An Efficient Index Method for Multi-Dimensional Query in Cloud Environment

Youzhong Ma[1,2(✉)], Xiaofeng Meng[2], Shaoya Wang[3], Weisong Hu[3], Xu Han[2], and Yu Zhang[2]

[1] School of Information and Technology, Luoyang Normal University, Luoyang, China
{mayouzhong,xfmeng,hanxumelody,zhangyu1990}@ruc.edu.cn
[2] School of Information, Renmin University of China, Beijing, China
[3] NEC Labs China, Beijing, China
{wang_shaoya,hu_weisong}@nec.cn

Abstract. The explosion of the data in many applications brings big challenges to the traditional relational database management systems, they are in trouble with the scalability when deal with very large volume data. The cloud-based databases provide a promising approach to manage massive data because of their native good scalability, fault tolerance and high availability, while they can not provide efficient multi-dimensional queries processing on the non-rowkeys. In real applications, many queries are focused on many attributes, at the same time we can not predicate all the query requirements and the query requirements always changes. In this paper we proposed an efficient index solution layered on the key-value store that can deal with multi-dimensional queries efficiently on large scale data in cloud environment, and the solution can support adding new index dynamically for the new query requirements. We implemented a prototype based on HBase and performed comprehensive experiments to test the scalability and efficiency of our proposed solution.

Keywords: Multi-dimensional index · Cloud computing · HBase

1 Introduction

The explosion of the data in many applications brings big challenges to the traditional relational database management system (RDBMS), the RDBMS can provide efficient multi-dimensional query processing because of its lots of index structures, such as KD trees [1], R-trees [2] et al., but RDBMS has big trouble with the scalability when deal with massive data. The cloud-based databases provide an effective way to deal with the big data because of its native good scalability, fault tolerance and high availability, but the cloud-based databases, such as Bigtable [3] and HBase, can only support efficient query on the rowkey, they can not provide efficient multi-dimensional queries on the non-rowkeys. This shortcoming limits the widespread usage of the cloud-based databases in many applications.

W. Qiang et al. (Eds.): CloudCom-Asia 2015, LNCS 9106, pp. 307–318, 2015.
DOI: 10.1007/978-3-319-28430-9_23

With the development of GPS technology and widely spread of smart phones, location based service(LBS) and location based social network(LBSN) have been used by many users. LBS providers can collect users' location information via their smart phones, based on these information, LBS providers can offer many interesting service for the users, e.g. recommending nearest Peking Duck restaurant. The users can also find the nearby friends through LBSN. In order to conduct the above services, the system needs to support multi-dimensional query, such as on time, location and other attributes. In addition, in many other applications, the queries are focused on multiple attributes, such as e-commerce, the internet of things applications. At the same time we can not predicate all the query requirements and the query requirements always changes. In this paper we proposed an efficient index solution layered on the key-value store that can deal with multi-dimensional queries efficiently on large scale data in cloud environment, and the solution can support adding new index dynamically for the new query requirements. We implemented a prototype based on HBase and performed comprehensive experiments to test the scalability and efficiency of our proposed solution. The main contributions of the paper are as follows:

1. we propose an efficient index solution layered on the key-value store that can deal with multi-dimensional queries efficiently on large scale data in cloud environment, and the solution can support adding new index dynamically for the new query requirements.
2. We propose a Region Split Tree as the global index to reorganize the regions based on the selected key attributes, so we can locate the related regions through the Region Split Tree for a given query.
3. We develop a prototype system based on HBase and perform comprehensive experiments to test the scalability and efficiency of our proposed index solution.

The organization of the paper is as the follows: Sect. 2 describes the related research works about the index techniques for cloud data management; Sect. 3 gives the system overview of our proposed index solution; In Sects. 4 and 5, we introduce the details of global index and local index respectively; Sect. 6 mainly gives the detailed procedure of the range query algorithm; In Sect. 7, we perform comprehensive experiments to test our proposed index solution; We conclude this paper in Sect. 8.

