Indexing Multi-dimensional Data in Modular Data Centers

Libo Gao, Yatao Zhang, Xiaofeng Gao(✉), and Guihai Chen

Shanghai Key Laboratory of Scalable Computing and Systems,
Department of Computer Science and Engineering,
Shanghai Jiao Tong University, Shanghai 200240, China
{nosrepus,confidentao}@gmail.com, {gao-xf,gchen}@cs.sjtu.edu.cn

Abstract. Providing efficient multi-dimensional indexing is critically important to improve the overall performance of the cloud storage system. To achieve efficient querying service, the indexing scheme should guarantee lower routing cost and less false positive. In this paper, we propose RB-Index, a distributed multi-dimensional indexing scheme in modular data centers with Bcube topology. RB-Index is a two-layer indexing scheme, which integrates Bcube-based routing protocol and R-tree-based indexing technology. In its lower layer, each server in the network indexes the local data with R-tree, while in the upper layer the global index is distributed across different servers in the network. Based on the characteristics of Bcube, we build several indexing spaces and propose the way to map servers into the indexing spaces. The dimension of these indexing spaces are dynamically selected according to both the data distribution and the query habit. Index construction and query algorithms are also introduced. We simulate a three-level Bcube to evaluate the efficiency of our indexing scheme and compare the performance of RB-Index with RT-CAN, a similar design in P2P network.

Keywords: Multi-dimensional data · Distributed index · Modular data center

1 Introduction

Recent years have witnessed an increasing need of cloud storage systems due to the emergence of modern data-intensive applications. Various cloud storage systems are put forward to meet requirements like scalability, manageability and low latency. Examples of such systems include BigTable [4], DynamoDB [6],

This work has been supported in part by the National Natural Science Foundation of China (Grant number 61202024, 61472252, 61133006, 61422208), China 973 project (2012CB316200), the Natural Science Foundation of Shanghai (Grant No. 12ZR1445000), Shanghai Educational Development Foundation (Chenguang Grant No. 12CG09), Shanghai Pujiang Program 13PJ1403900, and in part by Jiangsu Future Network Research Project No. BY2013095-1-10 and CCF-Tencent Open Fund.

© Springer International Publishing Switzerland 2015
Q. Chen et al. (Eds.): DEXA 2015, Part II, LNCS 9262, pp. 304–319, 2015.
DOI: 10.1007/978-3-319-22852-5_26

Cassandra [9], HyperDex [11], etc. One of the requirements of these systems is to support large-scale analytical jobs and high concurrent OLTP queries. To achieve this, many works [1,5,7,12–14,16] have been devoted to designing a new indexing scheme and data management system. A typical indexing scheme is RT-CAN [7]. RT-CAN is a two-layer indexing scheme integrating R-tree structure and CAN-based routing protocol. Its higher-layer index, called global index, is built upon the local index. In RT-CAN, each server in the network builds an R-tree as the local index, then selects a set of R-tree nodes and publishes them into the global index. At the same time, a multi-dimensional indexing space is constructed. Each server in the network maintains a zone and stores the global index in its responsible zone. Consequently, the global index, composed of R-tree nodes from different servers, works as an overview index and is distributed across servers. When a query is given, we first check the global index to determine which servers may contain the required data and then search the local R-trees on the related servers to get the result.

The design of the distributed two-layer index makes RT-CAN efficient and robust, however, like most of other works, RT-CAN is conducted in P2P network, rather than in data centers. Unlike the P2P network, of which nodes may scatter widely in the real world and the latency among nodes may vary greatly, the data center interconnects a great number of servers via a specific Data Center Network (DCN) and reaches high-reliability, scalability and regularity in its structure. A specific type of data center is Modular Data Center (MDC) [8,18,19], in which thousands of servers are interconnected via switches and then packed into a 20- or 40-feet shipping container. It can be rapidly employed anywhere to meet different requirements of applications. As the MDC gains its popularity, it brings new challenges for researchers to design a new efficient indexing scheme to support query processing for it. There are two main challenges. First, the indexing scheme should utilize the MDC's architecture topology to improve the performance. Second, since in modern data-intensive applications, multi-dimensional data are commonly processed, like photos, videos, etc., the indexing scheme should support multi-dimensional indexing. Although RT-CAN supports multi-dimensional data indexing, it requires that the dimension of stored data cannot surpass the dimension of the overlay network, which means that RT-CAN is not scalable in terms of data dimension. Therefore, to index higher-dimensional data, the original overlay network should be expanded and more servers should be added to build the index, which costs a lot. Moreover, the high-dimensional space is usually sparse. Directly building a high-dimensional indexing space is not efficient.

