Hash-Based Indexing for Cloud Databases

Yu-lung Lo[1], Choon-Yong Tan[2], Min-Hsuan Lai[3]

Department of Information Management, Chaoyang University of Technology
168, Jifeng E. Rd., Wufeng District, Taichung, 41349 Taiwan

[1]yllo@cyut.edu.tw

[2]kevin861119@hotmail.com

[3]s10014618@cyut.edu.tw

***Abstract*— Currently, there are many Cloud platform providers support large-scale database services. However, most of these Cloud platform architectures only support simple keyword-based queries and can't response complex query efficiently due to lack of efficient in multi-attribute index techniques. The existing multi-attribute index structures for Cloud platform are based on traditional *R*-tree, *k*-d tree and Quad tree, but there is still without study for evaluating these schemes yet. In this paper, we propose a new multi-attribute index structure, which combines hash-based scheme and tree indexing, for Cloud platform to manage the huge and variety data. Our experimental results demonstrate that our proposed index structure outperforms existing tree-based only indexing.**

***Keywords*—cloud computing, cloud database, cloud data indexing, multi-attribute index, hashed distribution**

1. INTRODUCTION

The Cloud computing is an emerging business solution. It can address the requirements of each software service to distribute the storage space and all kinds of the service on the resource pool. The user does not need to purchase any hardware or software and flexible to upgrade amount of resource according their own actual demand from provider. The Cloud system generated business opportunity and future trend in the software industry. The Cloud computing is a virtual computation resource which may maintain and manage by itself, normally for a lot of large-scale server cluster structures including computation servers, storage servers, the bandwidth resources and so on [23]. Cloud platform compose by a number of computer resources and store a large number of data, and provide services to millions of global user. Resource allocation usually computes in Cloud platform and make user feel that owns personal infinite resources. Providing scalable database services is one of most important issue for many applications of the Cloud platform. The Cloud platform simplifies to provide a large-scale distributed database system however performing indexing and searching in such a database on the Cloud platform has become new challenges to realize.

The traditional distributed system structure lacks of scalability and reliability therefore it cannot be directly applied to this new platform. Due to the diversity of applications, database services on the Cloud must support large-scale data analytical tasks and high concurrent On-Line Transaction Processing (OLTP) queries. When unexpected large searching enquiries occur, it may happen that users meet the situation of out of supported by system resource and disable of quality of service [25]. However, currently the Cloud platform only supports simple keyword-based queries. It can't answer complex queries efficiently due to lack of efficient index techniques. There were few research reports proposed indexing schemes for Cloud platform to manage the huge and variety data. These schemes create global index for master nodes and local index for each slave (or storage) node. To prevent the bottleneck, the global index is distributed and maintained in several master nodes. The local index manages the local data in a slave node for local data search and the global index manages the tree node in local index for searching entries of slave nodes. All these index structures are based on existing index structures, such as R-tree [8], *k*-d tree [6] and Quad tree [5], which can support multi-attribute / multi-dimensional indexing or spatial indexing. In this research, we would like to survey and evaluate the existing multi-attribute index schemes then develop a new and more efficient multi-attribute indexing in Cloud platform.

The remaining of this paper is organized as follows: in Section 2, we review the existing multi-attribute indexing schemes for Cloud

platform. In Section 3, we discuss the hash based indices. After that, our proposed hash-based multi-attribute index structure for Cloud platform is presented in Section 4. Moreover, we demonstrate our experimental results for comparing the proposed index scheme to existing multi-attribute indexing structures in Section 5. Finally, we give our conclusion in the last section.

2. RELATED WORKS

The concept of Cloud computing evolves from internet search engines' infrastructure. The differences between Cloud computing and DBMS are that the Cloud computing does not adopt order-preserving tree indexes, such as B-tree or hash table [7]. Aguilera et al. [1] proposed a fault-tolerant and scalable distributed B-tree for their Cloud systems. Although B-tree has been widely used as single attribute index in database systems, it is inefficient in dealing with indices composed of multi-attributes [25]. To improve the weakness of Cloud computing, to build a multi-attribute index may support more types of queries on Cloud computing platforms. Therefore, Zhang et al. proposed an Efficient Multi-dimensional Index with Node Cube for Cloud computing system [25], Wang et al. built the RT-CAN index in their Cloud database management system [20], and Ding et al. presented the Quad-tree based index structure for cloud data management [4]. All these index schemes are based on *k*-d tree, R-tree and Quad tree. The brief introductions for these schemes as well as R-tree, *k*-d tree and Quad tree are as follows.

2.1 Efficient Multi-dimensional Index with Node Cube

In 2009, Xiangyu Zhang et al. [25] proposed an efficient approach to build multi-dimensional index for Cloud computing system. In this approach, they build local *k*-d tree index for each slave nodes due to *k*-d tree can efficiently support point query, partial match query and range query. To prune irrelevant nodes on query processing, they construct a node cube for each slave node. A node cube indicates the range of value on each indexed attribute in this node. After they build a cube for each slave node, they maintain the cubes on master nodes with an R-tree. The reason of choosing R-tree for cube information is that the R-tree was designed for managing data regions and in their scenario the cubes are multi-dimensional data regions. They call this index approach EMINC: Efficient Multi- dimensional Index with Node Cube as shown in Figure 1.

With the node cube information in EMINC, query processing can be improved by pruning irrelative nodes in the nodes locating phase. And in order to keep cube information available and useful, insertion and deletion on slave nodes that may change their cubes should inform master nodes for update of cube.

The EMINC has some limitations and under some occasions, the performance could still be poor. The authors extend EMINC to use multiple node cubes to represent a slave node in which data records on a slave node will be represented by multiple node cubes. The shape and amount of node cubes is dependent on the method used for cutting the original single node cube.

