Advance Access publication on 20 December 2018
doi:10.1093/comjnl/bxy132

Accelerate Data Retrieval by Multi-Dimensional Indexing in Switch-Centric Data Centers

Xinjian Luo, Xiaofeng Gao* and Guihai Chen

Shanghai Key Laboratory of Scalable Computing and Systems, Department of Computer Science and Engineering, Shanghai Jiao Tong University, Shanghai 200240, China
**Corresponding author: gao-xf@cs.sjtu.edu.cn*

Data centers, receiving increased attention in data management and analysis communities, have posed new challenges in data-intensive applications, among which efficient querying processing holds a critical position. To accelerate the efficiency of multi-dimensional data retrieval, we propose a distributed multi-dimensional indexing scheme for switch-centric data centers in this paper. We first propose FR-Index, a two-layer indexing system integrating both Fat-tree topology and R-tree indexing structure. In the lower layer, each server indexes the local data with R-tree, while in the upper layer the distributed global index depicting an overview of the whole dataset. Based on the Fat-tree topology, we design a specific indexing space partitioning and mapping strategy for efficient global index maintenance and query processing. Furthermore, we develop a cost model to dynamically update FR-Index. Experiments on Amazon's EC2 platform, comparing FR-Index with RT-CAN and RB-Index, show that the proposed indexing schema is scalable, efficient and lightweight, which can significantly promote the efficiency of query processing in data centers.

Keywords: multi-dimensional data; distributed index; switch-centric data center

Received 13 April 2018; revised 7 September 2018; editorial decision 21 October 2018
Handling editor: Yannis Manolopoulos

1. INTRODUCTION

Massive storage systems have received increasing attention in both academia and industry nowadays. Various distributed storage systems, such as GFS [1], Cassandra [2], Dynamo [3], were designed to satisfy the increasing requirements of data-intensive applications.

An attractive challenge for large-scale distributed storage systems is how to retrieve specified data from massive dataset efficiently. Empirically, designing appropriate and effective index is a typical solution for this challenge. Refs [4–6] designed distributed indexing schemes in P2P systems. Each of them deploys a P2P topology on the distributed server cluster as an overlay for data mapping and routing queries. However, the underlying topologies of P2P networks are actually logically defined, which means the nodes are scattered geographically and the connections between nodes are unstable, leading to unreliable services [7].

In recent years, as a type of infrastructure, *Data Centers* are playing increasingly vital role in cloud services. In data centers, a great number of servers are organized into a specific topology by data center network (DCN). For example, Cisco employs fat-tree topology to support efficient communications in its data centers [8]. With the rapid development of data centers, some critical issues for performance optimization have been proposed. One of them is to construct efficient indexing schemes for storage systems deployed on data centers. Different from the logical topologies in P2P networks, however, DCN defines specific physical topologies, connecting nodes with a strict manner. Thus, it is impractical to simply transplant indexing schemes in P2P networks to data centers.

Gao *et al.* [7, 9–11] have proposed some network-aware indexing techniques, which are typically two-layer indexing schemes. In two-layer indexing framework, each server indexes its data via a *local index* structure, such as R-tree. Then some index nodes in each server are selected and published among the cluster as *global index*. Although those works presented excellent proposals, they have not considered

the multi-dimensional indexing in switch-centric DCN's. In fact, the multi-dimensional indexing strategies are widely adopted by many commercial applications. A simple application is the photo search via the metadata tags. A photo object could be specifically expressed as $\{t_1, t_2, \ldots, t_n\}$, where t_i denotes its features and geographic information, such as latitude, longitude, topic, color, size, etc. Typical queries could be searching photos in a specific location or with specific features. To efficiently support these queries, a multi-dimensional indexing strategy designed for data centers is critical.

This paper targets the problem of constructing distributed multi-dimensional two-layer indexing scheme on the Fat-tree topology of switch-centric DCN. There are two main challenges to construct the multi-dimensional indexing scheme in Fat-tree: first, how to partition and map the index space, a hypercube, to the servers interconnected with tree-like topology; and second, how to publish and update the global index distributed among the cluster.

We designed **FR-Index**, a two-layer indexing system fully taking advantage of the Fat-Tree topology and R-tree indexing technology. In FR-Index, each node plays two roles, i.e. storage node and overlay node. In the lower layer, each server, acting as storage node, stores the multi-dimensional data and indexes them with R-tree structure. In the upper layer, the indexing space of the entire datasets is partitioned and designated to each server, and the global index is selected from each servers' local index and scattered among servers based on their overlay position. The distributed global index depicts an overview of the whole dataset.