2 Related Works

Many research works have been done to study the index techniques on the cloud data management, they can be divided into many different categories according to their index techniques:double-level index framework, distributed index, bitmap-based index, Hadoop framework based index, index solution based on key-value stores and other index techniques specialized in processing some kind of data type.

Wu et al. [4] are the first to explore the index techniques for the cloud data management, they firstly proposed a double-level index framework in the cloud environment. The double-level index framework includes two parts: global index and local index. In cloud environment, the data is always stored at different storage nodes in a distributed way, each local index is built for the data at every storage node, and the global index is built based on the local index. In order to improve the query efficiency and eliminate the bottleneck of the centralized index paradigm, the computer nodes are organized into overlay networks such as CAN and Chord.

Several index approaches have been proposed based on the double-level index framework. [5] is one kind of the double-level index solutions. [5] builds up one B^+-tree at each storage node, then some index nodes of each B^+-tree are selected based on a cost model, these index nodes are reorganized into the global index using BATON overlay network. But [5] can only support point query or range query on single attribute, can not deal with multi-dimensional queries. Wang et al. [6] and Ding et al. [7] proposed other different index solutions respectively to support multi-dimensional query for the cloud data management. Wang et al. [6] built one R-tree to index the local data on each compute node, and organized the compute nodes into a CAN overlay network, the global index was constructed by selecting portion of the local R-tree index nodes to publish into the CAN overlay network. Ding et al. [7] used MX-CIF quad tree as the local index and Chord overlay network as the global index. Efficient B-tree, Wang et al. [6] and Ding et al. [7] all use P2P overlay network to organize the global index, this scheme has good scalability, but it needs additional network cost when deal with a query. Zhang et al. [8] and A-tree [9] both adopt the centralized index scheme at the global index. Zhang et al. [8] also adopted the local index plus global index structure, it used the K-d tree for local data, and in the global index level he adopted the centralized index scheme by using R-tree to organize the portion of the local K-d tree nodes.

IHBase, THBase, CCIndex [10] and MD-HBase [11] proposed some index solutions based on the key-value store. IHBase [12] and ITHBase [13] are two open source projects that provide transactional and indexing extension for hbase. CCIndex [10] is a kind of secondary index solutions based on Key-value store proposed by Zhou et al., in [10], one secondary index table was built for each indexed column. And in order to reduce the random read, the detailed information of each record was pushed into the secondary index table, so that the random read can be changed into sequential read. The author also proposed some optimization methods to support multi-dimensional query based on the several secondary indexes. CCIndex is easily to be implemented, but it has several drawbacks. Firstly it needs much more additional storage space when there are many indexed columns; secondly CCIndex does not support adding or removing index after the table was created. MD-HBase, as a scalable multi-dimensional data infrastructure, was proposed in Shohi et al. [11]. In [11], the author transformed the data from multi-dimensional space into one dimension by using linearization techniques such as Z-ordering and used the z-order value as the rowkey. In order to reduce the false positive scan during the query, the author divided the space

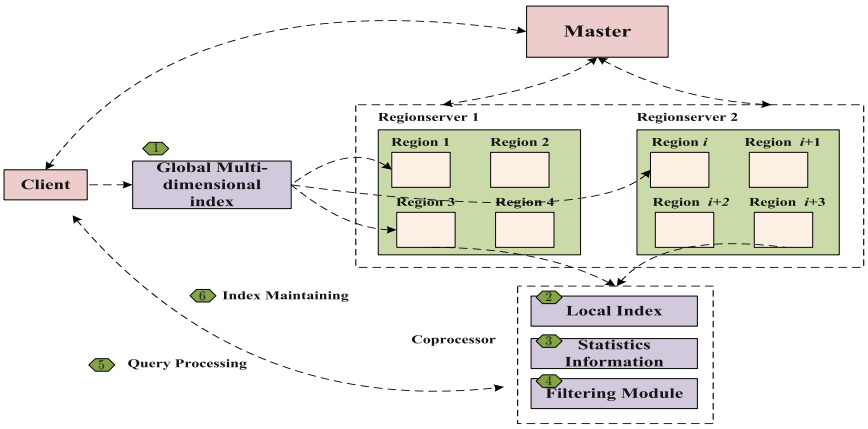

Fig. 1. System overview

into subspaces by using K-d tree and Quad-tree, and then constructed the index layer using the longest common prefix naming scheme.

