International Journal of Grid and Distributed Computing
Vol. 9, No. 2 (2016), pp.1-16
http://dx.doi.org/10.14257/ijgdc.2016.9.2.01

A Novel Distributed Index Method for Cloud Computing

Dongyu Li

Batotou Vocational & Technical College Department of Computer and Information Engineering, China

Abstract

For the performance and maintenance cost of the existing index methods, this paper presents a distributed multi access entrance B+ tree index method, which has some features that are important in practice, namely (1) achieve efficient parallel interval queries, and (2) and low maintenance cost for index structure. Our scheme relies on four key techniques to achieve efficient parallel interval queries and low maintenance cost for index structure: (1) It gives a routing table to each leaf node of distributed B+ tree, so as to interval search can be completed from any leaf node of any storage node and break through the Bottleneck caused by the root node in the distributed B+ tree; (2) It construction of balanced binary tree in different levels of a node, and choose select the relevant nodes for its routing table; (3) It uses the layer-by-layer transitivity of B+ tree node splitting information to sense node split position so as to only update routing information within subtree of corresponding particle at the time of node splitting; (4) it uses balanced structure of B+ trees to realize only update the single route information of relevant nodes at the time of node splitting, without adjusting the route tables in a large area. Our scheme has been implemented and evaluated, and the performance results are encouraging.

Keywords*: Cloud Computing; Distributed B+ tree; Index; Performance; Maintenance Cost*

1. Introduction

With rapid development of Internet application, complex query technology represented by interval query has been the hotspot of current researches[1,2,3]. Owing to the great volume and distribution of data as well as concurrent access of massive users, data index in cloud computing environment presents many new characteristics, such as great volume of index data and concurrent access of massive users[3-6]. These new characteristics have brought new challenges for complex query represented by interval query in cloud computing environment. (1) The great volume of index data requires large-scale distributed storage for index data. Therefore, interval query demands search across multiple nodes. As a result, an efficient search method is needed to accurately locate storage nodes and rapidly find out index data. (2) Concurrent access of massive users does not allow performance bottleneck in the process of interval query, so as to guarantee high efficiency and concurrence of the query operation.

At present, index methods in distributed system include Hash, regional bitmap, distributed B tree, prefix Hash tree, skip list, distributed balanced binary tree, *etc*.

(1) Hash method means to carry out Hash computation for keywords of data and then acquire data according to Hash value. Chord[7] is the typical example of Hash index method. It adopts loop as its static topological graph and allocates positions of storage nodes and storage locations of data objects on the basis of safe consistent Hash function[7]. This is a simple and effective method and supports precise query, but it does not support interval query. LSH[8] is a distributed index constructed on

ISSN: 2005-4262 IJGDC

the basis of Chord. It acquires identity in attribute value interval through the position sensitive Hash algorithm. This method supports interval query, but the return query results match the query conditions only under a certain probability. It cannot return all data objects that meet query conditions.

(2) Regional bitmap[9] carries out sequence subdivision for index attribute values in a global scope, and endows each data object with global number and local index bitmap. This method supports interval query, but all search operations have to be broadcasted to various storage nodes for local search. Thus invalid search will be caused for many nodes without storage search values.

(3) Distributed B tree [10,11,12] organizes index data into B+tree and allocates all nodes in this tree into various storage nodes. During access, search should be carried out from root to leaf nodes. This method supports interval query, but root node is the performance bottleneck. DB-tree[10] and P-tree[11] realize concurrent positioning and access by buffering all non-leaf nodes, but splitting of any internal nodes has to be synchronously updated in the buffering nodes, so the maintenance cost is quite high.

(4) Prefix Hash tree[13] adopts a prefix tree similar to binary tree to encode the data key and then allocates data objects with the same prefix into the corresponding nodes in an overlay network via DHT[6] technology. This method supports interval query, but the root node of prefix tree is still the performance bottleneck of search.

(5) Skip list[14] organizes the index nodes into an orderly list, with each node recording the information of routing nodes in a distance of 2i(i=0,1,…). This method has fundamentally avoided performance bottleneck of search, but it rigidly assigns routing nodes. This makes all nodes update routing information when nodes are inserted, so the maintenance cost is high.

(6) Distributed balanced binary tree [15] organizes index nodes by adopting balanced binary tree on the basis of skip list method. It maintains four neighborhood relations in each node: father node, child node, adjacent nodes in linear order, and nodes with a distance of 2^i at the same level. This method has high search efficiency, but it requires balance adjustment of the tree during node splitting. Balance adjustment needs to update a large amount of routing information, so the maintenance cost of routing information is still quite high.

For the performance and maintenance cost of the existing index methods, this paper presents a distributed multi access entrance B+ tree index method(named as MDB+-tree), which can achieve efficient parallel interval queries and low maintenance cost for index structure. First, It gives a routing table to each leaf node of distributed B+ tree, so as to interval search can be completed from any leaf node of any storage node and break through the Bottleneck caused by the root node in the distributed B+ tree; second, it construction of balanced binary tree in different levels of a node, and choose select the relevant nodes for its routing table; third, It uses the layer-by-layer transitivity of B+ tree node splitting information to sense node split position so as to only update routing information within subtree of corresponding particle at the time of node splitting; last, it uses balanced structure of B+ trees to realize only update the single route information of relevant nodes at the time of node splitting, without adjusting the route tables in a large area.

2. Analysis on Traditional Index Method

2.1 B+ tree Index

An m order B+ tree is a balanced multiway search tree (as shown in Figure 1). It possesses the following characteristics: (1) each node contains k($\lfloor m/2 \rfloor \leq k \leq m$) pairs of <keyword, pointer>, pointer in the leaf node points to the recorded physical storage

location, and pointer in non-leaf node points to a subtree; (2) all leaf nodes are at the same level and are linked a list via pointers, they store all keywords, which are arranged from small to big; (3) all non-leaf nodes only contain the largest keyword in the subtree.