To partition and map the multi-dimensional indexing space into Fat-tree, we selectively reduce and divide the index dimensionality. Efficient query processing methods are proposed to accelerate data retrieval based on this indexing system. A cost model based on two-state Markov chain model and Fat-tree routing protocol are proposed to dynamically update FR-Index. We also conducted extensive experiments on Amazon EC2 platform to evaluate the performance of our proposal. Comparisons between FR-Index and RT-CAN/RB-Index illustrate the availability and efficiency of FR-Index. To summarize, the contributions of this paper are concluded as follows:

- We propose a distributed two-layer indexing scheme, FR-Index, for efficient data retrieval in switch-centric data centers with tree-like topology. FR-Index fully takes advantage of the Fat-tree routing protocol and R-tree structure to accelerate data retrieval.
- We design a novel indexing space partitioning and mapping strategy to publish the global index among the cluster. We propose a cost model based on Markov model and Fat-tree routing protocol to dynamically maintain and update FR-Index.
- We conduct experiments on Amazon EC2 platform to validate the performance of FR-Index. Comparisons between our scheme and RT-CAN/RB-Index exhibit the efficiency and availability of FR-Index.

The remainder of this paper is organized as follows. In Section 2, we introduce some related work. The architecture of FR-Index system is discussed in Section 3, followed by the query processing strategy in Section 4. Section 5 discussed the index maintenance and updating in FR-Index. By comparing with RT-CAN and RB-Index, we evaluated the performance of FR-Index in Section 6. In the end, we summarized our work in Section 7.

2. RELATED WORK

In this section, we first introduce the Fat-tree topology in DCNs, then introduce the distributed two-layer indexing framework together with two critical design criteria: index completeness and uniqueness.

2.1. Fat-tree topology

As the core infrastructure in cloud systems, data centers employ switches and high-speed links to connect a great deal of servers, among which DCN play an important role in communications and computing. Compared to traditional P2P networks, DCN structures are designed to satisfy higher requirements such as high scalability, high availability, energy-saving and robustness.

In [12], the DCN topologies are divided into three categories based on the structural features: server-centric, switch-centric and enhanced architectures. *Switch-centric* architecture, which could be further divided into flat, tree-like and unstructured topologies, is a main category in DCN architectures. In this type of DCN, the switches are enhanced for networking and routing requirements, while the functions of servers remain unmodified. In *server-centric* DCN, however, low-end switches are simply used for message forwarding, while servers are used for networking and routing. Since servers are programmable and powerful, more intellective topologies (usually recursively defined) are designed for server-centric DCNs. As for the *enhanced* architectures, wireless or optical devices are generally employed for capacity enhancement.

As a subclass of switch-centric DCN's, tree-like DCN's connect devices by links similar to a multi-rooted tree. Switches are divided into multiple layers, and servers are linked to the bottom-layer switches. Most tree-like DCN's tend to divide lower layer switches and servers into some substructures, like *pod* in **Fat-tree**.

A k-pod Fat-tree consists of three layers of k-port switches. In each pod, the $k/2$ aggregation layer switches and the $k/2$ edge layer switches interconnect as a complete bipartite graph. Every switch in aggregation layer connects to $k/2$ switches in core layer. Every switch in the edge layer connects to $k/2$ servers. Thus, a k-pod Fat-tree can support

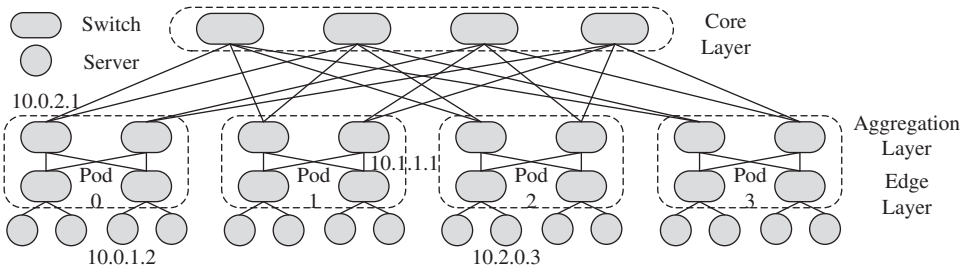

FIGURE 1. A fat-tree topology with four pods.

connecting $k^3/4$ servers. Ref. [8] introduces different IP addressing rules for switches and servers. For pod switches, the form *10.pod.swi.1* acts as their IP addresses, where $pod \in [0, k-1]$ denotes the pod number, and *swi* denotes the position of the switch in the pod (in $[0, k-1]$, starting from left to right, bottom to top). The address of a server is *10.pod.swi.ID* where *pod* and *swi* follow the address of the edge switch which the server connects, and *ID* (in $[2, k/2+1]$, starting from left to right) denotes the server's position in that subnet. Figure 1 illustrates a fat-tree topology with four pods and examples of the addressing scheme. Because of its simple architecture, high availability and strong robustness, fat-tree has been employed in many enterprises' data centers such as Cisco [12].

2.2. Distributed indexing systems

2.2.1. *Related multi-dimensional indexing schemes*

The performance of data retrieval in traditional cloud systems relies heavily on parallelism, i.e. parallel scanning all the data. Such methods are simple but inefficient and could easily incur immense network traffic. In recent years, many studies have been conducted to design efficient indexing schemes for distributed systems. For example, Sioutas *et al.* designed an ART^+ structure [21] to support efficient range queries on large-scale, decentralized environments. The outer level of ART^+ is an dynamic and fault-tolerant ART structure [22], and the cluster-node in ART^+ is organized into a D^3-tree [23], a enhanced dynamic version of D^2-Tree [24]. ART^+ is highly fault-tolerant and can improve the overall query efficiency meanwhile achieving load-balancing.