3 System Overview

In order to provide efficient multi-dimensional query on large scale data and deal with the upcoming new query requirements on other attributes, we propose a hybrid index solution layered on the key-value store(HBase). The overview of the system is depicted in Fig. 1. There are mainly five components in our solution: global index, local index, statistical information, query processing and index maintaining. The global index is always a multi-dimensional index and mainly responsible for the selected key attributes; the local index is built on the non-key attributes or the new attributes for each region; statistical information module is used to collect the statistical information of the data in each region; query processing module is responsible for executing the queries based on the above index and statistical information; lastly, the index maintaining module is used to add new index on other attributes.

4 Global Index

It is well known that the query performance of the multi-dimensional index (R-tree) decreases dramatically as the dimensionality increases. Especially when the dimensionality is very high, the performance of the multi-dimensional query will be almost the same with that of full scanning the whole data. So in our solution we just create multi-dimensional index for the selected key attributes, not for all the attributes. The aim of the multi-dimensional index is to divide the data into several disjoint partitions on the selected attributes and each partition will be stored in a region in HBase, so we propose a Region Split Tree as the

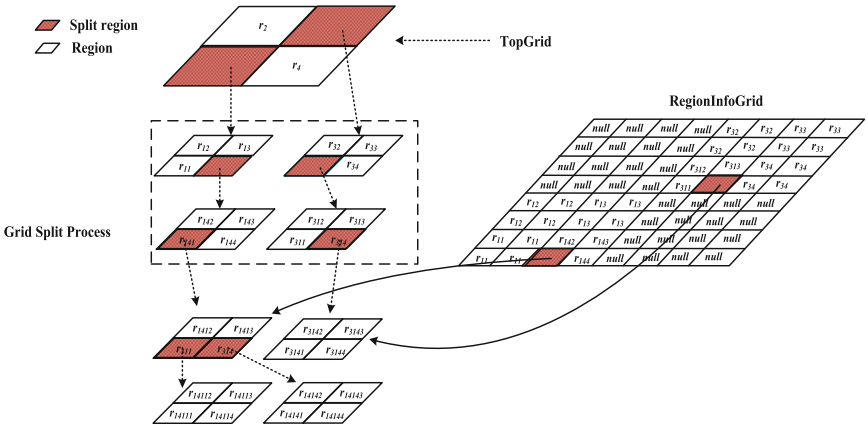

Fig. 2. Global index: region split tree

global multi-dimensional index. Figure 2 displays the overview of the Region Split Tree.

Actually Region Split Tree is a hybrid index by combining gird index and tree index. The basic idea of the grid index is to divide the k-dimensional space based on a orthogonal grid, the whole space is divided into many k-dimensional rectangle subspaces, and these rectangle subspaces are called grid directory. The strength of the grid index is that it can find the final results through limited times of access to the external storage. The main problem of the grid index is the storage of the grid directory, when the dimensionality is very high, the grid directory will be very large and the split will add many new grid directory items. So in our solution we make some modifications based on the original grid index.

TopGrid. Region Split Tree has two entrances: TopGrid and RegionInfoGrid. TopGrid is a coarse grained grid that is used to pre-split the space into partitions, each partition corresponds to a region of the HBase and the data of each partition is stored in the region. At the beginning the information that is stored in the TopGrid is the region names, as the data increases, if the number of the records in a specific partition exceeds a predefined threshold, we need to split the partition into several new partitions, at the same time we update the information that is stored in the TopGrid using the address of the new partitions. Finally all the regions can be indexed using the Region Split Tree. The granularity of the TopGird depends on the distribution of the original data, the data volume and other factors.

