node *F* and *G* increase 1 and the reference count of index node *E* decrease 1. Then, Root *A'* points to *E'* as show in figure 3(b). The node *F* is replicated as *F'* and so on. Finally, the node *F* is updated in B+ tree *A* by traditional B+ tree modification algorithm while the index node *F'* in clone tree *A'* keep the same. The node split in B+ tree *A* has no effect on clone tree *A'*.

Usually, data deleted from B+ tree would lead to index node merge operation. Therefore, an index node merge algorithm is considered in our clone model.

As can be seen from figure 4(a), there are two trees, one is the original tree *A*, the other is the clone tree *A'*. Assume a delete operation which remove some keywords from index node *F* triggered nodes merge. The detail description about nodes merge procedure is show in figure 4. In figure 4(b), on the path from node *A* to *F*, those nodes whose reference counts greater than 1 are copied. *F* and *G* merged into a new index node *H*. Then, because the original B+ tree is no longer balanced, it

needs to be adjusted. The index node *B* and *E* merged into *I*. At last, the previous root node *A* is removed according to traditional B+ tree delete algorithm. In figure 4 (d), the original B + tree has completely changed while cloning B+ tree didn't has any effect. In our multi-version index system, the newly B+ tree *I* records current data information, at the same time, cloning B+ tree *A'* maintains historical information.

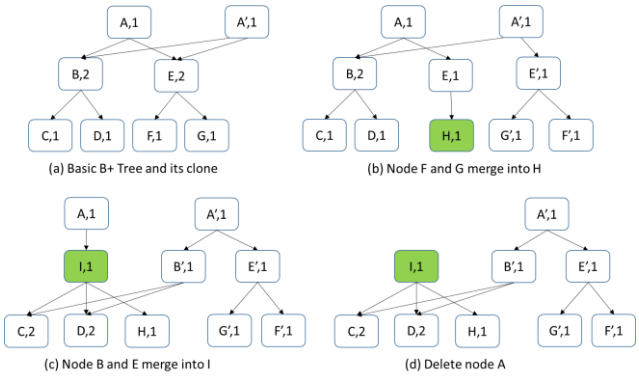

Figure 4. The procedure of index node merge

In order to save the storage space, cloning model should share the original node as much as possible. Therefore, the reference count of a node needs to be considered to determine whether it will be duplicated in our clone model. When data modification occurred, the CSD function is called to detect the value of a reference count. If a data update operation lead to a large number of index nodes adjustments, the efficiency of clone-base B+ tree maintenance algorithm will deteriorate for lots of CSD function recall. Fortunately, due to the branching factor of a B+ tree is usually big enough, the index adjustment is always small and local. The whole structure is stable. Let n represents the number of a B+ tree's leaf node, the time efficiency of our clone-base B+ tree maintenance algorithm is $O(log_2 n)$.

As for the algorithm, the CSD function is independent encapsulation, we only add the CSD function to where the node needs to be modified. The pseudocode of CSD function is below:

Input

curRoot: the root node of the modifying index tree currently.

curNode: the index node is modified currently.

refNode: the brother of *curNode* that needs to be merged with curNode.

Algorithm 1 CSD (*curRoot, curNode, refNode*)

```
1: tmpNode=curRoot;
2: preNode=null;
3: tmpKey=curNode.entries.get(0).getKey();
4: WHILE tmpNode != curNode
5:    FOR tmpNode.entries.get(i=0).getKey() TO
            tmpNode.entries.get(i=entries.size()).getkey()
6:      IF tmpKey>=tmpNode.entries.get(i).getKey()
            && tmpKey<tmpNode. Entries.get(i+1).getKey()
        THEN
7:          preNode = tmpNode;
8:          tmpNode=tmpNode.children.get(i);
9:      IF tmpNode.refCount > 1 THEN
10:         tmpNode.refCount --;
11:         preNode.children.set(i, tmpNode.clone());
12:         cloneNode=preNode.children.get(i);
13          cloneNode.refCount = 1;
14:     IF tmpNode.children!=null THEN
15:       FOR TNode=tmpNode.children.get(0) TO
              tmpNode.children.get(tmpNode.children.size())
16:       TNode.refCount ++;
          ENDFOR
17:     IF refNode != null THEN
18:       i = preNode.children.indexof(refNode);
19:       refNode.refCount --;
20:       preNode.children.set(i, refNode.clone());
21:       preNode.children.get(i).refCount=1;
22:     IF refNode.children!=null THEN
23:       FOR TNode= refNode.children.get(0) TO
              refNode.children.get(refNode.children.size())
24:       TNode.refCount ++;
          ENDFOR
25:   ENDFOR
26: ENDWHILE
```

In algorithm1, there are three input parameters which are the root node of current index tree, the index node which will be modified, and the node that will be merged with *curNode*. Firstly, the current index tree is traversed from root node *curRoot* to the modifying node *curNode* (line 4-8). Within the traversing process, the previous node of the *curNode* is stored in *preNode* (line 7-8). Then, if the reference count of the scanned index node is larger than one, this index node is cloned (line 9-13). The reference count of the original node is minus by one. If this modified index node named as *tmpNode* is not a leaf node, the reference count of its all child nodes need to be updated (line 14-16). At last, if index node merge happened which means the *refNode* is not null, the process is from the bottom up while the nodes traverse above is top-down (line 17-24). For all children of this *refNode*, their reference count is plus by one (line 22-24).

