R^2-Tree: An Efficient Indexing Scheme for Server-Centric Data Center Networks *

Yin Lin, Xinyi Chen, Xiaofeng Gao**, Bin Yao, and Guihai Chen

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

Abstract. Index plays a very important role in cloud storage systems, which can support efficient querying tasks for data-intensive applications. However, most of existing indexing schemes for data centers focus on one specific topology and cannot be migrated directly to the other networks. In this paper, based on the observation that server-centric data center networks (DCNs) are recursively defined, we propose *pattern vector*, which can formulate the server-centric topologies more generally and design R^2-Tree, a scalable two-layer indexing scheme with a local R-Tree and a global R-Tree to support multi-dimensional query. To show the efficiency of R^2-Tree, we start from a case study for two-dimensional data. We use a layered global index to reduce the query scale by hierarchy and design a method called *Mutex Particle Function* (MPF) to determine the potential indexing range. MPF helps to balance the workload and reduce routing cost greatly. Then, we extend R^2-Tree indexing scheme to handle high-dimensional data query efficiently based on the topology feature. Finally, we demonstrate the superior performance of R^2-Tree in three typical server-centric DCNs on Amazon's EC2 platform and validate its efficiency.

Keywords: data center network, cloud storage system, two-layer index

1 Introduction

Nowadays, cloud storage systems such as Google's GFS [7], Amazon's Dynamo [4], Facebook's Cassandra [2], have been widely used to support data-intensive applications that require PB-scale or even EB-scale data storage across thousands

* This work was partly supported by the Program of International S&T Cooperation (2016YFE0100300), the China 973 project (2014CB340303), the National Natural Science Foundation of China (Grant number 61472252, 61672353, 61729202 and U1636210), the Shanghai Science and Technology Fund (Grant number 17510740200), CCF-Tencent Open Research Fund (RAGR20170114), and Guangdong Province Key Laboratory of Popular High Performance Computers of Shenzhen University (SZU-GDPHPCL2017).

** Corresponding author.

of servers. However, most of the existing indexing schemes for cloud storage systems do not support multi-dimensional query well.

To settle this problem, a load balancing two-layer indexing framework was proposed in [18]. In two-layer indexing scheme, each server will: (1) build indexes in its local layer for the data stored in it, and (2) maintain part of global indexing information which is published by the other servers from their local data.

Based on the two-layer indexing framework, many efforts focus on how to divide the potential indexing range and how to reduce the searching cost. Early researches are mainly focused on Peer-to-Peer (P2P) networks such as RT-CAN [17], while later researches gradually turn to data center networks (DCNs) such as FT-INDEX [6], RT-HCN [12], etc. However, most of researches only focus on one specific network. The design lacks expandability and usually only suits one kind of network. Due to the differences in topology, it is always hard to migrate a specific indexing scheme from one network to another.

In this paper, we first propose a *pattern vector* P to formulate the topologies. Most of the server-centric DCN topologies are recursively defined and a high-level structure is scaled out from several low-level structures by connecting them in a well-defined manner. *Pattern vector* fully exploits the hierarchical feature of the topology by using several parameters to represent the expanding method. The raise of the *pattern vector* makes the migration of the indexing scheme feasible and is the cornerstone of generalization.

Then we introduce a more scalable two-layer indexing scheme for the server-centric DCNs based on P. We design a novel indexing scheme called R^2-Tree where a local R-Tree is used to support query for multi-dimensional local data and a global R-Tree helps to speed up the query for global information. We start from two-dimensional indexing. We reduce the query scale by hierarchy through building global indexes with a layered structure. The hierarchical design prevents repeated query process and achieve better storage efficiency. We also propose a method called *Mutex Particle Function* (MPF) to disperse the indexing range and balance the workload. Furthermore, we extend R^2-Tree to high-dimensional data space. Based on the hierarchy feature of the topology, we assign each level of the topology to be responsible for one dimension of the data. To handle data whose dimension is higher than the levels of the topology, we use *Principal Component Analysis* (PCA) to reduce the dimension. Besides, we design a mapping algorithm to select the nodes in local R-trees as public indexes and publish them on the global R-Trees of corresponding servers.

We evaluate the performance of range and point query for R^2-Tree on Amazon's EC2. We build two-layer indexes on 3 typical server-centric DCNs: **DCell** [10], **Ficonn** [13], **HCN** [11] with both two-dimensional and high-dimensional data and evaluate the query performance. Besides, by comparing the query time with **RT-HCN** [12], we show the technical advancement of our design.

The rest of the paper is organized as follows. The related work will be introduced in Sect. 2. Section 3 introduces the pattern vector to generalize the server-centric architectures. We elaborate the procedure of building two-layer index and the algorithm in Sect. 4 and depict the query processing in Sect. 5.

Section 6 exhibits the experiments and the performance of our scheme. Finally, we draw a conclusion of this paper in Sect. 7.

