An Efficient Distributed B-tree Index Method in Cloud Computing

Huang Bin[1,*] and Peng Yuxing[2]

[1]*School of Mathmatic and Computer Science, Guizhou Normal University, Guiyang 550001, China;* [2]*School of Computer, National University of Defense Technology, Changsha 410073, China*

Abstract: To support online index and range queries, the Distributed B-tree is adopted to index the mass and rapidly increasing data in cloud computing. But current Distributed B-tree has three defects: low degree of concurrency, frequent node splitting and high cost of updates in clients. For above mentioned defects, this paper presents efficient distribute B-tree index in cloud computing environment, which effectively enhances the performance of the distributed B-tree index. First, it improves concurrent access by the distributed B-tree high concurrency access method based on node split history. Second, it reduces the splitting frequency by the method of dynamic changing node size. Finally, it reduces node update cost in all client buffers by the regional delayed update method. Experimental results show that, this method has high performance in cloud computing environments.

Keywords: Cloud computing, distributed B-tree, efficient, splitting frequency, splitting histories.

1. INTRODUCTION

Now, a lot of Internet applications that are based on massive data and provide various types of information services arise in cloud computing environment, such as Delicious [1], Flickr [2], Google Base [3], etc., and these massive application data rapidly increase [4]. In these systems, the keys of each data are processed by hash method, and accordingly all of the data are distributed to multiple storage nodes, so as to realize scalable storage for rapidly increasing massive data. Therefore, the hash function becomes the main index of data, and the required data can be quickly accessed according to the hash value of keys [5, 6, 7]. However, in addition to the data query *via* keys, users also turn to other properties for point search or range search [8]. For example, in an online video system (such as Youtube [9]), each video contains a variety of information, including video ID, program name, upload time, times of plays. The users can quickly access the video *via* its ID that is the key of each video, but they can also search for video by inputting program name or defining upload time range. Constructing secondary index is an important method to improve the query efficiency of non-key attribute. At present, in the cloud computing environment, inverted index, the commonly used secondary index, can scan all storage nodes by multiple MapReduce [10] processes and generate inverted files. Inverted index is an off-line batch process, and it cannot realize timely query of newly inserted data [8]. For example, the record inserted into Google Base cannot be accessed by users until it is re-indexed next time (maybe one day later). Therefore, the application system needs to provide an index method with online construction and range queries.

B-tree index can realize online indexing and range queries; however, because of the massive data in cloud computing environment, centralized B-tree cannot meet the demand in storage of mass index data, while distributed B-tree [11, 12] can meet this demand by distributing all nodes into storage servers. But existing method to construct distributed B-tree shows low performance, for the following reasons: (1) low degree of concurrency. For transaction method, when multiple users' concurrently operate B+ tree, these operated nodes and their ancestor nodes will be locked, which leads to all users serially operating B-tree, seriously affecting the operation efficiency. (2) High cost of update. Firstly, the node size of existing distributed B-tree is fixed, which will cause nodes to frequently split. The existing distributed B-tree needs to update all internal nodes in client buffers when some node is spited, which greatly affects the system performance.

According to the existing defects of the traditional distributed B-tree, including low degree of concurrency, frequent node splitting and high update cost, this paper presents efficient distribute B-tree (mark as EDB) index in cloud computing environment, including distributed B-tree high concurrency access method based on node split history and the regularly changing method of node size and regional delayed update, effectively enhancing the performance of distributed B-tree index.