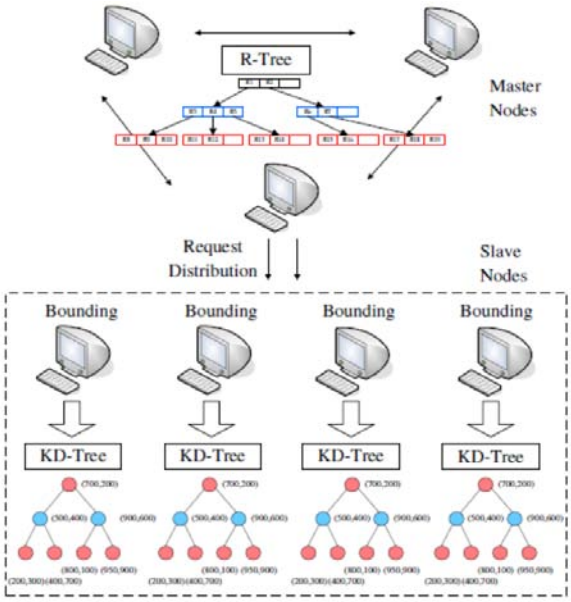

Figure 1. Framework of EMINC [25]

2.2 RT-CAN Index

The RT-CAN is a multi-dimensional indexing scheme proposed by Jinbao Wang et al. in 2010 [20]. RT-CAN integrates CAN-based routing protocol [18] and the R-tree based indexing scheme to support efficient multi-dimensional query processing in a Cloud system.

CAN (Content Addressable Network) [18] is a scalable, self-organized structured peer-to-peer overlay network. The RT-CAN index is built on a shared-nothing cluster, where application data are partitioned and distributed over different servers. In this approach, the global index composes of some R-tree nodes from the local indexes and is distributed over the cluster. The global index can be considered as a secondary index on top of the local R-trees. This design splits the processing of

a query into two phases. In the first phase, the processor looks up the global index by mapping the query to some CAN nodes. These CAN nodes search their buffered R-tree nodes and return the entries that satisfy the query. In the second phase, based on the received index entries, the query is forwarded to the corresponding storage nodes, which retrieve the results via the local R-tree. The index structure and data service of RT-CAN is shown in Figure 2.

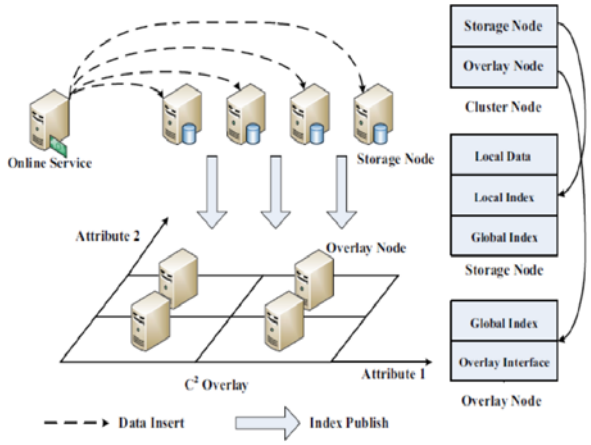

Figure 2. Data service of RT-CAN index [20]

2.3 Quad-Tree Based Index Structure

A Quad tree [5] is a tree data structure in which each region are defined by squares in the plane, which are subdivided into four equal-sized squares for any regions containing more than a single point show in Figure 3. (These are also called PR Quad trees, and we always refer to this variant of Quad trees in this paper.) So each internal node in the underlying tree has four children and regions have optimal aspect ratios (which is useful for many types of queries). Unfortunately, the tree can have arbitrary depth, independent even of the number of input points. Even so, point insertion and deletion is fairly simple. This data structure was named a Quad tree by Raphael Finkel and J.L. Bentley in 1974. A similar partitioning is also known as a Q-tree.

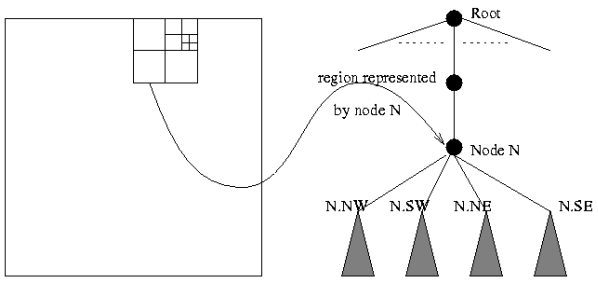

Figure 3. Nodes in a point Quad tree

In 2011, Linlin Ding et al. proposed an efficient quad-tree based index structure for cloud data management [4]. This paper presents an efficient quad-tree based multi-dimensional index structure, called QT-Chord, which integrates Chord-based routing mechanism and Quad tree based index scheme to support efficient multi-dimensional query processing in a cloud computing system.

There are two levels of QT-Chord index structure, global index level and local index level shown in Figure 4. Numerous compute nodes are organized in a cloud computing system to provide their services to end users. The data of user are divided into data chunks and then stored on different compute nodes. To realize efficient local multi-dimensional data management, each compute node builds its local index by an improved MX-CIF Quad-tree index structure, named IMX-CIF Quad tree.

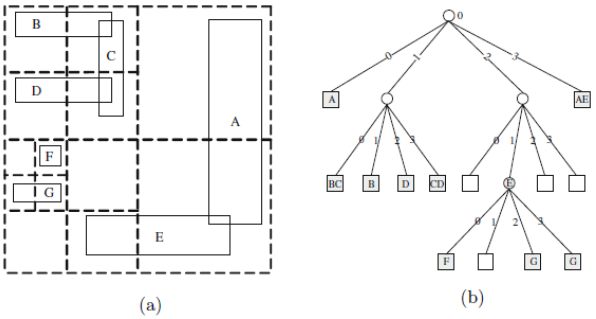

Figure 4. Framework of Quad tree based multi-attribute index [4]

2.4 *k*-d Tree

The *k*-d tree (short for *k*-dimensional tree) was proposed by Jon Louis Bentley in 1975 [3]. The *k*-d tree is a space-partitioning data structure for organizing points in a k-dimensional space. For example, the definition of a 2-d tree is a binary tree satisfying the following two conditions: (with root a level 0)

1. For node *N* with *level* (*N*) is even, then every node *M* under *N.llink* has the property that *M.xval* < *N.xval*, and every node *P* under *N.rlink* has the property that *P.xval* ≧ *N.xval*.
2. For node *N* with *level* (*N*) is odd, then every node *M* under *N.llink* has the property that *M.yval* < *N.yval*, and every node *P* under *N.rlink* has the property that *P.yval* ≧ *N.yval*.