2 Related Work

Data Center Network. Our work aims to construct a scalable, load-balance, and multi-dimensional two-layer indexing on data center networks (DCNs). The underlying topologies of DCN can be roughly separated into two categories. One is the tree-like switch-centric topologies where switches are used for interconnection and routing like the Fat-Tree [1], VL2 [8], Aspen Tree [16], etc. The other one is the server-centric topology, in which the servers are not only used to store the data, but also perform the interconnecting and routing function. Typical server-centric topologies include data centers such as HCN [11], DCell [10], FiConn [13], Dpillar [14], and BCube [9]. Server-centric architectures are mostly recursively defined structures. Our work exploits this hierarchical feature and put forward a pattern vector which can generalize the server-centric topologies.

Two-Layer Indexing. Two-layer indexing [18] maintains two index layers called local layer and global layer to increase parallelism and support efficient query for different data attributes. Given a query, the server will first search its global index to locate the servers which may store the data and then forward the query. The servers which receive the forwarded query will search their local index to retrieve the queried data. Early two-layer index works focus on P2P network, like RT-CAN [17] and the DBMS-like indexes [3]. Subsequently with the rapid development of DCNs, a universal U^2-Tree [15] is proposed for switch-centric DCNs. Apart from that, RT-HCN [12] for HCN and an indexing scheme for multi-dimensional data for BCube [5] are both efficient indexing schemes for server-centric DCNs. Their works are mostly confined to a certain topology. With the generalized pattern vector, we design a highly extendable and flexible indexing scheme which can suit most of the server-centric DCNs.

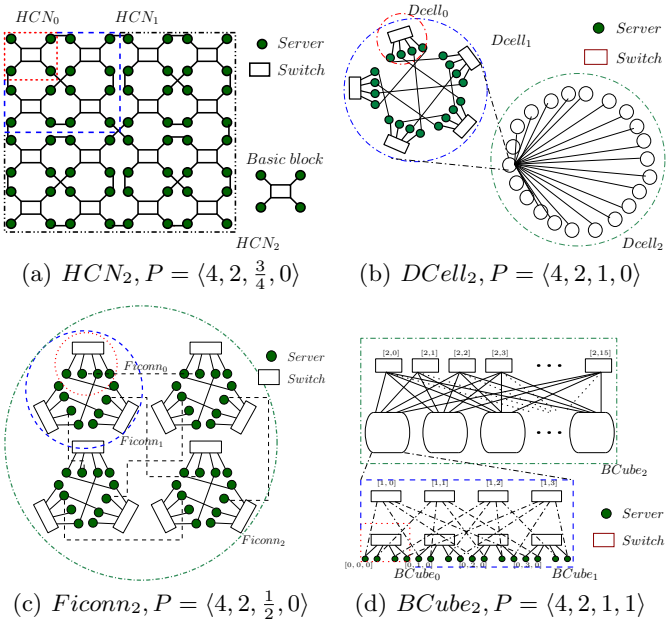
3 Recursively Defined Data Center

Server-centric DCN topologies have a high degree of scalability, symmetry, and uniformity. Most of the server-centric DCNs are recursively defined, which means that a high-level structure grows from a fixed number of low-level structures recursively. This kind of topologies has a favorable feature to design layered global index. However, due to the diversity of different kinds of topologies, with different number of Network Interface Card (NIC) ports for switches and connection methods, it is hard to migrate a specific indexing scheme from one topology to another. Thus, finding a general pattern for server-centric topologies is of great significance for constructing a scalable indexing scheme. We observe that the scaling out of the topology obeys some certain rules. The ratio of available servers which are actually used for expansion is fixed for every specific topology. In this section, we propose a *pattern vector* P to as a high-level representation to

Table 1. Symbol Description

Sym.	Description	Sym.	Description
h	Total height of the structure	na_i	Number of servers available to expand
k	Port number of mini-switch	nu_i	Number of servers actually used to expand
α	Expansion factor (≤ 1)	pir_j	potential indexing range of server j
β	Connection method denoter	g_i	Number of ST_{i-1} in ST_i (g_0=1)
ST_i	A level-i structure	q_i	Position of the meta-block in level-i
mbr	Minimum bounding rectangle	a_i	Position of the server in level-i

formulate the topologies. For clarity, we summarize the symbols in Table 1. Besides, we also show in Fig. 1 some typical server-centric topologies with the given pattern definition, including **HCN** [11], **DCell** [10], **Ficonn** [13] and **BCube** [5].



(a) $HCN_2, P = \langle 4, 2, \frac{3}{4}, 0\rangle$ (b) $DCell_2, P = \langle 4, 2, 1, 0\rangle$

(c) $Ficonn_2, P = \langle 4, 2, \frac{1}{2}, 0\rangle$ (d) $BCube_2, P = \langle 4, 2, 1, 1\rangle$

Fig. 1. Typical server-centric topologies represented by pattern vector P

To formulate the topology completely and concisely, 4 parameters are chosen for *pattern vector*. In the bottom right of Fig. 1(a), we show the basic building block, which contains a mini-switch and 4 servers. The port number of mini-switches which defines the basic recursive unit is denoted as k while the number of levels in the structure which defines the total recursive layers is denoted as h. Thus, in Fig. 1(a), $k = 4$, $h = 2$. Besides, the recursively scaling out rule for each topology is defined by the expansion factor and the connection method denoter, which are denoted as α and β and are explained in Def. 1, 2.