In this paper, we try to transplant the two-layer indexing scheme on MDC and the new indexing scheme should be scalable in terms of dimension. We present our RB-Index, a distributed indexing scheme for multi-dimensional query processing in MDC with Bcube [2] topology. Bcube is a server-centric architecture for MDC. Lots of mini-switches and links are used to form its hyperspace-liked network structure, which results in some attracting features like low-diameter, high-bandwidth and fault tolerance. In RB-Index, we adopt

R-tree as the local index. For the global index, instead of setting up one indexing space, we set up several distinct indexing spaces based on the feature of Bcube topology and build different global index for each space. The dimension of these global index is selected according to both data distribution and query habit. We design the rule to map servers into indexing spaces and publish R-tree nodes into the global index. A cost model is proposed to select the least-cost set of R-tree nodes to publish. We also discuss the query processing algorithms. In the end, we simulate a three-level Bcube and compare the performance of RB-Index with RT-CAN.

The contribution of this paper is threefold: (1) A multi-dimensional indexing scheme is proposed for MDC with Bcube topology. We design a strategy to build several indexing spaces, on which different global indexes with different dimension are set up. (2) We propose the rule to select the indexed dimension according to both the data distribution and query habit. The indexed dimension can be replaced along with the change of query habit, so theoretically RB-Index can support any-dimensional data indexing. (3) We simulate a three-level Bcube and evaluate the efficiency of RB-Index.

The rest of paper is organized as follows. Section 2 discusses the related work. Section 3 gives an overview of RB-Index. Section 4 introduces the global index construction, including the dimension selection, mapping scheme and the cost model. In Sect. 5, we present the query algorithms including point query, range query and KNN query. Section 6 illustrates the simulation and proves the efficiency of our indexing scheme. Finally, Sect. 7 gives the conclusion.

2 Related Work

Efficient indexing scheme is crucial for fast data retrieval in cloud infrastructures. Due to the unprecedented scale of data, extending the traditional indexing technologies, like B-tree and R-tree, has drawn the attention in distributed environment. One part of these efforts can be classified as the primary index. The primary index usually adopts key-value based indexing strategies. Given a key, the index will efficiently locate the objects linked with the key by utilizing a range index like a distributed B^+-tree [10] or by a multidimensional index like SD-Rtree [3]. However, these indexes do not support query on other data attributes. In real world, data like videos and photos usually have more than one keys. Therefore, supporting efficient secondary index becomes a very useful feature for many applications.

To handle these application cases, a new indexing scheme with two-layer structure, called RT-CAN, was proposed in [7]. In RT-CAN, servers are organized into an overlay network and the overlay network forms the global indexing space. The dimension of the overlay network is determined by data dimension, so after the data being mapped into the indexing space, every data attribute can be queried with the index. There are also some similar designs in [12,13]. All of these works are implemented in P2P network, until recently, several research work [5,14,16] implemented such a design in data centers. One problem of these

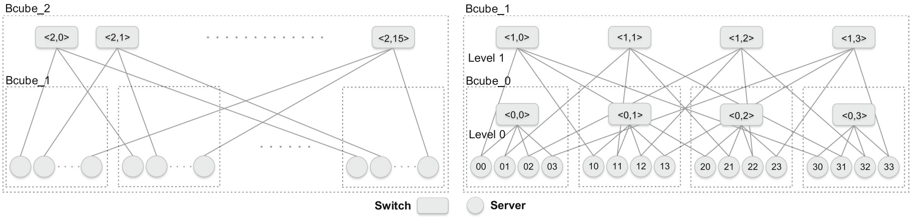

Fig. 1. A $Bcube_2$ with n=4 (left) is constructed from 4 $Bcube_1$ (right) and 16 4-port switches.

works is that the index is not scalable with respect to dimension. Take RT-CAN as an example, the scale of the overlay network determines the query efficiency under different number of dimension. When the dimension of stored data increases, the overlay network should also be extended and more server nodes should be added to guarantee the efficiency. Thus, this introduces great cost, especially in data centers where the physical layout and scale is usually fixed.

To address this problem, we want to build a scalable two-layer indexing scheme. Our index is designed for MDC. The scale of MDC is usually limited with the size of a shipping container. This property pushes such problem of dimension-scalability into an extreme condition and solving this problem becomes urgent and significant.

The Modular Data Center is proposed to meet the growing flexibility of customers. Many companies has already deployed the MDCs. Sun first presented an MDC in 2006 with up to 2240 servers and 3PB storage. Later, HP and IBM employed their own MDCs. The design of the MDC architecture is driven by application needs. A typical architecture is called Bcube [2].

Bcube is a server-oriented network for container-based, MDCs. By using much mini-switches and links, Bcube accelerates one-to-x traffic and provides high network capacity for all-to-all traffic. The construction is recursively-defined. $Bcube_0$ is the smallest module that consists of n servers connecting to an n-port switch. A high-level $Bcube_k$ employs $Bcube_{k-1}$ as a unit cluster and connects n such clusters with n^k n-port switches. Figure 1 illustrates an example of $Bcube_2$ with $n = 4$, which consists of 64 servers. The servers and switches are labeled in a string: $a_k a_{k-1} \dots a_0 (a_i \in [0, n-1], \ i \in [0, k])$, where a_i represents port number of the switch in the ith level to which the server connected.

3 System Overview

In this section, we first define the indexing space for $Bcube_k$. Then we design the rule to mapping servers in the $Bcube_k$ into its corresponding indexing space and assign the potential index range. Finally, we give an overview of the RB-Index.