2. RELATED RESEARCH

2.1. Distributed B-tree

In a traditional distributed B-tree [12], all nodes are distributed into multiple servers, and all internal nodes are buffered in each client, to realize multi-user concurrent access and improve the access efficiency. With delayed update, the node in client buffer can be updated accordingly to the corresponding node in servers, so that the synchronous update

*Address correspondence to this author at the School of Mathematics and Computer Science, Guizhou Normal University, Guiyang 550001, P.R. China; Tel: 18874501373; E-mail: hb415@163.com

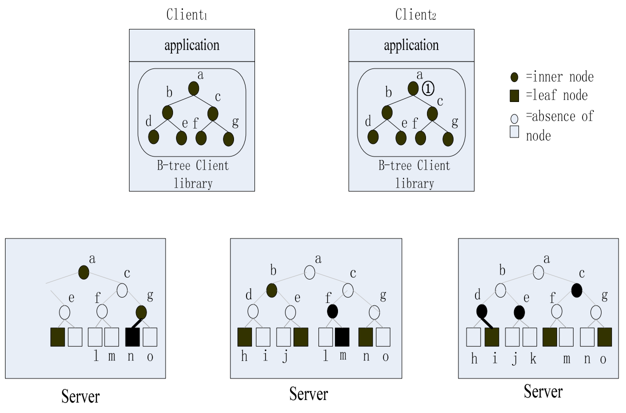

Fig. (1). Distributed B-tree.

costs are distributed to many operations, to reduce delayed time for some update operation.

Similarly, with B-tree in stand-alone environment, concurrent modification will also occur to distributed B-tree, namely, when data is inserted to the distributed B-tree, a leaf node and its several ancestor nodes may split. For example, in the distributed B-tree shown in Fig. (**1**), when an index value is inserted into node h, if the node h is full, then h will split, and then an index value will be inserted to node d; if the node d is full, then d will split, which may cause the nodes b and a to split. Node splitting can cause failure of concurrent modification operation. For example, when nodes h and i split simultaneously, data x and y will be inserted to node d, but it is possible that, node d will split into two nodes d and d' (when it is full). d' is distributed to another storage node, when y (or x) arrives, because of x (or y) cannot find the insertion position and fail to complete the insert operation.

2.2. B-tree Concurrency Control Method

In the stand-alone environment, the link established among nodes can eliminate concurrent modifications, but it is not suitable for distributed B-tree in cloud computing environment, because of delayed update mode, in the interval when a client visits the same node again, the corresponding node in server may split several times. According to the distribution strategy, the new split nodes may be distributed to other storage nodes, therefore, the second visit won't be realized unless the client passes through multiple server nodes.

A distributed B-tree concurrency construction method based on optimistic transaction concurrency control [13, 14] can eliminate the necessity to pass through multiple server nodes. But when the index data is inserted each time, distributed transaction will lock all the nodes distributed from the root to a leaf node, so the concurrent ability is low. For example, in Fig. (**1**), client 2 and client 1 insert index values respectively into nodes j and o. Client1 will lock node a in server 1, node b in server 2, node e in server 3 and node j in server 1 in sequence; Client 2 will lock node a in server 1, node c in server 3, node g in server 1 and node o in server 1 in sequence. If the client 1 locks node a first, then client 2 must wait for a moment.

By analyzing the process of client 1's and client 2's concurrent insert index value, we conclude that the entire path will be locked in each insertion operation, so all operations request to lock the root node, and the sequentially locking root node leads to serial operation of insertion into B-tree, leading to quite a low efficiency of construct index.

2.3. Method of Synchronous Update

With the lazy update method [15], the operation costs to update an object and its copies are distributed into the following subsequent operations, reducing the high costs of an update operation. However, for existing distributed B-tree when the nodes in client buffer are updated, all nodes buffered will be updated, thus high cost of update is still required.

3. THE CONCURRENCY ACCESS METHOD BASED ON NODE SPLIT HISTORY

3.1. The Recording Method of Node Split History

Each server contains a splitting log of B-tree nodes to record the split history in the server. According to the log structure shown in Fig. (**2**), each split of node is recorded in the log, in the structure of <LowValue, UpValue, ServerIP, IndexFileName, Version, preRecord>. LowValue and UpValue are respectively the minimum and maximum values of the index nodes; ServerIP is the machine number of storage node; IndexFileName is the name of the index node; Version is the version number of the index node; preRecord is a pointer pointing to the previous splitting record. In the log, the split histories of all nodes are connected into a linked list.

