He et al. Journal of Cloud Computing: Advances, Systems and Applications (2016) 5:10
DOI 10.1186/s13677-016-0060-1

Journal of Cloud Computing: Advances, Systems and Applications

RESEARCH Open Access

Dynamic multidimensional index for large-scale cloud data

Jing He[1,3], Yue Wu[1], Yunyun Dong[2], Yunchun Zhang[3] and Wei Zhou[3*]

Abstract

Although several cloud storage systems have been proposed, most of them can provide highly efficient point queries only because of the key-value pairs storing mechanism. For these systems, satisfying complex multi-dimensional queries means scanning the whole dataset, which is inefficient. In this paper, we propose a multidimensional index framework, based on the Skip-list and Octree, which we refer to as Skip-Octree. Using a randomized skip list makes the hierarchical Octree structure easier to implement in a cloud storage system. To support the Skip-Octree, we also propose a series of index operation algorithms including range query algorithm, index maintenance algorithms, and dynamic index scaling algorithms. Through experimental evaluation, we show that the Skip-Octree index is feasible and efficient.

Keywords: Cloud storage, Multidimensional index, Distributed index, Skip-Octree, Skip list, Octree

Introduction

Large-scale data management is a crucial aspect of most Internet applications. Emerging cloud computing [1–3] systems can provide users with cheap and powerful facilities for storage. As an attractive paradigm, cloud applications are required to deliver scalable and reliable management as well as process extensive data efficiently. However, most existing cloud storage systems generally adopt a distributed hash table (DHT) approach to index data, in which the data are then organized in the form of key-value pairs [4]. Thus, current cloud systems can only support keyword searches and access data through "point-query".

However, using only point queries is insufficient. Many multidimensional requirements exist for certain applications. For example, in location-based services, users often need to find an object based on its longitude, latitude, and time. In addition, they must query multiple attributes to return results immediately. Single key-value queries have clearly been unable to meet this demand. As a current solution, we can run a batch program such as a Hadoop task and scan all datasets to obtain results.

Multidimensional data structures are of considerable interest in many fields, including computational geometry, computer graphics, and scientific data visualization. Researchers have proposed multidimensional data structures such as R-tree [5], Quadtree [6, 7], and Octree [8], all of which enable efficient performance in data storage and searching systems. Quadtree is commonly used in the two-dimensional space, whereas Octree is more popular in the three-dimensional space common in many application systems. However, these traditional data indexes are normally used in a single machine or the peer-to-peer (P2P) system. Currently, with the emergence of the era of big data [9], the traditional data indices have several disadvantages such as lower storage capacity and slower efficiency.

Based on the aforementioned analysis, we have determined that the current cloud storage system performs poorly with respect to multidimensional and range queries. In addition, although traditional Octree conducts multidimensional searches effectively, it is unable to support the needs of today's big data. This is our motivation for integrating the multidimensional Octree into and developing an auxiliary dynamic index structure in a cloud environment.

This study proposes a dynamic index framework for multidimensional data in a cloud environment called Skip-Octree. Skip-Octree uses the concept behind a skip

* Correspondence: zwei@ynu.edu.cn
[3]National Pilot School of Software, Yunnan University, Kunming, Yunnan 650091, P.R. China
Full list of author information is available at the end of the article

list to improve the efficiency of the traditional Octree, and adopts double-layer Skip-Octree to construct an efficient and flexible cloud index. The main contributions of this study are listed as follows:

(1) A double-layer cloud index based on skip list and Octree is proposed in this study. To the best of our knowledge, ours is the first study to construct an auxiliary cloud index using an Octree structure. This combined index is decentralized and scalable.
(2) The skip lists are used to complete the hierarchical query of underlying Octrees. They also realize the linear indexing in a multidimensional indexing mechanism and speed up the searching process.
(3) Index maintenance algorithms and dynamic index scaling algorithms for load balancing are proposed in this study. The experiment results show the Skip-Octree index is feasible and efficient.

The remainder of this paper is organized as follows. Section 2 reviews related work. Section 3 describes the skip list, Octree, and presents a new framework of Skip-Octree on their basis. Section 4 illustrates the design of the relevant algorithms regarding Skip-Octree. Section 5 conducts tests for the algorithms related to the architecture and discusses the results of our experiments. Concluding remarks are given in Section 6.

Related works

Some existing cloud storage systems include: Google's Bigtable [10], GFS [11], and its open source implementation Hadoop [12], Amazon's Dynamo [13], and Facebook's Cassandra [14]. As a de facto standard for cloud storage systems, Hadoop has been widely used in many businesses including Yahoo, Linkedin, and Twitter. On a large scale, Hadoop allows multiple petabytes of data storage across hundreds or thousands of physical storage servers or nodes. However, lower performance of complex queries (such as range and multidimensional queries) in Hadoop presents an obstacle in its development.

Recent studies have shown that an index can dramatically improve the performance of cloud storage systems. Several studies [15–23] focusing on efficient indexes in cloud storage systems have been conducted. The study in [15] proposed a Trojan index to improve runtime performance. Its injects technology at the appropriate places by means of user defined functions (UDFs) only that affect Hadoop internally. In general, the embedded-index model is a kind of tight coupling solution. It integrates the index itself into a Hadoop framework closely to achieve high performance block selection. To decouple an index and storage system, a generalized search tree for MapReduce systems was designed in study [16]. In study [17], a global distributed B-tree index was built to organize large-scale cloud data. This method has high scalability and fault tolerance. However, it consumes considerable memory space to cache index information in the client, and it is unsuitable for processing multidimensional queries. The studies in [18, 19] proposed an improved B+ tree index. This solution adopt a double-layer index framework. The B+ tree index is built for each local data node that indexes only data on that node. By means of an adaptive algorithm, a proportion of the local B + tree nodes are published to the global index. They are efficient for single attribute queries. An R tree and content-addressable-network (CAN)-based multidimensional index schema called RT-CAN was proposed in study [20]. In RT-CAN, a CAN [21] overlay is constructed on top of the local R-tree indexes. In addition, a dynamic index node selection algorithm and cost model were proposed for RT-CAN. This solution provides high performance for multi-attribute queries. Similar to RT-CAN, a VA-file and CAN-based index framework was presented in study [22], which improves query performance by eliminating false positive queries in RT-CAN. The study in [23] adopted a compressed bitmap index to construct a cloud index, which can save considerable storage cost compared to other index structures.