3.1 Indexing Space

As is mentioned, Bcube is a recursively defined structure and a $Bcube_k$ is constructed by n $Bcube_{k-1}$s. This property guarantees that a $Bcube_k$ contains n^k $Bcube_0$s and n^{k+1} servers. To efficiently index multi-dimensional data, we construct a $(k+1)$-dimension indexing space for a $Bcube_k$. All the data mapped into the space is normalized, so the range on each dimension is the same and the indexing space is actually a hyperspace. We set the dimension of the indexing space to the level of $Bcube_k$, hoping that the servers in the network can be mapped into the indexing space uniformly and each one can take the responsibility for the indexes in a specific zone. We will talk about the mapping scheme later and here we want to give a closer view of the indexing space by sharing an example.

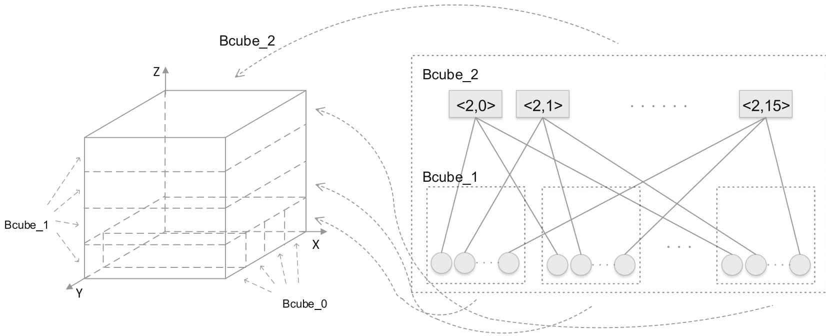

Fig. 2. The structure of indexing space for $Bcube_2$ with n=4.

Figure 2 gives an example of the indexing space for $Bcube_2$. Since the level of $Bcube_2$ is three, the indexing space is a cube. Here $n = 4$, so four $Bcube_1 s$ are arranged along with z-coordinate to form the cube, while each $Bcube_1$ contains four $Bcube_0 s$. The servers in $Bcube_0$ are arranged along with x-coordinate. The construction of a higher dimension indexing space takes the similar way. First, n servers in each $Bcube_0$ are placed uniformly along with the first dimension, then the process continues iteratively until finally n $Bcube_{k-1}$ are arranged uniformly along with the $(k+1)$th dimension.

Through this construction, the indexing space has some features. Referring to Bcube's label pattern, we know if two servers' labels have exactly one digit difference, then these two servers are adjacent and the path length is one. More generally, the path length between any two servers is not longer than $k+1$. Taking the indexing space in Fig. 2 as an example, the path length of any two servers, which are in line with the same coordinate, is one; the path length of any two servers on a plane, which is parallel to a coordinate plane, is not longer than two.

3.2 Mapping Scheme and Potential Index Range

Previously, we introduced the indexing space for $Bcube_k$. We set the dimension of the indexing space to the level of $Bcube_k$, so all n^{k+1} servers in the network can be mapped uniformly into the indexing space. Each server will hold a zone in the indexing space and take the responsibility for the indexes in that zone. We call that zone the potential index range. When building the global index, each server selects a set of R-tree nodes from its local R-tree, then these R-tree nodes will be published to the servers of which potential index range intersects with the node range. Here, we introduce the rule to allocate the potential index range to each server.

Suppose the indexing space is bounded by $\mathbf{B}=(B_0, B_1 \ldots B_k)$, where B_i is $[l_i, l_i + \gamma], i \in [0, k], \gamma \in \mathbb{R}^+$, the potential range of server t is $pir(t)$. Here, we denote a server t with a label string $a_k a_{k-1} \ldots a_0$ ($a_i \in [0, n-1], i \in [0, k]$). Equivalently, t equals to $\sum_{i=0}^{k} a_i n^i$. Then $pir(t)$ can be calculated by the following function.

$$\begin{aligned} pir(t) &= pir(a_k a_{k-1} \ldots a_0) \\ &= ([l_0 + a_0\frac{\gamma}{n}, l_0 + (a_0+1)\frac{\gamma}{n}], \ldots, [l_k + a_k\frac{\gamma}{n}, l_k + (a_k+1)\frac{\gamma}{n}]). \end{aligned} \quad (1)$$

A close observation reveals that in $Bcube_k$, the labels of servers in the same $Bcube_0$ only have the first bit, a_0, different. According to Eq. (1), these servers will be arranged along with the first coordinate. Figure 3 presents the arrangement of servers in the $3D$ space. We assume that l_0, l_1 and l_2 all equal to zero. Each of four $Bcube_1$s has 16 servers which are placed in a plain parallel to the xy-coordinate plain. Every server occupies a $3D$ zone. For example, $pir(000) = ([0, \frac{1}{4}], [0, \frac{1}{4}], [0, \frac{1}{4}])$, while $pir(133) = ([\frac{3}{4}, 1], [\frac{3}{4}, 1], [0, \frac{1}{4}])$