Due to this function is trigger before any modification to the index node, the branching factor of the B+ tree is important to the efficiency of clone-based index maintenance. If the branching factor is low, the probability of this CSD function is called is high. On the opposite, the higher branching factor will make the probability lower.

5. Experimental evaluation

The test platform used in this paper is Intel Core Duo T9900 3.06GHz 2 core 2 threads CPU, 8GB memory, 64 bit windows8.1 operating system. JDK edition is 1.7.0_60, eclipse edition is 4.3.2. To ensure that the Java virtual machine has enough memory to use, the execution parameter -Xmx4000m -Xms3000m is added at run time. Specifically, 4GB available memory and 3GB available stack is allocated to Java virtual machine dynamically.

By this optimized parameters setting, there will be no stack overflow even if the amount of the data is too large or function recursive too many times.

Compared with fully backup the index tree, cloning can save huge amount of storage spaces. Meanwhile, clone an index node is so quickly, its time consume is almost can be neglected. However, due to there exists multi-version index trees for clone-based index structure at one time, index maintenance, such as inserting or removing a data, is turn to costly. In this section, many laboratories are built to evaluate the performance of data inserting and data deleting for clone-based B+ tree index.

Limited to the platform performance, each B+ tree index in our experiments only has one original tree and one clone tree. To guarantee the accuracy of experiment, a certain number of data are randomly generated and stored in a link list. Using this link list, it makes the order that each time inserting is the same.

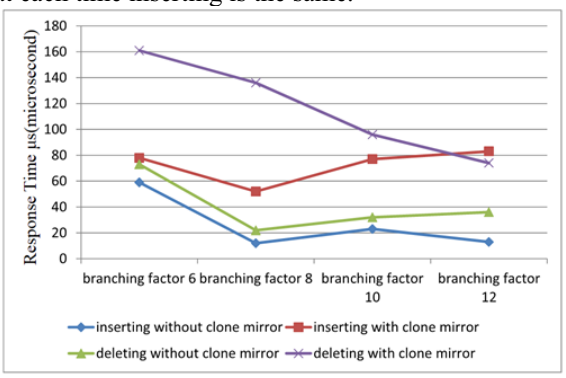

Figure 5. Time cost when inserting or deleting one thousands of data

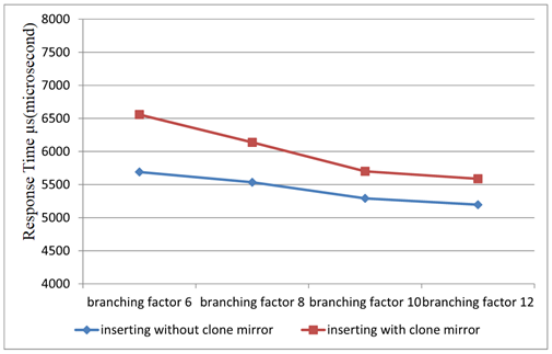

Figure 6. Time cost when inserting one million of data

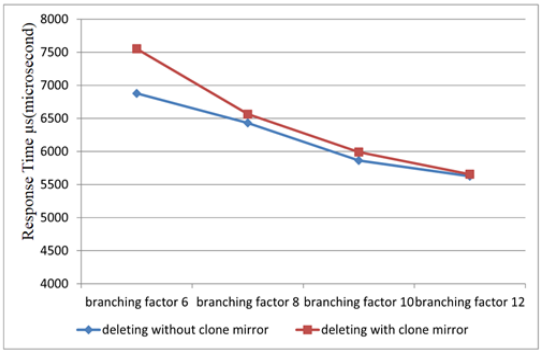

Figure 7. Time cost when deleting one million of data

Figure 5 shows that the addition of clone mirror makes the maintenance time cost much higher than original B+ tree structure. However, with the amount of data which inserting or deleting into the index tree increasing sharply, the extra time cost turns gently. In figure 6 and figure 7, when the amount of data that inserting or removing is one million, the operation response times which with clone mirror and without clone mirror have little difference. The loss of efficiency is acceptable. The reason is that the clone mirror is only duplicating a small part of index nodes which are modified. Under the big data environment, these duplicated index nodes is too small to the whole data set.