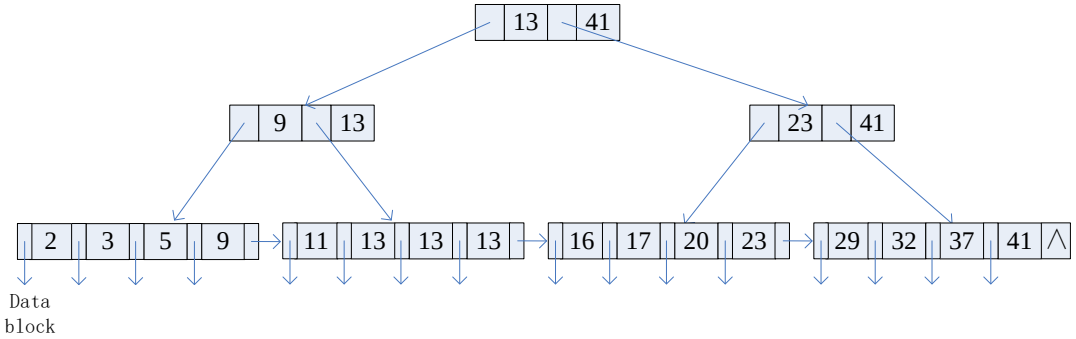

Figure 1. B+ Tree

The advantage of B+ tree index is that it has excellent search performance in centralized environment. The disadvantage of B+ tree index is: the root node of B+ tree is easy to become performance bottleneck. For instance, in the figure, when two users conduct simultaneous access to the data object of key=3 and key=29, the process for access to key=3 is data block where a→b→d→key=3 is located, and the process for access to key=29 is data block where a→c→g→key=29 is located. For both of the accesses, we have to gain access to the root node a. At the root node, concurrent access turns into serial access, which will affect the access performance.

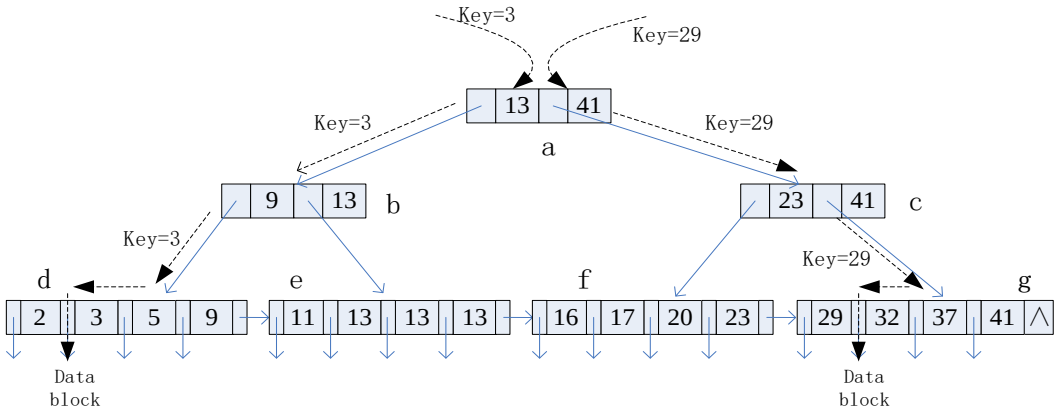

Figure 1. Parallel Access B+ Tree

2.2 Balanced Binary Tree Index

Balanced binary tree is a binary sort tree. For each node, the absolute value of depth difference between left and right subtrees will not exceed 1. Besides, both its left subtree and right subtree are balanced binary trees. Balanced binary tree makes all keywords distribute in various nodes of the tree. Each node contains k($\lfloor m/2 \rfloor \leq k \leq m$) pairs of <keyword, pointer>, and four neighborhood relations are maintained in each node: father node, child node, adjacent nodes in linear order, and nodes with a distance of 2^i at the same level, as shown in Figure 2.

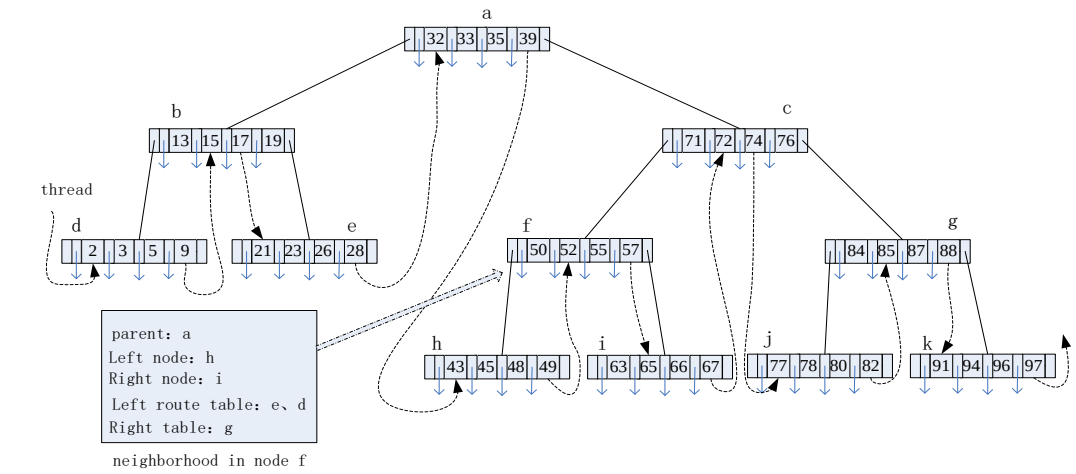

Figure 2. The Balance Binary Tree Index Method

The advantage of balanced binary tree index is that we can get access to the data object from any node. For instance, if the data object of key=96 is queried in the figure and f is the entry, the right routing table of f shows that the value of node g is the most approximate to the search key value, so the access request is delivered to node g. Node g presents that the search key value is located in its child node, so the access request is delivered to node k. Finally, the data object of key=96 is found in node k. The disadvantage of balanced binary tree index is: a large number of neighborhood relations have to be updated during node splitting, so the maintenance cost is high. For example, if 64 is inserted into node i, the node will split into i' and i'', and the binary tree needs to be adjusted. See Figure 3 for the neighborhood relation of node f and the new binary tree after adjustment. The figure shows that hierarchical relation of the node changes greatly after the adjustment, so great changes will also happen to routing table of all nodes. These changes require traversal of whole binary tree and computing node levels again, and routing table of all nodes should be updated timey.