Although some multidimensional indexes exist in cloud environments, an Octree-based multidimensional indexing remains nonexistent.

Octree is a kind of extended Quadtree data structure, which was proposed by Dr. Hunter in 1978 and is widely used for three-dimensional space. It is most often used to partition a three-dimensional space by recursively subdividing it into eight octants. Its tree structure has an advantage in terms of spatial decomposition, so it has been widely applied in the past years. The study of Octree has mainly focused on the analysis and improvement of traditional Octree algorithms. Meanwhile, Octree has also often been used in many 3D applications [24, 25]. The use of Octrees for 3D computer graphics was pioneered by Donald Meagher at Rensselaer Polytechnic Institute, as described in the study in [26]. In the study in [27], the author proposes a hybrid spatial index structure called ORSI, which is based on Octree and R tree. The experimental results show that the hybrid structure has more advantages than previous use of R tree on a 3D spatial index.

Current big data applications such as 3D spatical are a burden on traditional data indexes, not only in terms of space, but also high cost of storage. In addition, current cloud storage systems usually adopt a key-value model to organize data to retrieve data efficiently. This model only supports exact matching and thus does not work well with multidimensional data applications. Therefore, building a dynamic cloud storage index framework for multidimensional data is necessary.

In this study, we propose a novel skip list and Octree-based dynamic index. As far as we know, ours is the first work to set up an auxiliary cloud index using a skip list and Octree structure.

Framework of the Skip-Octree index

In cloud storage systems, a whole dataset is distributed and stored on multiple data servers. Therefore, query performance is mainly affected by two aspects. One is the manner in which to locate the corresponding data servers that stored user required data effectively. The other is the manner in which to improve the efficiency of data access on each local data server. In this study, a new double-layer cloud indexing framework based on Octree and skip list is proposed.

Background of Octree and skip list

Octree is a type of multidimensional data structure with which a multidimensional data space is recursively divided into eight equal subspaces (namely quadrants) until a quadrant contains only one data object. In addition, Octree is an adopted tree-based storage structure. For an Octree, an original data space is represented as a root node. Then, eight quadrants which act as eight children nodes of the root are generated by space partition. However, under the condition in which data is both sparsed and skewed, the query performance of Octree is worse than sequence retrieve. Hence, the compressed Octree was proposed in the study in [28]. In a compressed Octree, all empty paths are removed. Compared with R tree [29], the space division method of compressed Octree is simpler, and no space overlap occurs. Therefore, compressed Octree is used to index local data in this study. For simplicity, the compressed Octree is also called Octree in our cloud index framework.

The skip list [30] is a randomized data structure that organizes elements with hierarchical ordered link lists. Thus, it is an extension of the ordered list. Because query processing on each layer can skip many elements, a skip list can provide adequate query performance with a balanced binary tree. In addition, because a randomized algorithm is adopted to maintain balance rather than employing strictly enforced balancing, the insertion and deletion operations in a skip list are much simpler and considerably faster than the balanced binary tree. Furthermore, skip list is well suited to parallel computation applications. The insertion can be performed in parallel using different positions of the ordered list without rebalancing the global data structure. Skip list has been embedded in some popular key-value store databases such as Leveldb and Redis.

Strictly speaking, skip list is not a search tree, but its expected time complexity is $O(\log_2 n)$, which is similar to a binary search tree. In our Skip-Octree, the idea of skip list is utilized to accelerate the data retrieval efficiency of Octree.

Skip-Octree index specification

Octree is an efficient three-dimensional space partition method. However, in a cloud environment, extensive data can enlarge Octree to such an extent that it becomes inaccessible. In this section, our proposed index structure called Skip-Octree is described. Skip-Octree provides a hierarchical view of the compressed Octree to allow for logarithmic expected-time querying.

Design of Skip-Octree

Based on the randomizing idea of a skip list, the original dataset is randomly divided into subsets with a probability of 1/2. In addition, an individual Octree is constructed for each dataset.

In Fig. 1, Q_0, Q_1, and Q_2 are three datasets, where Q_0 is the original dataset, Q_1 contains approximately half the data of Q_0 and which is a subset of Q_0, and Q_2 is a subset of Q_1. The query request is processed from right to left, that is, from the smallest Octree to the largest. For each non-empty subspace, a pointer links it between different layers of the Octree. For example, if a user wants to search a keyword k, the hierarchical Octree index performs this query request at Q_2. Then, because k is not found on Q_2, this query request is redirected to Q_1. Finally, Q_0 receives this query request and obtains k. Because this query procedure has similar properties to those of a skip list, the hierarchical Octree is essentially a skip list reconstruction.

Definition of Skip-Octree The Skip-Octree is defined by a sequence of subsets Li of the input points S with $L_0 = S$ and builds a compressed Octree Qi for each Li. For $i > 0$, Li is sampled from L_{i-1} by maintaining each point with a probability of 1/2. For each Li, a compressed Octree Qi is built for the points in Li. Therefore, Qi can be seen as forming a sequence of levels in the skip list such that L_0 and $Ltop$ are the bottom and top levels, respectively.

As Fig. 2 illustrates, a skip list is a randomized data structure in which level 0 is denoted as L_0 that records all original data. In the same manner, L_1 records approximately half of the data of L_0 and L_2 records approximately half those of L_1. In Skip-Octree, L_0, L_1, and L_2 correspond to the three hierarchal Octree Q_0, Q_1, and Q_2. The multidimensional data space is partitioned by Octree to obtain multiple level subspaces. The skip list is used to organize these hierarchical data points and accelerate query performance. In a skip list, the same nodes between the upper and lower layers are associated with the pointer. Thus, with the pointer pointed to the root node in the topmost layer, we can find the specific keyword by having the pointer move down. In addition, with the locality sensitive hashing function [31], the