Definition 1 (Expansion factor) *Expansion factor α defines the utilization rate of the servers available for expansion. It can be proved that for every server-centric architecture, α is a constant and different server-centric architectures will have different α, which is given by: $\alpha = \lfloor nu_i/na_i \rfloor$.*

To explain, we use the symbol ST_i to represent the level-i structure. When ST_i scales out to ST_{i+1}, we define na_i as the number of available servers in ST_i that could be used for expansion, while we will use part of them for real expansion, and the total number of those used servers are defined as nu_i. Naturally, $na_i \geq nu_i$. We notice that for each topology, the ratio of servers used for expansion and available servers is surprisingly fixed. Therefore, we can denote a parameter α as $\lfloor nu_i/na_i \rfloor$ to depict the expansion pattern for each topology abstractly, which satisfies $0 < \alpha \leq 1$. For example, in Fig. 1(a), every time when HCN_i grows to HCN_{i+1}, $\alpha = \frac{3}{4}$, since three of four available servers will be used for topology expansion.

Definition 2 (Connection method denotor) *Connection method denotor β defines the connection method of servers, where $\beta = 1$ means the connection type is server-to-server-via-switch, like BCube in Fig. 1(d); and $\beta = 0$ means the connection type is server-to-server-direct, like DCell in Fig. 1(b).*

Definition 3 (Pattern vector) *A server-centric topology can be uniformly represented using a Pattern vector $P = \langle k, h, \alpha, \beta \rangle$, where k is the port number of mini-switches, h is the number of the total level, α is the expansion factor and β represents the connection method.*

To practice, let us first define g_{i+1} as the number of ST_i's in the next recursive expansion ST_{i+1}. Obviously, g_i can be calculated by: $g_i = \alpha \cdot na_{i-1} + 1$. Then take an eye on Fig. 1 again. Each of the subgraph exhibits a topology with $h = 2$. According to their different expansion rules, we can easily calculate the corresponding *pattern vector* values. Actually we can use *pattern vector* to construct brand new server-centric topologies, which could provide similar QoS service as other members in the server-centric family. For example in Fig. 2, for a given *Pattern Vector* $P = \langle 3, 3, \frac{1}{3}, 0 \rangle$, we can depict a new server-centric DCN.

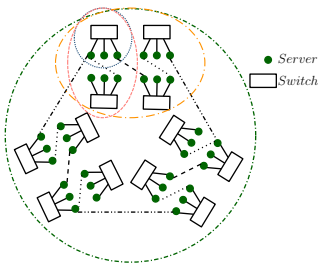


Fig. 2. A new-defined server-centric topology, $P = \langle 3, 3, \frac{1}{3}, 0 \rangle$

4 R^2-Tree Construction

When we use a *pattern vector* to depict any server-centric topologies generally, we can design a more scalable two-layer indexing scheme for efficient query processing requirements. We name this novel design as **R^2-Tree**, as it contains two R-Trees for both local and global indexes. A local R-Tree is an ideal choice for maintaining multi-dimensional data in each server and a global R-Tree helps to speed up the query in the global layer. In this section, we first discuss the hierarchical indexing design for two-dimensional data as an example, and then extend it to multi-dimensional version.

4.1 Meta-Block, Meta-Server and Representatives

Hierarchical global indexes design can avoid repeated query and achieve better storage efficiency. To build a hierarchical global layer, we divide the two-dimensional indexing space into $h+1$ levels of meta-blocks, defined as Def. 4.

Definition 4 (Meta-block) *Meta-blocks are a series of abstract blocks which are used to stratify the global indexing range. For a topology with $P = \langle k, h, \alpha, \beta \rangle$, the meta-blocks can be divided into $h+1$ levels.*

For a recursively defined structure with *pattern vector* $P = \langle k, h, \alpha, \beta \rangle$, we divide the total range in each dimension into g_h parts, where g_h is the number of ST_{h-1} in ST_h, and we can get ${g_h}^2$ meta-blocks on level-(h-1). Similarly, we divide the range in each dimension of meta-blocks in the second level into g_{h-1} parts and for each meta-block in second level, we get ${g_{h-1}}^2$ lower level blocks in the next layer. In this way, we can know that in the level-0, there are $\prod_{i=1}^{h} {g_i}^2$ meta-blocks. Thus, the total number of meta-blocks is given by Eqn. (1):

$$Total = \sum_{j=1}^{h} \prod_{i=j}^{h} {g_i}^2 + 1 \tag{1}$$

Each meta-block is assigned an $(h+1)$-tuple $[q_h, q_{h-1}, ..., q_1, q_0]$ in which q_i represents the meta-block's position in level-i. For example in the left part of Fig. 3, the level-0 block at the top left corner is assigned with $[0, 0, 0]$, while the level-1 block at the top left corner is assigned with $[1, 0, 0]$. To simplify the partition and search progress, we merge the $(h+1)$-tuple of each meta-block as a code ID named *mid*, which can be calculated by Eqn. (2).

$$mid_h = \sum_{i=0}^{h} \left(q_i \times \prod_{j=0}^{i} {g_j}^2 \right) \tag{2}$$

Figure 3 is an example for such range division process. Here in the left subgraph, the lowest level meta-blocks are coded as $0, 1, ..., 143$ and the second level meta-blocks are coded as $144, 153, ..., 279$. The highest level meta-block which covers the whole space is coded as 288.