3.3 Two-Layer Index Architecture

In a distributed storage system, data are randomly distributed over servers. Each server builds a local R-tree for local data retrieval. When given a query, instead of searching all the local R-trees, the global index is given to guide the query to the related servers, which significantly reduces the query region. One

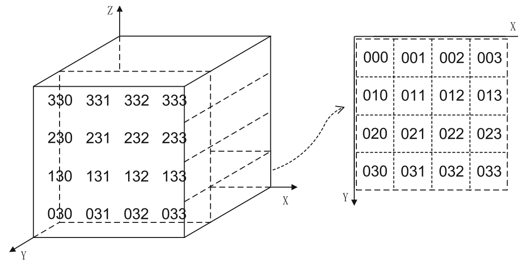

Fig. 3. Potential index range of servers

way to achieve this goal is to build a centralized global index. Queries are first sent to several master servers which possess the global index, then, according to the results, they are broadcast to the related servers for local queries. However, this strategy may face problems when the number of queries increases. Several master servers will be the bottleneck of the query performance. An alternative is to distribute global indexes to all servers and each server is responsible for a portion of global index in its potential index range. By balancing the workload, the performance of query is guaranteed. However, both of the indexing schemes require that the dimension of the indexed data cannot surpass the dimension of the indexing space. Therefore, to index the high-dimensional data, we need to build a high-dimensional indexing space. In this paper, we propose RB-Index, which is a distributed two-layer indexing scheme supporting scalability in terms of data dimension.

Previously, we have introduced the indexing space formed by $Bcube_k$ and the dimension of the space is limited to $k+1$. Usually, k is not a big number and the dimension of data may be far more than $k+1$. If we try to reduce the dimension of data and map them to a low-dimensional indexing space, when users query on some unindexed dimension, all the servers need to be searched to get the result. In RB-index, rather than building only one global indexing space, we build $n+1$ indexing spaces, containing one $(k+1)$-dimension indexing space for $Bcube_k$ and n k-dimension indexing spaces for n $Bcube_{k-1}s$. The global indexes built in these indexing spaces are classified into two types, **the main global index** and **the subsidiary global index**. The main global index is the index in the $(k+1)$-indexing space. Among all the global indexes, the main global index has the highest dimension, $k+1$, and is distributed across all the servers in the network. The subsidiary index is built in the k-dimension indexing space. In each of n $Bcube_{k-1}$s in $Bcube_k$, we build a subsidiary index, which is distributed only across servers in that $Bcube_{k-1}$. Therefore, there are n subsidiary global indexes in all. Each server t in the network is mapped into exactly two indexing spaces and responsible for two potential index ranges, $pir(t)$ for main global index and $pir'(t)$ for subsidiary global index. Theoretically, each time RB-Index can support $((n+1)k+1)$-dimension indexing. If the data dimension is less than $((n+1)k+1)$, extra zero vector can be appended to the data. If the data dimension exceeds $((n+1)k+1)$, we first pick $((n+1)k+1)$ dimensions from data and gradually rebuild the subsidiary indexes according to the query habit by replacing the indexed dimension.

For clarity, we summarize symbols with their meaning in Table 1. Some of them will be used in the description of the rest of this paper.

4 Index Construction

In this section, we first introduce the adaptive rule to determine the dimension of the indexing space, then we present the way to publish the R-tree nodes into the indexing space. Finally, a cost model is introduced to select a proper set of R-tree nodes from a local R-tree.

Table 1. Symbol description

Sym	Description	Sym	Description
t	Code for servers	k	The level of Bcube
$\mathbf{N_t}$	t's R-tree node set for publishing	n	Number of switch ports
$pir(t)$	t's potential index range in main global index	$pir'(t)$	t's potential index range in subsidiary global index

4.1 Dimension Selection

RB-Index builds several indexing spaces to support high-dimensional data indexing. The dimension of these indexing spaces need to be selected properly to face all kinds of query. An adaptive method is to select the dimension according to the query habit. It is common that during a certain period of time, users may tend to query on some certain combinations of the dimension. We build the index for them. Once the query habit changes, we first judge whether the new query habit exists, then we rebuild the indexing space based on the new query habit. There is no denying that rebuilding the indexing space will introduce a great cost, however, the primary aspect we consider here is that the gained efficiency should surpass the cost. In RB-Index, the dimension of the main global index is fixed once being established, while the dimension of the subsidiary global index will be replaced along with the change of query habit.

As is mentioned, the main global index has $k+1$ dimension and is distributed across all the servers in the network. We want to distribute the main global indexes as uniformly as possible to avoid the hot spots during query, so principle component analysis (PCA) [17] is adopted to select dimension for the main global index. PCA is a traditional dimension-reduction method. By analyzing the distribution of data, PCA always picks up top x dimension on which data are most uniformly distributed. Since the data are randomly distributed across the servers in the network, we use distributed PCA [15] to get the first $k+1$ dimension.