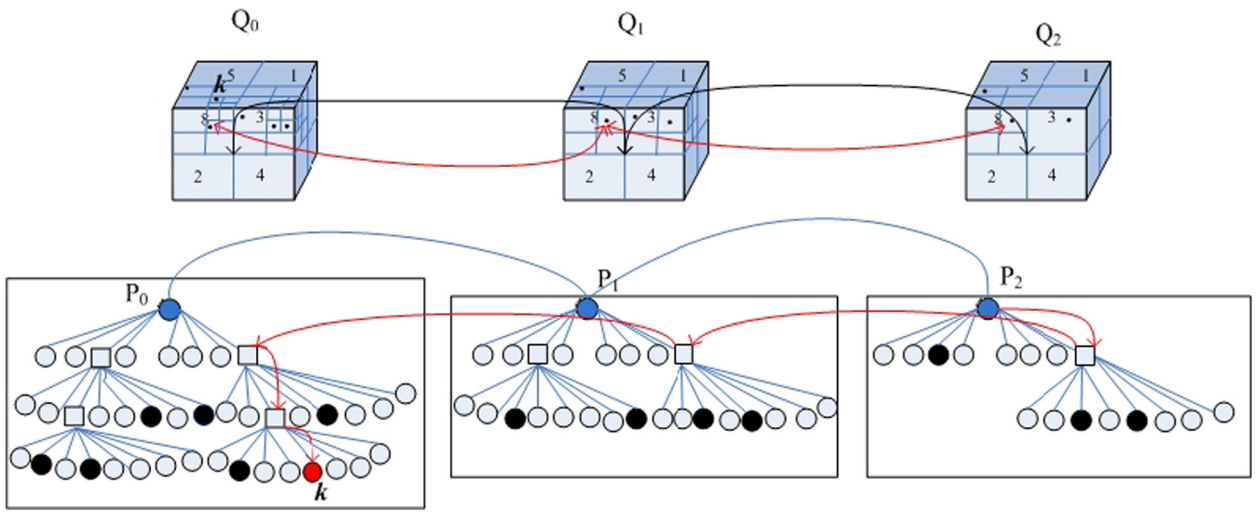

Fig. 1 Overview of a hierarchical Octree

points that belong to the same quadrant in the Octree map to the adjacent position of the skip list sequence.

Time complexity Given a point x, the searching time for x in a randomized multidimensional Skip-Octree of n points is $O(log_2 n)$.

Proof Assume n points exist in a multidimensional data space with a probability of ½. The original dataset is divided into at most $log_2 n$ subsets. Thus, the layers of the Skip-Octree are $log_2 n$. The query proceeds in a top-down fashion from the root consuming $O(log_2 n)$ time. Simultaneously, the query proceeds forward on each layer only if the search key x is smaller than the current keyword. Otherwise, it skips down to the next layer according to the parent-child link. The forward move time is $O(1)$. Therefore, the search time for x in Skip-Octree is $O(log_2 n)$ overall.

Extend Skip-Octree to index cloud data

In a distributed storage system, a large-scale dataset is usually divided into multiple small data units (known as data shards) by means of horizontal partitioning. These data shards are then stored in different computer nodes in the cloud computing environment based on the principle of load balancing. To improve query performance, a traditional global distributed index can be built for the whole dataset. However, with respect to big data, the global distributed index itself consumes much more memory space, and maintaining the index becomes difficult. Therefore, a double-layer hierarchical structure is adopted in our Skip-Octree-based cloud index. The overall framework of our Skip-Octree-based cloud index is shown in Fig. 3.

In the upper layer, the whole data space is partitioned into multiple subspaces according to the Octree structure. Each local data server is then assigned some of these subspaces. In the lower layer, a Skip-Octree is built