Now we need to assign some representative servers in charge of each meta-block from a server-centric DCN structure.

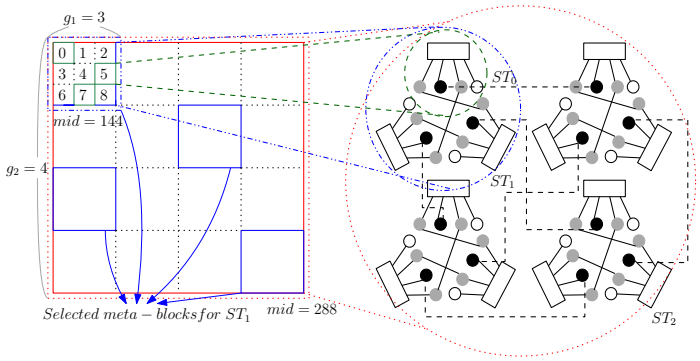


Fig. 3. Mapping meta-blocks to meta-servers

Definition 5 (Meta-server) *For each level-i structure ST_i, we can also denote it using pattern vector as $ST_i = \langle k, i, \alpha, \beta \rangle$, which can be an excellent representative to manage several corresponding meta-blocks, so it is also named as meta-server.*

Respectively, the right part of Fig. 3 shows a $Ficonn_2$ topology ($P = \langle 4, 2, \frac{1}{2}, 0 \rangle$). ST_2 denotes the meta-server in level-2 while ST_1 is the level-1 meta-server and ST_0 is the level-0 meta-server. Figure 3 also shows a **mapping scheme** to map the meta-blocks to the meta-servers. At level-i, there are g_i ST_i's, ${g_i}^2$ meta-blocks, so we map g_i meta-blocks to each ST_i. For each ST_i, we hope to select meta-blocks sparsely, so we formulate a *Mutex Particle Function (MPF)* to complete this task, motivated by mutex theory in physics. The mapping function will be described in Sect. 4.2.

Figure 3 illustrates this mapping rule thoroughly. The meta-block in the first-layer is mapped to the first-layer meta-server (ST_2). Since ST_2 contains 4 second-layer meta-server (ST_1), the first-layer meta-block contains 4^2 second-layer meta-blocks. Therefore each ST_1 is in charge of 4 second-layer meta-blocks. Similarly, each meta-block which is mapped to the first ST_1 can be divided into 3^2 parts and be mapped to the third-layer meta-server (ST_0) accordingly. After mapping meta-blocks to meta-servers, as meta-servers are just virtual nodes, we should select physical servers as representatives of meta-servers.

Definition 6 (Meta-server representative) *To achieve fast routing process, we select the connecting servers between ST_{i-1}'s as the representatives of ST_i.*

In Fig. 3, the grey nodes are the representatives for ST_0 and the black nodes are the representatives for ST_1. Selecting representatives in this method guarantees that the query in the upper layer of the meta-blocks can be forwarded to the lower layer in the least number of hops, and more than one representative to a meta-server guarantees a degree of redundancy.

Algorithm 1: Mutex Particle Function (MPF)

Input: A meta-server ST_i
Output: S_i: a set of meta-blocks which are mapped to meta-server ST_i

```
S_i = {∅};
Select a meta-block in this layer randomly and add it into S_i, and set the
  centroid of this mapped set as the center of this node;
while |S_i| < ∏_{j=i+1}^{h} g_j do
    From the set of the non-mapped meta-blocks, select one whose centroid is
      mostly far away from the centroid of the mapped set. Add this node into
      the mapped set of this meta sever, and re-calculate the centroid of the
      mapped set;
```

4.2 Mutex Particle Function

Once the queries appear intensively in a certain area, all the nearby meta-blocks will be searched at a high frequency. Therefore, a carefully designed mapping scheme is needed to balance the request load. We propose *Mutex Particle Function (MPF)* in this subsection. As its name illustrated, we regard the meta-blocks assigned to the same meta-server as the same kind of particles and like mutual exclusion of charges, same kind of particles should be mutually exclusive with each other. That means in two-dimensional space, the distance between the same kind of meta-blocks should be as far as possible. Every time we select a meta-block to a meta-server, we choose the furthest one from the centroid of the meta-blocks which have been chosen. Algorithm 1 describes *MPF* in detail.

4.3 Publishing Local Tree Node

In the process of building R^2-Tree indexes, we first build local R-Tree for every server based on their local data. Then to better locate the servers, information about local data and the corresponding server will be published to global index layer. We first select the nodes to be published from the local R-Trees, which starts from the second layer of local R-Tree to the end layer where all the nodes are leaf nodes. For the layer before the end layer, we select the nodes which have no published ancestors with a certain probability to publish. For the end layer, we publish all the nodes whose ancestors have not been published. In this way, we guarantee the completeness of the publishing scheme. Moreover, we make sure that the nodes in the higher layer have a higher possibility to be published so to reduce the storage pressure in global index layer. After the selection of the local R-Tree node, we find the minimum potential indexing range of a meta-server which covers this selected node exactly. Then, we publish the local R-Tree node to the corresponding representatives in the format of (mbr, ip), where mbr is the minimum bounding rectangle of the local R-Tree node, and ip means the ip address of the server where this node is stored. For each server, it will build a global R-Tree based on all the R-Tree nodes published to it. Global R-Tree can accelerate the speed in searching global indexes and forward the query.