In $Bcube_k$ with n, there are n $Bcube_{k-1}$s, which means that there are n subsidiary global indexes. For each subsidiary global index, we select k dimension according to the query habit. There are some restrictions on selecting dimension. First, the dimension of the subsidiary global index cannot totally be contained in the global index. Second, the dimension of any two subsidiary global indexes cannot be exactly the same. These two requirements are raised to avoid redundancy among the global indexes. Initially, we list the top n frequently occurred query patterns. Here, the query pattern is defined as a group of k dimensions that are usually queried together. Each subsidiary global index is initialized with one of these query patterns. As more and more queries happen, we use LRU-algorithm, a caching algorithm, to choose one subsidiary index to be rebuilt. The subsidiary index that is least recently hit will be required to be rebuilt. Here *a hit* means the global index contains some dimension of a given query. We also assume that only when a query does not hit any global index, including

one main global index and n subsidiary global indexes, the system will trigger rebuilding the least recently hit subsidiary index. In practice, to avoid frequent index rebuilding, the frequence of a query pattern should exceed a threshold in a time quantum before triggering the rebuilding of indexing space. Otherwise, no rebuilding will happen.

4.2 Publishing Scheme

As is discussed above, every server builds a local R-tree to accelerate local data retrieval. To build the global indexes, each server adaptively selects a set of R-tree nodes, $\mathbf{N_t} = \{N_t^1, \ldots N_t^n\}$, from its local R-tree and publishes them into the global index.

We use the same mapping scheme as in [7]. Since the publishing scheme of both the main global index and the subsidiary global index is similar, we take the main global index as an example. The format of the published R-tree node is (ip, mbr), where ip records the physical address of the server which publishes the node and mbr gives the range of the R-tree node. For each selected R-tree node, the center and radius are two criteria used for mapping. The center decides the position where the R-tree node is mapped, while the radius decides whether the node is mapped to one server or several servers. A threshold, R_{max}, is set to be compared with the radius. The detailed process of the mapping scheme is as follows: given an R-tree node, we calculate its center and radius. First, the node is directly mapped to the server whose potential index range contains the center. Then the radius is compared with R_{max}. If the radius is larger than R_{max}, then the node will also be mapped to the servers whose potential index range intersects with the R-tree node range.

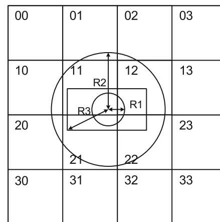

Fig. 4. Mapping scheme in $2D$ place

Figure 4 shows an example in $2D$ space. Suppose that the radius of the indexed R-tree node N is R_3 and R_{max} equals R_2, then N will only be mapped to $\{11\}$. If we set R_{max} to R_1, then since R_3 is larger than R_1, N will be mapped to servers $\{11, 12, 21, 22\}$. In higher-dimensional space, the mapping scheme works in the same way.

4.3 Cost Model

In RB-Index, each server chooses a set of nodes from its local R-tree and publishes them into the global index. [7] raises two properties as the basic requirements in selecting the local R-tree node set: index completeness and unique index. These two criteria guarantee that the selected set of R-tree nodes covers the whole local data range with the least redundance. Based on these requirements, a cost model is introduced here to optimize the index selection further. We combine the cost model raised in [7,13] and take advantage of both the cost models.

In the rest of the paper, we use hop number as the metric to evaluate the routing cost. A hop refers to the trip from one server to its neighbor. Since servers in Bcube are connected in a quite close physical distance, the communication delay between any two servers is approximately equal. Thus, using hop number to evaluate the real routing cost is reasonable. The cost model considers two aspects to value the cost of publishing a local R-tree node $C(N)$: index maintenance cost $C_m(N)$ and query process cost $C_q(N)$,

$$C(N) = C_m(N) + C_q(N) \tag{2}$$

The maintenance cost refers to the cost of essential node update operations: split and merge. Once the published node is split or merged, additional routing cost is caused by republishing the node. Given that the average routing cost between any server in $Bcube_k$ is $k+1$, the routing cost of splitting includes deleting the original node and publishing two new nodes. Similarly, the routing cost of merging includes deleting two old nodes and publishing a new node. Both of the two operations cost $3(k+1)$ approximately. Assume that the probabilities of splitting and merging for node N are $p_{split}(N)$ and $p_{merge}(N)$, we can get Eq. (3) as follows:

$$C_m(N) = 3(k+1)(p_{split}(N) + p_{merge}(N)) \tag{3}$$

Now the problem becomes how to estimate the value of $p_{split}(N)$ and $p_{merge}(N)$. In [7], a two-state markov chain model is applied on each R-tree node to calculate its $p_{split}(N)$ and $p_{merge}(N)$. However, this takes a great amount of computation. Here, we only use this method to calculate $p_{split}(N)$ and $p_{merge}(N)$ of the leaf node. For non-leaf nodes, we use the method in [13]. Suppose $p_1(N)$ is the probability of insertion in node N and $p_2(N)$ is the probability of deletion in node N. The relationship between p_1, p_2, p_{split} and p_{merge} can be formulated as:

$$p_{split} = \frac{(\frac{p_2}{p_1})^{\frac{3m}{2}} - (\frac{p_2}{p_1})^m}{(\frac{p_2}{p_1})^{2m} - (\frac{p_2}{p_1})^m}, \quad p_{merge} = \frac{(\frac{p_2}{p_1})^{2m} - (\frac{p_2}{p_1})^{\frac{3m}{2}}}{(\frac{p_2}{p_1})^{2m} - (\frac{p_2}{p_1})^m} \tag{4}$$