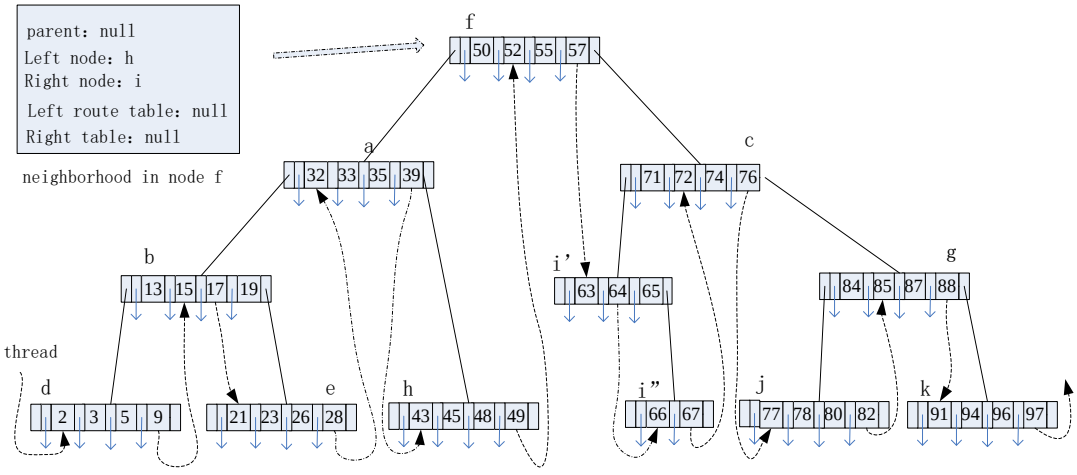

Figure 3. Adjusting the Balance Binary Tree Index

3. The Definition of Distributed Multi-Entrance B+ Tree

For B+ tree index performance bottleneck and balanced binary tree index maintenance cost, we propose distributed multi-access entry B+ tree, featured in the following three specific aspects:

(1) It distributes all index nodes to each storage node and gives a routing table to each leaf node of B+ tree. Through the routing table, interval search can be completed from any leaf node of any storage node so as to avoid access bottlenecks;
(2) It construction of balanced binary tree in different levels of a node, and choose select the relevant nodes for its routing table;
(3) It uses B + tree structure to maintain routing information, effectively reducing the routing information maintenance cost, which is mainly reflected in: ① It uses the layer-by-layer transitivity of B + tree node splitting information to sense node split position so as to only update routing information within subtree of corresponding particle at the time of node splitting; ② it uses balanced structure of B + trees to realize only update the single route information of relevant nodes at the time of node splitting, without adjusting the route tables in a large area;

Figure 4 is a sketch indicating that a multi-access entry B + tree is distributed in three storage nodes, wherein each storage node holds a plurality of nodes of the B + tree (marked in black solid blocks). To understand it better, we draw a complete logical multi-access entry B + tree in the upper half of the sketch and draw the B + tree structure on each storage node, in which the hollow block indicates the index nodes are not in the storage node but in other service storage node.

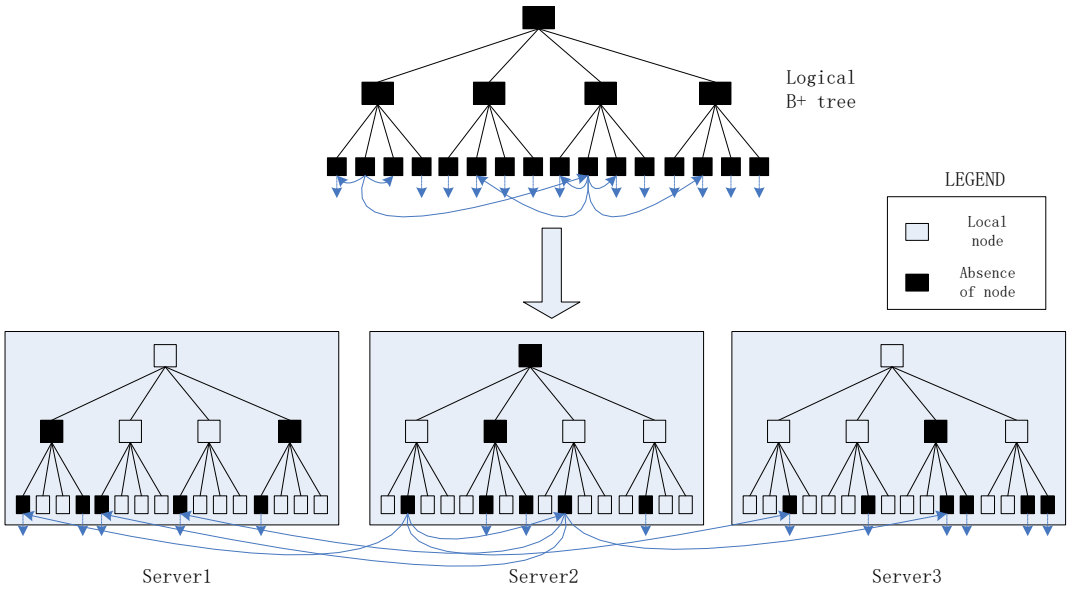

Figure 4. Distributed Multi-Entrance B+ Tree

According to the above structure, the interval query in distributed multi-access entry B+ tree (set search interval as $[k_l,k_u]$) process becomes as follows:

(1) Find the leaf node containing k_l(or the leaf node whose upper limit is less than k_l and is maximum) in the storage nodes accepting query request, and use it as the lookup entry;
(2) Find the next leaf node containing k_l (or the leaf node whose upper limit is less than k_l and is maximum) in the routing table of the entry leaf node and route the query request to the host node of the next leaf node;
(3) Repeat steps (1) and (2) and eventually find the leaf node where k_l is located;
(4) According to the ordered pointer list among the leaf nodes, traverse from the leaf node where k_l is located to the leaf node where k_u is located. The key values between k_l and k_u are the key values which should be found. According to their pointers, find the data.

In a distributed multi-access entry B + tree, all the leaf nodes can become query entry,

and thus it can achieve multi-user parallel query. In addition, nodes are randomly distributed. Thus, each storage node query load is even.

