Efficient R-Tree Based Indexing Scheme for Server-Centric Cloud Storage System

Yang Hong, Qiwei Tang, Xiaofeng Gao, Bin Yao, Guihai Chen, *Senior Member, IEEE*, and Shaojie Tang

Abstract—Cloud storage system poses new challenges to the community to support efficient concurrent querying tasks for various data-intensive applications, where indices always hold important positions. In this paper, we explore a practical method to construct a two-layer indexing scheme for multi-dimensional data in diverse server-centric cloud storage system. We first propose RT-HCN, an indexing scheme integrating R-tree based indexing structure and HCN-based routing protocol. RT-HCN organizes storage and compute nodes into an HCN overlay, one of the newly proposed sever-centric data center topologies. Based on the properties of HCN, we design a specific index mapping technique to maintain layered global indices and corresponding query processing algorithms to support efficient query tasks. Then, we expand the idea of RT-HCN onto another server-centric data center topology DCell, discovering a potential generalized and feasible way of deploying two-layer indexing schemes on other server-centric networks. Furthermore, we prove theoretically that RT-HCN is both space-efficient and query-efficient, by which each node actually maintains a tolerable number of global indices while high concurrent queries can be processed within accepted overhead. We finally conduct targeted experiments on Amazon's EC2 platforms, comparing our design with RT-CAN, a similar indexing scheme for traditional P2P network. The results validate the query efficiency, especially the speedup of point query of RT-HCN, depicting its potential applicability in future data centers.

Index Terms—Distributed index, R-tree, data center network

1 Introduction

Cloud storage systems keep gaining attentions from both academia and industry nowadays. From classical systems for general data services, such as Google's GFS [1], Amazon's Dynamo [2], Facebook's Cassandra [3], to newly designed systems with specialities, such as Haystack [4], Megastore [5], Spanner [6], various distributed storage systems were constructed to satisfy the increasing demand of online data-intensive applications that require massive scalability, efficient manageability, reliable availability, and low latency in the storage layer. Many works have been proposed for designing new indexing scheme and data management system to support large-scale data analytical jobs and high concurrent OLTP queries [7], [8], [9], [10].

To achieve query efficiency, most existing mature cloud storage systems employ a pure key-value data model or some of its variants, which are lacking in built-in support for multi-dimensional index, as can be seen from Dynamo and BigTable. Actually, two-layer index [7] is just an efficient and suitable framework for multi-dimensional querying in Cloud systems. This framework has been put into successful practice in [8], [9], [10]. However, they construct their global indices on the P2P networks, like BATON [11] and CAN [12]. We know that P2P networks give better illustrations for connections on the logic level than the Internet level, but their underlying topologies are actually undefined and the nodes may scatter widely with unbounded physical hop distance, bringing instability of performance.

As is known, more and more cloud systems today favor an infrastructure called *data center*, which consists of large number of servers interconnected by a specific *Data Center Network* (DCN) [13], [14], [15], [16], [17], [18], [19], [20], [21], [22], [23], [24], [25]. For instance, Cisco applies Fat-Tree topology as its DCN architecture for provably efficient communication [13]. Designing efficient indexing scheme on data center topologies has been put on the agenda. Different from P2P network, DCN is more structured with low equipment cost, high network capacity, and support of incremental expansion. Particularly, DCN has specific physical topologies, with the connections among nodes strictly defined. It is not wise to simply transplanted the indexing scheme for P2P networks onto DCN topologies. Such infrastructures bring new challenges for researchers to design efficient indexing scheme to support query processing for various applications.

In this paper, we show the construction of a distributed multi-dimensional indexing scheme for server-centric data center networks. We start from one of the mainstream server-centric topologies, named *Hierarchical Irregular Compound Networks* (HCN) [26], [27], as an example topology. HCN is mathematically a simple and beautiful topology due to the inherent regularity and symmetry, which brings in convenience for the indexing building and potential for expansion. Correspondingly, we propose a two-layer indexing scheme, RT-HCN. Since datasets are distributed among different servers, we can use an R-Tree like indexing structure to index locally stored multi-dimensional

- *Y. Hong, Q. Tang, X. Gao, B. Yao, and G. Chen are with the Shanghai Key Laboratory of Scalable and Computing and Systems, Department of Computer Science and Engineering, Shanghai Jiao Tong University, Shanghai, China 200240. E-mail: hongyang729@msn.cn, 1421940251@qq.com, {gao-xf, yaobin, gchen}@cs.sjtu.edu.cn.*
- *S. Tang is with the Naveen Jindal School of Management, University of Texas at Dallas, Richardson, Texas, U.S. 75080. E-mail: shaojie.tang@utdallas.edu.*

Manuscript received 23 Apr. 2015; revised 17 Dec. 2015; accepted 19 Jan. 2016. Date of publication 4 Feb. 2016; date of current version 27 Apr. 2016.
Recommended for acceptance by A. Aboulnaga.
For information on obtaining reprints of this article, please send e-mail to: reprints@ieee.org, and reference the Digital Object Identifier below.
Digital Object Identifier no. 10.1109/TKDE.2016.2526006

data for each server. Next, RT-HCN selectively distributes these local indices across servers as their global indices. To avoid single master server bottleneck, each server only maintains partial global index for its potential indexing range. Based on the characteristics of HCN, we design an index mapping method to promote the distribution of the global index onto computer servers. Corresponding query processing algorithms are also presented then. We also prove theoretically that RT-HCN is both query-efficient and space-efficient, by which servers will not maintain too may redundant indices while a large number of users can concurrently process queries with not relatively high routing cost. Before evaluation, we discuss our work on expanding our RT-HCN indexing scheme into another data center topology DCell. We compare our design with RT-CAN [8], a similar design for traditional P2P network, in terms of performance on multi-dimensional and skewed data. Experiments validate the query efficiency with low false positives of our proposed scheme and depict its potential implementation in data centers.

Our contribution of this paper is threefold:

- we are the first to propose a distributed multi-dimensional indexing scheme for server-centric DCN structures, and also the first to develop a general and practical method to expand this two-layer indexing scheme onto other data center networks;
- we present a specialized mapping technique to improve global index distribution in the network, bringing query-efficiency and load-balancing for the cloud system; we besides combine practical techniques to solve data skewness and querying false positives, greatly increasing the adaptability and querying performance of RT-HCN;
- we theoretically prove the efficiency of RT-HCN, and compare it numerically with RT-CAN [8], an indexing scheme for P2P network. Experiments on real platforms show that our scheme performs excellently for point query.

The rest of this paper is organized as follows. Section 2 summarizes the related work. Section 3 introduces the overview of our system, including a coding method of HCN nodes and meta-servers. In Section 4 we illustrate the procedure of two-layer indexing construction with a wealth of details, including some techniques for multi-dimension indexing, data skewness handling, and false positives controlling. Section 5 briefly depicts the query processing algorithms. In Section 6 we expand the RT-HCN indexing scheme to fit another server-centric data center networks, DCell. Section 7 proves the efficiency of our design and performs numerical experiments in comparison with the RT-CAN [8]. Finally, Section 8 provides a conclusion and gives some discussion about future work.

2 Related Works

2.1 Two-Layer Indexing

To get a better speed for data retrieval in these distributed storage, usually a mechanism for parallel querying is required. A two-level indexing framework is a good choice. A key idea about the combination of two-level indexing with the overlay networks was first proposed in [7]. A general mode is offered: in the indexing framework, processing nodes are organized in a structured overlay network, and each processing node builds its local index to speed up data access. A global index is built by selecting and publishing a portion of the local index in the overlay network, while the global index is distributed and maintained in all nodes over the network. Several attempts have already been carried out on P2P overlay networks. RT-CAN [8] was proposed as a multi-dimensional indexing scheme built on top of local R-tree indices and organized on a CAN overlay network. It helps to provide efficient data retrieval service for large-scale shared-nothing clusters. Different from this, CG-index [9] organizes computer nodes into a structured P2P network, BATON, and builds B-tree indices to support high throughput one-dimensional queries. The generalizable work in [10] presents an extensible indexing framework to support DBMS-like indices in the cloud based on the observation that many P2P overlays are instances of the Cayley graph [28]. This extensible framework is able to support multiple types of distributed indices simultaneously, such as hash, B-tree-like and multi-dimensional index, significantly reducing the maintenance cost and providing the needed scalability.

2.2 Data Center Networks

What our work pursues is a scalable method of adjustment for two-layer indexing building on DCN networks. One of the two core components is that our underlying topology is a specific server-centric DCN structure. Data center network is the network infrastructure for a data center, which connects a large number of servers via high-speed links and switches. Compared to traditional cloud system which is usually based on P2P network, specially and carefully designed DCN topologies fulfill the requirements with low-cost, high scalability, low configuration overhead, robustness and energy-saving. DCN structures can be roughly divided into two categories, one is switch-centric, which means that the function of switches is enhanced to accommodate the need of the interconnection and routing, while the servers require no modification. There are three kinds of it, tree-like, flat and optical switch based. The Fat-Tree [13], VL2 [14] and Aspen trees [15] belongs to the tree-like kind. The other category is server-centric, which means that each server enables the functions of interconnection and routing while the switches require no change and only provide easy crossbar function. Among them, DCell [16], FiConn [17], [18], Dpillar [19], SWCube and SWKuatz [20] are designed for mega data centers. While BCube [21], MDCube [22], uFix [23], snowflake [24], hyper-fat-tree network (HFN) [25] are topologies for the modular data centers. Chronologically, they usually have more advantages than the former designs. HCN [26], [27], the topology chosen in our system falls into the server-centric topology. It is a well-designed network for data center and offers a high degree of regularity, scalability, and symmetry. Different from traditional P2P network, the physical connection of DCN is known, we can guarantee the processing time by calculating out the physical hops needed for a given query, while in P2P network only logical hops of the overlay network can be estimated. We have to be aware of the physical topology when we are discussing DCN and that is why we need to improve

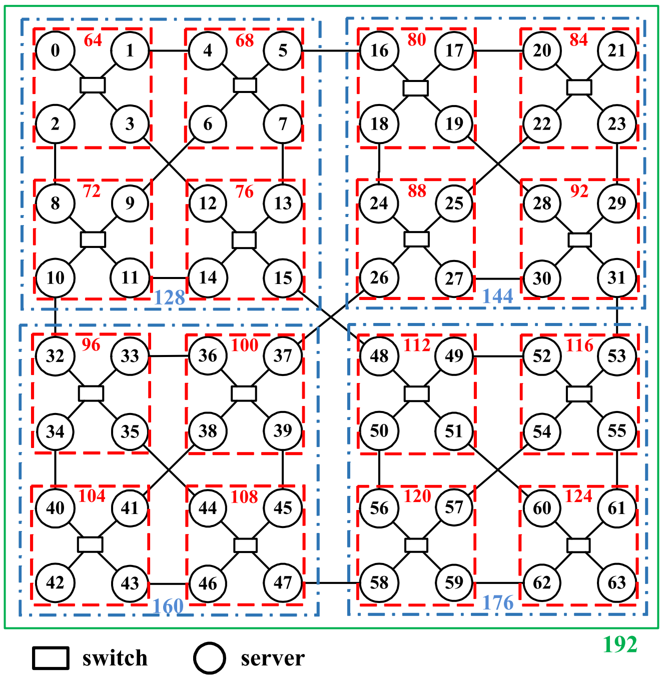


Fig. 1. HCN(4,2) with coding for meta-server.

the mapping technique for distributing global index to fix a given network.