RegionInfoGird. When the data is skewed, the depth will be very high, it will cost too much time to locate the related regions by traversing the whole Region Split Tree from the TopGird to the bottom. In order to solve this problem, we proposed RegionInfoGrid that is a fine grained grid compared with TopGrid. RegionInfoGird is mainly used to store the information of the regions that lie at a given level (L) of the Region Split Tree. The names of the regions from $Level_2$

to $Level_L$ can be stored in the RegionInfoGrid. At $Level_L$, if a sub-grid is split again, then the address of the new sub-grids will be stored in the RegionInfoGrid. The granularity of the RegionInfoGird is the same as that of the Grid ar $Level_L$.

The information of RegionInfoGrid can be stored in main memory, If too large, we can store all the grids information into HBase table. We create an index table to store the grid directory items, the rowkey of the index table is the combination of the y-coordinate and x-coordinate, the value corresponding to each rowkey is the region name that the subspace belongs to. When we execute a query, if the query contains the selected attributes, we can get the names of the related regions by accessing the index table. Because each region may correspond to several different sub grids, we need to filter the duplicated region names.

Region Split Procedure. Given the TopGrid, we need to record the number of the records in each sub-grid, if it is over a threshold, we need to divide the sub-gird again, so on and so forth. The number of the new partitions the sub-gird is divided into must be suitable, not too big or too small. Supposing the dimensionality of the key-attributes is n, the split number is N, then:

$$N = 2^k; \quad k = \begin{cases} 2n, & n = 2 \\ n, & n > 2 \end{cases}$$

During the procedure of the gird splitting, the information stored in RegionInfoGrid needs to be updated accordingly.

Rowkey Formulation Scheme. Because the data in HBase is organized into different regions based on their rowkeys, and the rowkey is in each region is sequential, we need to design a suitable key formulation scheme. When we divide the space using Region Split Tree, we have to make sure that the data that is close in the original space is likely to be stored in the same region. So we use the z-order value as the rowkey of each record, such coding scheme can make sure that the rowkey is continuous in each region.

5 Local Index

In addition to the selected key attributes on which we build global multi-dimensional index, we have to consider other non-key attributes for satisfying the query requirements. We plan to create local index for the non-key attributes, and the local index is built for each region. The structure of the local index can be selected according to the characteristics of the attributes, we can select R-tree for those attributes which are always used together, if one attribute is always used dependently, we can use B-tree for such attribute. We have two kinds of solutions for the local index: real time processing and batch processing.

5.1 Real-Time Processing

In real time processing mode, we have to index each record when it is inserted into the region. In order to keep the consistency between the local index and

the actual data inserted into the region, we can make use of the Coprocessor technique proposed in HBase recently. When a record is inserted into a region, it will trigger a new task that is used to insert the record into the local index through Coprocessor mechanism, the record will be inserted into the region only after the record has been inserted into the local index successfully.

5.2 Batch Processing

Although the real time processing mode can make sure that the data can be indexed timely, it will bring additional burden to the system and affect the performance of the insertion. In order to maintain the high inset throughput, we can adopt the batch processing mode. According to the batch processing mode, we don't create local index for the data when they are inserted into the region. After the data of one region becomes stable, that is to say no more data will be inserted in to this region again, we can create the local index for each region by scanning the whole data in the specific region. MapReduce paradigm can be used to speed up the batch processing procedure, map task is enough for the MapReduce job and each map task is responsible for one region.

5.3 Region and Local Index Localization

HBase always moves the regions among the region servers according to some predefined load balance strategy. While the local index is built for each region and it is stored as a file on the HDFS, at the beginning each region and its corresponding local index are at the same region server. In order to reduce the communication cost, we have to move the local index together with the region. We have two methods to choose: one is that we move the local index as long as the region has been moved; another one is that we can check the local index and regions occasionally and move the local index in batch.

6 Query Processing

In this section we mainly describe the detailed procedure of the range query processing based on our proposed index solution.

6.1 Range Query Processing

$Q(E_s, E_n)$ is a multi-dimensional query, E_s and E_n are the query conditions on the selected attributes and non-selected attributes respectively. Algorithm 1 display the detailed procedure of the multi-dimensional query. Firstly we need to decide whether the query contains the selected attributes or not, that is to say if E_s is empty, we have to send the query to all the regions, so we set the names of all the regions to the related region set S_Q (line 4), otherwise we can get the related regions through querying the global index (line 6). For each region R in the related regions set S_Q, we firstly decide whether R contains the desired

results or not based on the statistical information, if not, region R can be skipped (line 9,10), otherwise we need to process R (line 11–21). When process region R, we need to decide whether it is necessary to use the local index or not, if yes, we can get the final results through the local index (line 12,13), if not, we need to scan the whole region to find the final results (line 14–19).