Where *xval* and *yval* denote the coordinates of *x* and *y*, respectively; and *llink* and *rlink* are the pointers to the left child node and right child node,

respectively. For instance, a two-dimensional space consists of some data points as shown in Figure 5 Such that we can create a 2-d tree for these data points as shown in Figure 6, and the space is partitioned as in Figure 5.

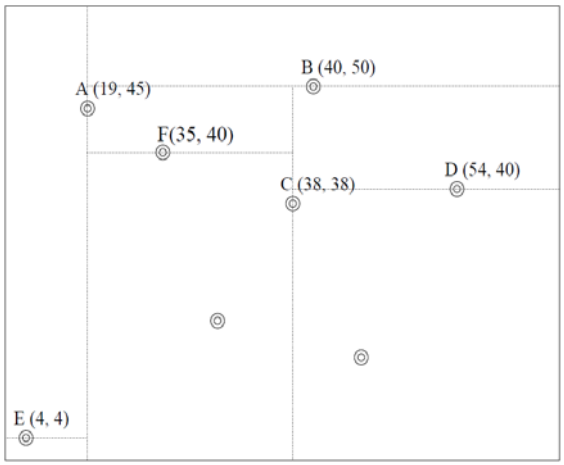

Figure 5. 2-d space with data points

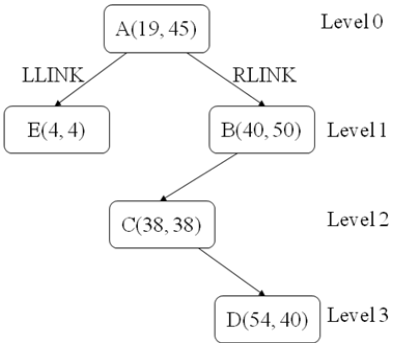

Figure 6. An example of 2-d tree

2. 5. R-tree

The R-tree was proposed by Antonin Guttman in 1984 [8]. R-tree is a tree data structure used for spatial access methods. It groups nearby objects and represents them with their minimum bounding *d*-dimensional rectangle in the next higher level of the tree. Each node of the R-tree corresponds to the minimum bounding *d*-dimensional rectangle that bounds its children. Since all objects lie within this bounding rectangle, a query that does not intersect the bounding rectangle can also not intersect any of the contained objects. In another words, R-tree uses the bounding boxes to decide whether or not to search inside a sub-tree. At the leaf level, each rectangle describes a single object; at higher levels the aggregation of an increasing number of objects. R-tree is a balanced search tree which organizes the data in pages and is designed for storage on disk.

In an R-tree for two-dimensional space, it has an associated order *k* and each non-leaf node contains a set of at most *k* rectangles and at least $\lceil k/2 \rceil$ rectangles. For example, there are three rectangles regions containing nine objects as shown in Figure 7. An R-tree for this 2-d space can be created as in Figure 8.

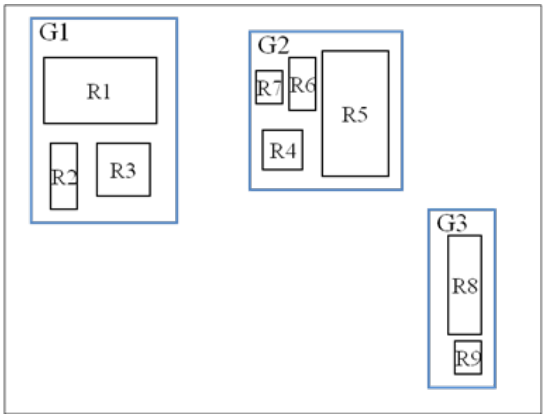

Figure 7. Rectangles for 2-d space

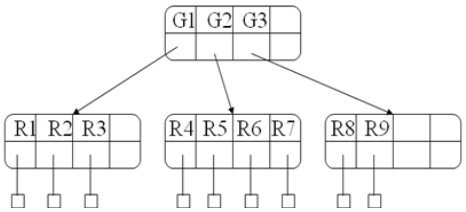

Figure 8. R-tree for 2-d space

3. HASH BASED INDICES

3.1 Hash Functions

There were researchers focused on hash index such as Aho et al. in [2], Lloyd in [13], Moran in [15], and Ramamohanarao et al. in [17]. Multi Attribute Hash (MAH) indexing has been used in preference to indexing schemes such as B-trees because these schemes are primary key indexing schemes and do not perform well when multiple non-primary keys are required in an operation.

Hashing is used to locate and retrieve items in a database because it really does the work in a faster manner and likes to find the item using the shorter hashed key than to find it using the original value.

A hash table [19] is a data structure that associates keys with values. The basic operation supports to efficiently find the corresponding value. It is done by transforming the key using the hash function into a hash, a number that is used as an index in an array to locate the desired location (bucket) where the values should be.

3.2 Traditional Iterative Hash Structure

The traditional one-way hash functions [12] have a common iterative structure as Figure 9.

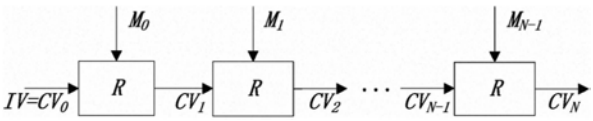

Figure 9. Traditional iterative hash structure

The input message was divided into N blocks (M_0, M_1, …, M_{N-1}) with b bits in each block, and the last block must be padded while the length was less than b. A compress function R was employed in each iterative process. There are two inputs in the function R, one of which is the message block M_{i-1} with b bits length, and the other is the output of the last iteration CV_{i-1} with n bits length. For the first block, CV_0 is equal to a fixed initial value IV, which was named as Initial Vector with n bits length. The hash value of message M is defined as the output of the last iteration. The algorithm can be described as :

$$CV_0 = IV$$
$$CV_i = R(CV_{i-1}, M_{i-1}), \quad 1 \le i \le N$$
$$H(M) = CV_N$$

3.3 Xiaos' Parallel Hash Structure

Xiaos have proposed an algorithm for parallel keyed hash [12][22] construction, whose structure can ensure the uniform sensitivity of hash value to the message. The mechanism of both changeable-parameter and self-synchronization is utilized to achieve all the performance requirements of hash function. The simplified algorithm structure is shown in Figure 10.