4.4 Multi-Dimensional Indexing Extension

The R^2-Tree indexing scheme can also be extended to multi-dimensional space. In our design, multi-dimensional indexing takes advantage of the recursive feature of the topologies to divide the hypercube space and let one level of the structure be in charge of one dimension. In this paper, we will not discuss circumstance where the data dimension is extremely high like image data. This may be solved by LSH-based algorithms, but it is another story from our bottleneck-avoidable two-layer index framework.

Potential Index Range. For a server-centric DCN structure with h levels, we can construct an $(h+1)$-dimensional indexing space. If the dimension of the data exceeds $h+1$, methods like principle component analysis (PCA) can be applied to reduce the index dimension. We assign one level of the structure to maintain the global information in one dimension. Since the number of parts in each dimension should be equal to the number of the lower layer structures ST_{i-1} in ST_i which is denoted by g_i, we divide the indexing space in dimension i into g_i parts (k for dimension 0) and every ST_{i-1} in this level will be responsible for one of them. Figure 4 shows the indexing design in detail.

4.5 Potential Indexing Range

As we have mapped several meta-blocks to a meta-server, the potential indexing range of a meta-server is the sum of ranges of those meta-blocks. Taking uniformly distributed data as an example, since there are $\prod_{j=i+1}^{h} {g_j}^2$ meta-blocks in level-i, the two-dimension boundary ($[l_0, u_0]$, $[l_1, u_1]$) can be divided into $\prod_{j=i+1}^{h} g_j$ segments for each dimension in level-i. The range of the highest level meta-block is $pir_h = ([l_0, u_0], [l_1, u_1])$. The range of meta-blocks for each dimension is given by:

$$
\begin{aligned}
pir_{i_0} &= \left[l_{i_0} + (q_i \bmod g_{i+1}) \times \frac{u_{i_0} - l_{i_0}}{g_{i+1}}, l_{i_0} + (q_i \bmod g_{i+1} + 1) \times \frac{u_{i_0} - l_{i_0}}{g_{i+1}} \right] \\
pir_{i_1} &= \left[l_{i_1} + (\lfloor q_i \div g_{i+1} \rfloor) \times \frac{u_{i_1} - l_{i_1}}{g_{i+1}}, l_{i_1} + (\lfloor q_i \div g_{i+1} \rfloor + 1) \times \frac{u_{i_1} - l_{i_1}}{g_{i+1}} \right]
\end{aligned}
\tag{3}
$$

In Eqn. (3), the subscript of pir means the level of the meta-block and 0 means the first dimension while 1 means the second dimension. u_i and l_i represent the boundary of the higher level meta-block which just covers it, q_i means the position of meta-block in level-i and i satisfies $0 \leq i < h$.

If data is not uniformly distributed, we use the *Piecewise Mapping Function (PMF)* [19] method to balance the skew data. The goal of PMF is partitioning the data evenly into some buckets. We use the cumulative mapping to evenly divide the data into buckets by using hash function.

In HCN_2, with $P = \langle 4, 2, \frac{3}{4}, 0 \rangle$ which is shown in Fig. 4, the potential indexing range of each server is represented by the purple cuboid. The servers in the level-0 structure will be combined together and ST_0 will manage the potential

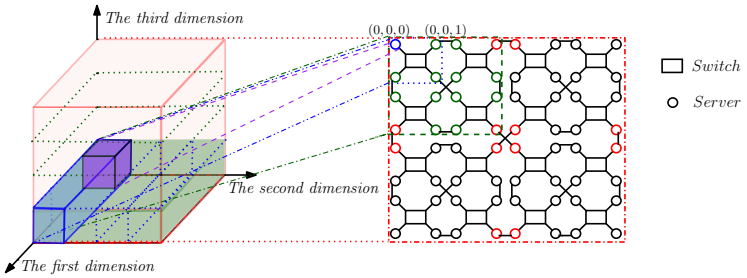


Fig. 4. Potential Indexing Range of HCN_2

indexing range represented by the blue long cuboid. The level-1 structure ST_1 consists of 4 ST_0's and will manage the green cuboid consisting of 4 blue cuboids. At the highest level, the data space it manages will be the whole red cuboid.

Suppose the indexing space is bounded by $B = (B_0, B_1, ..., B_h)$, and B_i is $[l_i, l_i + w_i]$, $i \in [0, h]$, the potential range of server s is $pir(s)$. Similar to meta-blocks, each meta-server is also assigned an $(h+1)$-tuple $[a_h, a_{h-1}, ..., a_1, a_0]$ in which a_i represents the meta-block's position in level-i.