For any non-leaf node N, suppose its children are $c_1, c_2, \ldots, c_i$, then p_1 and p_2 can be calculated by the Eq. (5):

$$p_1(N) = \prod_{j=1}^{i}(1 - p_{split}(c_j)), \quad p_2(N) = \prod_{j=1}^{i}(1 - p_{merge}(c_j)) \tag{5}$$

Therefore, once we get $p_{split}(N)$ and $p_{merge}(N)$ of leaf nodes, we can get the update probabilities of the internal node iteratively.

The second cost is query processing cost. We mainly consider false positive. Due to the overlay between R-tree nodes, it is common for the global index to guide the query to servers which actually does not contain the needed datum. Suppose $R(N)$ represents the range of node N and $D(N)$ represents the range of data in node N, then the probability of false positives p_{fp} can be simply defined in Eq. (6). And the average routing cost of query processing can be calculated in Eq. (7).

$$p_{fp}(N) = \frac{R(N) \cap D(N)}{R(N)} \tag{6}$$

$$C_q(N) = (k+1)p_{fp}(N) \tag{7}$$

After getting the maintenance cost (Eq. 3) and query processing cost (Eq. 7), we can get the cost of a node N:

$$C(N) = (k+1)(3p_{split}(N) + 3p_{merge}(N) + p_{fp}(N)) \tag{8}$$

5 Query Processing

In this section, we show how the global index can be applied to efficiently process high-dimensional data queries.

5.1 Point Query

Before we talk about point query, we make some explanations: here, point query refers to full-dimensional point query, rather than partial-dimension point query, since partial point query can be regarded as range query, which will be introduced later. Point query can be processed in either one of the global indexes. Usually it is processed in the main global index since the dimension selection of the main global index determines that data are most uniformly distributed in that index and the main indexing space is better grained with more responsible servers. Given a point p=(x_0, ...,x_d), the process of point query $Q(p)$ can be divided into two phases:

In the first phase, we forward the query to server t whose potential index range contains the point. Then we generate a circle centered at point p with radius R_{max}. All the servers whose potential index range intersects with the circle should be searched. We take advantage of multi-pathes between servers in Bcube and forward the query in parallel. We check the main global index buffered in these servers and get related R-tree nodes whose range contains the point. In the second phase, according to the result R-tree nodes, we continue the query on local R-trees in the corresponding physical servers. For point query processing, suppose the circle covers M servers and the average path length between any two server is $(k+1)$, then the routing cost of forwarding the query in the global index is $O((k+1)*M)$. Besides, forwarding query to the related servers for local search brings additional cost. If the overlay of R-tree is properly controlled, this process will have a tiny cost.

5.2 Range Query

Suppose the dimension of the stored data is d. The range query can be either the d-dimension range query or partial-specific range query. We denote a range query as $Q(range)$, where $range$=$([l_{d_1}, u_{d_1}] \dots [l_{d_t}, u_{d_t}])$, $\{d_1, d_2 \dots d_t\}$ is a subset of $\{1 \dots d\}$. The range query processing can be divided into three phases.

In the first phase, we choose one global index which matches the query most. Here, matching means having the most number of the same dimensions with the query dimension. We prefer to choose the global index which guides query to lest number of servers. If the dimension of the range query is not indexed, then we will check whether this query happens frequently in the past time. If so, one subsidiary global index which is the least recently used will be rebuilt. Otherwise, we will omit the following phases and directly broadcast the range query to all the servers and query on local R-trees. In the second phase, first, we send the query to servers whose potential index range intersects with the query range. Then due to the publishing scheme, the query should also be forwarded to some other servers to get the full result. We calculate the center of the range query and its radius. In the global index space, if only part of dimension of the query is indexed, then the center maybe a line or a plain. The server t is searched if and only if $|t.pir.center - range.center| < radius + R_{max}$. The buffered R-tree nodes in these servers will be searched. In the third phase, according to result R-tree nodes, the query is forwarded to the related physical servers to continue searching on their local R-trees.

The cost of range query processing is related to the radius of the range. As we can see from the process, if the radius is larger, more servers will be searched to get the buffered R-tree nodes. The worst case is broadcasting the query to all the servers. However, this case will not always happen. Thus, compared with the cost of broadcasting the query to all servers, the cost of range query processing with the help of RB-Index is reduced significantly.

5.3 KNN Query

We denote the KNN query $Q(p, K)$, which requires the K nearest neighbors for the point p. Again, we choose the main global index. The reason is the same as point query. We first generate a circle C centered at p with a given radius R_{init}. R_{init} is set according to the data distribution and the value K. During the process of the query, if the result of range query $Q(C)$ contains K nearest data, then the KNN query complete. Otherwise, we extend the circle radius with δ until the result of range query returns enough nearest data. The cost of KNN query is related to the value K. With higher K value, bigger range needs to be searched, which reduces the query efficiency.