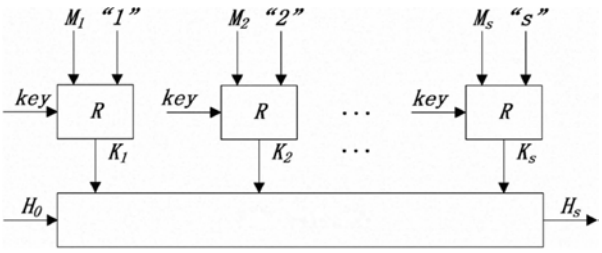

Figure 10. parallel keyed hash construction

The hash function is used to transform the multi-attribute index into the table index (the hash) of an array element (k-d tree or R-tree) where the corresponding value is to be sought. Ideally, the hash function should map each possible key to a unique slot index.

3.4 Multi-attribute Hash Indexes

Multi-attribute hash indexes were discussed in Advanced Database Systems by Keamey [10]. The Figure 11 presents the method of partitioning a multi-attribute index [10]. The space in a multi-attribute index should be partitioned into the fewest blocks required to store the key values. In the Figure 11, there are two blocks. Each block contains two record pointers. Figure 12 shows a grid file with two attributes and three components in grid file. In the first component, there are an array of key values and pointers for each attribute (1&2). The values of each attribute are stored in an array (directory in an extendible hashing index) together with a pointer. The pointer points to a row or column in the grid. In the second component, the first multi-attribute index in the array (1) points to row 0 in the grid. However, a grid file may have any number of attributes. Each additional attribute requires an additional array and a new dimension for the grid. A grid of pointers to disc blocks is stored. Each position in the grid corresponds to one combination of attribute values. More than one position in the grid may point to the same disc block.

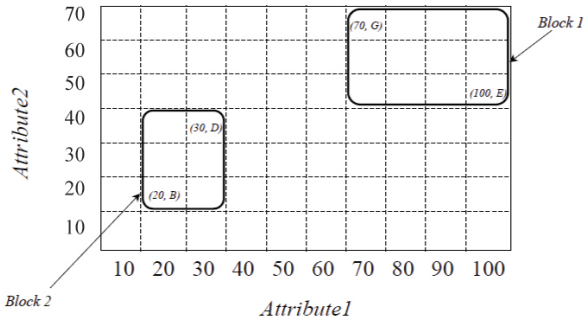

Figure 11. Partitioning a multi-attribute index [10]

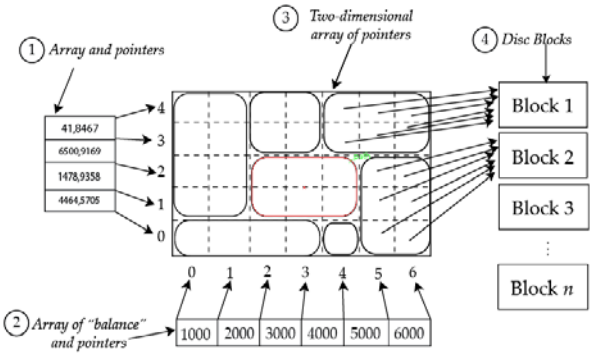

Figure 12. Grid file [10]

3.5 R-tree Hash

In 2010, Guobin Li and Jine Tang [11] proposed an HR-tree index based on hash address. The traditional hash address sorting algorithm does not have to compare the keywords and move elements. Hash function is a mapping from a set of keywords to an address set; hash (key) is the image of the record whose keyword is the key in the address records. In Figure 13, it shows HR-tree index structure of the region in the high dimensional space database which needs to transform high dimensional space address into one-dimensional address, divide the spatial query region and form MBR (minimal bounding rectangle). The keyword of hash function is the upper-left and lower-right coordinates of MBR. The image of the keyword in the address set is the center of MBR. To decide the center of the MBR, it calculates the center of the outer MBR first then estimates the center of the internal MBR contained.

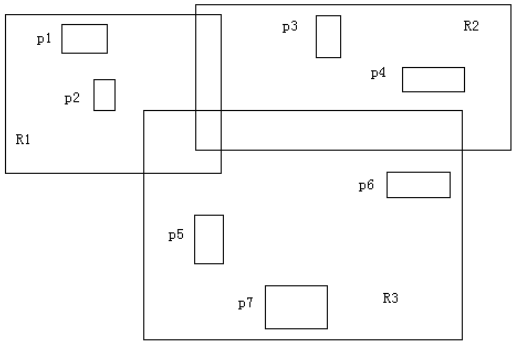

Figure 13. HR-tree index structure of the region

4. PROPOSED INDEX STRUCTURE

4.1 Hash-Based Multi-Attribute Indexing

The structure of our hash-based multi-attribute index for databases in the cloud platform consists of a hash table in master node as the global index and a tree-based index in each slave node as the local index. The hash table for global index only indicates the data allocated in which slave node. In addition, the tree indexes in local slave nodes can help to search data efficiently. The dimensions of hash table are decided by the number of attributes of a table indexed. The Figure 14 represents a hash-based 2-attribute index structure in which a two dimensional hash table is created in master node for global index and a 2-*d* tree, such as R-tree or *k*-d tree, is also created in each slave node for local index. In this index structure, the number of grids in the hash table is equal to or greater than the number of slave nodes. The numbers labeled in the grids of the hash table denotes that the data in the designated grid is allocated at corresponding slave node. Therefore, when a query is issued, the query data is hashed by hash function to determine the belonged grid in global hash table then searches in the slave node indicated by the grid. Comparing to the existing global tree index, our approach maintains a hash table as index can save searching time in the global index.