Lemma 1 *For a server s which is represented by tuple $[a_h, a_{h-1}, a_{h-2}, ..., a_0]$, its potential indexing range of pir is:*

$$\begin{aligned} pir\,(s) &= pir\,([a_h, a_{h-1}, ..., a_0]) \\ &= \left(\left[l_0 + a_0 \frac{w_0}{k}, l_0 + (a_0+1)\,\frac{w_0}{k} \right], ..., \left[l_h + a_h \frac{w_h}{g_h}, l_h + (a_h+1)\,\frac{w_h}{g_h} \right] \right) \end{aligned} \quad (4)$$

Publishing Scheme. Each server builds its own local R-tree to manage the data stored in it. Meanwhile, every server will select a set of nodes $N_k = \{N_k^1, N_k^2, ..., N_k^n\}$ from its local R-tree to publish them into the global index. Similar to the two-dimension situation, the format of the published R-tree node is (mbr, ip). ip records the physical address of server and mbr represents the minimum bounding rectangle of the R-tree node. For each selected R-tree node, we will use center and radius as the criteria for mapping. We set a threshold named R_{max}, to compare with the given radius. Given an R-tree node to be published, we first calculate the center and radius. Then, the node will be published to the server whose potential index range covers the center. If radius is larger than R_{max}, the node will be published to those servers whose potential indexing range intersects with the R-tree node range.

5 Query Processing

5.1 Query in Two-Dimensional Space

Point query. The point query is processed in two steps: (1) The first step happens among the meta-servers to locate the servers which may possibly store

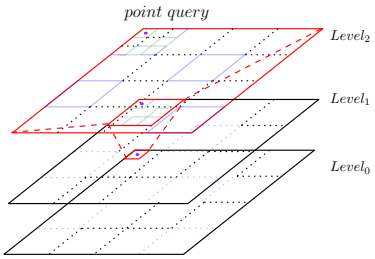


Fig. 5. An example of the point query process in R^2-Tree

the data. The query point $Q(x_0, x_1)$ will be first forwarded to the nearest level-h level representative which represents the largest meta-block. Then the query will be forwarded to level-(h-1) representative with corresponding meta-block whose potential indexing range covers Q. The process goes on until the query is forwarded to a level-0 structure. All the representatives which receive the query will search their global R-Trees and forward the query to local servers. (2) In the second step, the servers will search their local R-Trees and return the result. In all, only $(h + 1)$ representatives will be searched in total.

Figure 5 shows a point query example in the global R-Tree on the same topology shown in Fig. 3. Traditionally, we need to perform the query in all servers in the DCN. However, if the hierarchical global indexes are used, we only need to perform query in much fewer servers. For example, for the point query represented by the purple node, the querying process will go through the global index from $Level_2$ to $Level_0$ with 3 representatives, and then the query will be forwarded to the servers who possibly store the result. Therefore, from this case, we can see the effectiveness of this indexing scheme.

Range Query. The range query is similar to point query which is also a two-step processing. Given a range query $R([l_{d_0}, u_{d_0}], [l_{d_1}, u_{d_1}])$, as the same as the processing in point query, we begin query from the largest meta-server to the smallest meta-server which can just cover the range R and then the forwarded servers will search their local R-Trees to find the data. The only difference is that in point query the smallest meta-server must be a physical server.

5.2 Query in High-Dimensional Space

Point Query. The point query is a two-step processing. Given a point query $Q(x_0, x_1, x_2, ..., x_d)$, we first create a super-sphere centered at Q with radius R_{max}. We search all the servers whose potential indexing range intersects with the super-sphere. To increase query speed, we forward the query in parallel. After getting the R-tree nodes which cover the point query, we forward the query to the servers which contain these nodes locally.

Range Query. The range query $R([l_{d_0}, u_{d_0}], ..., [l_{d_h}, u_{d_h}])$ will be sent to all the servers whose potential indexing range intersects with range query R. These

servers will search their global indexes and find the corresponding R-Tree nodes. The query will be forwarded to those local servers. The cost of range query is less than directly broadcasting to all the servers.

6 Experiments

To validate R^2-Tree indexing scheme, we choose three existing server-centric data center network topologies including **DCell** ($P = \langle 4, 2, 1, 0\rangle$), **Ficonn** ($P = \langle 4, 2, \frac{1}{2}, 0\rangle$), **HCN** ($P = \langle 4, 2, \frac{3}{4}, 0\rangle$) to test the performance of our indexing scheme with them on the platform of Amazon's EC2. We implement our R^2-Tree in Python 2.7.9. We use in total 64 instance computers. Each of them has two-core 2.4GHz Intel Xeon E5-2676v3 processor, 8GB memory and 8GB EBS storage. The bandwidth is 100 Mbps. The scale of the DCN topologies ranges from level-0 to level-2. The experiments involve 3 two-dimensional datasets: (1) Uniform_2d which follows uniform distribution, (2) Zipfian_2d which follows zipfian distribution, and (3) Hypsogr which is a real dataset obtained from the R-Tree Portal [1] and one uniform three-dimensional datasets. The detailed information of our experiments is shown in Table 2.