6 Performance Evaluation

In this section, we simulate a $Bcube_2$ with $n = 4$ and evaluate the performance of RB-Index on it. The data are generated and distributed randomly across the

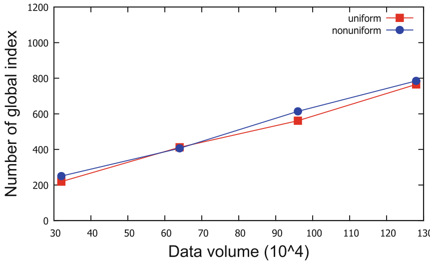

Fig. 5. Average number of global index in one server

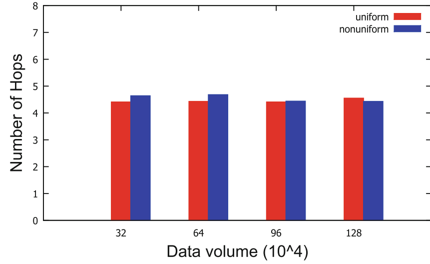

Fig. 6. Point query

servers in the Bcube. We generate two different datasets to evaluate the performance of RB-Index. In the uniform dataset, we generate 320000 to 1280000 data. These data have 11 attributes and values of attributes are uniformly distributed between $(0, 1)$. In the nonuniform dataset, data are generated following the $80/20$ rule, which means 80 percentage of data are concentrated in 20 percentage of the space. For both of the datasets, we disseminate data to servers and keep every server roughly maintain the same number of data. A $Bcube_2$ with $n = 4$ contains 64 servers and each server builds a local R-tree indexing local data. For each internal R-tree node, the maximum number of entries is set to 10. Initially, we choose the last but one level from R-trees to publish into global indexes, since these nodes are not frequently updated and have the modest false positive.

In the following simulation, we calculate the number of hops to evaluate the query performance of RB-Index. As is mentioned, in MDC the communication delay between any two servers is approximately equal. Thus, using hop number to evaluate the real routing cost is reasonable. We mainly focus on point query and range query since in RB-Index, KNN query can be achieved by several range queries. We also build the RT-CAN [7] in $Bcube_2$ as a reference object. Due to the algorithm of point query, the point query processing of RT-CAN and RB-Index is actually same, so we only compare the range query performance of RT-CAN with RB-Index.

Before evaluating query performance, we first estimate the space cost of the global index under different data volume. We record the number of buffered R-tree nodes in each server and calculate the average number of buffered R-tree nodes in a server. The result is shown in Fig. 5. As the data volume increases, the average number of buffered R-tree nodes increases as well. In addition, the nonuniform dataset causes slightly more published R-tree nodes than the uniform data. This may be resulted from the property of R-tree and the publishing scheme. Although holding two portions of different global indexes in each server introduces more space cost, we still think it is a tolerant cost and the gained efficiency overweighs the space cost.

For the point query, to evaluate the performance, we adopt the single-path strategy, which means when we get the result R-tree nodes through the global index, we continue local searching from one server to another, rather than in parallel. Once we find the data, the query terminated. For the range query, we

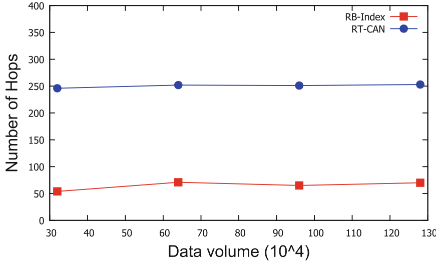

Fig. 7. Range query (uniform dataset)

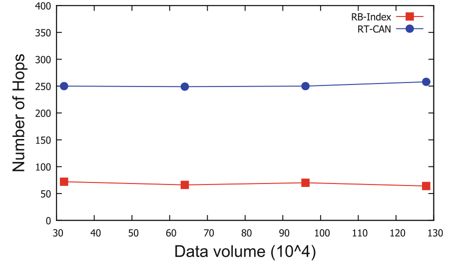

Fig. 8. Range query (nonuniform dataset)

limit the size of query range. Each time the search range will cover approximately 10 percentage of the range in each dimension. We respectively conduct 1000 randomly generated queries and take the average result. Figure 6 shows the average routing cost of point query. The result shows that even with increasing data volume, the routing cost still maintains stable. Figures 7 and 8 show the average routing cost of range query. Since RT-CAN only indexes three dimension of data, the average routing cost is almost the cost of broadcasting query in the network. In contrast, RB-Index improves the performance by more than 50 % in both uniform dataset and nonuniform dataset.