In addition, the above experiments verify the influence of the branching factor. In figure 7, when the branching factor is 6, due to deleting operation lead to some index nodes merge, then, the response time with clone mirror is much higher than the case without clone mirror because there is more CSD function call. Otherwise, when the branching factor is 12, the time cost is almost the same whenever with clone mirror or without clone mirror. The reason is the additional CSD function call turns to small.

6. Conclusion

This paper presents a double layer index structure based on B+ tree, this structure overcomes the throughput limitation triggered by single point server when index overload increase. Then a practical clone-based B+ tree model is designed. With clone mirror, our index tree is not a single version tree again, it is a multi-version tree. The different value of a record within a period of time can be traced through these different versions. Our clone model only backup modified nodes and share the other nodes in a B+ tree, therefore, this solution has advantages both in time complexity and space complexity. Finally, experimental results show that this clone-based B+ tree model has high performance and availability under dynamic cloud environment.

Acknowledgement

This work is supported by the National Natural Science Foundation of China (61363021); Science Research Fund of Yunnan Provincial Education Department (2014Y013);

References

1. Moshe Ba. *Linux File Systems*, 1nd ed., McGraw-Hill Companies, New York. (2001).
2. Vijayan Prabhakaran, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. *Analysis and Evolution of Journaling File Systems*, USENIX Annual Technical Conference (USENIX Association) (2005).
3. Marcos K. Aguilera, Wojciech Golab, Mehul A. Shah. *A practical scalable distributed B-tree*,

Proceedings of the VLDB Endowment, **1(1)**, pp. 598-609. (2008).

4. Brian F. Cooper, Raghu Ramakrishnan, Utkarsh Srivastava Adam Silberstein, Philip Bohannon, Hans-Arno Jacobsen, Nick Puz, Daniel Weaver and Ramana Yerneni, *PNUTS: Yahoo!'s hosted data serving platform*, Proceedings of the PVLDB. Auckland, New Zealand, pp. 1277-1288. (2008).
5. Giuseppe DeCandia, Deniz Hastorun, Madan Jampani, Gunavardhan Kakulapati, Avinash Lakshman, Alex Pilchin, Swaminathan Sivasubramanian, Peter Vosshall and Werner Vogels, *Dynamo: amazon's highly available key-value store*, Proceedings of the 21st ACM Symposium on Operating Systems Principles. Stevenson, Washington, USA, pp. 205–220. (2007).
6. Avinash Lakshman, Prashant Malik. Cassandra: a decentralized structured storage system, ACM SIGOPS Operating Systems Review, **44(2)**, pp. 35-40. (2010).
7. Fay Chang, Jeffrey Dean, Sanjay Ghemawat, Wilson C. Hsieh, Deborah A. Wallach. Mike Burrows, Tushar Chandra, Andrew Fikes, Robert E. Gruber. Bigtable: A distributed storage system for structured data. ACM Transactions on Computer Systems, **26(2)**, pp. 1-26. (2008).
8. Craig Franke, Samuel Morin, Artem Chebotko, John Abraham, and Pearl Brazier. *Distributed semantic web data management in HBase and MySQL cluster*, IEEE International Conference on Cloud Computing. New York, pp. 105-112. (2011).
9. Sai Wu, Kun-Lung Wu. An indexing framework for efficient retrieval on the cloud, IEEE Data Engineering Bulletin. (2009).
10. Sai Wu, Dawei Jiang, Beng Chin Ooi and Kun-Lung Wu. *Efficient B-tree Based Indexing for Cloud Data* Processing, Proceedings of the VLDB Endowment, **3(1)**, pp.1207–1218. (2010).
11. H. V. Jagadish , Beng Chin Ooi , Quang Hieu Vu. *BATON: A Balanced Tree Structure for Peer-to-Peer Networks*, Proceedings of the 31st international conference on Very large data bases. Trondheim, Norway, pp. 661-672. (2005).
12. Jinbao Wang, Sai Wu, Hong Gao, Jianzhong Li, and Beng Chin Ooi., *Indexing multi-dimensional data in a cloud*, Proceedings of the 2010 ACM SIGMOD International Conference on Management of data. Indianapolis, Indiana, USA, pp. 591-602. (2010).
13. S. Ratnasamy, P. Francis, M. Handley, R. Karp, and S. Shenker. A scalable content-addressable network. ACM Sigcomm Computer Communication Review, **355(4)**:161-172. (2002).
14. LinLin Ding, Baiyou Qiao, Guoren Wang and Chen Chen, *An efficient quad-tree based index structure for cloud data management*, Proceedings of the 12th International Conference on Web-Age Information Management. Wuhan, China, pp. 238-250. (2010).
15. Rodeh O., B-trees, shadowing, and clones, ACM Transactions on Storage (TOS), **3(4)**, pp. 2. (2008).