Actually, similar works on the DCN-based indexing scheme has yielded good results. FT-Index [29] is a distributed indexing scheme based on Fat-Tree topology. In FT-Index, the B$^+$-tree and the Interval tree are adopted to organize data set locally and globally. FT-Index has a good performance on false positives and space-efficiency, with the help of two assistant tools FT-Gap and FT-Bloom. In another work [30], the B$^+$-tree is combined with Segment tree, and the two-layer indexing is implemented on three tree-like switch-centric DCN topologies. However, either FT-Index or the later one performs worse for general range query than point query. This is essentially the limit of B$^+$-tree indexing for one-dimensional data. Inspired by this, we want to enrich the types of querying in our new indexing scheme.

2.3 R-Tree in Distributed Environment

The other core components is a multi-dimension oriented indexing structure: we use the R-tree. R-trees [31], [32], developed for indexing multi-dimensional data, are the most popular indices for spatial query processing due to their simplicity and efficiency. The R-tree extended the B-tree from one dimension to multi-dimensions for indexing static features. Spatial features and their relationships were recognized and stored in a tree structure so that features could be more quickly retrieved by tracing along the tree structure. R-tree's advantages in multi-dimensional query has additional value in the distributed systems. With the proliferation of location-based e-commerce and mobile computing, the need for moving objects querying begins to arise. Assuming object movement trajectories are known as priori, Saltenis et al. [33] proposed the Time-Parameterized R-tree (TPR-tree) for indexing moving objects. By combining with the improved UK-means algorithm, Kao et al. [34] uses an R-tree index to solve the problem of clustering

TABLE 1
Symbol Description

Sym	Description
n	Code for server and meta-server
S_n	The server with code n
M_n	The meta-server with code n
$\mathbf{R}_n$	Representatives for M_n
$\mathbf{N}_n$	S_n's node set for publishing
h	The highest level of HCN
$\mathbf{B}$	Data boundary
$\mathbf{B}_n$	Potential index range for M_n
R_n^i	The ith representative for M_n
N_n^i	The ith publishing node for S_n

uncertain objects whose locations are described by probability density functions. In [35] Zheng et al. developed efficient algorithms to answer fuzzy objects querying, by extending the R-tree indexing structure and deriving several highly effective heuristic rules.

3 System Overview

We focus on the server-centric DCN structure for the distributed indexing scheme construction, and take HCN as the first example to illustrate our design. In this section, some basic features of HCN topology will be introduced and illustrated. Then, we explain some preliminary definitions for further illustration of index construction strategy.

3.1 Hierarchical Irregular Compound Network

HCN (Hierarchical irregular Compound Network) is a well-designed network for data center and offers a high degree of regularity, scalability, and symmetry. A level-h HCN with n servers in every single unit is denoted as HCN(n,h). HCN is a recursively defined structure. A high-level HCN(n,h) employs n low level HCN$(n,h-1)$ as a unit cluster and connects all the clusters by means of a complete graph. HCN$(n,0)$ is the smallest module (basic construction unit) that consists of n dual-port servers and an n-port mini-switch. For each server, its first port is used to connect with the mini-switch while the second port is employed to interconnect with another server in different smallest modules for constituting a larger network. Fig. 1 illustrates an example of HCN with $n=4$ and $h=2$, which consists of 64 servers. Each server is labeled according to a coding process, which will be introduced in Section 3.2.

It is easy to see that there is always multiple routes between any two servers in HCN and this is called multi-path routing which provides good network features like high bandwidth, good balancing and fault tolerance. This is the main aspect we want to concern during index construction and also becomes a main reason why special designed index scheme for specific network is well worth being discussed.

For clarity, we summarize the symbols with their meanings in Table 1. Some of them will be described in the following sections.

3.2 Meta-Server and Coding Strategy

Given an HCN$(4, h)$, there are 4^{h+1} servers in total and are coded by n ranging from 0 to $4^{h+1} - 1$. Thus we use S_n to denote the nth server in the HCN. Since HCN itself is a recursively defined structure, there are also 4^{h-l} different HCN$(4, l)$'s $(0 \le l \le h)$ in one HCN$(4, h)$. We consider each S_n and HCN$(4, l)$ $(0 \le l \le h)$ as a meta-server in our system.

Definition 1. *A Meta-server is a single server or an HCN$(4, l)$ $(0 \le l \le h)$ considered entirely.*

Meta-servers together constitute the overlay network and facilitate global queries, which is going to be explained in detail later. Meta-servers are also coded by n and denoted as M_n. The coding strategy of M_n is given by:

$$M_n = \begin{cases} S_n, & 0 \le n < 4^{h+1} \\ \text{HCN}(4, q-1), & n \ge 4^{h+1} \end{cases}, \tag{1}$$

where the HCN$(4, q-1)$ exactly consists of servers ranging over $S_r, S_{r+1}, \ldots, S_{r+4^q-1}$, with q and r from the above equation represent the quotient and the reminder when the given n is divided by 4^{h+1}. From the function, we know that M_n is equivalent to S_n when $n < 4^{h+1}$, and when $n \ge 4^{h+1}$, M_n represents a specific HCN$(4, l)$ $(0 \le l \le h)$. For example, Fig. 1 shows that in an HCN$(4, 2)$, 64 meta-servers consist of only one server are coded with the number ranging from 0 to 63, 16 meta-servers formed by HCN$(4, 0)$ are coded with 64, 68, $\ldots$, 124 (shown as red squares), four bigger meta-servers formed by HCN$(4, 1)$ are coded with number 128, 144, 160, 176 (shown as blue squares), while the biggest green square formed by the whole HCN $(4, 2)$ is coded with 192.

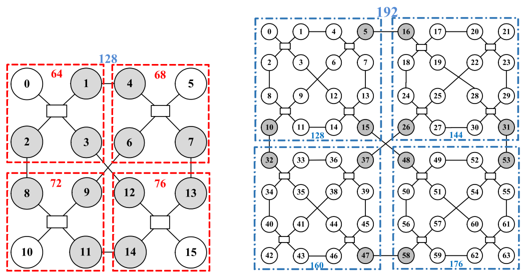
3.3 Representatives in Meta-Server

As is already mentioned, meta-servers form a higher-level overlay network and assist query processing in the network. However, as the meta-server is merely an abstract concept, we need to pick up several physical servers to be in charge of queries that are sent to corresponding meta-server from the overlay network.

Definition 2. *Representatives of a given meta-server M_n are several physical servers that actually deal with queries forwarded to M_n, which are picked out based on the following strategy.*

1) If $n < 4^{h+1}$, we choose S_n as the representative of M_n since $M_n = S_n$.
2) If $n \ge 4^{h+1}$, M_n, now, is actually an HCN$(4, l)$ $(0 \le l \le h)$, and we provide a special bit manipulation to find out its representatives. First we calculate the quaternary form of n, denoted as $q_0q_1 \cdots q_m$. Here $m > h$ stands since $n \ge 4^{h+1}$. Second, we pick up the first $m - h$ bits as shown in Equation (2) and calculate the decimal number b. Then we replace each of the last b bits with q_*, where $q_* \ne q_{m-b}$ and will get several newly formed number. Finally we calculate the decimal form of the last $h + 1$ bits of the new numbers and servers coded with those numbers are the representatives for this M_n.

$$\underbrace{\overbrace{q_0q_1\cdots q_{m-h-1}}^{m-h\text{ bits}}}_{=b(\text{decimal})}\ \underbrace{\overbrace{q_{m-h}\cdots q_{m-b}q_{m-b+1}\cdots q_m}^{h+1\text{bits}}}_{\text{replace part}}. \tag{2}$$



(a). $l = 0$ (b). $l = 1$

Fig. 2. The Representatives for Meta-servers in HCN (4,2).

For example, the grey nodes in Figs. 2a and 2b illustrates the representatives for corresponding meta-server HCN$(4, l)$ when l equals to 0 and 1. It is obvious that these representatives actually takes the advantages of HCN topology and offer good connectivity which will facilitate query processing.

We denote the representatives of M_n as set $\mathbf{R}_n$ and $\mathbf{R}_n$ has different number of entities in different situation. In case 1, $\mathbf{R}_n = \{R_n^1\}$ while in case 2, $\mathbf{R}_n = \{R_n^1, R_n^2, R_n^3\}$. Here R_n^i stands for the server which is the ith representative of meta-server M_n, and R_n^i is called an lth-level representative if and only if M_n is a meta-server formed by an HCN$(4, l)$ $(0 \le l \le h)$.

Now we give some explanation on choosing representatives. It is obvious to choose S_n as representatives for M_n when $n < 4^{h+1}$ because they are exactly the same. So we focus our discussion on case 2 here. According to the topology of HCN, it is not hard to find that all of the representatives we choose for M_n $(n \ge 4^{h+1})$ are servers that are more closely connected to other HCN$(4, l)$'s in the same level. Thus, our strategy cut the cost of queries forwarding and since there are three representatives for a single M_n, it also offers flexibility to choose the closest representative or the dullest one. This strategy also fits quite well with the multi-path routing of HCN. What's more, the following lemma and theorem show that our strategy also offers good scalability and balancing property.

Lemma 1. *Each meta-server M_n $(n \ge 4^{h+1})$ has exactly three representatives except for the highest level of meta-server with four.*

Proof. There are four different bits in quaternary number and our strategy replace the bits with a same bit different with q_{m-b} in Equation (2), so the result is three. The biggest $M_{(h+1)\cdot 4^{h+1}}$ is the only special one that has four representatives since the bit q_{m-b} does not exist in this case. □

Theorem 1. *Each server S_n will be the representative for exactly two different meta-servers.*

Proof. First, it is obvious that each server should be chosen as a representative for M_n where $n < 4^{h+1}$. Second, each meta-server with its number$\ge 4^{h+1}$ picks disjoint groups of representatives based on Equation (2), which means each server would also be chosen as a representative for only one meta-server M_n where $n \ge 4^{h+1}$. □

From Lemma 1 and Theorem 1, we can claim that in RT-HCN, the number of replications for every global indexing is well controlled as three or four, bringing in good space-efficient, and that different scales of querying tasks are evenly distributed over all the HCN nodes as different levels of representatives.