A 3-attribute index is shown in the Figure 15, which consists of a 3-d hash table in master node and a 3-d tree index in each slave node. The more attribute index can also be derived in the similar way.

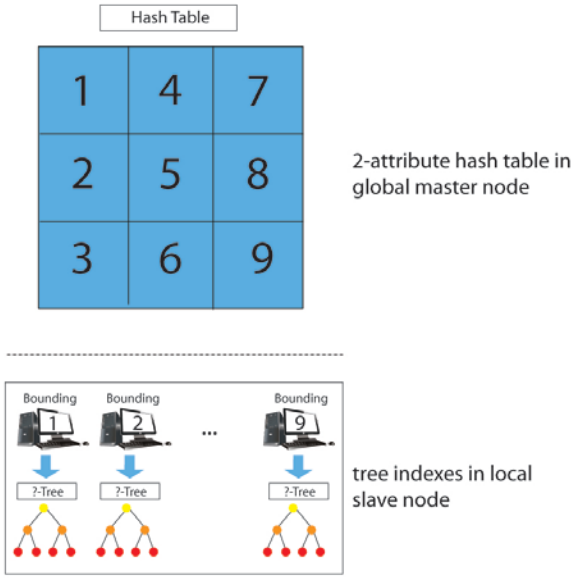

Figure 14. Hash-based 2-attribute index

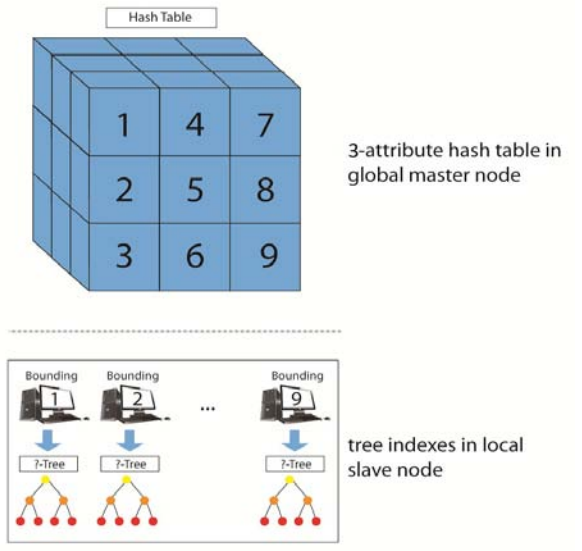

Figure 15. Hash-based 3-attribute index

4.2 Hash Function for Hash Table

There has been numerous hash functions proposed for distribution of data [16][19][21][22][24]. Most of them were used to address the particular data distribution however they may not benefit the hash table with multi-dimensional distribution. Suppose that an attribute will be partitioned into *n* segments for distribution, we organize the following two considerations for hash function design.

Consideration 1 – if most of queries specify this attribute by range, such as age from 20 to 30, the hash function should be designed as range distribution. An example of the hash function, *HF_range*( ), is shown in Equation (1).

$$HF\_range(x) = x / (MaxValue/n) \quad \ldots (1)$$

Where x is the value of attribute to be hashed, *MaxValue* is the possible maximum value of the attribute, and n is the number of segments to be hashed. This is a range distribution and suitable for range query.

Consideration 2 – if most queries specified this attribute by a certain value, such as employee ID is equal to 7654321, and normal distribution usually occurs in this attribute data, the hash function should be designed to uniformly fragment the data. The MOD function, which divides one numeric expression by another numeric expression and returns the remainder, can be one of such hash functions and the hash function, *HF_mod*(), is shown in Equation (2).

$$HF\_mon(x) = x \text{ MOD } n \quad \ldots(2)$$

Furthermore, the coordinate of a grid in multi-dimensional hash table can be determined by the multi-attribute hash function, *MA_HF*(), as shown in Equation (3).

$$MA\_HF(x_1, x_2, x_3, \ldots) = (HF_1(x_1), HF_2(x_2), HF3(x_3), \ldots) \quad \ldots (3)$$

Where x_1, x_2, x_3, … denote the values of attributes to be hashed, $HF_1()$, $HF_2()$, $HF_3()$, … denote the hash functions, which are either *HF_range*()s or *HF_mod*()s decided by consideration 1 and 2, used to convert the attribute value x_i. A record can be hashed by Equation (3) to determine which grid to enroll.

4.3 Best Fit Decreasing Strategy

The range distributed data may cause load imbalanced in slave nodes due to data may massed in some small range by the property of normal distribution. Hash distribution is usually used to address this problem. However, it is not to guarantee. If data is unbalancing distributed across the slave nodes, it will degrade the system performance and long response time for queries may happen. To resolve this problem, there is an accumulator for each grid of hash table in our hash-based multi-attribute index. The accumulator is used to keep trace the number of records enrolled in this grid and allocated to corresponding slave node. Therefore, the Best Fit Decreasing Strategy [9] can be applied to balance the data distribution in slave nodes.

In our proposed scheme, a database can be organized as a set of data grids in which data is enrolled. Such that the Best Fit Decreasing Strategy balancing the data distribution in slave nodes is achieved by assigning data grids from overflow local slave nodes to underflow nodes. In this strategy, the grids of hash table are first sorted into decreasing order according to the number of records inscribed in accumulator. Then, in assignment iteration, the grid which currently has the largest value (number of records) in accumulator is assigned to the slave node which currently has the smallest number of records assigned. This process is repeated until all the records enrolled in grids have been allocated.

We would like to note that the grids in our hash table are only used to indicate the slave nodes where data allocated such that the data structure of each grid consists of an accumulator and a pointer (to the slave node) only. The size of our hash table is fixed and won't grow when the size of database grows up.

5. PERFORMANCE STUDY

5.1 Experimental Methods

For simplify, we only randomly generated two dimensional coordinates and three dimensional coordinates as two-attribute and three-attribute records for creating two dimensional and three dimensional index structures. We build our proposed multi-attribute index structure by using hash table in master node as global index combine with three tree indexes, which are R-tree, *k*-d tree and quad-tree, in slave nodes as local indices. They will be abbreviated to "Hash table + R-tree", "Hash table + *k*-d tree", and "Hash table + quad-tree" in remaining of this paper.