3.2. High Concurrency Access Algorithm of Distributed B-tree

In a distributed B-tree, each node in server is endowed with a version number, and each node in client's buffer also contains corresponding version number. Due to the delayed update strategy, the buffer's version number is smaller than that of the corresponding node sometimes, which shows the modification of the node in the server is not synchronized to the corresponding node in the buffer.

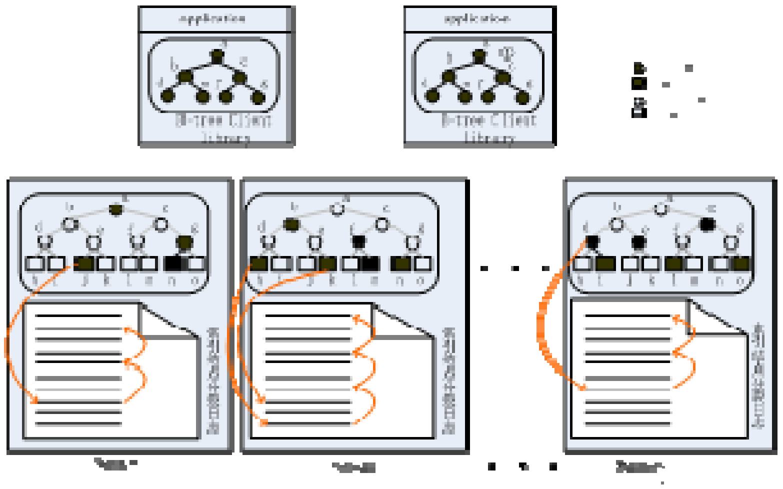

Fig. (2). Structure of node splitting logs.

In this paper, high concurrent access to distributed B-tree is realized according to the version number and node split history. Specifically, if the version number of accessed node is the same as that of the client buffer, the data of the node will be directly accessed to, or, this node's splitting log is accessed for finding the accessed node of next hop and its version number, and then transmits the access request to the next-hop node.

Based on the above idea, a <key, pointer> is inserted into a B-tree and the access is conducted according to the key, the distributed algorithm 1 and 2 are respectively explained below:

Algorithm 1: Insert (Key, Pointer)

```
For client:
1. Search for insertion position (host, node) and the node's version number (v) in the B-tree of client buffer;
2. Send the insertion request and v to the server host;

For server:
1. if (node. version==v) {
2.   insert <key, pointer> into node;
3.   if (node is full)
4.     update its ancestor node;
5. }else{
6.   search for the accessed object's position (host ', node') and v in the node's splitting log;
7.   send the access request and v to the server host';
8. }
```

Algorithm 2: Lookup (key)

```
For client:
1. Search for index node position (host, node) and its version number (v) in the B-tree of client buffer;
2. Send the search request and v to the server host;

For server:
1.search in node;
2 if (success) {
3.   return the results to client;
6. }else{
7.   search for the accessed object's position (host', node') and v' in the node's splitting log;
8.   if (success)
9.     send the access request and v to the server 'host';
10.  else
11     Return "Not exist" to the client;
12. }
```

According to the above algorithm, when modifying B-tree node, only the leaf node to be accessed is locked, thus the trouble that the path from root node to accessed leaf node will be locked in each access is eliminated; similarly, if modification of leaf node causes the modification of the internal node, only this node should be locked in each node modification, so this algorithm can greatly improve the efficiency of concurrent access.

4. METHOD TO CHANGE NODE'S SIZE

At present, the node's size of distributed B-tree (or order) is fixed. If the order is set smaller, there are a large number of nodes in B-tree, which means more frequent node splitting and lower performance of B-tree. Accordingly, the method to regularly adjust node size is designed, which contains two alternately performing processes, namely node quantity increase and node expansion, which effectively reduces the node split frequency and ensures even distribution of data.

(1) Node quantity increase process

In this process, fixed capacity (C) leaf node is set. With the data insertion, the leaf nodes split, and the quantity of leaf nodes (LN) gradually increases, until the quantity of new leaf nodes is the same as that of stored nodes (SN). It is assumed that this increase process is the ith times, then LN = × SN.