4 Indexing Construction

4.1 Potential Indexing Range

Before we begin our discussion of index construction, we illustrate another essential concept about our meta-servers. In order to construct the global index for multi-dimensional data in our overlay network, we have assigned each meta-server a potential indexing range. Thus, for any given queries, we can figure out which meta-server is responsible for it and then the query is processed by the representatives of that meta-server.

Definition 3. *Potential indexing range is an abstract attribute assigned to meta-server and indicates which meta-server (indeed the subordinate representatives) is (are) responsible for processing a given query.*

The multi-dimensional data forms a data boundary denoted as $\mathbf{B}$, which is a k-dimensional rectangle as the bounding box of the spatial data objects: $\mathbf{B} = (B_0, B_1, B_2, \ldots, B_d)$. Here d is the number of dimensions and each B_i is a closed bounded interval $[l_i, u_i]$ describing the extent of the data along dimension i. Next, we focus on the two-dimensional situation while for $d \geq 3$, we take a slightly different consideration in Section 4.5.

When $d = 2$, the data is bounded by $\mathbf{B} = (B_0, B_1)$, where B_0 is $[l_0, u_0]$ and B_1 is $[l_1, u_1]$. We calculate a quaternary number Q_n for each meta-server M_n and use it to help figure out the potential indexing range $\mathbf{B}_n$ for M_n. Suppose $q_0 q_1 \cdots q_m$ is the corresponding quaternary form for n, then Q_n is calculated according to the following two cases:

1) If $m \leq h$, we add $h - m$ consecutive 0's to the front of $q_0 q_1 \cdots q_m$ and construct an $h+1$ bit quaternary number $Q_n = 00\cdots 0 q_0 q_1 \cdots q_m$.
2) If $m > h$, $q_0 q_1 \cdots q_m$ can be split as the form explained in Equation (2). In this situation, we pick out the last $h+1$ bits and delete the replace part to get $Q_n = q_{m-h} q_{m-h+1} \cdots q_{m-b}$.

Now, we use the following iteratively defined function to calculate $\mathbf{B}_n$.

$$\mathbf{B}_n = pir(\mathbf{B}, Q_n)$$
$$= \begin{cases} pir\Big(\Big(\Big[l_0, \frac{l_0+u_0}{2}\Big], \Big[l_1, \frac{l_1+u_1}{2}\Big]\Big), Q'_n\Big), & q_0 = 0 \\ pir\Big(\Big(\Big[\frac{l_0+u_0}{2}, u_0\Big], \Big[l_1, \frac{l_1+u_1}{2}\Big]\Big), Q'_n\Big), & q_0 = 1 \\ pir\Big(\Big(\Big[l_0, \frac{l_0+u_0}{2}\Big], \Big[\frac{l_1+u_1}{2}, u_1\Big]\Big), Q'_n\Big), & q_0 = 2 \\ pir\Big(\Big(\Big[\frac{l_0+u_0}{2}, u_0\Big], \Big[\frac{l_1+u_1}{2}, u_1\Big]\Big), Q'_n\Big), & q_0 = 3 \\ ([l_0, u_0], [l_1, u_1]) & Q_n = \varnothing \end{cases}, \quad (3)$$

where Q_n is a quaternary number denoted as $q_0 q_1 \cdots q_i$ and Q'_n is a newly constructed quaternary number $q_1 q_2 \cdots q_i$.

For example in Fig. 3, two axes B_0 and B_1 denote the range of data in two dimensional space (w.l.o.g., we assume it generates a square area, rather than a rectangle). Then the potential indexing range for M_{80} (denoted as red square) should be $([\frac{l_0+u_0}{2}, \frac{l_0+3u_0}{4}], [l_1, \frac{3l_1+u_1}{4}])$; the *pir* for M_{144} (denoted as blue square) is $([\frac{l_0+u_0}{2}, u_0], [l_1, \frac{l_1+u_1}{2}])$, while the *pir* for M_{192} is $([l_0, u_0], [l_1, u_1])$.

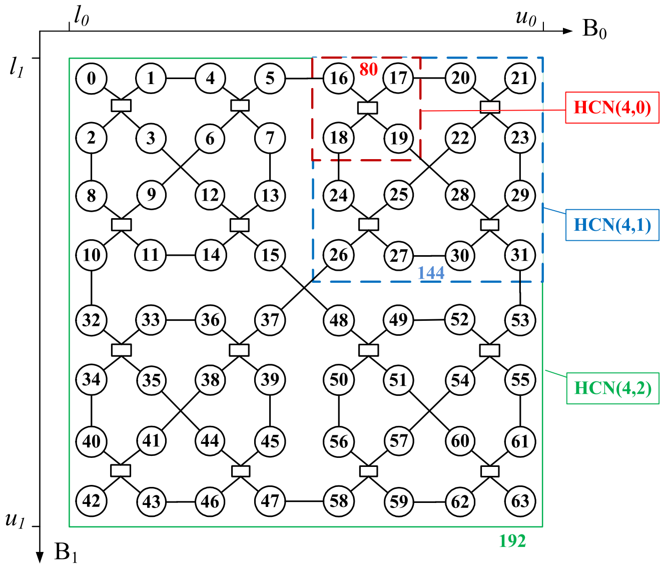


Fig. 3. Potential index range.

4.2 Cumulative Mapping

This section concerns about skewed data processing. Consider a one-dimensional skewed dataset, and there are some servers waiting to be assigned the potential indexing ranges. If the data is sorted, we can assign each server the range containing equally proportional data according to the ordered sequence. In this way, the number of data fall into the responsible range of each server is almost the same and skewness is eliminated. However, the ideal method is not practical in that we must collect all the data to do the sorting, which is both storage and time intolerable.

That being said, Zhang et al. [36] offered a *Piecewise Mapping Function (PMF)* to achieve an approximately uniform distribution for the skewed data. To construct the PMF, we evenly partition the data range into some buckets. Then the boundary coordinates of buckets are the coordinates used to compute the PMF. Consider 10 numbers $\{0.5, 0.9, 1.4, 2.5, 2.7, 3.2, 3.7, 5.4, 7.0, 8.9\}$ distributed in $[0, 10]$. If we divide them into five buckets with boundary coordinates $0, 2, 4, 6, 8$, and 10, the count of data in each buckets will be $3, 4, 1, 1, 1$, with a variance of 1.6. Then we can easily get the cumulative values of the boundary coordinates: $0, \frac{3}{10}, \frac{7}{10}, \frac{8}{10}, \frac{9}{10}, 1$. Based on the six pairs of boundary coordinate and cumulative value, we can build a piecewise function with five pieces well-reasonably. Then we map all the 10 real data distributed on $[0, 10]$ through this piecewise function to values distributed on $[0, 1]$, gaining respective approximate cumulative value, as shown in Table 2. After mapping, the same buckets gains a variance of 0, balancing the counts perfectly.

Formally, PMF is obtained as follows:

1) Take a random sample for the whole dataset to do the later computation on a small scale and keep the original feature of data distribution as well.
2) Equally divide the data space into N buckets, calculate the cumulative values of the $N+1$ boundary

TABLE 2
Mapping for Approximate Cumulative Value

$[0,2]$	$[2,4]$	$[4,6]$	$[6,8]$	$[8,10]$
0.5	2.5	5.4	7.0	8.9
0.9	2.7			
1.4	3.2			
	3.7			
0.075	0.21	0.44	0.64	0.85
0.135	0.31	0.54	0.77	0.945
$[0,0.2]$	$[0.2,0.4]$	$[0.4,0.6]$	$[0.6,0.8]$	$[0.8,1.0]$

coordinates of these N buckets, and build N piecewise linear functions PMF.

3) Based on the PMF, compute the mapped value for all real data in the sample set.
4) Treat the mapped values as a new sample set, repeat step 2 and 3 until a satisfied variance of the counts in each buckets has been achieved.

As for multi-dimensional data, we compute PMF's for different dimensions in the same way.

Cumulative mapping actually applies to both skewed and uniform data, as the consequence of mapping is a nearly uniform distribution. However, since space mapping is a relatively static manner, original distribution maybe essentially changed due to data updating, the whole indexing system will not benefit from the mapping any longer. To handle this, we can periodically check the variance of number of data in each bucket, and we shut down to rebuild the system in the case of its obvious fluctuation beyond some threshold.

4.3 Bloom Filter

A Bloom filter is a space-efficient probabilistic data structure, which can be used as an accelerator in point query. For a set S with n elements $\{e_1, e_2, \dots e_n\}$, a filter use a bit array of m bits, initially all set to 0. k independent hash functions are used, denoted as $f_1, f_2, \dots, f_k$, each producing an integer in $[1, m]$. For each element e_i, the bits at positions $f_1(e_i)$, $f_2(e_i), \dots, f_k(e_i)$ in the array of m bits are set to 1. In practical use, we need to choose the parameters m and k appropriately in order to minimize the inevitable false positives and maximize the filtering ability. According to [37], we draw several salutary lessons. For a previously defined Bloom filter, the probability of a false positive for an element not in the set, or the false positive rate, can be estimated as $f = (1 - e^{-\frac{kn}{m}})^k$. On one hand, given n and m, more hash functions provides more chances to find a 0 bit for an element that is not a member of S, but using fewer hash functions increases the fraction of 0 bits in the array. The optimal number of hash functions that minimizes f as a function of k is $\ln 2 \cdot \frac{m}{n}$. On the other hand, given n and optimal k, the length of bit array m needs to be essentially $n\log_2 e \cdot \log_2 \frac{1}{\epsilon}$ for any representation scheme with a false positive rate bounded by ϵ. For our experiments, the parameter selection for Bloom filters depends largely on these theoretical results.

As for multi-dimension, a direct idea in [38] is to generate a bloom filter for each dimension with the same series of hash functions. When a multi-dimensional data is to be checked, we only need to check its components on different dimensions with corresponding bit arrays and get the intersected results of yes or no. However, there are high probability that a misjudgment at any dimension would have probably brought in a final false positive. This is because the union information of the multi-dimension is not fully used. Based on this observation, we decide to concatenate all the multidimensional components together as a single string. Moreover, the Murmur hash we use to generator bloom filters is very good at processing long strings.

Specifically, bloom filter is added as an inherent property of our local R-tree nodes. In the construction of RT-HCN, each selected local R-tree node will create a bit array for all the data items it covers with Murmur hash function as its bloom filter. Then all the created filters are published along with the nodes to be buffered remotely. They will serve for the RT-HCN point query procedure. The details of index publishing and update of bloom filters is to be explained later.