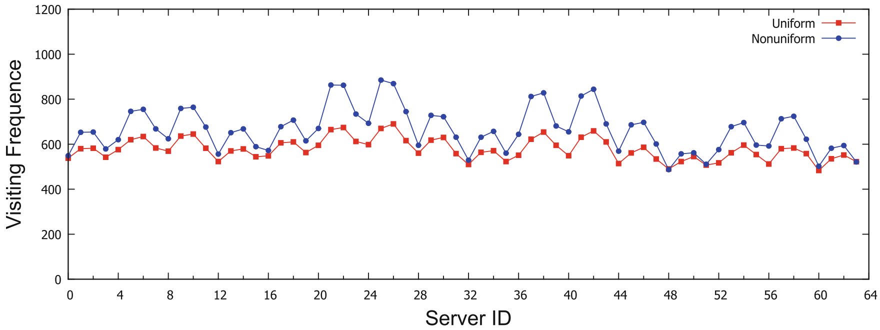

Fig. 9. Visiting frequence

Finally, we record the visiting frequence of each server in the network. As in real systems, both the point query and range query are processed, we conduct 2000 random queries with both point query and range query. The result is showed in Fig. 9. Under uniform dataset, the visiting frequence of each server is approximately same, while some servers mapped in the center of the indexing space are a bit more frequently visited since they possess more published R-tree nodes. Under nonuniform dataset, the overall visiting frequence decreases and due to the skewed distribution, some servers are particularly frequent-visited.

7 Conclusion

In this paper, we propose RB-Index, a multi-dimensional indexing scheme for Modular Data Center with Bcube topology. RB-Index adopts the two-layer indexing scheme, of which the global index is built upon local R-tree and is distributed over servers. Based on the characteristics of Bcube, we set up several indexing spaces to build several global indexes. We select the dimension of these indexing spaces adaptively to meet query habits. As a result, theoretically RB-Index can support any-dimensional data indexing. We propose the index construction rule and query algorithms to guarantee efficient data management in the network. In the evaluation, we simulate a 64-node Bcube and compare the query performance of RB-Index with RT-CAN, a most related previous work. Although building several global indexes increases space cost, the result verifies the query efficiency of RB-Index.

References

1. Chen, G., Vo, H.T., Wu, S., Ooi, B.C., Özsu, M.T.: A framework for supporting DBMSlike indexes in the cloud. VLDB **4**(11), 702–713 (2011)
2. Guo, C., Lu, G., Li, D., Wu, H., Zhang, X., Shi, Y., Tian, C., Zhang, Y., Lu, S.: BCube: a high performance, server-centric network architecture for modular data centers. In: SIGCOMM (2009)
3. du Mouza, C., Litwin, W., Rigaux, P.: SD-Rtree: a scalable distributed Rtree. In: ICDE pp. 296–305 (2007)
4. 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. In: OSDI, pp. 205–218 (2006)
5. Li, F., Liang, W., Gao, X., Yao, B., Chen, G.: Efficient R-tree based indexing for cloud storage system with dual-port servers. In: Decker, H., Lhotská, L., Link, S., Spies, M., Wagner, R.R. (eds.) DEXA 2014, Part II. LNCS, vol. 8645, pp. 375–391. Springer, Heidelberg (2014)
6. DeCandia, G., Hastorun, D., Jampani, M., Kakulapati, G., Lakshman, A., Pilchin, A., Sivasubramanian, S., Vosshall, P., Vogels, W.: Dynamo: amazons highly available key-value store. SIGOPS **41**(6), 205–220 (2007)
7. Wang, J., Wu, S., Gao, H., Li, J., Ooi, B.C.: Indexing multi-dimensional data in a cloud system. In: SIGMOD, pp. 591–602 (2010)
8. Hamilton, J.: An architecture for modular data centers. In: CIDR (2007)
9. Avinash, L., Prashant, M.: Cassandra: a decentralized structured storage system. ACM Spec. Interest Group Oper. Syst. (SIGOPS) **44**(2), 35–40 (2010)
10. Aguilera, M.K., Golab, W., Shah, M.A.: A practical scalable distributed B-tree. PVLDB **1**, 598–609 (2008)
11. Escriva, R., Wong, B., Sirer, E.G.: HyperDex: a distributed, searchable key-value store. In: SIGCOMM, pp. 25–36 (2012)
12. Sai, W., Wu, K.-L.: An indexing framework for efficient retrieval on the cloud. ICDE **32**(1), 75–82 (2009)
13. Wu, S., Jiang, D., Ooi, B.C., Wu, K.-L.: Efficient B-tree based indexing for cloud data processing. VLDB **3**(1–2), 1207–1218 (2010)

14. Gao, X., Li, B., Chen, Z., Yin, M., Chen, G., Jin, Y.: FT-INDEX: a distributed indexing scheme for switch-centric cloud storage system. In: ICC (2015)
15. Liang, Y., Balcan, M.-F., Kanchanapally, V.: Distributed PCA and k-means clustering. In: NIPS (2012)
16. Liu, Y., Gao, X., Chen, G.: Design and optimization for distributed indexing scheme in switch-centric cloud storage system. In: ISCC, pp. 804–809 (2015)
17. Jolliffe, I.T.: Principal Component Analysis, 2nd edn. Springer, New York (2002)
18. IBM: Scalable Modular Data Center. http://www-935.ibm.com/services/us/its/pdf/smdc-eb-sfe03001-usen-00-022708.pdf
19. Rackable Systems: ICE Cube Modular Data Center. http://www.rackable.com/products/icecube.aspx