Kokotinis *et al.* [25] proposed a NSM-tree index scheme based on M-tree and MapReduce framework. M-tree [26] works well for efficient range and kNN queries in many conditions. While its structure is similar to R-Tree, M-Tree could have large overlap areas and this problem is still hard to eliminate for now.

Space-filling curve is another popular technique that can map multi-dimensional data into 1D space. Multi-dimensional indexing schemes based on space-filling curve, such as Squid [27], CISS [28] and SCRAP [29], have been well studied.

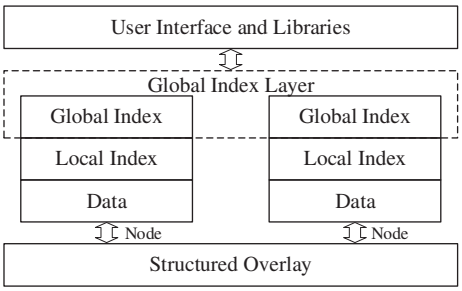

FIGURE 2. The distributed two-layer framework.

These schemes can achieve good locality and load balancing in low-dimensional scenarios, but query locality can become very poor for even three dimensions because of the curse of dimensionality [29]. In addition, schemes of space-filling curve are generally built on P2P overlay networks. In data centers storing large-scale data items, indexing schemes with space-filling curves could cause query congestion because of its feature of dimensionality reduction [28].

In FR-Index, we adopt R-tree as the underlying indexing structure, given that R-tree is generally used in many multi-dimensional applications, offering superior performance and flexibility.

2.2.2. *Distributed two-layer indexing schemes*

To reduce the network traffic and index maintenance overhead, Wu and Wu [6] proposed a two-layer indexing framework based on overlay network, shown in Fig. 2, for efficiently data retrieval in cloud systems.

In the indexing framework, each server in cloud system plays two roles, namely, overlay node and storage node. To build a distributed two-layer indexing system, each server first builds a local index for its data; then, some index entries in each server are selected and disseminated in cluster as the global index. To process a query, the cluster first searches global index to find a responsible server S, and then forwards the query to S. Whereafter, data retrieval is processed through the local index of S and the results are returned to user. The framework supports most of existing index structures and is compatible with many distributed file systems like Dynamo

TABLE 1. Comparison between FR-Index and related two-layer indexing studies.

Year	Scheme	Index structure	Data dimension	Overlay	Network type
2010	CG-Index [5]	B^+-tree	1	BATON [13]	P2P
2010	RT-CAN [14]	R-tree	>1	CAN [15]	P2P
2011	QT-Chord [16]	quad-tree	>1	Chord [17]	P2P
2011	TLB-Index [18]	bitmap	1	BATON [13]	P2P
2015	RB-Index [9]	R-tree	>1	BCube [19]	DCN
2015	FT-Index [10]	B^+-tree	1	Fat-tree [8]	DCN
2015	U^2-Tree [11]	B^+-tree	1	Tree-like Topologies	DCN
2016	RT-HCN [7]	R-tree	>1	HCN [20]	DCN
2018	FR-Index	R-tree	>1	Fat-tree [8]	DCN

[3]. Some recent indexing schemes based on this framework are presented in Table 1.

One key observation of Table 1 is that roughly before 2015, most indexing schemes concentrated on P2P networks, which is not surprising since the two-layer indexing framework mentioned before was proposed based on the P2P overlay networks. After 2015, however, designing efficient indexing schemes in DCN became an important issue. Hong *et al*. [7] and Gao *et al*. [9] proposed different two-layer distributed indexing schemes for HCN (RT-HCN) and Bcube (RB-Index), respectively. Though FR-Index, RT-HCN and RB-Index are all two-layer schemes, there exist significant differences between FR-Index and the other two schemes.

Hierarchical irregular compound network (HCN) [20] is a recursively defined architecture based on n-port switches and dual-port servers. Hong *et al*. [7] designed 2D RT-HCN based on $HCN(4, n)$ and provided a perspective of indexing extension for data with three or more dimensions. However, since $HCN(4, n)$ is a typical 2D overlay network, the perspective for indexing extension seems too complicated and impractical. For FR-Index, however, datasets with any number of dimensions can be supported, which is more scalable and practical.

RB-Index [9] is another two-layer indexing scheme designed for Bcube, where a $Bcube_k$ is constructed by connecting n lower level $Bcube_{k-1}$ with n^k n-port switches. In RB-Index, the index partitioning scheme constructed in $Bcube_k$ can support datasets with up to $k+1$ dimensions. An RB-Index system is composed of $n+1$ indexing spaces, where one $(k+1)$-dimensional space is for $Bcube_k$ and n k-dimensional spaces are for n $Bcube_{k-1}$ s. Such architecture can make RB-Index support up to $((n+1)k+1)$-dimensional data. However, if the number of data dimensions is less than k, extra zero vector will be appended to the data, leading to plenty of extra storage cost; if the number of data dimensions is more than $((n+1)k+1)$, some dimensions will not be indexed, leading to a full cluster scan which could greatly degrade the entire system performance. In FR-Index, we employ a set of indexing instances, with each instance in charge of three dimensions, to cover all the data dimensions, which is not only more flexible but also more efficient. Comparisons between RB-Index and FR-Index will be discussed in Section 6.4.