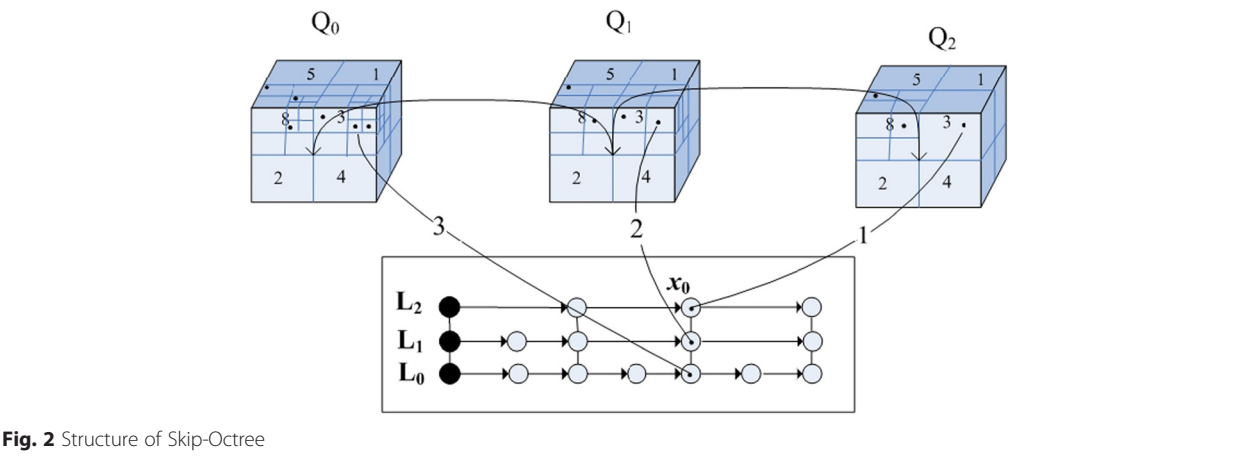

Fig. 2 Structure of Skip-Octree

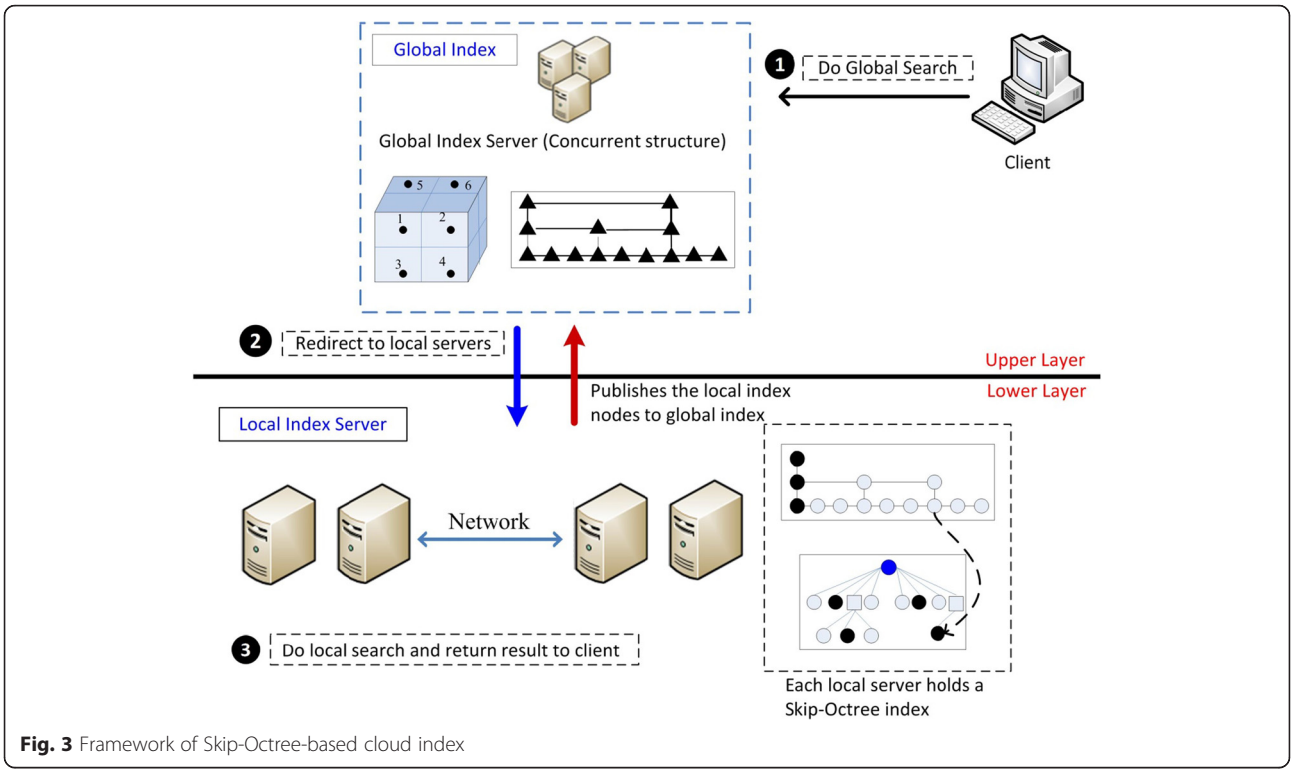

Fig. 3 Framework of Skip-Octree-based cloud index

to index data stored in each local data server. In addition, every local Skip-Octree publishes some of its own index nodes to construct a global Skip-Octree index. By combining the aforementioned two layer indexes, scanning data nodes that do not contain query results can be mostly avoided.

The query process is divided into three phases as shown in Fig. 3: (1) A query request is first send to the global index server, which performs index retrieval on global Skip-Octree to find the local data servers that may contain the query results; (2) the query request is then redirected to the corresponding local data servers; (3) finally, each selected local data server begins retrieving the data on its own local indexes, and returns the query results to the end user.

Index operating strategy

Range query processing

Range queries are widely used in cloud applications. For example, when we want to know product sales within a certain period, the search condition is a multidimensional range. In this case, the keyword index is unable to meet the user demand efficiently.

Skip-Octree can support multidimensional queries. Because we use Octree to store the data in the underlying structure. The general steps of this algorithm are as follows:

Algorithm 1 illustrates the global index process of a range query in the Skip-Octree framework. First, the function *lookup* is used to access the upper global servers to locate the first index node whose keyword is longer than *Rmin*. This index node is then mapped to the specific node of the local server (Lines 1–4). Second, the query message is forward to *Ni*'s neighbor, which invokes a similar algorithm to determine whether it is a node whose range satisfies the search range. This operation is conducted repeatedly until we find the range that is beyond the search range (Lines 5-11). Finally, local index retrieval is performed on the corresponding local data server (Line 12).

Index maintenance

In practice, the performance of inserting and deleting data also must be considered in the Skip-Octree architecture. In a cloud environment, the index maintenance process mainly consists of two steps. First, the global index server calculates the hash values of required keywords (inserting or deleting) according to the evaluation function, and then searches for the specific quadrants that contain those keywords. Second, a local index maintenance process is performed on each located local data server.

Because the skip list is a randomized data structure, the number of levels of an inserted keyword x set as random, which is generated by a random function *randomLevel()*.

Algorithm 2 provides a detailed description of the data inserting process on a cloud Skip-Octree. The locating

phase (Lines 1-3) is similar to the query process previously discussed. By calculating the hash value of the input keyword, we can find the quadrant that contains this coordinate. Simultaneously, the central node of this quadrant is mapped to the root node of the underlying local server. It next determines whether the Octree is empty; it starts the local index process if the Octree is not empty (Lines 4–5). The local index retrieval starts from the root node of the highest level Octree, and scans the skip list from the top down (Lines 6-7). When the value of the current pointer is less than the input keyword, the pointer moves forward. Otherwise, the pointer skips to the next level containing the parent and child links until the position of the new keyword is found on the lowest level of Skip-Octree (Lines 8-14). For each selected level, the keyword is inserted and the whole cloud Skip-Octree is refreshed (Lines 15-21).

For a given set which has n points in Skip-Octree, each level requires O(1) time for a pointer move and keyword comparison. Furthermore, the search time top-down on the skip list is $O(\log_2 n)$ because the height of the skip list is $O(\log_2 n)$ under the probability of 1/2. Therefore, the efficiency of inserting data on Skip-Octree is $O(\log_2 n)$.

The process of deleting data is similar to that of inserting data in Skip-Octree. It must be noted that if only one keyword is deleted on a certain level in the Skip-Octree, the height of the skip list must be modified. The specific algorithm is detailed as follows:

As Algorithm 3 illustrates, the input keyword is converted to the form of a hash key by the global index in Skip-Octree. The local data server that contains this key word is then located (Lines 1-3). In the local index, because the same keyword may appear on different levels of the skip list, Lines 4-14 are used to find the position of the input keyword X. If this keyword is not found, the deletion operation cannot be performed (Line 15). Otherwise, this keyword is removed from the local index. In addition, in the event a link list in the Skip-list is empty, the height of this skip list is reduced (Lines 16-21). Finally, the whole cloud Skip-Octree is refreshed (Lines 22-24). Similar to the data insertion operation, the efficiency of data deletion on Skip-Octree is $O(\log_2 n)$.