4. Constructing Distributed Multi- Entrance B+ Tree

4.1 B+ Tree Node Number

In order to properly build the routing table, in addition to the index file name, each node in the tree has also been given a logical number. The index file name is fixed, and logical number varies with the node splitting. Logical number is in the form as follows: D(0):D(1):…:D(i):…:D(h-1)， i=0,1,…,h (h is the height of the B+ tree), D (i) = 0,1,2,.... For example, in Figure 5, the index file name of the root node is a, and its logical number is D (0) = 0; the son index file names of root node are: b0, b1, b2 and b3, which are numbered as D (0): D (1) = {0:0,0:1,0:2,0:3}; the son node number of 0:0 is D (0): D (1): D (2) = {0:0:0,0:0:1}, the son node number of 0:1 is D (0): D (1): D (2) = {0:1:0, 0:1:1,0:1:2,0:1:3 }, and so on

.

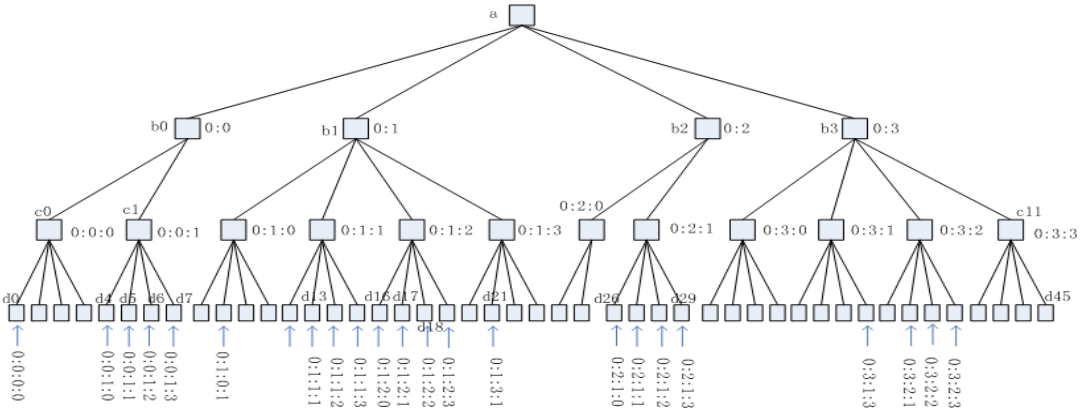

Figure 5. Node Number

According to the above numbering method, different leaf nodes in the same tree have the same prefix. For example, the common prefix of node 0:1:1:0, 0:1:1:1 is 0:1:1; the nodes in the same location in different subtrees have the same suffix. For example, the node 0:1:0:1 and node 0:1:1:1 are respectively located in sub-tree of the root of 0:1:0 and sub-tree of the root of 0:1:1, but they are the second child of their sub-trees respectively. Thus, they have a common suffix, namely 1.

4.2 Building Distributed Multi-Entry B+ Tree

Through establishing different connection in leaf nodes and in non-leaf nodes of each layer, a distributed B + tree will become a multi-entry distributed B + tree. The connection in leaf node is mainly used for fast query while the connection in non-leaf node is mainly used for building and maintaining the connection in leaf nodes.

(1) Non-leaf node data structure

For each non-leaf node, it records two types of neighbor relationships: parent-child relationship (PC) and neighbor brother relationship (NB). Parent-child relationship is used to insert a new branch to the parent node at the time of splitting. Neighbor brother relationship is composed of left neighbor brother relationship (LNB) and right neighbor brother relationship (RNB), mainly used to search for the subtrees with its brother nodes as the root. In response to these two types of neighbor relations, non-leaf node uses the data structure shown in Figure 6 to describe it

.

Node Name		ID	
Parent			
LNB			
RNB			

Figure 6. Data Structure for Non-Leaf Node

For node D (0):...: D (i-1): D (i) ($0 \leq i < h-1$), the algorithm choosing neighbor brother is such as Algorithm 1.

Algorithm 1: chooseBrother(n: non-leaf node) //the algorithm for non-leaf node selecting neighbor brother
1 **construct** a balanced binary tree LBT from all left brother nodes of n; 2 LNB={t|t is a node belonging to the path from the root node to the bottom right node in LBT} 3 **construct** a balanced binary tree RBT from all right brother nodes of n; 4 RNB={t|t is a node belonging to the path from the root node to the bottom left node in RBT}

For example, for node 0:1:1 and the node 0:3:0 in Figure 5, the balanced binary tree constituted by their left and right brothers are shown in Figure 7 (a) and Figure 7 (b). Their data structures are shown in Figure 4-10 (a) and Figure 4-10 (b).

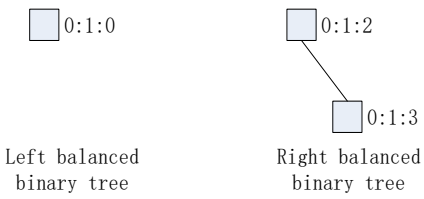

Figure 7(a). Balanced Binary Tree of 0:1:1

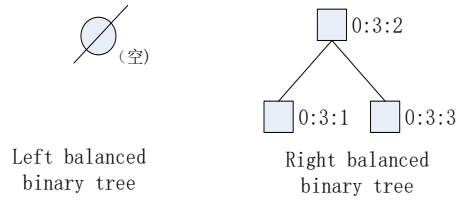

Figure 7(b). Balanced Binary Tree of 0:3:0

Node Name	C2	ID	0:1:1
Parent	0:1		
LNB	0:1:0		
RNB	0:1:2		

Figure 8(a). Neighbor Relations of 0:1:1

Node Name	C8	ID	0:3:0
Parent	0:3		
LNB			
RNB	0:3:1, 0:3:2		

Figure 8(a). Neighbor Relations of 0:3:0

(2) The Data Structure for Leaf Node

For each leaf node, it records three types of neighbor relationships: parent-child relationship (Parent), successor relationship (successor) and neighbor routing relationship (NR). As the non-leaf node, the parent-child relationship is used to insert a new branch at the time of splitting; successor relationship is used to find the keys and pointers meeting the scope; neighbor routing relationship is used for query routing, and it is composed of the left routing table (LRT) and the right routing table (RRT). LRT is related only to the left associated leaf nodes, and RRT is related only to the right leaf nodes. In response to these three types of neighbor relations, leaf nodes are described in the data structure in Figure 7. To meet the requirement of routing update, a "layer" field is designed for each item of routing table to indicate which layer of the tree the node is located.