(2) Node expansion process

This process allows doubled leaf node capacity. It is assumed that this expansion process is ith times, and then the capacity of leaf node is $C = 2^{i-1} \times C_D$. Because of incomplete data after leaf node expansion, each leaf node can receive data. This process won't terminate until some leaf node will split.

5. REGIONAL DELAYED UPDATE METHOD

When a client accesses a leaf node, and if this leaf node has split or merged after the last visit, the parent node of this leaf node in this client should be updated. If the parent node in the server also has split or merged, the same update operation should be conducted. The nodes in client buffer will be updated with the following two strategies: (1) if n keys k_i+1, k_i+2,…, k_i+n, (split by child nodes on server) are added between key k_i and k_j, sub-trees of B (k_i+1),… , B (k_i+n) with n keys as roots will be buffered, and sub-tree B (K_i) and B (K_j) individually pointed by k_i and k_j will be respectively deducted by sub-tree B (k_i+1),… B (k_i+n); (2) if n keys k_i+1, k_i+2,…, k_i+n, (split by child nodes on server) are reduced from between key k_i and k_j, then sub-trees B (k_i+1),… , B (k_i+n) with n keys as roots will be eliminated, and k_i sub-tree B (k_i) pointed by k_i will be buffered.

Below, we prove the correctness of the regional sub-tree delayed update method.

Proof:

The following four situations to discuss:

(1) When the nodes in buffer remain in the same state with those in server, the user can correctly locate data;

(2) In the server, node b and its sub node c are updated, but in buffer only the node b is updated, which is caused by other descendant nodes' splitting. According to our update strategies, the update of node c will cause no changes in node b, or, the update of node b in buffer will also cause the change of node c. With the help of node splitting history log, whether leaf node splits or not, the data can still be correctly accessed.

(3) In the server, node b and its ancestors a are both updated, but in buffer only the node b is updated and the splitting of node b does not lead to that of node a. There is one path from a to b in buffer, thus a can access to all of b's leaf nodes.

(4) In the server, node b, its ancestor a and descendant c are all updated, but in buffer only the node b is updated, which is similar to case (2) and (3), and accordingly, the data can be properly accessed to.

To sum up, regional sub-tree delayed update method can guarantee the correctness of access to data in each client.

Comparative to traditional delayed update method, all of the entire B+ tree's internal nodes in buffer are compared and updated; while with regional sub-tree delayed update method, only a sub-tree's nodes are compared and updated, with the implication of greatly reduced updated nodes, thereby greatly improving the access performance.

6. PERFORMANCE EVALUATION

6.1. Experimental Environment

Our testing infrastructure had 126 machines on 4 racks connected by Gigabit Ethernet switches. Intra-rack bisection bandwidth was ≈14Gbps, while inter-rack bisection bandwidth was ≈6.5Gbps. Each machine had two 2.4GHz Intel Xeon CPUs, 4GB of main memory, and two 7200RPM SCSI disks with 200GB each. Machines ran Red Hat Enterprise Linux AS 4 with kernel version 2.6.9.

There are total 175 pairs of <Key, Pointer> values in each node of Tree B. one <key, Pointer> value takes up 22 bytes, including a Key, which is a 8 byte integer with value range of [0,109], and a pointer, which takes up 14 bytes. It consists of 2 parts, namely the IP address (4 bytes) and offset (8 bytes). Before experiment, B Tree has already had 400 nodes and 64,000 <key, pointer> pairs in 4 servers.

The nodes of B Tree are placed in server. Loads are generated from client, while server and client are located in different computers respectively. The memory of each server provides 32M buffer to Tree B and each client runs in 4 threads. They all access the same Tree B.

6.2. Concurrency Ability

We now evaluate the concurrency ability of our EDB strategy. For the purpose of comparison, we implement the Link strategy and distributed transaction strategy. In the experiment, there are three kinds of loads:

The inserting load: all operations are inserting operation. New keys are randomly generated uniformly at random from a space of 109 elements and inserted into the B-tree.