Algorithm 1. Range Query Processing

Input: $Q(E_s, E_n)$
E_s: conditions on selected attributes
E_n: conditions on non-selected attributes
Output: R_Q
1: $R_Q \leftarrow \emptyset$
2: $S_Q \leftarrow \emptyset$ /*Initialize the related region set to empty*/
3: **if** ($E_s == \emptyset$) **then**
4: $S_Q \leftarrow$the names of all the regions
5: **else**
6: $S_Q \leftarrow$ getRegions($Q(E_s, E_n)$, RST)
7: **end if**
8: **for each** region R in S_Q **do**
9: **if** (R doesn't contain the desired result) **then**
10: continue
11: **else**
12: **if** (need to query localindex) **then**
13: $R_Q \leftarrow R_Q \cup searchLocalIndex(R, E_s, E_n)$
14: **else**
15: **for each** record r in R **do**
16: **if** $r \in (E_s, E_n)$ **then**
17: $R_Q \leftarrow R_Q \cup r$
18: **end if**
19: **end for**
20: **end if**
21: **end if**
22: **end for**

6.2 Get Related Regions Through Global Index

Algorithm 2 displays how to get the related regions based on the global index: Region Split Tree, it mainly contains three steps. Firstly we get the sub-girds by querying the TopGird of the Region Split Tree, if the elements of the query result are all Regions (that is to say no region has been split), we return the result directly (line 3–6); if some regions have been split, we need to query the RegionInfoGird of Region Split Tree again (line 7–10). If the elements of the query result are all Regions, we combine the result and previous result S_Q, then retrun S_Q (line 11–13); otherwise we need to query the Region Split Tree continuously by using the entrances retrieved from the RegionInfoGird (line 14–18). Finally the related region set S_Q

is returned, the Range Query Processing Algorithm will continue to process the related regions.

Algorithm 2. Get Related Regions Through Global Index

Input: $Q(E_s, E_n)$
E_s: conditions on selected attributes
E_n: conditions on non-selected attributes
RST: global index: Region Split Tree
Output: S_Q: the related regions set
1: $S_Q \leftarrow \emptyset$ /*Initialize the related region set to empty*/
2: $subGridSet \leftarrow \emptyset$ /*The temporal set to store the query result*/
3: $subGridSet \leftarrow getGirds(Q, RST.TopGird)$
4: **if** (*the regions in subGridSet are Region*) **then**
5: $S_Q \leftarrow subGridSet$
6: return S_Q
7: **else**
8: $S_Q \leftarrow Regions\ in\ subGridSet$
9: $subGridSet \leftarrow \emptyset$
10: $subGridSet \leftarrow getGirds(Q, RST.RegionInfoGird)$
11: **if** (*the regions in subGridSet are Region*) **then**
12: $S_Q \leftarrow S_Q \cup subGridSet$
13: return S_Q
14: **else**
15: $S_Q \leftarrow S_Q \cup Regions\ in\ subGridSet$
16: $S_Q \leftarrow S_Q \cup$
17: $getRegions(Q, RST.RegionInfoGird.Entrances)$
18: **end if**
19: **end if**
20: return S_Q

7 Experiment Evaluation

In this section we will perform comprehensive experiments to test the performance of our prototype. We will compare our proposed index RegionSplitTree with other two index: UQE-Index [14] and EMINC [8], UQE-Index [14] is an Update and Query Efficient index for massive IOT data in cloud environment, EMINC [8] index refers to Efficient Multi-dimensional Index with Node Cube. The prototype is implemented based on HBase-0.94.5 and Hadoop-1.0.3 The experiments were performed on an in-house cloud platform, the cloud platform size varies from 4 to 16 nodes that are connected with 1 Gbit Ethernet switch, the configuration is: CPU: Q9650 3.00 GHz, memory: 4 GB, disk:1 TB, os: 64 bit Ubuntu 9.10 server. We mainly focus on the insert throughput, query processing performance such as rang query, point query.