Dynamic index scaling

In a distributed system, the greater the amount of data that a machine processes, the bigger is its index. Simultaneously, load balancing is a major problem. To solve this, our Skip-Octree framework is dynamically scaled. This means a local data server can migrate some of its data to other servers or merge together the data of a local data server. In this manner, the parallel load balancing processing of multiple servers is realized. Furthermore, a statistical approach is used in Skip-Octree to monitor the load status of the cloud systems. After a local data server periodically sends its load statistics to the global index server, statistical information is analyzed at the global server to determine the loading factor for each local server. Based on these loading factors, the global index server decides whether certain migrations must be invoked.

In Skip-Octree, an overloaded local data server can split its local Octree, then migrate some of its Octree nodes to a new or adjacent server. We offer the following strategies to deal with such splits in Octree:

In this algorithm, S_1 is the server that must split its local Octree and S_2 represents the server that accepts the migration data. First, a temporary list *newList* is created to store migrated data during the data transformation process (Line 1). Then, all data within l in S_1 is found and exported to *newList* (Lines 2-6). The skip list for S_1 is modified by means of data removal (Line 7). After the data are imported to S_2, the Octree on S_1 is split into two parts (Lines 8-9). At last, because location information is changed on the local index, the global index is refreshed for each published local index node (Lines 10-11). The function of refreshGlobal(*newlist[i]*) consists of two steps: locate the original published index node, and update its meta-index information with new local index data.

Figure 4a represents the original Octree on Server 3, and Fig. 4b is the structure of the split operation when completed. Given a three-dimensional data space, much data are in the third and eighth quadrants. Initially, all data are stored on the same server. However, big data may lead to index memory overflow. Therefore, some data on Server 3 must be transferred to another data server. In Fig. 4b, the whole data space is divided into two subspaces. The data within the eighth quadrant is migrated to Server 4. Server 3 saves the remaining data.

In addition to the split operation, the Skip-Octree framework offers a merging algorithm, which is used to accumulate data from different local data servers. As previously discussed, the splitting algorithm can transfer some Octree data to a new or adjacent server. After migrating, our merging algorithm can help combine migrating with current data. Moreover, if a local server crashes, we can use the merging algorithm to transfer the data derived from it to another available local server before removing it from the Skip-Octree framework.

Algorithm 5 describes the process of data merging in Skip-Octree. Here, S_1 is the server that needs to transfer its local Octree, whereas S_2 is the server that accepts the migrated local Octree. First, all data in S_1 is located and buffered in a temporary *migrateList* (Line 1). Second, the function *insertValue()*, which finds the proper position for inserted data, is called to insert each data set from

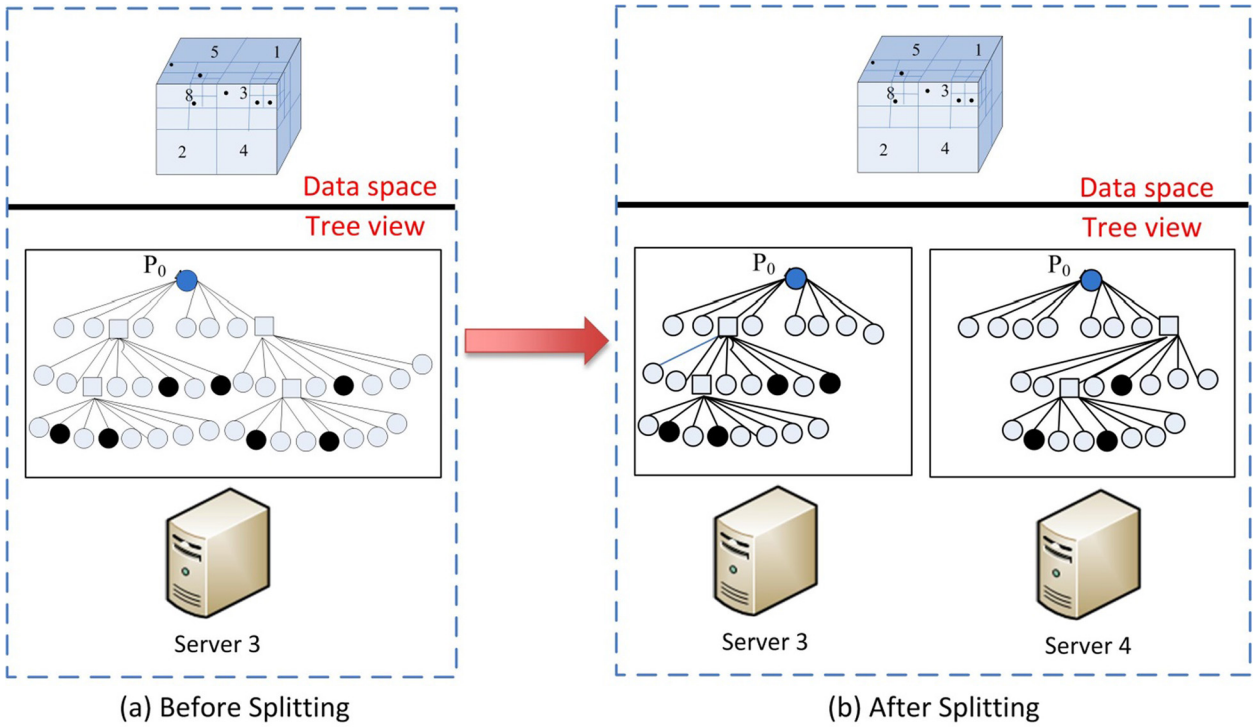

Fig. 4 Process of Octree splitting

migrateList to S_2 (Line 2-9). During the process of inserting data, determining whether the migrating data repeat with the data in S_2 is crucial. Each duplicated value is removed (Lines 5-8). If the indexNode is a published node, the relevant global index node must be refreshed (Lines 9-11). Finally, the Octree in S_2 is refreshed to ensure normal operation after merging, and the storage space of S_1 is released because empty (Line 12–13).