The searching load: all operations are searching operations. The starting point skey and ending point ekey of each searching range are all randomly picked up from the key set in Tree B

The hybrid load: the operations include inserting operation and searching operation. The key generation method of these two operations is same as the methods mentioned above.

With fixed node quantity, and increased server and client amount, the test results of the three loads (shown in Figs. (**3-5**)) displays that EDB method is better than Link method and transaction method in any of the three cases. EDB method is better than the transaction method, because, in each visit, all accessed nodes from the root to the leaf nodes deserve distributed lock, with the latter method, while only the nodes to

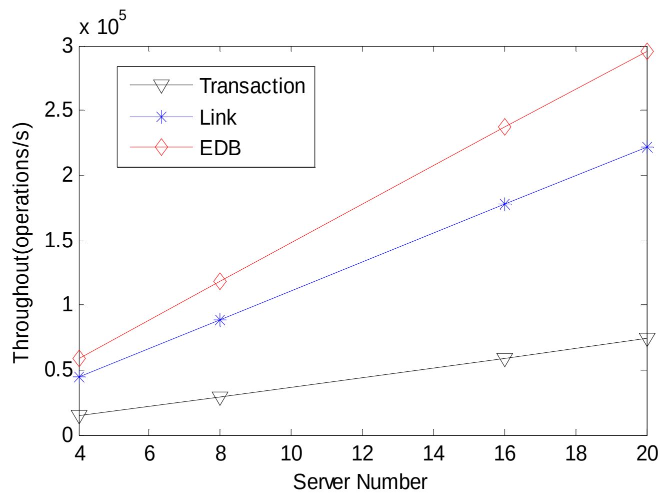

Fig. (3). Throughput under lookup-type load.

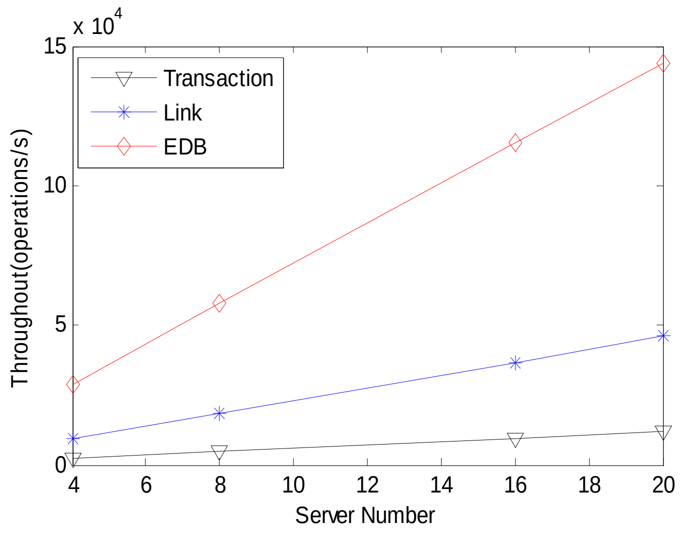

Fig. (4). Throughput under mixed load.

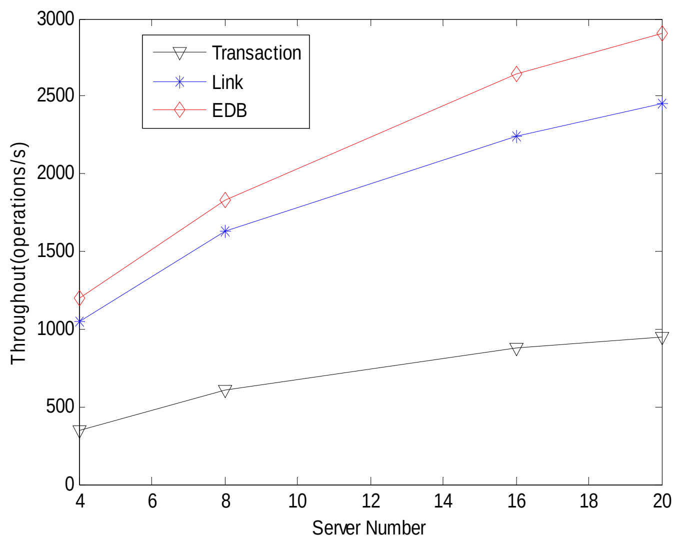

Fig. (5). Throughput under inserted load.