The experiments are performed on two different data set: uniform distribution data set and skewed distribution data set, each data set contains 200 million records. The data sets are synthetic data, and they are generated in the following scheme: in the distribution uniform data set, each record has six attributes: time, latitude, longitude, att_1, att_2, contastantString. The values of longitude and latitude are uniformly distributed in the range [1, 10000], the time is the system time when the data is generated, ConstantString is used to tune the record size, att_1 and att_2 are two additional attributes on which we need to build local index. In the skewed distribution data set, each record has the same attributes as the uniform distribution data set, the difference is that longitude, latitude, att_1 and att_2 are skewed following *zipf* like distribution, and we set the skew factor as 0.5.

7.1 Performance of Insert Throughput

Figure 3 shows that the insert throughput of RegionSplitTree, UQE-Index and EMINC. We can see that the insert throughput of RegionSplitTree is up to 6000 rec/s, it is two times of EMINC. We also can see that UQE-Index is the best, it is two times of RegionSplitTree. The main reason is that UQE-Index can only create index on three attributes: time, latitude and longitute, while RegionSplitTree creates index on five attributes: time, latitude, longitute, att_1, att_2. So the cost of RegionSplitTree will be higher than that of UQE-Index.

7.2 Performance of Point Queries

Figures 4 and 5 show that the performance of point query for uniform data set and skewed data set. When the number of the computer nodes exceeds 8, the point query performance of RegionSplitTree is always better than that of UQE-Index for both uniform data set and skewed data set. But for the skewed data set, the performance of UQE-Index decreases as the number of the computer nodes increases, the main reason is that the data is skewed, although the computer nodes increase, the data is still inserted into some fixed computer nodes; on the other hand, communication cost will increase as the computer nodes increase. From the experiment result we can see that RegionSplitTree is more suitable for the skewed data set than UQE-Index.

7.3 Performance of Range Queries

Figure 6 shows the performance of range query for uniform data set. From the figure we can see that RegionSplitTree has the best performance when the selectivity is lower than 0.01 %, while the performance of RegionSplitTree becomes worst when the selectivity is more than 0.1 %. But Fig. 7 shows that the range query performance of RegionSplitTree is the always the best for all the selectivity on skewed data set.

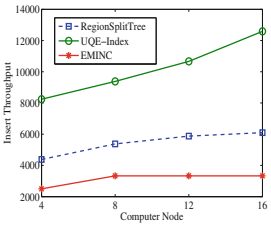

Fig. 3. Insert throughput

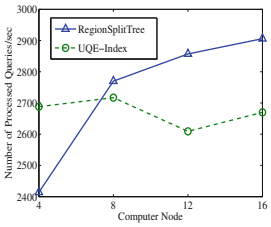

Fig. 4. Point query-uniform

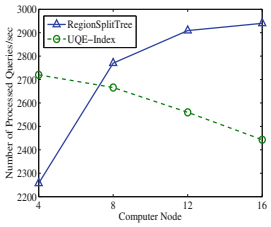

Fig. 5. Point query-skewed

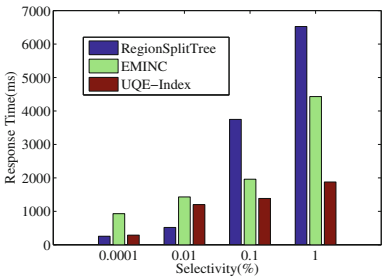

Fig. 6. Performance of range queries (uniform)

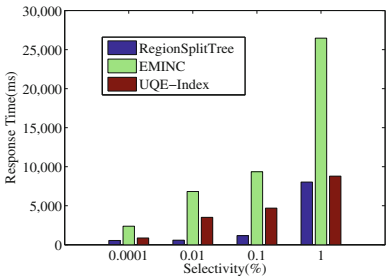

Fig. 7. Performance of range queries (skewed)

8 Conclusions and Future Work

In this paper we proposed an efficient index solution layered on the key-value store that can deal with multi-dimensional queries efficiently on large scale data in cloud environment, and the solution can support adding new index dynamically for the new query requirements. We pick up some important attributes which are often used in the queries as the key attributes and create global multi-dimensional index for the key attributes, we proposed a Region Split Tree as the global index. We build up local index for non-key attributes if needed, the ocal index can be R-tree or B-tree. Finally, we implemented a prototype based on HBase, and comprehensive experiment evaluations have been done to analyze our solution's efficiency and scalability.