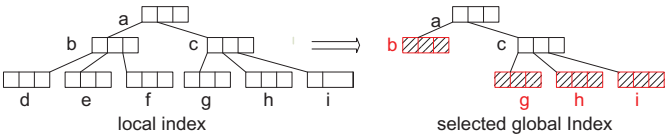

FIGURE 3. Index completeness and uniqueness.

2.2.3. *Index completeness and uniqueness*

In FR-Index, when selecting a portion of local index nodes as the global index, completeness and uniqueness must be guaranteed [14].

DEFINITION 1 (Index uniqueness). *Suppose server N_i selects a set of index nodes S_{lg} from its local index S_l to publish as the global index. To satisfy the index completeness, if and only if for any data item d_i stored in server N_i, d_i is contained in one of the nodes of S_{lg}.*

DEFINITION 2 (INDEX UNIQUENESS). *Suppose server N_i selects a set of index nodes S_{lg} from its local index S_l to publish as the global index. To satisfy the index uniqueness, if and only if for any index node $n_i \in S_l$ and its ancestor node $n_j \in S_l$, $n_i \in S_{lg} \rightarrow n_j \notin S_{lg} \wedge n_j \in S_{lg} \rightarrow n_i \notin S_{lg}$.*

Take Fig. 3 for example. Suppose the tree in the left part, say, a R-tree, is the local index S_l of server N_i. A set of nodes in S_l needed to be selected and published in the cluster as global index. To satisfy the index completeness and uniqueness, we selected these shaded nodes with red border (nodes $\{b, g, h, i\}$) in the right tree as the global index.

3. FR-INDEX

All servers in the data center participate in constructing and maintaining the indexing instances of FR-Index. In addition, we set an individual server as a historical data collector (called **Collector** for short) of FR-Index. The collector will collect some historical data as the basis of some decisions we made.

An FR-Index system is composed of a set of *index instance*s denoted by $\mathcal{I} = \{I_1, I_2, \ldots, I_w\}$. An index instance I_i indexes an 'indexing space' denoted by $I_i.space$, which is composed of several selected dimensions of the dataset. The FR-Index collector generates $I_i.space$ and informs all servers of $I_i.space$. Each server builds a local R-tree to index its local data on the dimensions contained by $I_i.space$. Then each server publishes a portion of its local R-tree index nodes among the cluster to compose the global index. Thus, we get distributed global index and each server maintains a portion of the global index. An index instance I_i can be regarded as a combination of all local indexes and global index which are built on $I_i.space$.

Suppose that a dataset is composed of n attributes. Every item in the dataset can be regarded as an object in a n-dimensional space which can be denoted by $D = \{d_0, \ldots, d_{n-1}\}$. Then we define that $I_i.space \subseteq D$. Additionally, a multi-dimensional query is denoted as $q(Ctr)$, where $Ctr = \{ctr_1, \ldots, ctr_u\}$ is a set of query criteria on u dimensions. We take a set $Qd = \{qd_1, \ldots, qd_u\}$ to represent the u dimensions. Obviously, $Qd \subseteq D$.

To help understand these symbols, we take a SQL query q for example. Assume the query conditions of q is '*where NAME* = '*Jack*' *and GENDER* = '*male*' *and No* = '123'', then the query criteria of q is $Ctr = \{NAME = \text{'}Jack\text{'}, GENDER = \text{'}male\text{'}, NO = \text{'}123\text{'}\}$, and the corresponding query dimensions of q is $Qd = \{NAME, GENDER, NO\}$.

The symbols and notations used in this paper are summarized in Table 2, some of which will be defined in the following sections.

TABLE 2. Notations and symbols.

Term	Definition
I_i	The ith index instance in FR-Index system
$I_i.space$	The indexing space of index instance I_i
n	The number of dimensions of datasets
D	The dimensions of datasets: $D = \{d_0, \ldots, d_{n-1}\}$
B	The bounding box covering all the spatial data objects: $B = (b_0, \ldots, b_{n-1})$
b_i	The ith interval of B: $b_i = [l_i, u_i]$
$q(Ctr)$	The multi-dimensional query with query criteria Ctr
Ctr	The query criteria for a specific query: $Ctr = \{ctr_i, \ldots, ctr_u\}$
Qd	The dimensions of Ctr: $Qd = \{qd_1, \ldots, qd_u\}$
S_t	The tth server
pir_t	The *potential indexing range* of S_t
IN_t	The node set selected from S_t as global index: $IN_t = \{in_t^1, \ldots, in_t^k\}$

3.1. Selecting indexing dimensions

In most cases, a query would not cover many dimensions, which means that for a query $q(Ctr)$, the cardinality of the corresponding query dimension Qd would not be too large. Thus, it is necessary to reduce the dimensions of the proposed multi-dimensional index. However, a single index which is built on a few dimensions might not facilitate processing all queries. To better manage the indexing system, we build a set of index instances $\mathcal{I}$ and set each $|I_i.space|$ as a fixed value in FR-index.