Experimental evaluation

To evaluate the performance of Skip-Octree architecture, we developed a simulator extended from Peersim [32]. The testing computer had an Intel Core i5 4200 M, 2.4 GHz CPU, 8 GB RAM, and a 320 G disk space running CentOS6.0 (64 bit). It was used to simulate different data nodes that extend from 10000 to 50000. In the simulator, the number of server nodes is set to 16, the type of keywords is a string, and the length of a keyword is 24. At each query, the the number of nodes is 500. For comparison, we also conducted an experiment using a traditional Octree. To guarantee the accuracy of the experimental data, we calculated the average of 10 runs of each experiment.

Figure 5 shows the performance comparison of three-dimensional range queries between Skip-Octree and traditional Octree. In this experiment, given 16 local data servers, the amount of data first increased from 1000, then grew in multiples of 1000. The search range was a radius of 0.1 cubes. We can see that Skip-Octree performs better than does the traditional Octree. The reason is that skip list realizes a hierarchical Octree structure with probability of 1/2. Through skip list, extensive data can be found rapidly without searching a huge Octree. This experiment also confirmed the feasibility of Skip-Octree's multidimensional indexing structure.

Index maintenance performance is a crucial indicator used to evaluate the effectiveness of an index structure. As shown in Fig. 6, eight data servers were created to build the cloud storage environment. As the amount of inserted data increased, the response time of the deletion operation increased. However, when the amount of data was the same, Skip-Octree always consumed less time than did the traditional Octree. Because Skip-Octree realized hierarchal Octree, considerable useless data was ignored during the deletion process.

The insertion operation in Skip-Octree is similar to that of the deletion. As shown in Fig. 7, when the amount of inserted data is small, Skip-Octree consumes nearly the same amount of time as does the traditional Octree. When the amount of inserted data increases, the Skip-Octree shows its performance advantage. The reason is that the skip list can more quickly determine inserted data positions by ignoring lots of data.

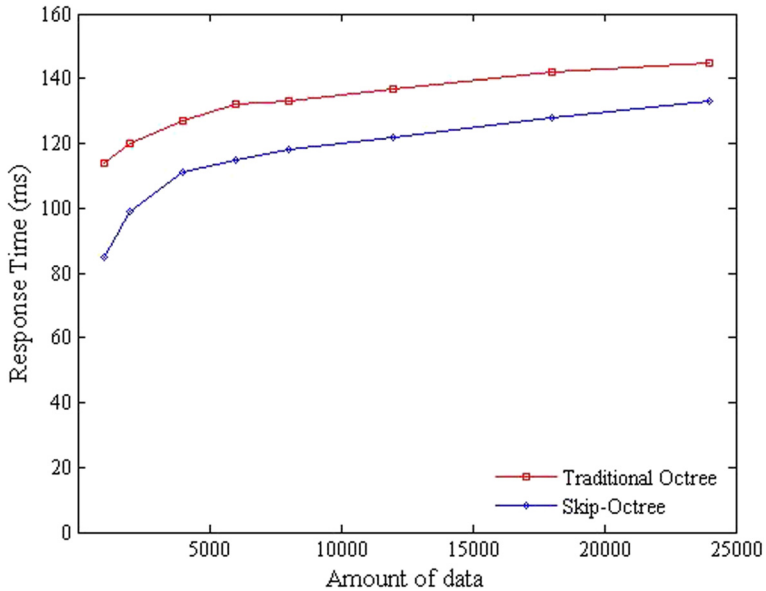

Fig. 5 Performance of range query

To achieve load balancing for the cloud storage environment, our Skip-Octree is dynamically adjusted by means of splitting and merging. In this experiment, the number of data servers was set to eight, and the amount of data increased from 10000 to 40000. Figure 8 shows a performance comparison of a given range query between a static Skip-Octree and dynamic adjusting Skip-Octree. Obviously, dynamic Skip-Octree was more efficient than static Skip-Octree, as load balancing is critical for a distributed storage system. Moreover, with each increase in the amount of data, the amount of time consumed for dynamic Skip-Octree actually decreased. The reason is that when executing a given query request in a determined cluster, if the amount of data is small, the number of local data servers selected by a dynamic skip list is greater. Otherwise, with an increasing scale of stored data, the required data are just a small portion of the entire dataset, with the resulting set stored in a few data servers. The retrieval time for a small number is less than for a large number of local indexes.