4.4 Indexing Publishing

As is introduced before, each data server S_n build an R-tree index for its local data to facilitate multi-dimensional search. Then, S_n adaptively selects a set of index nodes $\mathbf{N}_n = \{N_n^1, N_n^2, \dots, N_n^{d_n}\}$ from its local R-tree and publishes each N_n^i to the representatives of a specific meta-server whose potential indexing range just covers the minimum bounding range of N_n^i. Bloom filters of the corresponding nodes is also generated and published. The choose of the publishing nodes starts from the second level of R-tree to an ending level in a probabilistic manner. For each level before the ending level, we publish nodes who have no published ancestors with a fixed probability. For the ending level, we publish all the left nodes who have no published ancestors. In this way, the principle of index completeness and unique index when doing index publishing [8] are both satisfied. Meanwhile, various sizes of R-tree nodes get a chance to be published, which conforms to the RT-HCN's design ideas of "indexing hierarchical management". We set the ending level as antepenult level and the probability as 0.7, giving more publishing chances for the larger nodes with the purpose of checking the bloom filters of high level nodes as soon as possible in the point query procedure and keeping the total number of global indices at a relatively low level. The format of the published R-tree nodes is (n, mbr), where n indicates the origin server for storing the data and mbr is the minimal bounding range of the published R-tree node. After receiving the published nodes, representatives buffers the index in memory. Algorithm 1 describes the process of index publishing.

Algorithm 1. Index Publishing (For S_n)

```
1 $\mathbf{N}_n$ =getSelectedRTreeNode($S_n$)
2 for each $N_n^i \in \mathbf{N}_n$ do
3     Find the least $n'$ s.t. $\mathbf{B}_{n'}$ fully covers $N_n^i.mbr$
4     Get the representatives $\mathbf{R}_{n'}$ for $M_{n'}$
5     for each $S_k \in \mathbf{R}_{n'}$ do
6         $S_k$ inserts $(n, N_n^i.mbr)$ into its global index set
```

Fig. 4 provides a simple example of a local R-tree. If the node R_1 is selected to be published, it should be published to server S_{17}, S_{18}, S_{19}, which are representatives of the HCN

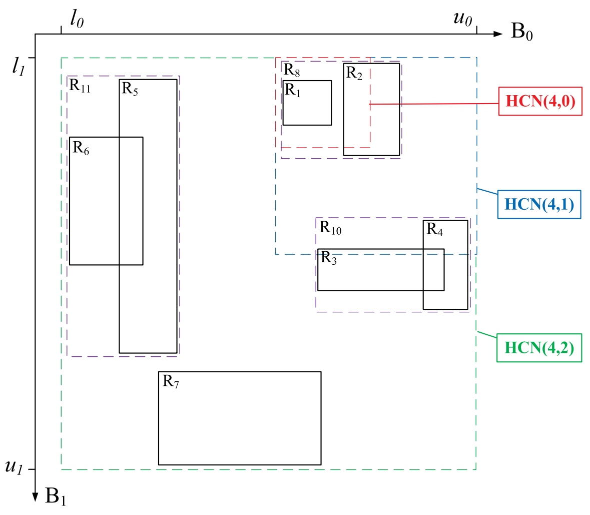
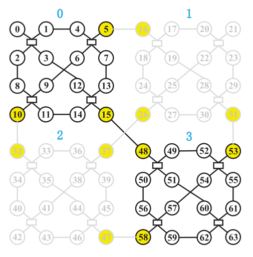
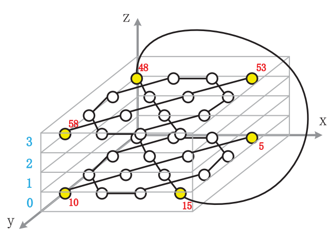


Fig. 4. Index distribution.

$(4, 0)$ shown in red. Similarly, if the node R_3 is selected to be published, it should be published to server S_{16}, S_{26}, S_{31}, which are representatives of the HCN$(4, 1)$ shown in blue.

The average routing cost for one-to-one traffic in HCN is $O(2^{h+1})$, and the cost for information transmission between representatives of the same meta-server is $o(2^{h+1})$, thus the cost to publish an index node should be $O(2^{h+1})$ on average, and it equals to $O(\sqrt{N})$ when $n = 4$, where N is the total number of servers in the given HCN. For more general situation, the cost is given by $N^{-\log_2 n}$ and can be reduced as n get larger. We also want to mention here that the cost shown above is exactly physical hops between servers while previous works claims that it takes only $O(\log N)$ to publish index in P2P network but they are only discussing hops in the overlay network. Since the physical connection of P2P network is unclear, the physical hops can be hard to exam and constrain. Another improved feature for index publishing in our system is that under this design we can make sure that each index node is published to exactly only three servers in the system. However, the strategy used in [8] publishes the larger index nodes to many servers as long as the range of a auxiliary circle centered with the publishing node's center overlaps with the responsible query range of them. Then, the amount of index nodes published in the system is hard to control.

As for index maintenance, we consider the index update triggered by local data insertion. If an insertion causes the range of a leaf node to be expanded, we simply published this leaf node with a newly generated bloom filter to the right place, which costs 1 or 3 or 4 HCN routing messages (depending on the number of representatives) with information of the newly published node and at most dozens of local hashing. If an insertion does not cause any expansion, it needs to update the remote bloom filter of its published ancestor node and there must exist such one. In this case, the updating still costs 1 or 3 or 4 HCN routing messages (depending on the number of representatives) with information of new bit array positions to be set to 1 and several times of local hashing. In this way, the correctness of point and range query can both be guaranteed. We also implement another version of index updating when bloom filter is not used for our experiment, which is the same with RT-CAN: when published node is split due to local insertion, we just delete the remote published ones and republish two new nodes, which costs triple HCN routing messages comparing with the former updating method.

(a). Planar View (b). 3D View

Fig. 5. Two views of HCN$(4, 2)$.

4.5 Multi-Dimension Indexing

When dimension is 3 or higher, we weaken the concept of potential indexing range. HCN can not naturally support the one-one mapping between topology and data space like RT-CAN but has its own advantage that it is a recursively defined multi-level topology. We can use the levels to process the dimensions. An HCN(n, h) consists of n HCN $(n, h-1)$, or n^2 HCN$(n, h-2)$, etc. Similarly, a d-dimensional space can be divided into $\frac{d}{n}$ or $\frac{d}{n^2}$ $(d-1)$-dimension space, to be further processed by HCN$(n, h-1)$ or HCN $(n, h-2)$. And so it goes on. Finally, the last two-dimensional space is just what we have already discussed. To achieve this, we need to find an appropriate way of allocating HCN levels.

In our design, an HCN$(4, h)$ can be used to process data of at most $(h+2)$-dimension. Given the top level h and the data dimension $(A_1, A_2, \ldots, A_D)$ $(D \leq h+2$, the order here makes no sense), we use the following principles to allocate the total $(h+1)$ HCN levels for different dimensions:

1) Let l_2 be the number of levels shared by dimension A_1 and A_2. For $d \geq 3$, let l_d be the number of levels allocated to dimension A_d, then $l_d = l_{d-1}$ or $l_d = l_{d-1} - 1$.
2) $\Sigma_{i=2}^{D} l_i = h + 1$.

Take HCN$(4, 2)$ for example. If the target dataset is two-dimensional, we calculate the Meta-servers and corresponding potential indexing ranges just as the previous discussion does, which means we allocate total three HCN levels for two-dimensional data. In the case of three-dimension, we allocate two levels for the first two dimensions, and 1 level for the third dimension according to the above principles. Fig. 5b is the $3d$ viewpoint of planar HCN$(4, 2)$ in (a) with 4 levels of HCN$(4, 1)$ with the ability to partition a three-dimensional data space. To be simple, we only highlight the 0_{th} and 3_{rd} HCN$(4, 1)$ unities as dark regions. In another word, we use four HCN$(4, 1)$ to divide one dimension, and use one single HCN$(4, 1)$ to process the other two dimensions. Above all, the previous concepts of Meta-server, representative and potential indexing range only make sense in the single HCN$(4, 1)$ for two-dimension processing. As for the node publishing of three-dimensional R-tree, each node appears as a cuboid in a 3-d space in Fig. 5b. If a selected R-tree node overlaps with some of the four subspaces along the z-axis, we will distribute the same copies of it into these overlapping subspaces for the same representatives along the z-axis's direction. Generally speaking, with one more dimension, the number of global indices will be quadrupled. Correspondingly, querying process for multi-dimensional data will first determine which smallest subsequences for 2-d processing are involved in. This step just consists of lots of simple partitions of the search space.

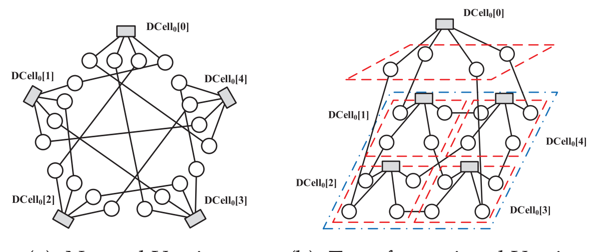
5 Query Processing

RT-HCN can process kinds of queries, such as point query, range query, and k-NN query. In this section, we explain how different queries for two-dimensional data are executed in RT-HCN.

The point query in RT-HCN is a two-stage processing, similar to most other two-layer indexing schemes. First stage happens among the Meta-servers, where a query point $Q(v_0, v_1)$ is first forwarded to the nearest hth level representative S_{n_h} which represents the largest Meta-server covering Q. S_{n_h} returns qualified results through checking the bloom filters of its buffered global indexing nodes, afterwards, it forwards the query to the nearest $h-1$th level representative $S_{n_{h-1}}$ which represents the $h-1$th Meta-server covering Q and $S_{n_{h-1}}$ does the same querying and forwarding jobs till the level of a single server. After all the qualified results received, the query processing goes into the second stage of remote querying on local R-trees. In all, a point query needs to search $h+1$ servers in total, which actually are representatives from 0th to hth levels. Our index publishing scheme does bring out some extra cost for some queries, and we present the theoretical analysis about the comparison of this extra routing cost with the simplest one-to-one point query in Section 7.1.

The range query is a general version of the point query. We regard a range $R([a_0, b_0], [a_1, b_1])$ as a large queried point and find out the smallest HCN$(4, t)$ that fully covers this large queried point. Then, a point query procedure is carried out until the tth level. During the query before tth level, the query is always forwarded to the Meta-server whose potential indexing range covers the queried range R while from then on, the query shall be forwarded to all the Meta-servers whose potential indexing range overlaps with R, guaranteeing the correctness and completeness of results.