Note that different from most recursively defined server-centric topologies, the architecture of Fat-tree strictly consists of three layers, i.e. core layer, aggregation layer and edge layer (see Fig. 1). Benefiting from the hierarchical architecture of Fat-tree, a 3D hypercube (indexing space) could be appropriately partitioned and designated to servers of edge layer, regardless of the order k of Fat-tree. We will show how to determine a set of indexing spaces with three dimensions in FR-Index.

FR-index **Collector** collects query samples by requesting servers for their logs. To accelerate the process of log collecting, an optional method is *stratified random sampling*. For example, we can only send log requests to some servers randomly selected, where the sample quantity can be customized or self-tuned. The collector then analyses those query samples to make decisions about indexing spaces:

- **Collector** traverses all query samples and extracts each query's dimensions Qd. A histogram is maintained to record the occurrence frequency P_i of every different Qd_i. Then, all different Qd's are sorted by its occurrence frequency in descending order, denoted as a collection $\mathcal{Q} = \{Qd_1, Qd_2, \ldots, Qd_m\}$.
- **Collector** selects the first x sets in $\mathcal{Q}$ by calculating an integer x which satisfies $\sum_{j=1}^{x} P_j \geq P_{thr}$, where $P_{thr} \in [0, 1]$ is a threshold to control the performance of FR-Index. Generally, a higher P_{thr} may incur more index instances accompanied by more maintaining costs and faster query processing. Now $\mathcal{Q}$ is pruned into $\mathcal{Q}_p = \{Qd_1, Qd_2, \ldots, Qd_x\}$. We regard that $\mathcal{Q}_p$ depicts the feature of $P_{thr} \times 100\%$ of all historical queries as well as all subsequent queries.
- Based on $\mathcal{Q}_p$, **Collector** finds a collection $\mathcal{D}_{ans} = \{Dc_1, Dc_2, \ldots, Dc_y\}$ which has the following three properties:

  (a) $\forall\, i \in \{1, 2, \ldots, y\}$, $Dc_i \subseteq D$ and $|Dc_i| = 3$.
  (b) $\forall j \in \{1, 2, \ldots, x\}$, if $|Qd_j| < 3$, $\exists\, i \in \{1, 2, \ldots, y\}$, such that $Qd_j \subseteq Dc_i$.
  (c) $\forall j \in \{1, 2, \ldots, x\}$, if $|Qd_j| \geq 3$, $\exists\, i \in \{1, 2, \ldots, y\}$, such that $Dc_i \subseteq Qd_j$.

Each set in $\mathcal{D}_{ans}$ will become a 3D indexing space on which an index instance will be built. All of these index instances constitute an FR-Index system. Since $\mathcal{Q}_p$ depicts the feature of $P_{thr} \times 100\%$ of all historical queries, properties (b)(c) of $\mathcal{D}_{ans}$ guarantee that our FR-Index can efficiently facilitate processing $P_{thr} \times 100\%$ of all subsequent queries.

3.2. Partitioning indexing space

The information of selected indexing spaces will be sent to all servers by **Collector**. Once a server received the information, it will build local R-tree index on the dimensions contained by the indexing space. To better illustrate our proposal, we will take one index instance as an example to show that how our system works.

As mentioned above, a server needs to maintain a portion of global index. A challenge is to determine the range of global index that the server should be responsible for. In the following discussion, we denote this range as *potential indexing range* (PIR). As a tree-like DCN, the hierarchical structure of Fat-tree offers a convenient and efficient way to partition the indexing space such that we can generate PIR for every server.

In the entire data center, all multi-dimensional data form a data boundary denoted as $B = (b_0, b_1, b_2, \ldots, b_{n-1})$, i.e. a n-dimensional hypercube. Here each b_i is a closed bounded interval $[l_i, u_i]$ describing a range which is covered by the data objects along dimension d_i. Suppose that we had chosen an indexing space $I_j.space = (d_0, d_1, d_2)$. Since $B' = (b_0, b_1, b_2)$ is the 'meaningful' subspace of $I_j.space$ for our work, we will consider $I_j.space = B' = (b_0, b_1, b_2)$ in the following discussion for simplicity.

In a k-pod Fat-tree, we code a server S_t by $t = (k/2)^2 pod + (k/2) swi + (ID - 2)$, where pod, swi and ID are parameters in the IP address of this server.