node name		ID		upper		lower	
Parent				successor			
LRT							
Node name	level	lower		upper		address	
RRT：							
Node name	level	lower		upper		address	

Figure 9. The Data Structure for Leaf Node

For the nodes with the number of D (0): D (1):…:D (i-1): D (i):…: D (h-1) (h is the height of the tree), the algorithm for constructing routing table is as shown in Algorithm 2:

Algorithm 2 buildRoutingTable(D (0): D (1):…D (i-1): D (i):…: D (h-1):leaf node)
//the algorithm for a leaf node building its routing table

```
1   construct a balanced binary tree LBT from all left brother nodes of D (0): D (1):…:D
    (i-1): D (i):…: D (h-1);
2   LRT←{t|t is a node belonging to the path from the root node to the bottom right node in
    LBT};
3   construct a balanced binary tree RBT from all right brother nodes of D (0): D (1):…:D
    (i-1): D (i):…: D (h-1);
4   RRT←{t|t is a node belonging to the path from the root node to the bottom left node in
    RBT};
5   for i=h-2, 0
6     for D(0):…:D(i-1):x∈LNB of D(0):…:D(i-1):D(i)
7       if  D(0):…:D(i-1):x: D(i+1) :…: D(h-1) exists then
8         LRT←D(0):…:D(i-1):x: D(i+1) :…: D(h-1);
9       else
10        LRT←the nearest D(0):…:D(i-1):x: D(i+1) :…: D(h-1) node on the left there
11    for D(0):…:D(i-1):y∈RNB of D(0):…:D(i-1):D(i)
12      if  D(0):…:D(i-1):y: D(i+1) :…: D(h-1) exists then
13        RRT←D(0):…:D(i-1):y: D(i+1) :…: D(h-1)
14      else
15        RRT←the nearest D(0):…:D(i-1):x: D(i+1) :…: D(h-1) node on the left there
```


For example, if the index value range of relevant nodes in Figure 5 is as shown in Figure 11, the data structure of node 0:1:2:1 is as shown in Figure 4-11, wherein LRT = {0:1: 2:1}, RBT = {0:1:2:2}; calculated through the parent node 0:1:2, the node 0:1:1:1 ∈ LRT and 0:1:3:1 ∈ RRT; Calculated through ancestor node 0:1, the tree with the root of 0:1 has different structure from the tree with the root of 0:0, and 0:0:2:1 does not exist. The left existing and the nearest node is 0:0:1:3. Thus, 0:0:1:3 ∈ LRT. Similarly, 0:2:1:3 ∈ RRT.

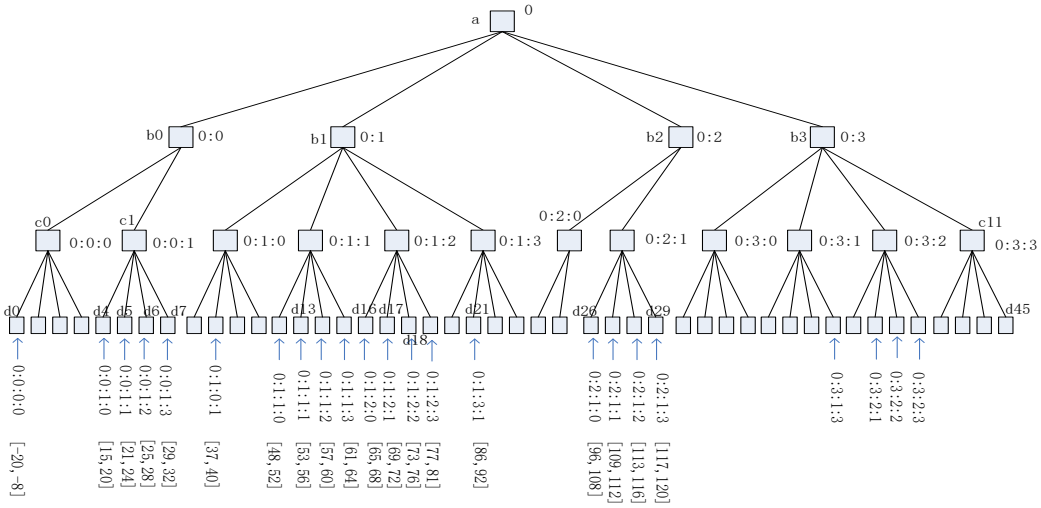

Figure 10. Range Value of each Node

Node	d17	ID	0:1:2:1	lower	69	upper	72
Parent	c4			successor	d18		
LRT							
node	level		lower	upper	pointer		
d16	4		65	68	pointer to d16		
d13	3		53	56	pointer to d13		
d7	2		29	32	pointer to d7		
RRT							
node	level		lower	upper	pointer		
d18	4		73	76	pointer to d18		
d21	3		84	88	pointer to d21		
d29	2		117	120	pointer to d29		

Figure 11. Neighbor Relations of 0:1:2:1

As can be seen from the above two algorithms, in constructing the leaf node and non-leaf node connectivity relationships, we select nodes by building balanced binary tree layer by layer so as to realize binary search using different size of subtrees as the lookup unit according to proximity of the keys and nodes: if the key and the node is closed to each other, the sub-tree with a smaller particle size will be used for binary search; if the key and the node is far from each other, the sub-tree with a larger particle size will be used for binary search; and then binary search will be performed by gradually decreasing size of subtrees. In this way, the search speed can be improved.

4.5 Routing and Maintenance & Update Algorithms

4.5.1 Routing Algorithm: The access process to DMA-B+ tree is the multiple routing process of the access request in the server node. The DMA-B+ tree node distribution features and connection relationship between nodes determines that the routing protocol needs to address the following two questions: (1) according to the definition of DMA-B+ tree, each server node is provided with many index nodes. When a server receives an access request, which leaf node should it choose as the access entry? (2) Which leaf node should it select as the "next hop"? We use the search key and the node distance metric method to solve these two problems. The method is as follows: $\mathbf{s = |upper - key| + |key - lower|}$

Based on data structure and distance metric method of leaf nodes, the routing algorithm of query key value of s_{key} is shown in Algorithm 3.