The k-nearest neighbours (k-NN) query returns the top-k nearest results to the data $Q(v_0, \ldots, v_{d-1})$ given the query (Q, k). Since the dataset has been mapped into an approximately evenly distributed data space, we consider the density of data in the mapped space as $\rho = \frac{K}{\Pi(U_i - L_i)}$, where K is the total number of data and the denominator is the size of the whole data space. ρ is thus the average number of data in a single unit of the data space. We can use ρ to estimate the ranges of k nearest results in such space. We will first query the range $\left(\left[v_0 - \gamma\sqrt[d]{\frac{k}{\rho}}, v_0 + \gamma\sqrt[d]{\frac{k}{\rho}}\right], \ldots, \left[v_{d-1} - \gamma\sqrt[d]{\frac{k}{\rho}}, v_{d-1} - \gamma\sqrt[d]{\frac{k}{\rho}}\right]\right)$, where d is the dimension, $\gamma\sqrt[d]{\frac{k}{\rho}}$ is a uniform offset, and γ is a scaling parameter typically equals to 0.5. If more than k results are returned, we select k nearest ones. Otherwise, we increase the offset linearly.



(a). Normal Version (b). Transformational Version

Fig. 6. Two forms of $DCell_1$ topology (with $n = 4$).

6 Scheme Generalization

Server-centric data center topologies generally are recursive topologies (usually recursively defined multi-level structures), in which a high-level structure consists of certain low-level structures and the structures at the same level are connected with each other in a well-defined way. As is mentioned in the beginning, HCN is our first target topology since it is relatively simple and regular. We further seek for methods to build similar two-layer indexing schemes on other structured server-centric data center networks. Compared to developing specific indexing schemes for different types of underlying topologies, a reusable and adaptable method for this procedure is much more admired.

DCell is a structure that has many desirable features for data centers. It uses servers equipped with multiple network ports and mini-switches to construct its recursively defined architecture. Any high-level DCell is constituted by connecting certain number of the next lower level DCells. $DCell_0$ is the basic building block in which n servers are connected to an n-port commodity switch. DCells at the same level are fully connected with each other. Given t servers in a $DCell_k$, t+1 $DCell_k$'s are used to build a $DCell_{k+1}$. The t servers in a $DCell_k$ connect to the other t $DCell_k$'s, respectively. This way, DCell achieves high scalability and high bisection width.

The DCell topology is our first attempt for generalization and is believed to be a good breakthrough point, derived from some internal connection between HCN and DCell. Fig. 6 shows the topology of $DCell_1$ with $n = 4$, referred as the level-1 DCell. The $DCell_1$ consists of five $DCell_0$s, ranging from $DCell_0[0]$ to $DCell_0[4]$. In fact, we can get different "pictures" of this topology from different angles. Fig. 6a is the normal version of $DCell_1$, which has been the most common form well known since its birth. However, what Fig. 6b illustrates is also the topology of $DCell_1$. This version can be transformed from Fig. 6a by "lifting up" one of the five building blocks (Here we choose $DCell_0[0]$) and doing some rearrangement on the wiring. Such wiring rearrangement is definitely feasible because as long as the fully connective feature at the same level is guaranteed, the topology must be a DCell. Hence, an HCN(4,1) appears in the transformational version of $DCell_1$. This inclusion relation is no by accident, which may be a specific result of the small $DCell_1$. Later, we will see this kind of relation also exists when the DCell topology scales up further.

Based on this internal connection between DCell and HCN, we can promote a similar two-layer indexing scheme, RT-DCell, on the DCell topology just like what we have done for HCN. The concrete method stated here basically

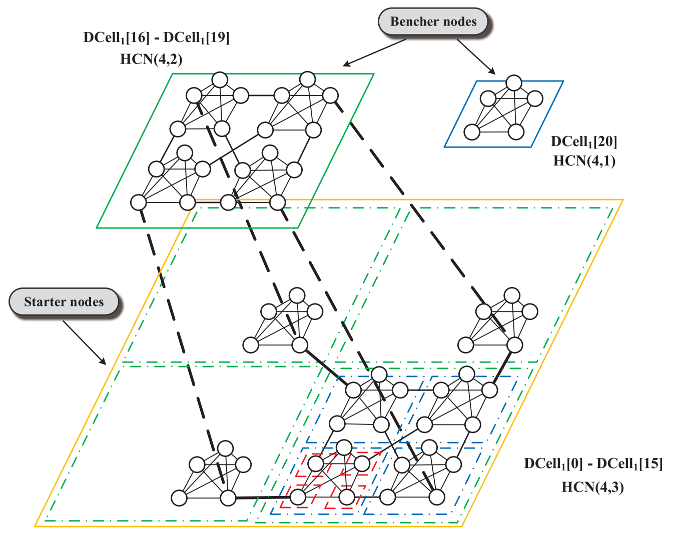


Fig. 7. $DCell_2$ topology (with $n = 4$).

follows the principles of RT-HCN scheme, and also brings new attractive features.

We take Fig. 6b as an illustration for RT-DCell. The part of HCN-like topology in $DCell_1$ is used for the indexing construction, and actually it is an HCN(4,1). The nodes left are treated as common data nodes with no indexing abilities. To be brief, in practice data is distributed all over the nodes in DCell while part of them help to build an indexing scheme. We consider the two kinds of nodes in the following definition.

Definition 4. *Starter nodes are the nodes constituting the highest level of an HCN-like topology as long as they can in a DCell topology. All the nodes left are called Bencher nodes.*

For starter nodes, they make up of the HCN topology. As in Fig. 6b for $n = 4$, a $DCell_1$ contains an HCN(4,1). Hence the concept of Meta-server, representative and potential indexing range for RT-HCN can be used directly without any modification. Data is distributed all over the nodes in DCell, but the global indexing nodes are published to starter nodes only.

For bencher nodes, they are not meaningless roles in RT-DCell scheme. DCell has a characteristic that all the one-level lower DCells are fully connected at any level. It means the HCN-like topology consisting of starter nodes is not a fixed one, and the set of starter nodes is not fixed, either. In Fig. 6b, any four of the five $DCell_0$s can make up of an HCN (4,1). In other words, if some $DCell_0$ unit used to build the HCN(4,1) does not work well due to machine fault or network congestion, it can be replaced by the bencher $DCell_0$, keeping the whole HCN indexing scheme complete and unaffected. We can change the HCN topology correctly and naturally by letting the indexing publishing procedure to produce replicated global indexing nodes and store redundant ones on bencher nodes. This behaviour is just like the bench players substitute the starters, and this cycle is sustainable.

To better understand the roles of bencher nodes in RT-DCell, we let the $DCell_1$ expands a step further.

Fig. 7 shows a $DCell_2$ topology with $n = 4$. It consists of 21 $DCell_1$ numbered from 0 to 20, each of which appears as a rectangular pyramid like the $DCell_1[20]$ at the top right corner. These $DCell_1$s should be fully connected with part of the links represented. A node in Fig. 7 actually represents a $DCell_0$ unit. Based on Definition 4, starter nodes are distributed in $DCell_1[0]$ to $DCell_1[15]$ limited in the yellow range, forming a highest level of HCN(4,3). In general, green range indicates an HCN(4,2), blue range indicates an HCN(4,1), and red range indicates an HCN(4,0). All left nodes from $DCell_1[16]$ to $DCell_1[20]$ are bencher nodes. Interestingly, these bencher nodes can be elegantly divided to form an HCN(4,2) and an HCN(4,1).

The two-layer indexing scheme normally runs on the HCN(4,3) topology, consisting of starter nodes. However, the bencher nodes offers rich alternatives for substitution. No matter what size of a sub-HCN topology in the HCN (4,3) encounters problems, bencher nodes can greatly smooth the bad effects by providing an HCN(4,1) for small scale faults and an HCN(4,2) for larger ones. This back up function from bencher nodes seems to be a customized service for the RT-HCN indexing architecture in RT-DCell. The faulty problem mentioned here is rather vital and makes sense in today's data centers, and need to be treated carefully in the theoretical design if possible. RT-DCell is the first generalization from RT-HCN scheme on other server-centric data center topologies, and we believe it is a highly fault-tolerant and robust indexing architecture.

7 Performance Evaluation

In this section, we evaluate the performance of RT-HCN theoretically and experimentally.

7.1 Theoretical Analysis for Extra Routing Cost

We propose a comparison of the routing cost between RT-HCN indexing routing strategy and the shortest routing strategy here. The shortest routing strategy is the most common routing strategy for ordinary distributed indexing mechanisms. It acts as a reference in the following analysis, reflecting the relative performance of the RT-HCN indexing routing strategy. We take the point query process for illustration.

The RT-HCN indexing routing strategy for point query can be described as four steps. First, a query comes at a random starting node S in HCN(n, h), and S forwards the query to the nearest hth level representative S_{n_h} which is responsible for it. Second, S_{n_h} forwards the query to the nearest $h-1$th level representative $S_{n_{h-1}}$, and this step recursively continues until the 0th level representative is reached. Third, each time a representative is reached, it returns back eligible indexing nodes to the starting node S. Fourth, S search for the final result with the received indexing nodes.

However, if we use the shortest routing strategy for point query, one vital precondition is that every node has buffered all the local indices of all the other nodes. As long as a query comes at a starting node S, it knows which node the final result is located at, with no need of any extra routing. Next, we make the extra routing cost more formalized to get a better understanding.

Following are the computation procedures of three variables defined for the two routing strategies.