We compare our approaches to the existing index structures of R-tree combining R-tree, R-tree combining *k*-d tree, and quad-tree combining quad-tree as global indices and local indices in the efficiency of memory cost and search time cost. These index structure are also abbreviated to "R-tree + R-tree", "R-tree + *k*-d tree", and "quad-tree + quad-tree". We note that both 2D R-tree and 3D R-tree have an associated order (or degree) four. To investigate the scalabilities of multi-attribute indices, the total numbers of record generated in our databases is varied from one millions to five millions records. We assume that the indexed attributes are normal distribution in our database such that the Equations (2) and (3) are used for created our proposed indices. Our experimentation consists of three parts -- memory cost, time cost for hit data search, and time cost for no hit data search.

Our experimental infrastructure includes one master node and twenty slave nodes to simulate Cloud computing platforms. Each computer had a Q8400 2.66G (1333MHZ) CPU with 4M: L2 cache, 2GB*1(DDR3 1066) 4*DIMM memory, and 500 GB disk. Machines ran on Windows XP Professional OS.

5.2 Memory Cost

In this section, the memories consumed by every variety of multi-attribute indices were investigated. The experimental results for two-attribute indices and three-attribute indices are shown in Figure 16 and Figure 17, respectively. From the two figures, our proposed hash table + *k*-d tree consumes the least memory among six index structures. The hash table + quad-trees also performs well and is just little worse than hash table + *k*-d trees. The main reason is that the memory consumed by a hash table as a global index is a constant size and independent to the size of database. Therefore, only the tree indices in slave nodes are affected by the database sizes in memory consuming for our proposed hash-based multi-attribute index structures.

The index structure R-tree + R-tree performs the worst for memory consuming. This may be due to that R-tree always stores data in leaf nodes and needs to create a number of non-leaf nodes for the minimum bounding rectangles. R-tree also needs to store more coordinates for bounding rectangles and needs more branch links to the child nodes. In contrast, the *k*-d tree likes the binary search tree in which data is stored in either leaf nodes or non-leaf nodes and *k*-d tree has only two branch links to the child nodes. In addition, *k*-d tree, in 2-dimensional or 3-dimensional, needs only one coordinate for each node and does not need to store the boundary for rectangle boxing. Thus it can explains that one of our proposed hash-based indices, hash table combining R-tree, does not perform well in this study. The Hash table + *k*-d trees comparing to R-tree + R-tree can save memory space up to 69.29% and 70.73% in 2-attribute and 3-attribute respectively, where there are five million records in indexes.

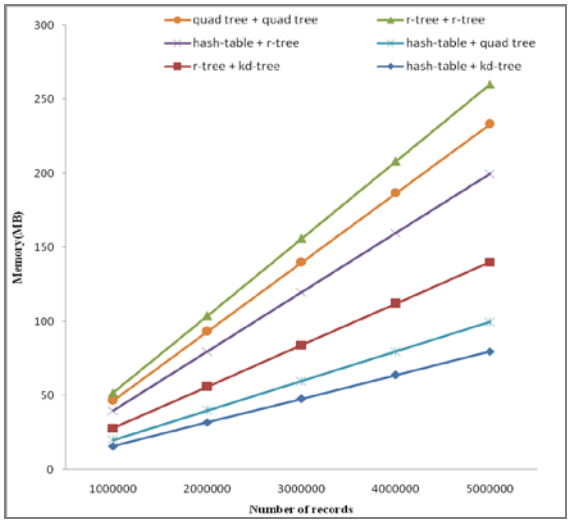

Figure 16. Memory cost for two-attribute indices

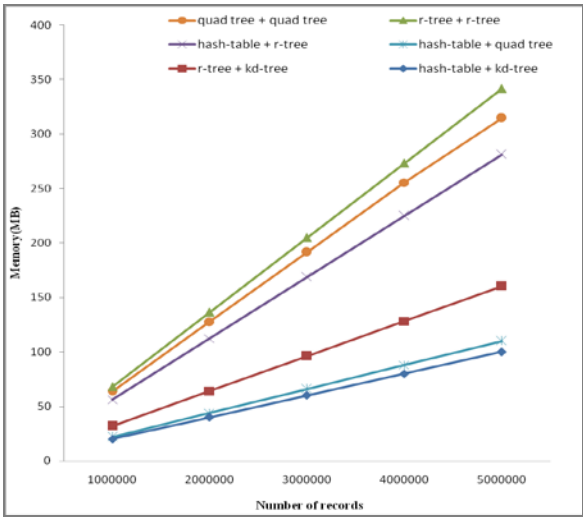

Figure 17. Memory cost for three-attribute indices

5.3 Time Cost for Hit Data Search

In this section, we evaluate the query search efficiency for all six of multi-attribute indices. We randomly pick 50,000 records from databases to be query examples then search in the indices to

insure the hit data search. We accumulated the time needed for all query searches. The experimental results are presented in Figure 18 and Figure 19. The Figure 18, demonstrating all the query data searching hit in two-attribute indices, shows that our three hash-based indices outperform the other three index approaches. Figure 19, which representing hit searching in three-attribute indices, also shows the similar behavior as in Figure 18 that hash-based indices outperform the other three. That is because the search in hash-based index is straightforward by calculating in hash function and finding the pointer in hash table then linked to the slave node for consequent searching in local data. Therefore, unlike searching in tree indices in which more comparisons have to perform in each travelled node, the search time needed in the hash table of global index is very short. The hash table + *k*-d tree is the most efficient index in this study. That is due to *k*-d tree can supports the most efficient searching comparing to the R-tree and quad-tree.

The R-tree + R-tree performs worst again. The reason is that R-tree only stores data in leaf nodes such that the queries should always search to the leaf node. Although R-tree has the advantage that it uses the minimum bounding boxes to decide whether or not to search inside a sub-tree or slave node, this advantage cannot benefit R-tree in this study due to the query searches were all hit. The Hash table + *k*-d trees comparing to R-tree + R-tree can save time cost up to 90.10% and 97.84% hit data searching in 2-attribute and 3-attribute respectively, where there are five million records in indexes.