Algorithm 3 route(S_{key}) // the routing algorithm
1 d_x=the node which has the smallest s with Skey; 2 if $dx.lower \le s_{key} \le dx.upper$ then the result is returned in the local search; 3 else 4 if s_{key}<lower then 5 d_y= the node which has the smallest s with S_{key} in the LRT of d_x; else 6 d_y= the node which has the smallest s with S_{key} in the RRT of d_x; 7 access request is redirected to the server storing d_y;

4.5.2 Update Algorithm: Splitting of DMA-B + tree node will cause changes in the neighbor relationship among the nodes. In order to ensure the correctness of route, it needs to update the node routing table in time.

In the B + tree, splitting of a leaf node may cause splitting of many ancestor nodes from the bottom up and splitting of node will insert a new branch in its parent node. This change is used to control related leaf nodes to hierarchically update routing information, wherein the algorithm is shown in Algorithm 4.

Algorithm 4 update()// Routing update algorithm;

1 receive messages(its format is < D(0):D(1):…:D(i), b, pointer>);
2 if (message types for the new branch message) {
3 branch b is inserted to node D(0):D(1):…:D(i);
4 if(D(0):D(1):…:D(i) is not full) {
5 for x=0,m-1 do { //m is the number of child nodes of D(0):…:D(i)
6 Rebuild LBT and RBT of D(0):…:D(i):x;
7 Achieve new LNB and RNB from LBT and RBT;
8 for y=0,k-1 do //k is the number of child nodes of D(0):…:D(i):x
9 Send <LNB, RNB, i> to the host node of D(0):…:D(i):x:y;
10 }
11 }
12 else{ //D(0):D(1):…:D(i) is full
13 D(0):D(1):…:D(i-1):D(i) is split into D(0):D(1):…:D(i-1):D(i) and D(0):D(1):…:D(i-1):D(i)+1;
14 Send< D(0):D(1):…:D(i-1),D(i)+1,pointer> to the host node of D(0):D(1):…:D(i-1);
15 }


```
16   }
17   else{                          // message types for updating routing table
18        if (D(0):D(1):…:D(i-1):D(i) is no-leaf node)
19           for   x=0,m-1 do // m is the number of child nodes of D(0):…:D(i)
5               Forward the message to the host node of D(0):…:D(i):x;
20        else{
21              all route entries with level n in LRT of D(0):D(1):…:D(i) are updated
                LNB;
22              all route entries with level n in RRT of D(0):D(1):…:D(i) are updated
                RNB;
23              }
24        }
```

For example, in Figure 4-7, node 0:0:1:1 splits into new nodes 0:0:1:2 and 0:0:1:1, its parent node 0:0:1 splits into new nodes 0:0:1 and 0:0:2; the numbers of the original nodes 0:0:1:2 and 0:0:1:3 are changed into 0:0:2:0 and 0:0:2:1. Splitting of node 0:0:1 will generate another child node 0:0:2 of its parent node 0:0. After splitting, the B + tree is as shown in Figure 4-14. It completes the routing information adjustment through the following steps:

(1) When the node 0:0:1 splits into new nodes 0:0:2 and 0:0:1, the child node of 0:0:1 and 0:0:2 (not including newly split nodes) will reconstruct balanced binary tree and update the routing table.

(2) When the node 0:0:1 splits, its parent node 0:0 will add a new child node) 0:0:2. The nodes 0:0:0 and 0:0:1 will reconstruct binary tree and get LNB and RNB. The LNB and RNB of 0:0:0 do not change, so its leaf nodes will suffer no routing update. LNB and RNB of 0:0:1 change, so its leaf nodes will suffer routing update.

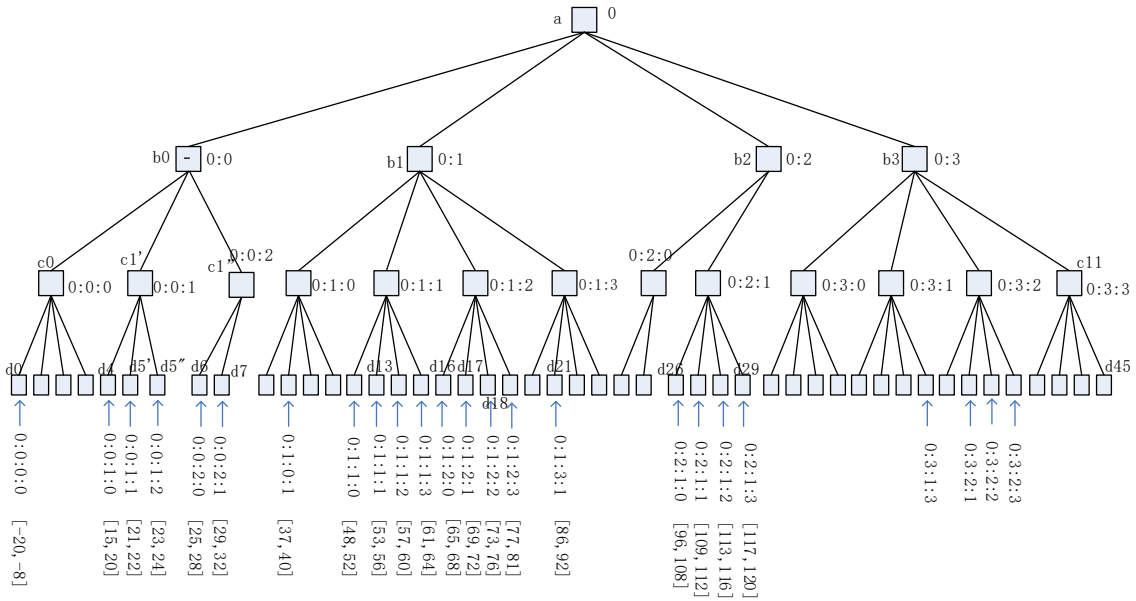

Figure 12. Node Splitting

6. Evaluation

We now evaluate the performance and scalability of our DPRD strategy. For the purpose of comparison, we implement the single access B+ Tree (recorded as SDB+-Tree) and distributed balance binary tree (recorded as DBA-tree) for comparing performance.

6.1 Experimental Environment

Our testing infrastructure had 126machines on 4 racks connected by Gigabit Ethernet switches. Intra-rack bisection bandwidth was ≈14Gbps, while inter-rack bisection bandwidth was ≈6.5 Gbps. 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 totally 175 pair 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 $[0,10^9]$, 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. In the following experiments, 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 10^9 elements and inserted into the B-tree.