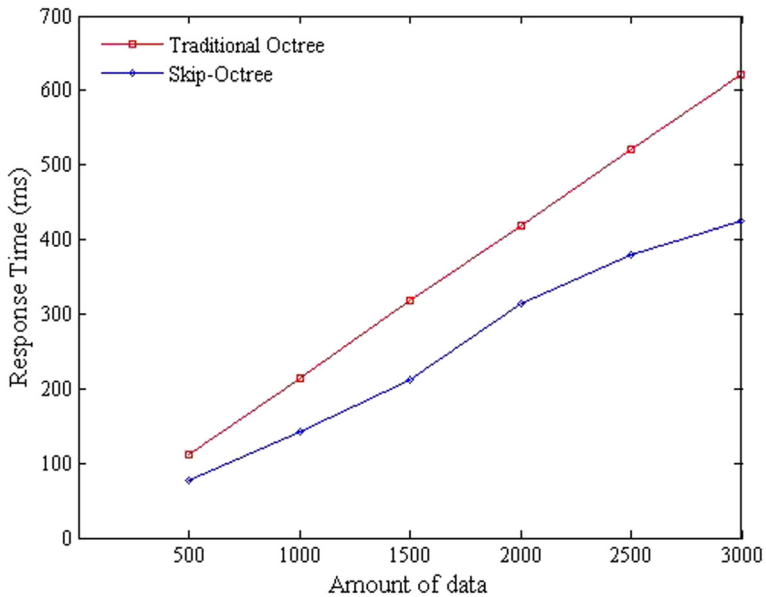

Fig. 6 Performance of the deletion process

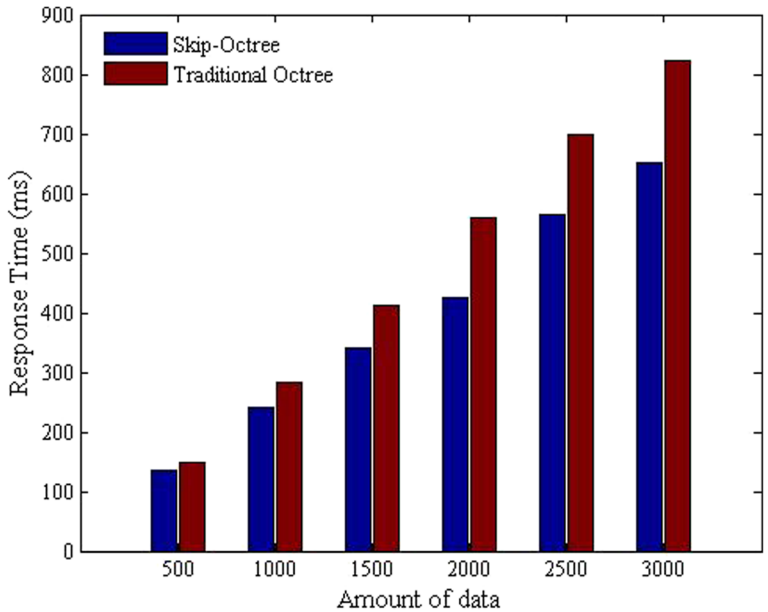

Fig. 7 Performance of the insertion process

The Skip-Octree is a double-layer cloud index that has more complex structure than a traditional single-layer index. In this experiment, the performance of a double-layer Skip-Octree was evaluated. As a comparison, a Skip-Octree having only an upper layer, a Skip-Octree having only a lower layer, and the traditional Octree were tested under the same conditions. Figure 9 shows 16 local data servers present in the cloud storage system, and the amount of data increases from 1000 to 50000. Our test queried 500 sets of data within the whole dataset. The double-layer Skip-Octree is the most efficient among them. The Skip-Octree having only an upper layer consumes more time than the traditional distributed Octree. This is because the upper layer index is built only of a global Skip-Octree, and the index is too deep when the amount of data is large. Although the traditional Octree is stored in multiple servers, its query speed is faster than that of Skip-Octree having only an upper layer.

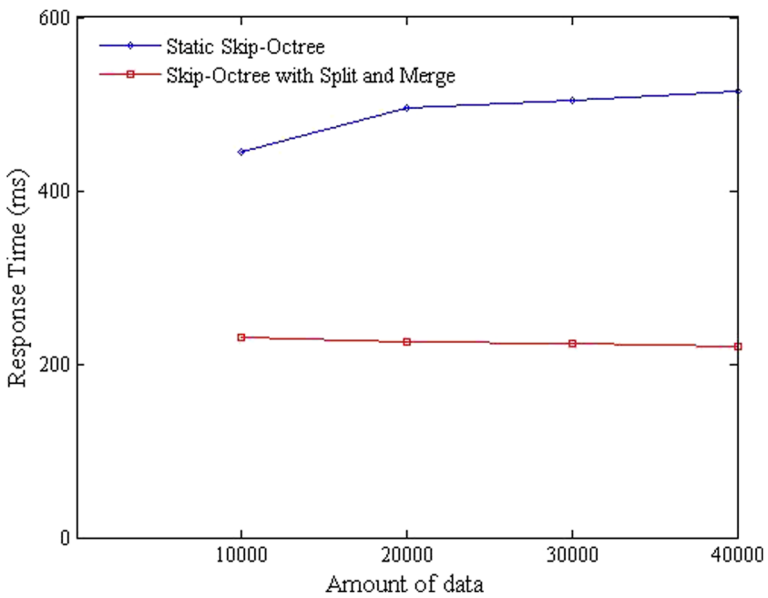

Fig. 8 Performance of the split and merge operations

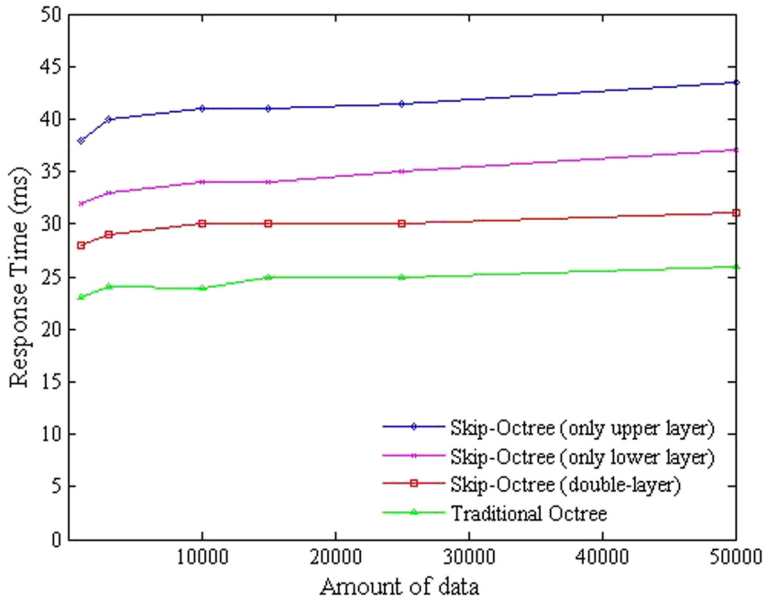

Fig. 9 Availability testing for a double-layer cloud index

Conclusion

This study provided a new multidimensional data index framework, called Skip-Octree, which combines the best features of two well-known data structures: Octree and skip lists. Some index operating algorithms that include multidimensional range querying, data insertion and deletion, and index splitting and merging were also proposed in this study. The experimental results show that our Skip-Octree is efficient. However, because a cloud storage system usually supports both transactional and data analysis operations simultaneously, frequent updates will conflict with data queries, thereby reducing data query efficiency. Means to enhancing data consistency in order to ensure query efficiency is a topic of future research.

Acknowledgments

This work is supported by the National Nature Science Foundation of China (61363021, 61540061), Science Research Foundation of Yunnan Province Education Department (2014Y013) and the Youth Program of Applied Basic Research Programs in Yunnan Province (2012FD0047).