1) L_h: the average distance between any starter node S and the nearest $h\text{th}$ level representative S_{n_h} in HCN (n, h). According to whether the starter node falls into the same sub-HCN with the $h\text{th}$ level representative, L_h has two probable values. It equals to L_{h-1} with the probability of $\frac{1}{n}$, and equals to $L_{h-1} + 2^{h-1}$ with the probability of $\frac{n-1}{n}$. L_0 is 0. Thus,

$$\begin{aligned} L_h &= \frac{n-1}{n}(L_{h-1} + 2^{h-1}) + \frac{1}{n}L_{h-1} \\ &= L_{h-1} + \frac{n-1}{n}2^{h-1} = \frac{n-1}{n}(2^h - 1). \end{aligned}$$

2) R_h: the average distance of the routing path which starts from the $h\text{th}$ level representative S_{n_h}, and ends up at the 0th level representative S_{n_0}. Similarly, for any two representatives of two adjacent levels, S_{n_i} and $S_{n_{i-1}}$, the average distance has two probable values according to whether they falls into the same sub HCN. R_h is their summation. Thus,

$$\begin{aligned} R_h &= \sum_{i=1}^{h}[\frac{1}{n}(2^k - 1) + \frac{n-1}{n}2^k] + 1 \\ &= 2^{h+1} - \frac{n+h}{n}. \end{aligned}$$

3) A_h: the average distance between any pair of nodes in HCN(n, h). Similarly, according to whether the two nodes fall into the same sub HCN, A_h has two probable values. In HCN(n, h), the probability of any two nodes falling into the same HCN$(n, h-1)$ is $\frac{\binom{n}{1}\binom{n^h}{2}}{\binom{n^{h+1}}{2}}$, while the probability is $\frac{\binom{n}{2}\binom{n^h}{1}\binom{n^h}{1}}{\binom{n^{h+1}}{2}}$ for the contrary. A_0 is 1. Thus,

$$\begin{aligned} A_h &= \frac{\binom{n}{1}\binom{n^h}{2}}{\binom{n^{h+1}}{2}}A_{h-1} + \frac{\binom{n}{2}\binom{n^h}{1}\binom{n^h}{1}}{\binom{n^{h+1}}{2}}(2L_h + 1) \\ &= \frac{n^h - 1}{n^{h+1} - 1}A_{h-1} + \frac{(n-1)n^h}{n^{h+1} - 1}(2L_h + 1) \\ &= \frac{2n-2}{2n-1}\frac{(2n)^{h+1} - 2(2n)^h + 1}{n^{h+1} - 1} - \\ &\quad \frac{n^{h+1} - 2n^h + 1}{n^{h+1} - 1}. \end{aligned}$$

The L_h here has been defined above.

In fact, A_h stands for the routing cost of the shortest routing strategy, while L_h and R_h stands for the first two steps of the RT-HCN indexing routing strategy, which brings in the major extra routing cost. Consider that HCN(n, h) usually has a small value for h, we take limits of the three variables by tending n to ∞.

1) $\lim_{n\to\infty} L_h = \lim_{n\to\infty} \frac{n-1}{n}(2^h - 1) = 2^h - 1.$
2) $\lim_{n\to\infty} R_h = \lim_{n\to\infty} (2^{h+1} - \frac{n+h}{n}) = 2^{h+1} - 1.$
3) $\lim_{n\to\infty} A_h = \lim_{n\to\infty} (\frac{2n-2}{2n-1}\frac{(2n)^{h+1}-2(2n)^h+1}{n^{h+1}-1} - \frac{n^{h+1}-2n^h+1}{n^{h+1}-1})$ $= 2^{h+1} - 1.$

The relative major extra routing cost can be represented as $\frac{L_h+R_h}{A_h} = 1 + \frac{2^h-1}{2^{h+1}-1} < \frac{3}{2}$. Regarding the vital precondition that the RT-HCN indexing routing strategy stores far less global indexing items than the shortest routing, we can draw a conclusion that this two-layer indexing scheme has perfect space efficiency while it may suffer a little burden for extra routing, which is quite reasonable.

TABLE 3
Experiment Settings

Parameter	Values
Cardinality	800,000, 3,200,000, **1,280,000**
Dimensionality	**2**, 3, 4
Distribution	**Uniform**, Zipfian, Real
Uniform Datasets	**Uniform_2d**, Uniform_3d, Uniform_4d
Skewed Datasets	Zipfian_2d, **Hypsogr**
Selectivity	0.01%, 0.1%
HCN Level	0, 1, **2**
Participants	**RT-HCN**, RT-CAN
M of Local R-tree	10

7.2 Numerical Experiments

To testify the proposed indexing scheme, we evaluate RT-HCN on Amazon's EC2 platform. We implement our indexing system in Python 2.7.9, with bloom filters implemented in C++. The instance each has a 3.3 GHz Turbo Intel Xeon processor, 4 GB memory and 8 GB EBS storage. The network links are 100 Mbps. Experimental network scale ranges from 4, 16 and 64, corresponding to the total number of servers supported by HCN$(4, 0)$, HCN$(4, 1)$, and HCN$(4, 2)$. The experiments involves four synthetic datasets and one real dataset, classified into two groups. One group follows uniform distribution, named as Uniform_2d, Uniform_3d, and Uniform_4d, generated for 2 to 4 dimensions. The other group follows skewed distribution, named as Zipfian_2d and Hypsogr. Zipfian_2d is a two-dimensional dataset, strictly follows zipfian distribution with skewness factor 0.8 in each dimension. Hypsogr is a real dataset obtained from the R-tree Portal, containing 76,999 two-dimensional MBRs. For each dataset, we generate 200000N data points, where N is the number of instances. Particularly, for Hyposogr, three steps are taken to generate 200000 N data items. We first extract from the original MBRs for distinct two-dimensional points. Then we do several minor offsets for each point as new data. Finally, we randomly rearrange the data sequence. Synthetic datasets are all distributed in the range [0, 2.5]. In each experiment concerning one dataset, we initially divide and distribute the whole dataset randomly over the instances, making all of them maintain roughly the same number of data items. Table 3 is the parameters used in our experiments, where default values are in bold.

Experiments are conducted in the following ways. For each network scale and specific dataset, we execute point query, range query, k-NN query, and update-point-mix query. We interact with the running HCN topology with another EC2 instance called client in a centralized manner, mainly in charge of query tasks distribution and information gathering. For each experiment, client instance continuously distribute the query tasks randomly over the HCN clusters. Query time (seconds consumed per 1,000 queries or 500 queries) is used as our performance metric. We execute 10,000 query tasks per kind of query, and record the processing time every 1,000 queries. Each test is repeated 10 times and the average results are used.

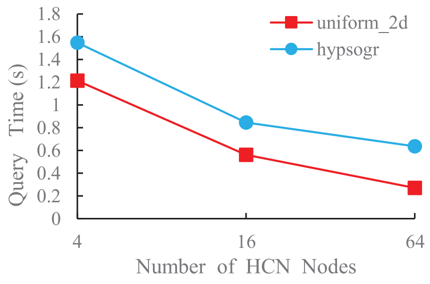


Fig. 8. Performance of point query.

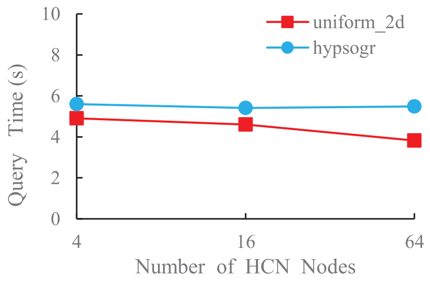


Fig. 10. Performance of k–NN query $(k = 8)$.

7.2.1 Query Performance

We test the performance of three query types on Uniform_2d and Hypsogr datasets.

For point query, the query sets are generated through a random extraction from the target dataset. Considering that the dense portion of a dataset is more likely to be queried, this method of query generation is natural. For RT-HCN, point query is not a special type of range query due to the bloom filters. The filters carry more reliable information about the existence of a point, saving much time wasted in the remote verification. Fig. 8 shows the performance of point query in different network scales. Processing time per 1,000 queries is the metric. Regarded collectively, point query gets acceleration to some extent as network scales up. This is because the bloom filters play great roles in the parallel processing. Considering the data features and indexing publishing rules, each representative will buffer lots of global indices with many interval overlaps. Given a querying point in a uniformly distributed dataset, we have reasons to believe that almost all the servers may store proximal points in the worst case, leading to the verification work spread all over the network. Theoretically, this method cannot make full use of the parallelism of the scaling network. What bloom filters contribute is the sharp cutoff of the candidate locations to be verified. For Uniform_2d, as the network scales from 4 to 16 and 16 to 64, RT-HCN obtains a speedup of 2.15 and 1.64, respectively. As for Hypsogr, the performance trend of point query is similar with the uniform dataset but a little higher in value. This has two reasons. One is the cumulative mapping helps to handle skewed datasets just as the uniform distribution. Another reason is that each point in the real dataset occupies more bytes, which may affect the efficiency of hashing in bloom filters.

For range query, the query sets are generated through different selectivities, which are defined as the percentage of searched space. Fig. 9 shows the performance of range query in different network scales. The metric is processing time per 500 queries. For Uniform_2d and 0.01 percent selectivity, when network scales 16 times up, we observe a total speedup of 1.39 times. A similar performance appears in hypsogr. However, when selectivity increases up to tenfold, the speedup almost disappears. Actually, the first step of RT-HCN querying algorithm concerning four top-level representatives do not benefit from the network scaling up. When range query involves small searched space, at least not all the servers are to be involved, and the parallelism can make some contribution. But as the selectivity increases, the worst case of searching spread all over the network will frequently occur in range query, which makes the overall performance largely depend on the first stage of candidate global index searching, leaving the four starting nodes as bottlenecks. As for RT-CAN with a similar indexing structure, it has different behaviors for range query, which will be shown later.

For k-NN query, query sets are basically the same as point query, with extra parameter k. As stated before, we use an efficient estimation of initial queried range through the density of data in the mapped space. Processing time per 1,000 queries is used. In Fig. 10 , we show the performance in different network scales when $k = 8$. For uniform dataset, performance improves slowly like the range query but the values are better. This is because the k-NN search range is much smaller. Real dataset has a similar performance as the estimation of initial queried range also applies for skewed data due to the work of cumulative mapping. Fig. 11 shows the performance of various values of k from 2 to 8 for the two datasets. Network scale is set to 64. The query time increases when parameter k increases, as the search space increases with k.

7.2.2 Update Performance

For index update experiment, we generate a mixed load of point queries and point insertions. To be specific, our client instance distribute the same querying tasks as in the point query test and generates a random local R-tree point insertion every five queries. To have a better view of update performance, we also do a comparative experiment with no bloom filters. Processing time per 1,000 queries is used.

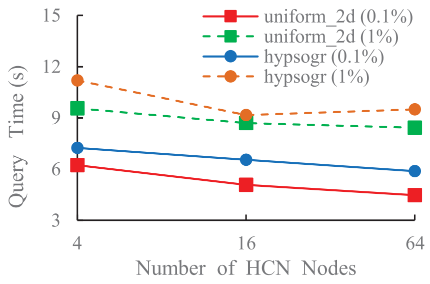


Fig. 9. Performance of range query.

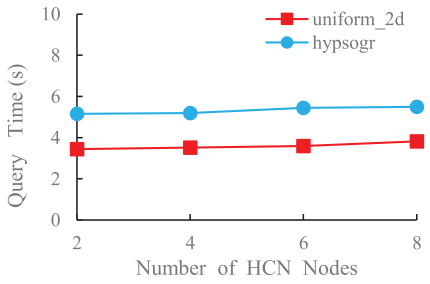


Fig. 11. Performance of k–NN query (64 node).