The searching load: all operations are searching operation. 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 with above mentioned methods.

6.2 Experiment Result and Analysis

In searching load, we test the searching performance of MDB+-tree in the intervals of s=0, s=0.04 and s=0.1 ($s = \frac{ekey-skey}{10^9}$), and the results are indicated in Figure 4-15. Judged from the figure: 1), point searching （s=0） has the best searching performance; 2) With the increase of interval size, the performance drops correspondingly. It may be caused the rapid increase of interval size and cross-node search; 3) in different searching intervals, the system's processing capacity increases linearly with the increase of server, proving that MDB+-tree has outstanding extendibility.

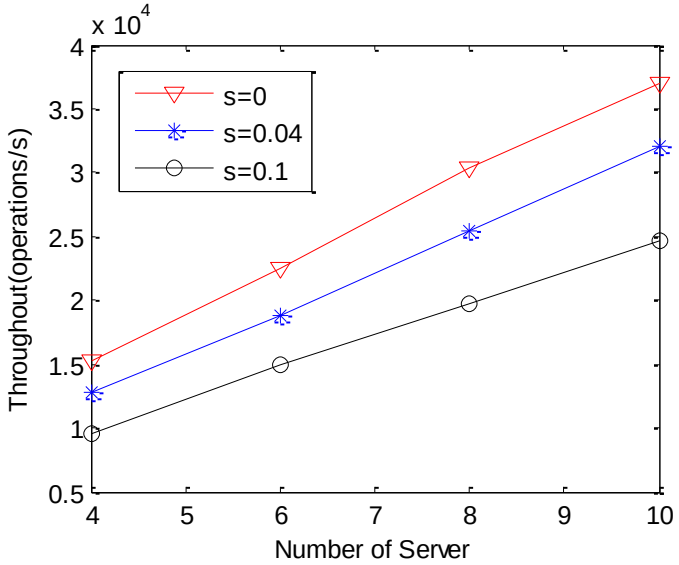

Figure 13. Query Throughout of MDB$^+$-tree

In searching load, this paper compares the searching performance of MDB+-tree, SDB+-tree and DBA-tree in the interval of s=0.04 and the results are indicated in Figure 4-16. Judged from the figure, since the root node is the bottleneck, SDB+-tree suffers poor extendibility, while MDB+-tree and DBA-tree offer much better extendibility. In addition, their performance is obviously better than that of SDB+-tree. Under the circumstance of searching load, MDB+-tree's searching performance is slightly better than that of DBA-tree, as MDB+-tree only needs to search leaf nodes of the same level, while DBA-tree distributes data nodes in different levels and the pointer changes among parent node, child node and infix sequence increase the switches of searching operation between servers.

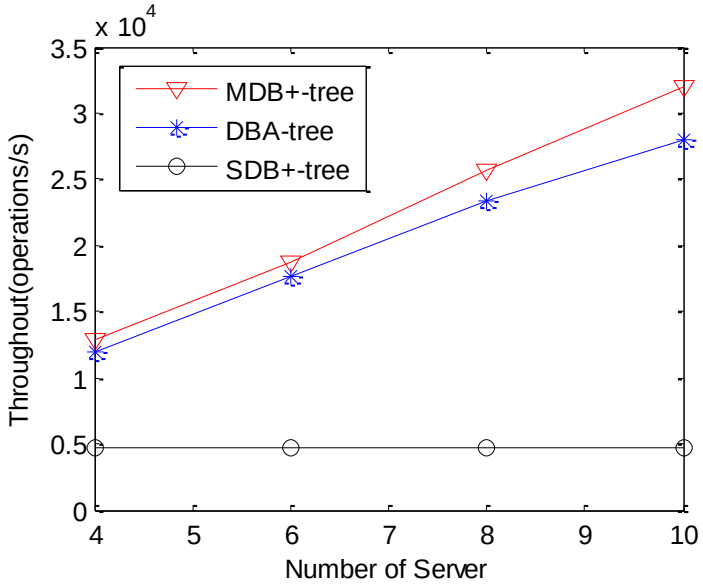

Figure 14. Performance Comparison of Three Trees

We compare the updating performance of MDB+-tree, SDB+-tree and DBA-tree in inserting load (sever number is set to 6). The results are indicated in Figure 4-17. Judged from the figure, the updating performance of the three trees grows with time in the level of logarithm. However, the SDB+-tree suffers poorest updating performance, since all updating operation need to pass root nodes and therefore offer poor concurrent ability. On the other hand, MBD+-tree and DBA-tree could insert operation from any node and offer outstanding concurrent ability. Meanwhile, MDB+-tree has better updating performance than that of DBA-tree. It's because: 1)MDB+-tree has relatively smaller updating range in the aspects of: A. it only implements route updating for leaf nodes under the non-leaf nodes of newly inserted branch rather than implementing route updating for all node in the tree; B. by building binary tree layer by layer, it could not only maintain the efficiency of searching but also reduce the node updating range, because when one node is split, the LNB and RNB of some brother nodes mayn't have any change. Therefore, the leaf nodes under these brother nodes don't need route updating. (2) it has less updating information, because this B+ tree structure is stable. In each route update, the leaf node only needs to modify small amount of route information in routing table.

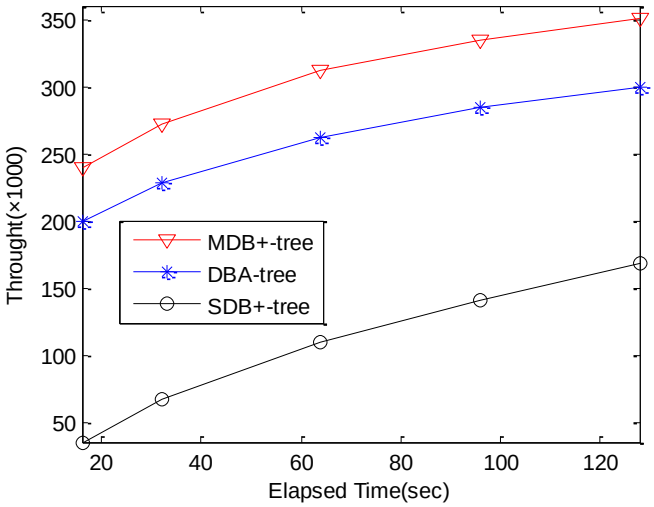

Figure 15. Updating Performance

We compare the capacity of MDB+-tree, SDB+-tree and DBA-tree under the circumstance of hybrid load. In this experiment, the server number is set to 6 and changes the ratio of searching operations in total operations. The results are indicated as Figure 4-18. Judged from the figure, in hybrid load, MDB+-tree has better performance than MDB+-tree and SDB+-tree.