Intuitively, we first partition the indexing space into k (the number of pods) parts along the first dimension, then $k/2$ (the number of edge switches in each pod) parts along the second dimension, finally $k/2$ (the number of servers connecting with each edge server) parts along the third dimension. Now, we gain $k^3/4$ equal-sized partitions of the indexing space and map each partition to a server. Therefore, for server S_t, its PIR denoted by pir_t can be expressed by Equation (1).

$$
\begin{aligned}
pir_t &= \{[l_0^t, u_0^t], [l_1^t, u_1^t], [l_2^t, u_2^t]\} \\
&= \left\{ \begin{bmatrix} l_0 + \dfrac{pod\,(u_0 - l_0)}{k}, l_0 + \dfrac{(pod+1)(u_0 - l_0)}{k} \end{bmatrix}, \right. \\
&\quad \begin{bmatrix} l_1 + \dfrac{switch\,(u_1 - l_1)}{k/2}, l_1 + \dfrac{(switch+1)(u_1 - l_1)}{k/2} \end{bmatrix}, \\
&\quad \left. \begin{bmatrix} l_2 + \dfrac{(id-2)(u_2 - l_2)}{k/2}, l_2 + \dfrac{(id-1)(u_2 - l_2)}{k/2} \end{bmatrix} \right\}
\end{aligned} \tag{1}
$$

Figure 4 shows an example for the index space partitioning strategy in a 4-pod FAT-tree.

3.3. Publishing in FR-index

To build global index, a server, say, S_t, adaptively selects a set of R-tree nodes, $IN_t = \{in_t^1, \ldots, in_t^n\}$, from its local index and publishes them as the global index. The index nodes in IN_t should cover all data stored in S_t, i.e. satisfying the index completeness in Section 2.2.3. Each in_t^i in IN_t will be published as a tuple of (ip_t, mbr_i), where ip_t is the IP address of S_t and mbr_i is the bounding box of in_t^i.

3.3.1. *Mapping scheme*

Wang *et al.* [4] proposed a novel mapping schema to regulate the publishing process. We adapt this schema for our system to distribute global index among the servers in Fat-tree.

For an R-tree node in_t^i in S_t to be published, we take the center $in_t^i.c$ and radius $in_t^i.r$ of its bounding box as the criteria for mapping. For example, assume the bounding box of in_t^i is $\{[l_0, u_0], [l_1, u_1], [l_2, u_2]\}$, then $in_t^i.c = (\frac{l_0+u_0}{2}, \frac{l_1+u_1}{2}, \frac{l_2+u_2}{2})$, and $in_t^i.r = \frac{1}{2}\sqrt{(u_0 - l_0)^2 + (u_1 - l_1)^2 + (u_2 - l_2)^2}$. Furthermore, suppose $in_t^i.c = (in_t^i.c_0, in_t^i.c_1, in_t^i.c_2)$, where $in_t^i.c_j$ $(j = 0, 1, 2)$ denotes the coordinate value on the jth dimension of $in_t^i.c$.

It is obvious that $in_t^i.c$ will be contained in a certain server's PIR, i.e. the designated indexing subspace. According to the indexing space partitioning scheme in Section 3.2, we can calculate this server's ip address $10.pod.switch.id$ by $in_t^i.c$:

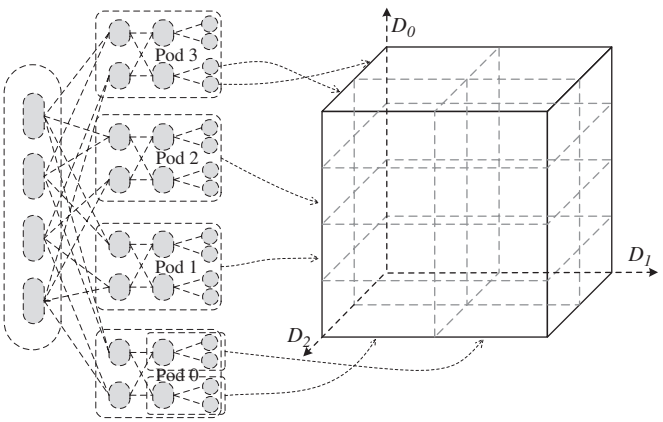

FIGURE 4. An example of index space partitioning in a 4-pod FAT-tree. The space was split along D_0 dimension first, then split along D_1 and D_2 in sequence.

$$\begin{cases} pod = \lfloor k\,(in_t^i.c_0 - l_0)/(u_0 - l_0)\rfloor \\ switch = \lfloor k\,(in_t^i.c_1 - l_1)/2(u_1 - l_1)\rfloor \\ id = \lfloor k\,(in_t^i.c_2 - l_2)/2(u_2 - l_2)\rfloor + 2 \end{cases} \quad (2)$$

For simplicity, we call this server the **initial server**, S_x. To publish in_t^i as the global index, S_t first sends in_t^i to S_x. Then S_x compares $in_t^i.r$ with a *predefined threshold*, say R_{thr}. If $in_t^i.r > R_{thr}$, then S_x determines the servers (called **candidate servers**) whose PIR intersects with the bounding box mbr_i of in_t^i, and in_t^i will be further sent to those servers as global index. If $in_t^i.r < R_{thr}$, the node will be stored in S_x only. The value of R_{thr} will be discussed in Section 4.