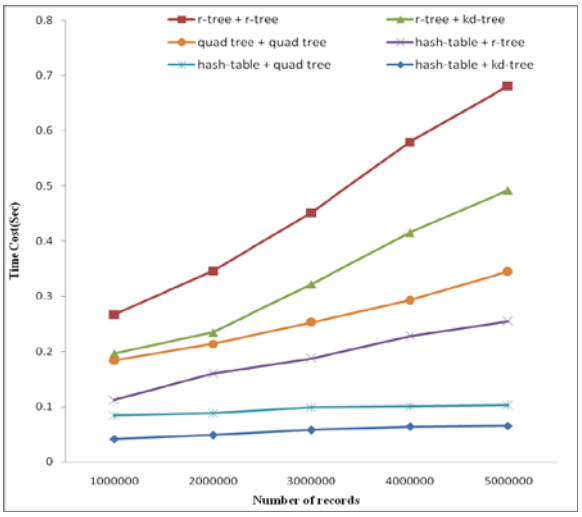

Figure 18. Hit data searching in two-attribute indices

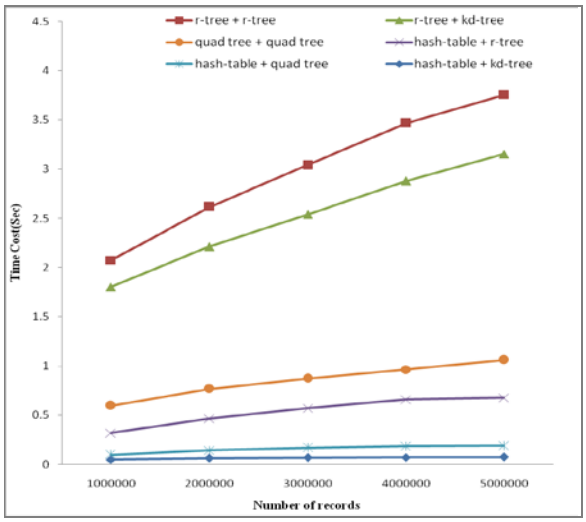

Figure 19. Hit data searching in three-attribute indices

5.4 Time Cost for No Hit Data Search

After we study the time consuming for hit data search, we would like to examine the time cost for no hit data search in this section. As discussed in last section, R-tree has the advantage for using the minimum bounding boxes to decide whether or not to search inside a sub-tree or slave node. If a query does not intersect the bounding rectangle, it will be filtered out quickly and not necessary searching down to the leaf nodes or slave nodes. Therefore, no hit data searching might benefit R-tree. In this experiment, we designed and randomly generated 50,000 query samples which cannot be found in our database to insure searching with no hit. These queries are also searched in four kinds of multi-dimensional index which represent two-attribute indices and three-attribute indices. Again, we accumulated the time needed for all query searches. Our experimental results are demonstrated in Figure 20 and Figure 21. Obviously, these two figures have the similar behaviors with that of the Figure 18 and Figure 19. Our proposed hash-based multi-attribute indices are more efficient than the other three tree-based indices again. The hash table + *k*-d tree is still the most efficient index for applying in either 2-attribute or 3-attribute data searching in this study. The hash table + quad-tree is also performs well.

Furthermore, although the curves of R-tree in Figure 20 and Figure 21. are slight lower than in Figure 18 and Figure 19, respectively, the advantage of filtering out no hit query for R-tree is not obvious. In the report of [14] by Michela and et al. have proved that R-tree is based on minimum bounding rectangles and the three

dimensional extension consists of minimum bounding boxes and techniques are often low in efficiency, as sibling nodes might overlap.

The Hash table + *k*-d trees comparing to R-tree + R-tree can save time cost up to 90.40% and 97.97% no hit data searching in 2-attribute and 3-attribute respectively, where there are five million records in indexes.

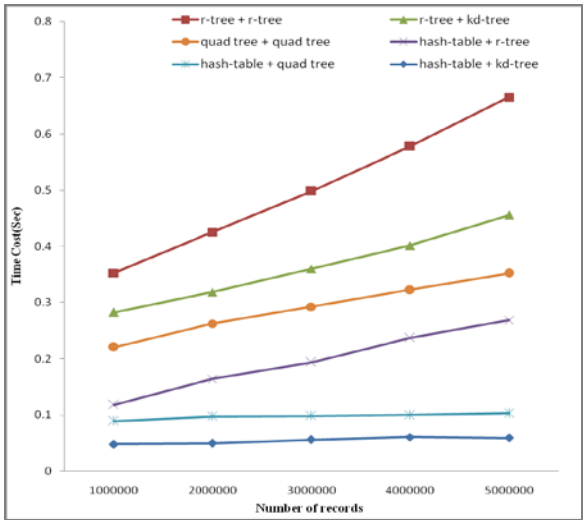

Figure 20. No hit data searching in two-attribute indices

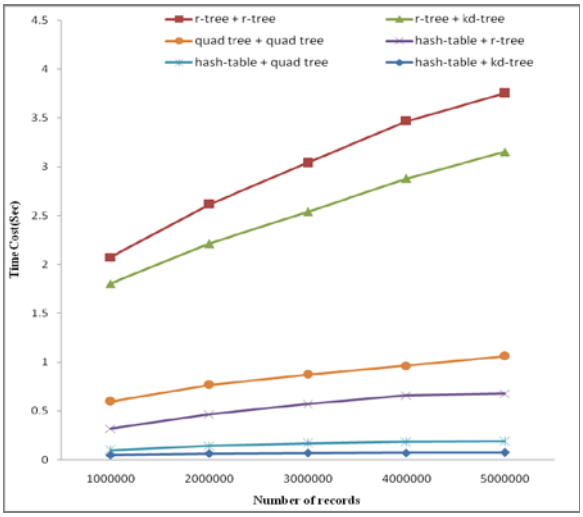

Figure 21. No hit data searching in three-attribute indices

6. CONCLUSIONS

There were few research reports proposed multi-attribute indexing schemes for Cloud platform to manage the huge and variety data to address the complex queries efficiently. All these existing indexing schemes for Cloud platform construct tree-based indices. In this research, we investigated the load balancing issue and also developed the hash-based multi-attribute index structures for Cloud platform. Our experimental results demonstrate that our proposed indices outperform the existing multi-attribute index schemes. Furthermore, among of our proposed indices the hash table in master node as the global index combining with *k*-d tree indexes in slave nodes for local indices is the most efficient structure in both memory consuming and supporting query search.