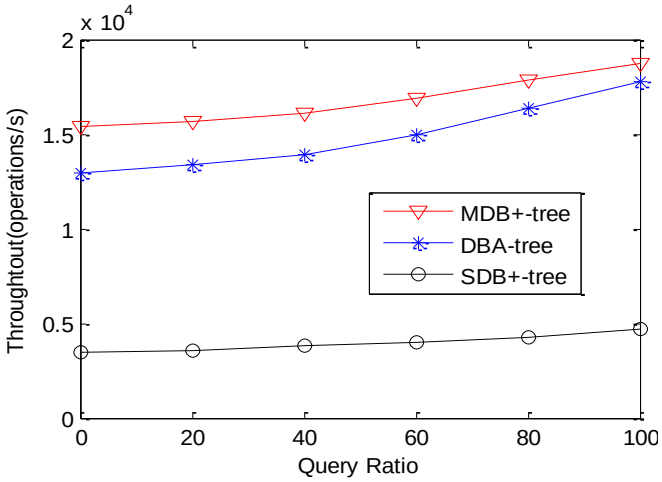

Figure 15. The Performance of Various Query Ratios

7. Conclusions

We presented the design of a scalable distributed B-tree. Our approach has some features that are important in practice, namely (1) achieve efficient parallel interval queries, and (2) and low maintenance cost for index structure. Our scheme relies on two key techniques to achieve efficient parallel interval queries: (1) It gives a routing table to each leaf node of distributed B+ tree, so as to interval search can be completed from any leaf node of any storage node and break through the Bottleneck caused by the root node in the distributed B+ tree; (2) It construction of balanced binary tree in different levels of a node, and choose select the relevant nodes for its routing table. Our scheme also relies on two key techniques to achieve low maintenance cost for index structure: (1) It uses the layer-by-layer transitivity of B+ tree node splitting information to sense node split position so as to only update routing information within subtree of corresponding particle at the time of node splitting; (2) it uses balanced structure of B+ trees to realize only

update the single route information of relevant nodes at the time of node splitting, without adjusting the route tables in a large area. Our scheme has been implemented and evaluated, and the performance results are encouraging.

References

[1] D. Li, J. Cao, X. Lu, "Delay-Bounded Range Queries in DHT-based Peer-to-Peer Systems", Proc. of IEEE ICDCS 2006. Lisboa, **(2006)**.
[2] Y. Zhang, D. Li, X. Lu, "SRQ: Scalable Multi-Attribute Range Queries over Constant-Degree DHTs" Technical Report NDUT-CS-PDL-08-01, **(2008)**.
[3] S.Wu, D. Jiang, B. C. Ooi, K. L. Wu, "Efficient Btree Based Indexing for Cloud Data Processing" VLDB, **(2010)**.
[4] S. Wu and K. L. Wu, "An Indexing Framework for Efficient Retrieval on the Cloud," IEEE Bulletin of the Technical Committee on Data Engineering, vol. 32, no. 1, **(2009)**, pp. 75-82,
[5] X. Zhang, J. Ai, Z. Y. Wang, J. H. Lu and X. F. Meng, "An Efficient Multi-Dimensional Index for Cloud Data Management", proc. of the first international workshop on Cloud data management, Hong Kong, **(2009)**, pp. 17-24.
[6] J. Wang, S. Wu, H. Gao, J. Z. Li and B. C. Ooi, "Indexing Multi- dimensional Data in a Cloud system", in proc. of the international conference on Management of data (SIGMOD'10), Indianapolis, Indiana, June **(2010)**, pp.591-602.
[7] I. Stoica, R. Morris, D. Karger, M. F. Kaashoek, and H. Balakrishnan, "Chord: A scalable peer-to-peer lookup service for internet applications", Proceedings of the 2001 Conference on Applications, Technologies, Architectures, and Protocols For Computer Communications, SIGCOMM '01, **(2001)**; San Diego, California, United States.
[8] A. Broder, M. Charikar, A. Frieze, M. Macher. Min-wise Independent Permutations. Proc. of the Thirtieth Annual ACM Symposium on Theory of Computing (STOC), Dallas, Texas, USA, **(1998)**, pp. 327–336.
[9] M. Bi-Ping,W.Teng-Jiao, L.Hong-Yan, Y. Dong-Qing. "Regional Bitmap Index: A Secondary Index for Data Management in Cloud Computing Environment", Chinese Journal of Computer. vol. 35, no. 11, **(2012)**, pp. 2306-2316.
[10] M.K. Aguilera, W. Golab and M.A. Shah, "A Practical Scalable Distributed B-Tree", Proc. of the VLDB Endowment, Vol. 1, Issue 1, **(2008)**.
[11] K. Aberer, P.C.Mauroux, A.Datta, Z.Despotovic, "P-Grid: A self organizing structured P2P system", ACM SIGMOD 2003, **(2003)**, pp: 29-33.
[12] A. Crainiceanu, P. Linga, J. Gehrke, J. Shanmugasundaram. Ptree: A P2P Index for Resource Discovery Applications. Proc. Of WWW 2004, New York, USA, **(2004)**.
[13] Y. Chawathe, S. Ramabhadran, S. Ratnasamy, "A Case Study in Building Layered DHT Applications", Proc. of ACM SIGCOMM 2005. Philadelphia, **(2005)**, pp. 97–108
[14] N. J. A. Harvey, M. B. Jones, S. Saroiu, "SkipNet: A Scalable Overlay Network with Practical Locality Properties", Proc. of USENIX USITS 2003. Seattle: USENIX Press, **(2003)**.
[15] H. V. Jagadish, B. C. Ooi, and Q. H. Vu. Baton: A Balanced Tree Structure for Peer-to-Peer Networks. Proc. of VLDB, **(2005)**, pp.661–672.