Figure 5 shows an example of global index mapping scheme in 2D space. Suppose the cluster consists of 16 servers numbered $01 - 16$, and the indexing space is partitioned into 16 subspaces which are respectively designated to these servers as their PIR's. The dashed box, with radius r and center in the PIR of server 06, denotes a node in_t^i to be published. Server 06 is the initial server. If $R_{thr} = r_1$, then in_t^i will be sent to candidate servers 05, 09 and 10 further by server 06. Eventually in_t^i will be stored in servers 05, 06, 09 and 10 as the global index. If $R_{thr} = r_2$, then in_t^i will be stored in server 06 exclusively.

3.3.2. *Publishing scheme*

The server addressing and routing protocol of Fat-tree ensure a special and independent communication link between any pair of servers, thus the initial server can forward an indexing node to candidate servers through different links. However, there are two problems during the communication between initial server and candidate servers: one is that these messages need to be queued up for transmission at the network port of initial server; the other is that initial server needs to build and maintain multiple links with other servers simultaneously. These two problems could greatly increase the time for data transmission and indexing construction.

In fact, we can regard initial server and candidate servers as a vertex set V. For any v_i, $v_j \in V$, edge e_{ij} represents the communication link between v_i and v_j, and the weight w_{ij} of

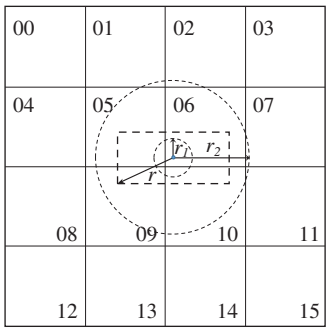

FIGURE 5. Example of global index mapping scheme in 2D space.

e_{ij} represents the number of switches on the corresponding link (the physical links of $v_i \to v_j$ and $v_j \to v_i$ are different in Fat-tree, but the two links have the same number of switches, thus regarded as one same edge). Edge set E consists of edges between all the vertex pairs in V. Therefore, the indexing node forwarding problem is transformed to the problem of finding a minimum spanning tree in the complete undirected graph with weight, $G = (V, E)$. Based on fat-tree routing protocol of the premise that disk I/O is faster than network transmission under the same data volume, we designed a *dimension-order partitioning and spreading* algorithm for indexing node publishing, which is shown in Algorithm 1.

In Algorithm 1, the current (initial) server S_{this} receiving indexing node *in* first traverses the candidate server set $\mathbf{S}$. For each server $S \in \mathbf{S}$, if its PIR pir_S is on the left (right) side of pir_{this} along the ith dimension, then S is removed from $\mathbf{S}$ and added to set $\mathbf{S_l}$ ($\mathbf{S_r}$) (lines $4 - 10$). If $\mathbf{S_l}$ is not empty, S_{this} then forwards $\mathbf{S_l}$ and indexing node *in* to its left neighbor server S_{ln}^i along the ith dimension, and S_{ln}^i will execute Algorithm 1 recursively (lines 11–14). Note that S_{this} can continue to execute the following codes (lines $15 - 18$) without getting the return value of line 14.

Now we compare the performance between Algorithm 1 and directly forwarding strategy. Suppose a 3D indexing space (d_0, d_1, d_2) is partitioned into $k_0 \times k_1 \times k_2$ PIR's, and there are k_i PIR's on the d_i $(i = 1, 2, 3)$ dimension. Now

Algorithm 1 Dimension-order partitioning and spreading (*dimorder* in brief)

Input: A candidate servers set $\mathbf{S}$, an index node *in*

1 $S_{this}.store(in)$;
2 $\mathbf{S_l} = \mathbf{S_r} = \emptyset$;
3 **for** $i = 0$ *to* 2 **do**
4 **for** *each* $S \in \mathbf{S}$ **do**
5 **if** S *is on* S_{this}*'s left along the* i*-th dimension* **then**
6 $\mathbf{S_l} = \mathbf{S_l} \cup S$;
7 $\mathbf{S} = \mathbf{S} - S$;
8 **if** S *is on* S_{this}*'s right along the* i*-th dimension* **then**
9 $\mathbf{S_r} = \mathbf{S_r} \cup S$;
10 $\mathbf{S} = \mathbf{S} - S$;
11 **if** $\mathbf{S_l} \neq \emptyset$ **then**
12 $S_{this}.send(\langle \mathbf{S_l}.in \rangle, S_{ln}^i)$;
13 $\mathbf{S_l} = \emptyset$;
14 $S_{ln}^i.dimorder(\mathbf{S_l}, in)$;
15 **if** $\mathbf{S_r} \neq \emptyset$ **then**
16 $S_{this}.send(\langle \mathbf{S_r}, in \rangle, S_{rn}^i)$;
17 $\mathbf{S_r} = \emptyset$;
18 $S_{rn}^i.dimorder(\mathbf{S_r}, in)$;

consider two servers S_1, S_2 and their PIR's pir_1, pir_2. If the intervals of pir_1 and pir_2 on d_0 dimension are different, then S_1 and S_2 dwell in different pods, and the communication between S_1 and S_2 needs to go through five switches. If the intervals of pir_1 and pir_2 are equal on d_0 dimension but different on d_1 dimension, then three switches are on the communication link between S_1 and S_2. If the intervals of pir_1 and pir_2 are different exclusively on d_2 dimension, then only one switch is on the communication link. Suppose one initial server needs to forward an indexing node to all the other servers in the cluster. The number of switches passed during Algorithm 1 execution is calculated using Equation (3), while the number of switches passed by directly sending messages to other servers is calculated using Equation (4).