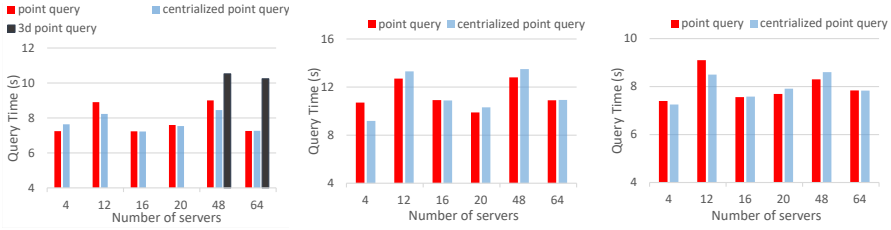
Table 2. Experiment Settings

Parameter	**Values**
DCN topologies	DCell, Ficonn, HCN
Structure level	0, 1, 2
Dimensionality	2, 3
Distribution	Uniform, Zipfian, Real
Uniform Datasets	Uniform_2d, Uniform_3d
Skew Datasets	Zipfian_2d, Hypsogr
Query Method	point query, range query, centralized point query

Our experiments are conducted as follows. For each DCN topology, we generate $2,000,000$ data points for each server. We execute 500 point queries and 100 range queries and record the total query time as the metric for each dataset. Additionally, to test the effectiveness of the *Mutex Particle Function*, we also perform centralized 500 point queries where all the query are confined to a certain area of the whole data space. By comparing the query time with **RT-HCN** [12], we show the superiority of our global R-Tree design. Besides, by counting the hop number for each point query and the average number of global indexes, we explain a trade-off between the query time and the storage efficiency.

In R^2-Tree, we propose hierarchical global indexes for two-dimensional data and divide the potential indexing range evenly for three-dimensional data. In Fig. 6, we show the point query performance of R^2-Tree in three different datasets. Since it is impossible to manipulate hundreds of thousands of servers in

[1] http://chorochronos.datastories.org/?q=node/21

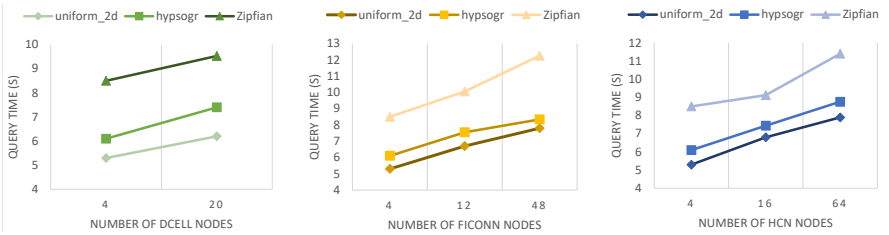


(a) Uniform (b) Zipfian (c) Hypsogr

Fig. 6. Point query performance

the experiments and a certain number of servers will be representative enough, the server number of **DCell** scales from 4 to 20, while the server number of **Ficonn** scales from 4 to 12 and 12 to 48, and for **HCN**, the server number scales from 4 to 16 and from 16 to 64. The two parallel columns represent the query time for the normal point query and the centralized point query respectively when the server number and the type of dataset are fixed. Based on the result that the query time for the centralized and non-centralized point query is close to each other when the other parameters are fixed, we show that the *Mutex Particle Function* balances the request load effectively.

We observe from Fig. 6 that the query time increases as the DCN structure scales out. By counting the global indexes stored in representatives in different levels, we notice an unbalance of the global information. The representatives in higher level tend to store more global indexes because they have larger potential indexing range. Since most of the chosen-to-published R-Tree nodes are from upper layer, the minimum bounding boxes are larger and will be more likely to be mapped to the meta-blocks which have larger potential indexing range. Nonetheless, in this way, we achieve higher storage efficiency since we do not need to store a lot of global information in each server. Besides, the global R-Tree helps to alleviate this bottleneck to a great extent. Among the three different datasets, we can see that the query time is the shortest for Uniform dataset and longest for Zipfian dataset.



(a) DCell (b) Ficonn (c) HCN

Fig. 7. Range query performance

The range query in Fig. 7 also shows a same tendency of query time increase as the structure scales out. From the comparison of query time between different topologies, we find that for the same level number and the same kind of dataset, **DCell** performs the best while **Ficonn** performs the worst. We calculate the number of hops among the servers for a point query to explain the inner reason. In Fig. 8, we can see that the number of hops increases as the structure scales out. For the same level structure, the number of hops for DCell is the least and the hop number for Ficonn is the largest. This can be explained by *expansion factor* α easily. Figure 9 explains the trade-off between query time and storage space clearly. Larger α means that the connection between servers is more compact, and the number of physical hops will reduce and therefore achieve better time efficiency. However, the store efficiency will decrease correspondingly since each server stores more global information in different levels. By Comparing the query hop numbers for 2d and 3d data in Fig. 8, we can see the efficiency for the hierarchical global indexing design. Since the potential indexing range is of different size, we only publish the tree node to the just-cover meta-block. This mechanism avoids the repeated query effectively, and therefore reduce the total query time. Besides, in Fig. 10, we compare the query time of **R²-Tree** to **RT-HCN** [12]. Global R-Tree accelerates the global query and *PMF* helps to balance the request load. Therefore, R^2-Tree shows superiority over **RT-HCN** [12].