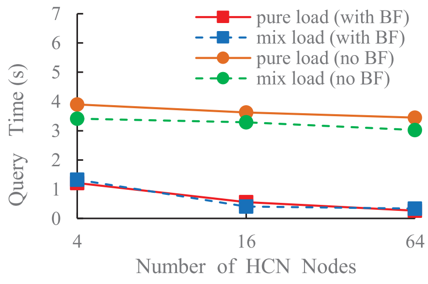


Fig. 12. Performance of update.

Results are shown in Fig. 12, where pure load is just the point query tasks. From the perspective of pure load, we see clearly how fast bloom filters are. In network scale of 4, point query performance of RT-HCN with bloom filters is 3.21 times the no-bloom-filter version. With the network growing up, RT-HCN with bloom filters continuously benefits from the parallelism and keeps gaining speedup, while the performance of no-bloom-filter RT-HCN grows slowly owing to the similar reasons as range query. As for the mixed load containing 20 percent data insertions, there are two kinds of performance. For RT-HCN without bloom filters, updating brings better performance, while for RT-HCN with bloom filters, they are very close. When filter is not used, the global updating is node-oriented, concerning the deletion and republishing of related global nodes through at most four routing messages, faster than the point query. When filter is used, the global updating is bit-array-oriented, concerning the new positions to be set to 1 of related published bloom filters through at most four routing messages. However, the update of published bloom filters still needs a linear scan as query. In all, RT-HCN can handle both queries and updates efficiently.

7.2.3 Multi-Dimensional Performance

We also conduct a simple experiment to verify the indexing ability for other dimensions. Based on the designing principles for dimension greater than 2, the HCN levels are allocated for extra dimensions. Thus, network scale of 4 only supports Uniform_2d. As the the network scales up for one level, a new dataset with one more dimension is supported. We present the results of point query for three uniform datasets in Fig. 13. Processing time per 1,000 queries is the metric. Undoubtedly, two-dimension has the best performance, since RT-HCN works in two different styles for two-dimension and other dimensions. When doing queries for dimensions greater than 2, lots of partition work needs to be done before locating a region for further two-dimensional processing.

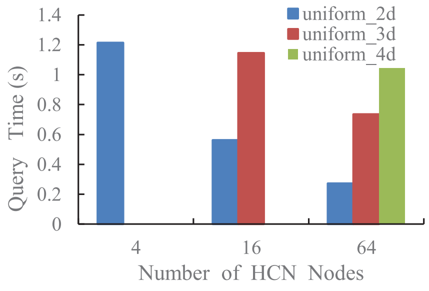


Fig. 13. Multi-dimension.

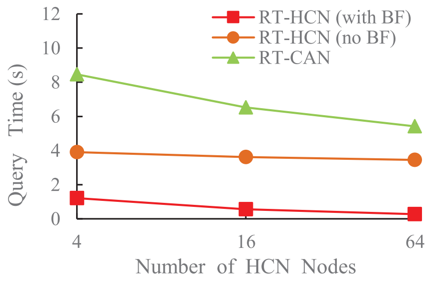


Fig. 14. Comparison of point query (uniform dataset).

7.2.4 Comparison with RT-CAN

RT-CAN is a well known two-layer indexing schemes on P2P networks, sharing similar designing philosophy with RT-HCN. It is the most relevant work we have learned about so far that can be experimentally implemented to compare with for our scheme. Hence we actually make RT-CAN run on the same batch of EC2 instances as RT-HCN, treating them to be logically connected like the P2P topology. We implement the main querying functions for RT-CAN in python, except for the index tuning mechanism, because we do not involve dynamic updating behaviors but static point and range querying tasks in this experiment. Three other notes need to be declared here. First, our local R-tree has at least 5 or 6 levels with branch limitation M set to 10, which is different from 3 levels in the RT-CAN paper. Second, the parameter R_{max} is set as the half of the side length of each CAN node zone, since the selection of R_{max} in RT-CAN paper has no common sense for different sizes of local R-tree. Third, we published the antepenult level nodes of local R-tree as global index, rather than the last but one level. We compare RT-HCN and RT-CAN on Uniform_2d and Zipfian_2d for different network scales. Point and range query are both tested. Processing time per 1,000 queries is used for point query while for range query, the metric is per 500 queries.

Figs. 14 and 15 show their performances of point and range query for Uniform_2d, respectively. In Fig. 14, we find that RT-CAN does not act very well, even slower than the RT-HCN without bloom filters. However, we do not regard this as the real performance of RT-CAN. Having checked out the number of global indices maintained on each server, we discover that each RT-CAN server buffers global indices 10 times more than RT-HCN server on average, which slow down the first stage of two-layer indexing. The situation may result from our selections of indexing publishing levels and R_{max}. We try different settings and never get a remarkable improvement. From another point of view, as the network grows up, the speedup of RT-CAN

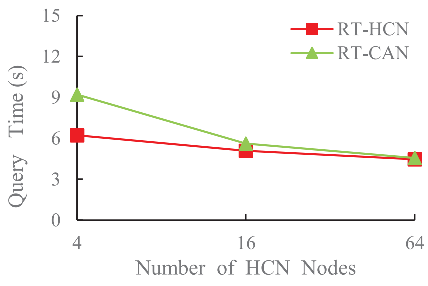


Fig. 15. Comparison of range query (uniform dataset).

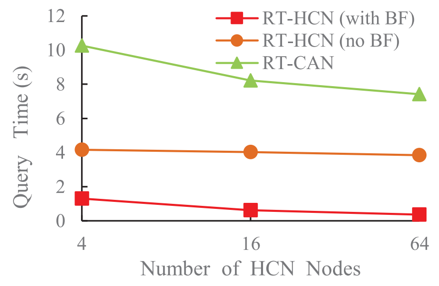


Fig. 16. Comparison of point query (zipfian dataset).

is better than RT-HCN, because RT-CAN has no bottleneck when doing global indexing search like the four top-level representatives in RT-HCN. But the number of global index buffered in RT-CAN servers is really hard to control due to its arbitrary redundancy of published nodes and often stays at a high value, its absolute query performance is limited. As for range query in Fig. 15, a similar pattern appears. Fig. 16 shows the point query performance for Zipfian_2d. For RT-HCN, cumulative mapping has made the zipfian distribution equivalent to the uniform distribution, thus RT-HCN has almost the same querying performance as Fig. 14. For RT-CAN, it does not offers any preprocessing mechanisms for data skewness. It is just passively adapted to handle skewness, thus suffering a performance degradation.

In short, RT-CAN indeed perform better than RT-HCN in regard of performance speedup as the network scale increases. However, the absolute performance depends largely on the selections of index publishing levels and R_{max} which decide the total number of published global indexing nodes. Moreover, RT-CAN is bad at processing skewed data. For RT-HCN, we equip it with bloom filters to bypass the top-level bottleneck and regain speedup for point query, achieving tremendous performance improvement. With the help of representative mechanisms and specific selection of publishing levels, the number of published global indices is controlled at a relatively low level, which also point out another significant fact: considering the enormous redundancy of published nodes, bloom filters are not appropriate for RT-CAN. Last but not least, RT-HCN use cumulative mapping to efficiently process data skewness.

8 Conclusion

In this paper, we propose an indexing scheme named RT-HCN for multi-dimensional query processing in data centers, which are the infrastructures for building cloud storage systems and are interconnected using a specific data center network. RT-HCN is a two-layer indexing scheme, which integrates HCN-based routing protocol [27] and the R-Tree based indexing technology, and is partially distributed on each server. Based on the characteristics of HCN, we present a specialized mapping technique to improve global index allocation in the network, resulting query-efficiency and load-balancing for the cloud system. We also combine practical techniques in face of data skewness and querying false positives, greatly increasing the adaptability and querying performance of RT-HCN. We prove theoretically that RT-HCN is both query-efficient and space-efficient, by which each server will only maintain a constrained number of indices while a large number of users can concurrently process queries with low routing cost. We also give an insight into the inner relation between the HCN and other data center topologies, and apply the two-layer indexing scheme to the DCell topology in a generalized way. We compare our design with RT-CAN [8], a similar design for traditional P2P network. Experiments validate the efficiency of our proposed scheme and depict its potential implementation in data centers.

Acknowledgments

This work has been supported in part by the China 973 project (2014CB340303), National Natural Science Foundation of China (Grant numbers 61202024, 61472252, 61133006, 61272443, 61428204, 61473109), the Opening Project of Key Lab of Information Network Security of Ministry of Public Security (The Third Research Institute of Ministry of Public Security) Grant number C15602, the Opening Project of Baidu (Grant number 181515P005267), the Scientific Innovation Act of STCSM (No.13511504200, 15JC1402400), Singapore NRF (CREATE E2S2), and the EU FP7 CLIMBER project (No. PIRSES-GA-2012-318939). The authors would like to thank Mr. Fan Li and Mr. Wanchao Liang for their contributions on the early versions of this paper. They would also thank Prof. Wenfei Fan and Prof. Deke Guo for sharing some vital materials to inspire our design, and the anonymous reviewers for their valuable comments to improve the quality of their paper. Xiaofeng Gao is the corresponding author.

References

[1] S. Ghemawat, H. Gobioff, and S. T. Leung, "The google file system," *ACM SIGOPS Oper. Syst. Rev.*, vol. 37, pp. 29–43, 2003.
[2] G. DeCandia, D. Hastorun, M. Jampani, G. Kakulapati, A. Lakshman, A. Pilchin, S. Sivasubramanian, P. Vosshall, and W. Vogels, "Dynamo: Amazon's highly available key-value store," *ACM SIGOPS Oper. Syst. Rev.*, vol. 41, pp. 205–220, 2007.
[3] A. Lakshman and P. Malik, "Cassandra: A decentralized structured storage system," in *Proc. ACM SIGOPS Oper. Syst. Rev.*, vol. 44, pp. 35–40, 2010.
[4] D. Beaver, S. Kumar, H. C. Li, J. Sobel, and P. Vajgel, "Finding a needle in haystack: Facebook's photo storage," in *Proc. 9th USENIX Conf. Oper. Syst. Des. Implementation*, 2010, vol. 10, pp. 1–8.
[5] J. Baker, C. Bond, J. C. Corbett, J. J. Furman, A. Khorlin, J. Larson, J. Leon, Y. Li, A. Lloyd, and V. Yushprakh, "Megastore: Providing scalable, highly available storage for interactive services," in *Proc. Conf. Innovative Data Syst. Res.*, 2011, vol. 11, pp. 223–234.
[6] J. C. Corbett, J. Dean, M. Epstein, A. Fikes, C. Frost, J. J. Furman, S. Ghemawat, A. Gubarev, C. Heiser, P. Hochschild, W. Hsieh, S. Kanthak, E. Kogan, H. Li, A. Lloyd, S. Melnik, D. Mwaura, D. Nagle, S. Quinlan, R. Rao, L. Rolig, Y. Saito, M. Szymaniak, C. Taylor, R. Wang, and D. Woodford, "Spanner: Googles globally distributed database," *ACM Trans. Comput. Syst.*, vol. 31, no. 3, p. 8, 2013.
[7] S. Wu and K. L. Wu, "An indexing framework for efficient retrieval on the cloud," *IEEE Data Eng. Bull.*, vol. 32, no. 1, pp. 75–82, Jan. 2009.
[8] J. Wang, S. Wu, H. Gao, J. Li, and B. C. Ooi, "Indexing multi-dimensional data in a cloud system," in *Proc. ACM SIGMOD Int. Conf. Manage. Data*, 2010, pp. 591–602.
[9] S. Wu, D. Jiang, B. C. Ooi, and K. L. Wu, "Efficient b-tree based indexing for cloud data processing," *Proc. VLDB Endowment*, vol. 3, no. 1-2, pp. 1207–1218, 2010.
[10] G. Chen, H. T. Vo, S. Wu, B. C. Ooi, and M. T. Özsu, "A framework for supporting dbms-like indexes in the cloud," *Proc. VLDB Endowment*, vol. 4, no. 11, pp. 702–713, 2011.
[11] H. V. Jagadish, B. C. Ooi, and Q. H. Vu, "Baton: A balanced tree structure for peer-to-peer networks," in *Proc. 31st Int. Conf. Very Large Data Bases*, 2005, pp. 661–672.