References

[1] M.K. Aguilera, W. Golab and M.A. Shah, "A Practical Scalable Distributed B-Tree," in *Proc. of the VLDB Endowment, Vol. 1, Issue 1*, August 2008.

[2] A. V. Aho and J. D. Ullman, "Optimal Partial-match Retrieval When Field are Independently Specified," in *Proc. of the ACm Transaction on Database Systems, Vol. 4, pp. 168-179*, June 1979.

[3] J.L. Bentley, "Multidimensional binary search trees used for associative searching", in *Communications of the ACM* , Vol. 18, Issue 9, pp. 509-517, September 1975.

[4] L. Ding, B. Qiao, G. Wang, and C. Chen, "An Efficient Quad-Tree Based Index Structure for Cloud Data Management," in *proc. of the 12th international conference on Web-age information management*, pp. 238-250, 2011.

[5] D. Eppstein, M. T. Goodrich and Jonathan Z. Sun, "The Skip Quad tree: A Simple Dynamic Data Structure for Multidimensional Data," in *Proceedings of the 21st Symposium on Computational Geometry (SGC)*, pp. 296-305, 2005.

[6] H. Garcia-Molina, J. D. Ullman, and J. Widon, Database System Implementation, Prentice Hall, Inc., Upper Saddle River, NJ, USA, 1999.

[7] C. Gong, J. Liu, Q. Zhang, H. Chen and Z. Gong, "The characteristics of Cloud computing," in *proc. of the 39th International Conference on Parallel Processing Workshops (ICPPW)*, pp.275-279, 2010.

[8] A. Guttman, "R-trees a dynamic index structure for spatial searching," in *proc. of the ACM SIGMOD*, June 1984.

[9] A. Henrich., H. W. Six, and P. Widmayer, "The LSD tree: Spatial access to multidimensional point and non-point objects," in *proc. Of the 17th International*

Conference on Very Large Data Bases, pp. 525-535, September, 1991.

[10] S. M. Kearney, "Advanced Database Systems- Multi-Attribute Indexing," BBIT4/SEM4 Advanced Database Systems.

[11] G.b. Li and J. Tang, "A new HR-tree index based on hash address," in *IEEE Signal Processing Systems (ICSPS), 2nd International*, Vol. 3, Issue 1, pp. 35-38, July 2010.

[12] P. y. Li; Y. x. Sui and H. j. Yang, "The parallel computation in one-way hash function designing," in *IEEE Computer, Mechatronics, Control and Electronic Engineering (CMCE) International Conference*, vol. 1, pp. 189 - 192, October 2010.

[13] J. W. Lioyd, "Optimal Partial-match Retrieval," in *BIT*, vol. 20, pp. 406- 413, 1980.

[14] B. Michela, B. schoen, D.F. Laefer and M. Sean "Storage, manipulation, and visualization of LiDAR data," in *proc. of the 3rd ISPRS International Workshop on 3D Virtual Reconstruction and Visualization of Complex Architectures (3D-ARCH)*, Trento, Italy, 25-28 February 2009.

[15] S. Moran, "On the Complexity of Designing Optimal Partial-match Retrieval Systems," in *ACM Transaction on Database Systems*, vol. 8, 543-551, December, 1983.

[16] B. Mozafari and N.H. Savoji, "A new collision resistant hash function based on optimum dimensionality reduction using Walsh-Hadamard transform," *the 9th International Conference on Information Technology* (*ICIT '06*), pp. 149-154, Dec. 18-21, 2006.

[17] .K. Ramamohanarao， J. Shepherd and R. Sacks-Davis, "Multi- attribute Hashing with Multiple File Copies for High Performance Partial-match Retrieval," in *proc. Of BIT*, vol. 30, pp. 404-423, 1990.

[18] S. Ratnasamy, P. Francis, M. Handley, R. Karp, and S. Shenker, "A scalable content-addressable network," in *proc. of conference of the ACM Special Interest Group on Data Communication (SIGCOMM)*, San Diego, CA, USA, 2001.

[19] M. Singh and D. Garg, "Choosing Best Hashing Strategies and Hash Functions," in *IEEE International Advance Computing Conference*, pp. 50 - 55, March 6-7, 2009.

[20] 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)*, pp.591-602, Indianapolis, Indiana, June 2010

[21] Y. Xia, S. Chen, and V. Korgaonkar, "Load Balancing with Multiple Hash Functions in Peer-to-Peer Networks," *the 12th International Conference on Parallel and Distributed Systems* (*ICPADS'06*), Minneapolis, MN, July 12-15, 2006.

[22] D. xiao, XF. Liao, and SJ. Deng, "Parallel keyed hash function construction based on chaotic maps," *Physics Letters A*, vol. 372, pp. 4682-4688, 2008.

[23] S. Zhang, S. Zhang, X. Chen and S. Wu, "Analysis and research of Cloud computing system instance," in *proc. of the second international conference on Future Networks*, pp. 88-92, Sanya, Hainan, January 22-24, 2010.

[24] D. Zhang, D. Agrawal, G. Chen, and A.K.H. Tung, "HashFile: An efficient index structure for multimedia data," *the 2011 IEEE 27th International Conference on Data Engineering* (*ICDE2011*), pp. 1103-1114, April 11-16, 2011.

[25] X. Zhang, J. Ai, Z. Y. Wang, J. H. Lu and X. F. Meng, "An Efficient Multi-Dimensional Index for Cloud Data Management," in *proc. of the first international workshop on Cloud data management*, pp. 17-24, Hong Kong, November 2009.