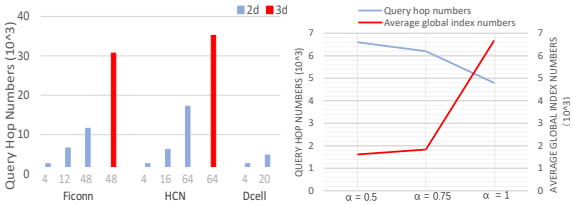


Fig. 8. Hop number

Fig. 9. Trade-off

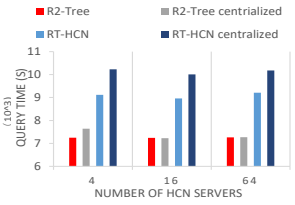


Fig. 10. Comparisions

7 Conclusion

In this paper, we propose an indexing scheme named **R²-Tree** for multidimensional query processing which can suit most of the server-centric data center networks. To better formulate the topology of server-centric DCNs, we propose a *pattern vector* P through analyzing the recursively-defined feature of these networks. Based on that, we present a layered mapping method to reduce query scale by hierarchy. To balance the workload, we propose a method called *Mutex Particle Function* to distribute the potential indexing range. We prove theoretically that R^2-Tree can reduce both query cost and storage cost. Besides, we take three typical server-centric DCNs as examples and build indexes on them based on Amazon's EC2 platform, which also validates the efficiency of R^2-Tree.

References

1. Al-Fares, M., Loukissas, A., Vahdat, A.: A scalable, commodity data center network architecture. In: ACM SIGCOMM Computer Communication Review. pp. 63–74 (2008)
2. Beaver, D., Kumar, S., Li, H.C., Sobel, J., Vajgel, P.: Finding a needle in haystack: facebook's photo storage. In: OSDI. pp. 47–60 (2010)
3. Chen, G., Vo, H.T., Wu, S., Ooi, B.C., Özsu, M.T.: A framework for supporting DBMS-like indexes in the cloud. Proceedings of the VLDB Endowment 4(11), 702–713 (2011)
4. Decandia, G., Hastorun, D., Jampani, M., Kakulapati, G., Lakshman, A., Pilchin, A., Sivasubramanian, S., Vosshall, P., Vogels, W.: Dynamo: Amazon's highly available key-value store. In: SOGOPS. pp. 205–220 (2007)
5. Gao, L., Zhang, Y., Gao, X., Chen, G.: Indexing multi-dimensional data in modular data centers. In: DEXA. pp. 304–319 (2015)
6. Gao, X., Li, B., Chen, Z., Yin, M.: FT-index: A distributed indexing scheme for switch-centric cloud storage system. In: ICC. pp. 301–306 (2015)
7. Ghemawat, S., Gobioff, H., Leung, S.T.: The google file system. In: SOSP. pp. 29–43 (2003)
8. Greenberg, A., Hamilton, J.R., Jain, N., Kandula, S., Kim, C., Lahiri, P., Maltz, D.A., Patel, P., Sengupta, S.: VL2: A scalable and flexible data center network. In: ACM SIGCOMM Computer Communication Review. pp. 51–62 (2009)
9. 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. ACM SIGCOMM Computer Communication Review 39(4), 63–74 (2009)
10. Guo, C., Wu, H., Tan, K., Shi, L., Zhang, Y., Lu, S.: DCell: A scalable and fault-tolerant network structure for data centers. ACM SIGCOMM Computer Communication Review 38(4), 75–86 (2008)
11. Guo, D., Chen, T., Li, D., Li, M., Liu, Y., Chen, G.: Expandable and cost-effective network structures for data centers using dual-port servers. IEEE Transactions on Computers 62(7), 1303–1317 (2013)
12. Hong, Y., Tang, Q., Gao, X., Yao, B., Chen, G., Tang, S.: Efficient R-Tree based indexing scheme for server-centric cloud storage system. IEEE Transactions on Knowledge & Data Engineering 28(6), 1503–1517 (2016)
13. Li, D., Guo, C., Wu, H., Tan, K.: Ficonn: Using backup port for server interconnection in data centers. In: INFOCOM. pp. 2276–2285 (2009)
14. Liao, Y., Yin, D., Gao, L.: Dpillar: Scalable dual-port server interconnection for data center networks. In: ICCCN. pp. 1–6 (2014)
15. Liu, Y., Gao, X., Chen, G.: A universal distributed indexing scheme for data centers with tree-like topologies. In: DEXA. pp. 481–496 (2015)
16. Walraed-Sullivan, M., Vahdat, A., Marzullo, K.: Aspen Trees: Balancing data center fault tolerance, scalability and cost. In: CoNEXT. pp. 85–96 (2013)
17. 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)
18. Wu, S., Wu, K.L.: An indexing framework for efficient retrieval on the cloud. IEEE Computer Society Data Engineering Bulletin 32(1), 75–82 (2009)
19. Zhang, R., Qi, J., Stradling, M., Huang, J.: Towards a painless index for spatial objects. ACM Transactions on Database Systems 39(3), 19 (2014)