[12] S. Ratnasamy, P. Francis, M. Handley, R. Karp, and S. Shenker, “A scalable content-addressable network,” in *Proc. Conf. Appl., Technol., Archit. Protocols Comput. Commun.*, 2001, pp. 161–172.

[13] M. Al-Fares, A. Loukissas, and A. Vahdat, “A scalable, commodity data center network architecture,” in *Proc. ACM SIGCOMM Conf. Data Commun.*, 2008, pp. 63–74.

[14] A. Greenberg, J. R. Hamilton, N. Jain, S. Kandula, C. Kim, P. Lahiri, D. A. Maltz, P. Patel, and S. Sengupta, “Vl2: A scalable and flexible data center network,” in *Proc. ACM SIGCOMM Conf. Data Commun.*, 2009, pp. 51–62.

[15] M. Walraed-Sullivan, A. Vahdat, and K. Marzullo, “Aspen trees: Balancing data center fault tolerance, scalability and cost,” in *Proc. 9th ACM Conf. Emerging Netw. Exp. Technol.*, 2013, pp. 85–96.

[16] C. Guo, H. Wu, K. Tan, L. Shi, Y. Zhang, and S. Lu, “Dcell: A scalable and fault-tolerant network structure for data centers,” *ACM SIGCOMM Comput. Commun. Rev.*, vol. 38, pp. 75–86, 2008.

[17] D. Li, C. Guo, H. Wu, and K. Tan, “Ficonn: Using backup port for server interconnection in data centers,” in *Proc. IEEE INFOCOM*, 2009, pp. 2276–2285.

[18] D. Li, C. Guo, H. Wu, K. Tan, Y. Zhang, S. Lu, and J. Wu, “Scalable and cost-effective interconnection of data-center servers using dual server ports,” *IEEE/ACM Trans. Netw.*, vol. 19, no. 1, pp. 102–114, Feb. 2011.

[19] Y. Liao, D. Yin, and L. Gao, “Dpillar: Scalable dual-port server interconnection for data center networks,” in *Proc. IEEE 19th Int. Conf. Comput. Commun. Netw.*, 2010, pp. 1–6.

[20] D. Li and J. Wu, “On the design and analysis of data center network architectures for interconnecting dual-port servers,” in *Proc. IEEE INFOCOM*, 2014, pp. 1851–1859.

[21] C. Guo, G. Lu, D. Li, H. Wu, X. Zhang, Y. Shi, C. Tian, Y. Zhang, and S. Lu, “Bcube: A high performance, server-centric network architecture for modular data centers,” in *Proc. ACM SIGCOMM Comput. Commun. Rev.*, vol. 39, pp. 63–74, 2009.

[22] H. Wu, G. Lu, D. Li, C. Guo, and Y. Zhang, “Mdcube: A high performance network structure for modular data center interconnection,” in *Proc. 5th Int. Conf. Emerging Netw. Exp. Technol.*, 2009, pp. 25–36.

[23] D. Li, M. Xu, H. Zhao, and X. Fu, “Building mega data center from heterogeneous containers,” in *Proc. 19th IEEE Int. Conf. Netw. Protocols*, 2011, pp. 256–265.

[24] X. Liu, S. Yang, L. Guo, S. Wang, and H. Song, “Snowflake: A new-type network structure of data center,” *Jisuanji Xuebao (Chinese J. Comput.)*, vol. 34, no. 1, pp. 76–86, 2011.

[25] Z. Ding, D. Guo, X. Liu, X. Luo, and G. Chen, “A mapreduce-supported network structure for data centers,” *Wiley Concurrency Comput.: Practice Exp.*, vol. 24, no. 12, pp. 1271–1295, 2012.

[26] D. Guo, T. Chen, D. Li, Y. Liu, X. Liu, and G. Chen, “Bcn: Expansible network structures for data centers using hierarchical compound graphs,” in *Proc. IEEE INFOCOM*, 2011, pp. 61–65.

[27] D. Guo, T. Chen, D. Li, M. Li, Y. Liu, and G. Chen, “Expandable and cost-effective network structures for data centers using dual-port servers,” *IEEE Trans. Comput.*, vol. 62, no. 7, pp. 1303–1317, Jul. 2013.

[28] M. Lupu, B. C. Ooi, and Y. C. Tay, “Paths to stardom: Calibrating the potential of a peer-based data management system,” in *Proc. ACM SIGMOD Int. Conf. Manage. Data*, 2008, pp. 265–278.

[29] X. Gao, B. Li, Z. Chen, and M. Yin, “Ft-index: A distributed indexing scheme for switch-centric cloud storage system,” in *Proc. IEEE IEEE Int. Conf. Commun.*, 2015, pp. 301–306.

[30] Y. Liu, X. Gao, and G. Chen, “Design and optimization for distributed indexing scheme in switch-centric cloud storage system,” in *Proc. IEEE Symp. Comput. Commun.*, 2015, pp. 804–809.

[31] A. Guttman, “R-trees: A dynamic index structure for spatial searching,” *ACM SIGMOD Rec.*, vol. 14, pp. 47–57, 1984.

[32] N. Beckmann, H. P. Kriegel, R. Schneider, and B. Seeger, “The r*-tree: An efficient and robust access method for points and rectangles,” *ACM SIGMOD Rec.*, vol. 19, pp. 322–331, 1990.

[33] S. Šaltenis, C. S. Jensen, S. T. Leutenegger, and M. A. Lopez, “Indexing the positions of continuously moving objects,” in *Proc. ACM SIGMOD Int. Conf. Manage. Data*, 2000, pp. 331–342.

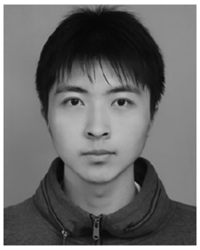
[34] B. Kao, S. D. Lee, D. W. Cheung, W. S. Ho, and K. F. Chan, “Clustering uncertain data using Voronoi diagrams and r-tree index,” *IEEE Trans. Knowl. Data Eng.*, vol. 22, no. 9, pp. 1219–1233, Sep. 2010.

[35] K. Zheng, X. Zhou, P. C. Fung, and K. Xie, “Spatial query processing for fuzzy objects,” *The Int. J. Very Large Data Bases*, vol. 21, no. 5, pp. 729–751, 2012.

[36] R. Zhang, J. Qi, M. Stradling, and J. Huang, “Towards a painless index for spatial objects,” *ACM Trans. Database Syst.*, vol. 39, no. 3, p. 19, 2014.

[37] A. Border and M. Mitzenmacher, “Network applications of bloom filters: A survey,” *Internet Math.*, vol. 1, no. 4, pp. 485–509, 2003.

[38] D. Guo, J. Wu, H. Chen, and X. Luo, “Theory and network applications of dynamic bloom filters,” in *Proc. IEEE INFOCOM*, 2006, pp. 1–12.



Yang Hong received the BS degree in computer science and technology from Nanjing University in 2014. He is a postgraduate student in the Department of Computer Science and Engineering at Shanghai Jiao Tong University, P.R. China. His research interests include data communication and engineering, data center networks, and distributed systems. Currently, he is working in the research group of Data Communication and Engineering under the supervision of Dr. Xiaofeng Gao.

Qiwei Tang is an undergraduate student in the Department of Computer Science and Engineering at Shanghai Jiao Tong University, China. His research interests include data communication and engineering and data center networks virtual reality. Currently, he is working in the research group of Data Communication and Engineering under the supervision of Dr. Xiaofeng Gao.

Xiaofeng Gao received the BS degree in information and computational science from Nankai University, China, the MS degree in operations research and control theory from Tsinghua University, China, and the PhD degree in computer science from the University of Texas at Dallas, Richardson. She is an associate professor with the Department of Computer Science and Engineering, Shanghai Jiao Tong University, China. Her research interests include data engineering, wireless communications, and combinatorial optimizations. She has produced a number of research papers in indexing and data retrieval, distributed database systems, and algorithm design.

Bin Yao received the BS degree and the MS degree in computer science from the South China University of Technology in 2003 and 2007, respectively, and the PhD degree in computer science from the Florida State University in 2011. He has been an associate professor in the Department of Computer Science and Engineering, Shanghai Jiao Tong University since 2011. His research interests are management and indexing of large databases, and scalable data analytics.

Guihai Chen obtained the BS degree from Nanjing University, ME engineering from Southeast University, and the PhD from the University of Hong Kong. He visited the Kyushu Institute of Technology, Japan, in 1998 as a research fellow, and the University of Queensland, Australia, in 2000 as a visiting professor. He is a distinguished professor with the Department of Computer Science, Shanghai Jiao Tong University. He has published more than 280 papers in peer-reviewed journals and refereed conference proceedings in the areas of wireless sensor networks, high-performance computer architecture, peer-to-peer computing, and performance evaluation. He is a senior member of the IEEE.

Shaojie Tang received the PhD degree in computer science from the Illinois Institute of Technology in 2012. He is currently an assistant professor of the Naveen Jindal School of Management at the University of Texas at Dallas. His research interest includes social networks, mobile commerce, game theory, e-business, and optimization. He received the Best Paper Awards in ACM MobiHoc 2014 and IEEE MASS 2013. He also received the ACM SIGMobile service award in 2014. He served in various positions (as a chair and TPC member) at numerous conferences, including ACM MobiHoc, IEEE INFOCOM, IEEE ICDCS, and IEEE ICNP.