In this paper we mainly focus on multi-dimensional range query, in the future we plan to extend our works to support other more complex query, such as KNN query, aggregate query. In addition to query on the simple data, we plan to deal with queries on the more complicated data, such as strings, vector data, graph data, an so on.

Acknowledgment. This work was partially supported by the grants from NEC Labs China; the Natural Science Foundation of China (No. 61070055, 91024032, 91124001); the Fundamental Research Funds for the Central Universities, and the

Research Funds of Renmin University (No. 11XNL010); National 863 High-tech Program (2012AA011001, 2013AA013204); Science and technology project of Henan Province (No. 152102210332).

References

1. Bentley, J.L.: Multidimensional binary search trees used for associative searching. Commun. ACM **18**, 509–517 (1975)
2. Guttman, A.: R-trees: A dynamic index structure for spatial searching. In: Conference on SIGMOD 1984, pp. 47–57 (1984)
3. Chang, F., Dean, J., Ghemawat, S., Hsieh, W.C., Wallach, D.A., Burrows, M., Chandra, T., Fikes, A., Gruber, R.E.: Bigtable: a distributed storage system for structured data. ACM Trans. Comput. Syst. **26**, 4 (2008)
4. Wu, S., Wu, K.L.: An indexing framework for efficient retrieval on the cloud. IEEE Data Eng. Bull. **32**, 75–82 (2009)
5. Wu, S., Jiang, D., Ooi, B.C., Wu, K.L.: Efficient B-tree based indexing for cloud data processing. In: VLDB, pp. 1207–1218 (2010)
6. Wang, J., Wu, S., Gao, H., Li, J., Ooi, B.C.: Indexing multi-dimensional data in a cloud system. In: SIGMOD Conference 2010, pp. 591–602 (2010)
7. Qiao, B., Wang, G., Chen, C., Ding, L.: An efficient quad-tree based index structure for cloud data management. In: Wang, H., Li, S., Oyama, S., Hu, X., Qian, T. (eds.) WAIM 2011. LNCS, vol. 6897, pp. 238–250. Springer, Heidelberg (2011)
8. Zhang, X., Ai, J., Wang, Z., Lu, J., Meng, X.: An efficient multi-dimensional index for cloud data management. In: CloudDB, pp. 17–24 (2009)
9. Papadopoulos, A., Katsaros, D.: A-tree: Distributed indexing of multidimensional data for cloud computing environments. In: CloudCom, pp. 407–414 (2011)
10. Zha, L., Wang, S., Xu, Z., Zou, Y., Liu, J.: CCIndex: a complemental clustering index on distributed ordered tables for multi-dimensional range queries. In: Ding, C., Shao, Z., Zheng, R. (eds.) NPC 2010. LNCS, vol. 6289, pp. 247–261. Springer, Heidelberg (2010)
11. Nishimura, S., Das, S., Agrawal, D., Abbadi, A.E.: MD-HBase: A scalable multi-dimensional data infrastructure for location aware services. In: Mobile Data Management 2011, pp. 7–16 (2011)
12. Kulbak, Y., Washusen, D.: IHBase (2010). http://github.com/ykulbak/ihbase
13. Kennedy, J., et al.: ITHBase:transactional and indexing extensions for HBase (2010). https://github.com/hbase-trx/hbase-transactional-tableindexed
14. Ma, Y., Rao, J., Hu, W., Meng, X., Han, X., Zhang, Y., Chai, Y., Liu, C.: An efficient index for massive IOT data in cloud environment. In: CIKM 2012, pp. 2129–2133 (2012)