Authors' contributions

Author JH provided the idea of this paper, carefully designed the framework, and drafts the manuscript. Author YD and YZ performed the experiments and presented performance analysis. Author YW and WZ reviewed and edited the manuscript. All authors read and approved the final manuscript.

Competing interests

The authors declare that they have no other competing interests.

Author details

[1]School of Computer Science and Engineering, University of Electronic Science and Technology of China, Chengdu, Sichuan 611731, P.R. China. [2]Research Center of Western Yunnan Development, Yunnan University, Kunming, Yunnan 650091, P.R. China. [3]National Pilot School of Software, Yunnan University, Kunming, Yunnan 650091, P.R. China.

Received: 30 January 2016 Accepted: 12 July 2016
Published online: 15 July 2016

References

1. Armbrust M, Fox A, Griffith R et al. (2010) A view of cloud computing. Commun ACM 53(4):50–58
2. Chauvel F, Song H, Ferry N et al. (2015) Evaluating robustness of cloud-based systems. J Cloud Comput 4(1):1–17
3. Hashem IAT, Yaqoob I, Anuar NB et al. (2015) The rise of big data on cloud computing: Review and open research issues. Inf Syst 47:98–115
4. Yang Y (2015) Attribute-based data retrieval with semantic keyword search for e-health cloud. J Cloud Comput 4(1):1–6
5. Kao B, Lee SD, Lee FKF et al. (2010) Clustering uncertain data using voronoi diagrams and r-tree index. IEEE Trans Knowl Data Eng 22(9):1219–1233
6. Nandi U, Mandal JK (2013) Efficiency and Capability of Fractal Image Compression With Adaptive Quardtree Partitioning. Int J Multimedia Its Appl 5(4):53–66
7. Eppstein D, Goodrich MT, Sun JZ (1997) The Skip Quadtree: A Simple Dynamic Data Structure for Multidimensional Data. J Comput 20(9):849–854
8. Zeng M, Zhao F, Zheng J et al. (2013) Octree-based fusion for realtime 3D reconstruction. Graph Model 75(3):126–136
9. Labrinidis A, Jagadish HV (2012) Challenges and opportunities with big data. Proc VLDB Endowment 5(12):2032–2033
10. Chang F, Dean J, Ghemawat S et al. (2008) Bigtable: A distributed storage system for structured data. ACM Trans Comput Syst 26(2):1–26
11. Ghemawat S, Gobioff H, Leung S-T (2003) The Google file system. ACM SIGOPS Operating Syst Rev 37(5):213–223
12. The Apache Software Foundation: Hadoop. http://hadoop.apache.org/. Accessed 22 June 2016.
13. Decandia G, Hastorun D, Jampani M et al. (2007) Dynamo: Amazon's highly available key-value store. In: The 21st ACM Symposium on Operating Systems Principles. ACM Press, New York, pp 205–220
14. Laskhmam A, Malik P (2010) Cassandra: a decentralized structured storage system. ACM SIGOPS Operating Syst Rev 44(2):35–40
15. Dittrich J, Quian'e-Ruiz J-A, Jindal A, Kargin Y et al. (2010) Hadoop++: Making a Yellow Elephant Run Like a Cheetah (Without It Even Noticing). Proc VLDB Endowment 3(1):518–529
16. Lu P, Chen G, Ooi BC et al. (2014) ScalaGiST: scalable generalized search trees for mapreduce systems [innovative systems paper]. Proc VLDB Endowment 7(14):1797–1808
17. Aguilera MK, Golab W, Shah MA (2008) A practical scalable distributed b-tree. Proc VLDB Endowment 1(1):598–609

18. Wu S, Jiang D, Ooi BC et al. (2010) Efficient B-tree Based Indexing for Cloud Data Processing. Proc VLDB Endowment 3(1):1207–1218
19. Zhou W, Lu J, Luan Z et al. (2014) SNB-index: a SkipNet and B+ tree based auxiliary Cloud index. Clust Comput 17(2):453–462
20. Wang J, Wu S, Gao H et al. (2010) Indexing multi-dimensional data in a cloud. In: Proceedings of the 2010 ACM SIGMOD International Conference on Management of data. Indianapolis, Indiana, USA., pp 591–602
21. Ratnasamy S, Francis P, Handley M et al. (2002) A scalable content-addressable network. ACM Sigcomm Comput Comm Rev 355(4):161–172
22. Cheng CL, Sun CJ, Xu XL et al. (2014) A Multi-dimensional Index Structure Based on Improved VA-file and CAN in the Cloud. Int J Autom Comput 11(1):109–117
23. Lu P, Wu S, Shou L et al. (2013) An efficient and compact indexing scheme for large-scale data store. In: 2013 IEEE 29th International Conference on Data Engineering (ICDE)., pp 326–337
24. Haber E, Schwarzbach C (2014) Parallel inversion of large-scale airborne time-domain electromagnetic data with multiple OcTree meshes. Inverse Problems 30(5):055011
25. Vo AV, Truong-Hong L, Laefer DF et al. (2015) Octree-based region growing for point cloud segmentation. ISPRS J Photogramm Remote Sens 104:88–100
26. Meagher D (2012) High-speed image generation of complex solid objects using Octree encoding., USPO, Retrieved 20 September 2012
27. Weijie GU, Jishui WANG, Hao SHI et al. (2011) Research on a Hybrid Spatial Index Structure. J Comput Info Syst 7(11):3972–3978
28. Aluru S, Sevilgen FE (1999) Dynamic compressed hyperoctrees with application to the N-body problem. In: *Proc.* In: 19th Conf. Found. Softw. Tech. Theoret. Comput. Sci., 1738., pp 21–33
29. Gaede V, Gunther O (1998) Multidimensional access methods. ACM Comput Surv 30(2):170–231
30. Xie Z, Cai Q, Jagadish H V, et al. (2016) PI: a Parallel in-memory skip list based Index. arXiv preprint arXiv:1601.00159.
31. Paulevé L, Jégou H, Amsaleg L (2010) Locality sensitive hashing: A comparison of hash function types and querying mechanisms. Pattern Recogn Lett 31(11):1348–1358
32. Montresor A, Jelasity M (2009) PeerSim: A scalable P2P simulator. In: IEEE 9th International Conference on Peer-to-Peer Computing. IEEE, New York, pp 99–100