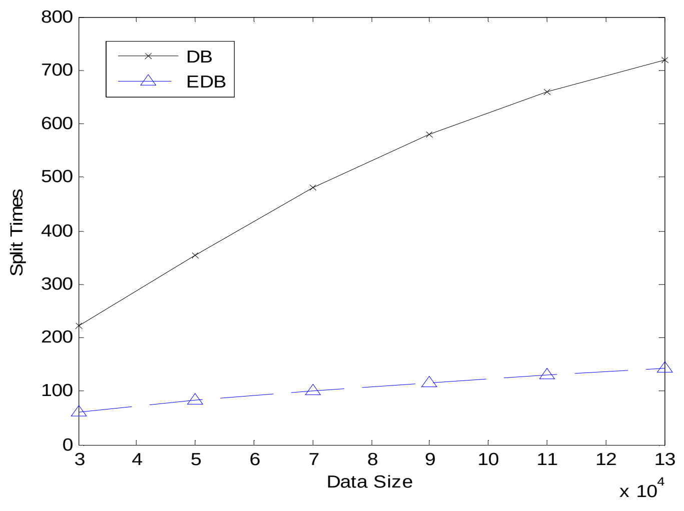

Fig. (6). Comparison of node splitting times.

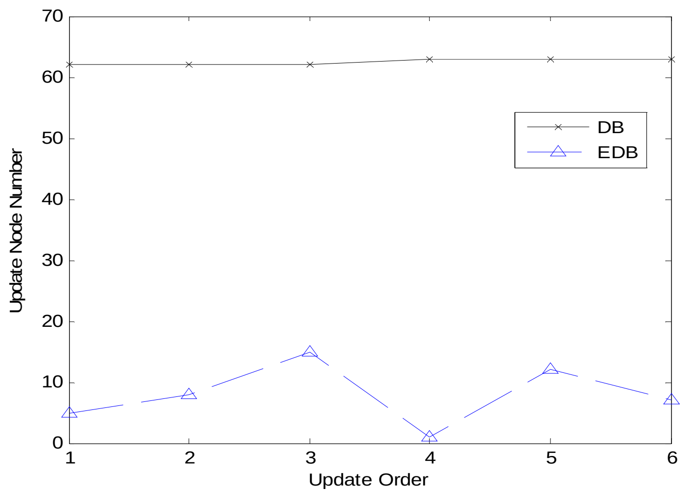

Fig. (7). Comparison of updated nodes.

be accessed will be locked. EDB method is better than Link method, because, with EDB method, the number of nodes to traverse is smaller than that with Link method.

In addition, EDB method and Link method have good expansibility, and the throughput increases with the increase in the number of servers in all of the three cases. For EDB, the growth rate of the throughput decreases with the increase in the number of servers in the inserting load and the hybrid load, which is mainly caused by the following two situations: (1) All of the data modifications are recorded in the log which thus then becomes bottleneck; (2) Massive insertion causes more frequent internal node splitting, thus more update in client buffer influences the access performance.

6.3. Split Frequency

In the experiments, traditional B-tree nodes' size is defined as 2KB, with EDB, nodes' original size is 2KB too. All nodes in EDB increase by 2KB in each expansion, and all data is randomly generated. In different data sizes, node split times is shown in Fig. (**6**). As it is shown, with the same data size, node split times with EDB is much less than that with traditional method.

6.4. Cost of Update in Buffer

In the experiments, the data are simultaneously inserted into distributed B-tree by one client, and the other client queries data from the distributed B-tree. In the contrast test, when the same data value is searched, the quantity of updated nodes in client buffer in each query is recorded (Fig. (**7**)). As is indicated, the quantity of nodes to be updated in client buffer is significantly reduced with EDB method than that with the traditional distributed B-tree. Because, in the traditional distributed B-tree, all nodes shall be buffered in each update, while only some of them will be updated with EDB method.

CONCLUSION

Our efficient distributed index EDB is used to eliminate the following questions. (1) Low degree of concurrency. When multiple users operate nodes, the transaction method needs lock operated nodes and all their ancestor nodes, which seriously affect the operational efficiency. EDB only needs to lock each operated node by recording node split history. (2) High cost of update. The node size of the existing distributed B-tree is fixed, which will cause frequent splitting of nodes. EDB decreases frequent splitting of nodes by allowing nodes' size to change regularly. Moreover, traditional distributed B-tree needs to update all internal nodes in client buffers when a node is split. EBD only updates some nodes by our regional delayed update strategy. We have described our efficient distributed index EDB and given the survey of the performance comparisons, and the performance results are encouraging.

CONFLICT OF INTEREST

The authors confirm that this article content has no conflict of interest.

ACKNOWLEDGEMENTS

The authors would like to thank for the support by National Basic Research Program of China (973 Program) under Grant No.2011CB302601, National High Technology Research and Development program of China (863 Program) under Grant No. 2011AA01A202, Science and technology program of Hunan Province under Grant 2013FJ4335 and 2013FJ4295.

REFERENCES

[1] http://www. delicious.com.
[2] Flickr website. http://www. flickr.com.
[3] Google base website. http:// base.google.com.
[4] A. Zhou, "Data intensive computing", *Communication of China Computer Federation*, vol. 5, no. 7, pp. 50-53, 2009.
[5] G. DeCandia, D. Hastorun, M. Jampani, G. Kakulapati, A. Lakshman, A. Pilchin, S. Sivasubramanian, P. Vosshall, and W. Vogels. "Dynamo: Amazon's highly available key-value store", In: *Proceedings of 21st ACM SIGOPS Symposium on Operating Systems Principles*, 2007.
[6] F. Chang, J. Dean, S. Ghemawat, W. C. Hsieh, D. A. Wallach, M. Burrows, T. Chandra, A. Fikes, and R. E. Gruber, "Bigtable: A distributed storage system for structured data", In: *Proceedings of the 7th USENIX Symposium on Operating Systems Design and Implementation (OSDI06)*, pp. 205-218, 2006.
[7] S. Ghemawat, H. Gobioff, and S. T. Leung, "The Google file system", In: *Proceedings of 19th ACM Symposium on Opreating Systems Principles*, *SOSP*, pp. 29-43, 2003.
[8] S. Wu, D. Jiang, B. C. Ooi, and K. L. Wu, "Efficient based indexing for cloud data processing", *Proceedings of the VLDB Endowment*, vol. 3, no. 1-2, pp. 1207-1218, 2010.
[9] Youtube Website. http://www. youtube.com.
[10] J. Dean, and S. Ghemawat, "Mapreduce: simplified data processing on large clusters," In: *Proceedings of 6th Conference on Symposium on Operating Design and Implementation OSDI*, 2004.
[11] M. K. Aguilera, W. Golab, and M. A. Shah, "A practical scalable distributed B-tree", In: *Proceedings of the VLDB Endowment*, vol. 1, no. 1, pp. 598-609, 2008.
[12] C. Tang, J. Gao, T. Wang, and D. Yang, "*Distributed B+ tree index system and building method*," C. N. Patent 101576915, Nov. 11, 2009.
[13] B. Yuri, F. K. Henry, and A. Silberschatz, "*Concurrency control protocols for management of replicated data items in a distributed database system*," U. S. Patent 5999931, Dec. 7, 1999.
[14] D. R. Naphtali, and S. Artyom, "*Efficient optimistic concurrency control and lazy queries for B-trees and other database structures*," U. S. Patent 5920857, Jul. 6, 1999.
[15] M. Ahmed, S. S. Singh, and M. J. Lee, "*Lazy updates to indexes in a database*," U.S. Patent 20090089334, April 2, 2009.

Received: September 22, 2014 Revised: November 30, 2014 Accepted: December 02, 2014