$$\begin{aligned} N_1 &= 5(k_0 - 1) + 3k_0(k_1 - 1) + k_0k_1(k_2 - 1) \\ &= k_0k_1k_2 + 2(k_0k_1 - 1) + 2(k_0 - 1) - 1. \end{aligned} \tag{3}$$

$$\begin{aligned} N_2 &= 5k_1k_2(k_0 - 1) + 3k_2(k_1 - 1) + (k_2 - 1) \\ &= k_0k_1k_2 + 2k_2(k_0k_1 - 1) + 2k_1k_2(k_0 - 1) - 1. \end{aligned} \tag{4}$$

Based on the comparison between N_1 and N_2, it is concluded that Algorithm 1 can significantly reduce network I/O when the publishing range is relatively large.

3.4. Skewed data processing

In reality, the datasets are more or less skewed, which means data are not uniformly distributed in the entire indexing space. If we directly adopt the indexing mapping and publishing scheme in Section 3.3, the workloads of global indexing maintenance and query processing among different servers will be greatly imbalanced. In this section, we revise the global indexing mapping scheme to cater for skewed datasets.

Zhang *et al.* [30] proposed a *piecewise mapping function* (PMF) to approximate *cumulative distribution function* (CDF) and uniformly map data points into [0, 1]. The CDF function $cdf(x)$ can return the percentage of data which $\leq x$. Therefore, we can adapt PMF to FR-Index for global indexing redistribution.

The objective of PMF is to evenly map data objects into m buckets in [0, 1]. Suppose O denotes a skewed dataset, and $|R|$ denotes the data range covered by O. m buckets with $m + 1$ boundary coordinates $\{bc_0, bc_1, \ldots, bc_m\}$ are needed for data mapping. These coordinates uniformly divide R into m pieces (or buckets). Let $bc_i.n$ ($i \in \{0, 1, \ldots, m\}$) represent the number of objects which $\leq bc_i$. Now given a point with coordinate p, we use the following equation to calculate the number i of bucket that p should be mapped into:

$$i = \left\lfloor \frac{p}{|R|} \times m \right\rfloor. \tag{5}$$

For example, consider 10 numbers $O = \{0.5, 0.9, 1.4, 2.5, 2.7, 3.2, 3.7, 5.4, 7.0, 8.9\}$ distributed in $R = [0, 10]$. We can partition these numbers into five buckets with boundary coordinates $\{0, 2, 4, 6, 8, 10\}$, then we have $\{bc_0.n, bc_1.n, \ldots, bc_5.n\} = \{0, 3, 7, 8, 9, 10\}$ and $\{cdf(bc_0), cdf(bc_1), \ldots, cdf(bc_5)\} = \{0, \frac{3}{10}, \frac{7}{10}, \frac{8}{10}, \frac{9}{10}, 1\}$

Formally, PMF uses Equation (6) to approximate the exact $cdf(p)$:

$$cdf(p) = \begin{cases} 1, & p = |R| \\ \dfrac{\frac{bc_{i+1}.n - bc_i.n}{bc_{i+1} - bc_i}(p - bc_i) + bc_i.n}{|O|}, & p \neq |R| \end{cases} \tag{6}$$

where $cdf(p) \in [0, 1]$.

Based on PMF, we design an index mapping scheme for skewed data. Suppose we has a local R-tree in space (d_0, d_1, d_2), from where a node in_t^i will be selected and published as a global index node. On dimension d_0, we can calculate the cdf of in_t^i:

(1) Take a dataset O for later calculation, whose data distribution should reflect the features of original skewed data. Specifically, O can be the original dataset, or a sample subset produced by stratified random sampling from the original dataset.
(2) Define the number m of buckets and their boundary coordinates on dimension d_0 for data mapping. Then, we calculate the cumulative counts of items $bc_i.n$ ($i \in 0, 1, \ldots, m$) for each bucket.
(3) Based on the data range $|b_0| = u_0 - l_0$ and the d_0-dimensional coordinate of in_t^i's center $in_t^i.c$, we obtain the number i of the bucket that in_t^i is mapped into by Equation (5), then calculate the CDF value $cdf_0(in_t^i.c)$ of $in_t^i.c$ on dimension d_0 subsequently.

In a similar manner, we can also calculate $cdf_1(in_t^i.c)$ and $cdf_2(in_t^i.c)$ on d_1 and d_2, respectively. Next, we determine the transformed coordinates $in_t^i.c' = (cc_0, cc_1, cc_2)$ for $in_t^i.c$:

$$cc_j = cdf_j(in_t^i.c) \times |b_j| + l_j \quad j \in \{0, 1, 2\}, \tag{7}$$

where $b_j = [l_j, u_j]$ is the data range of O on ith dimension. Finally, we get a transformed node $in_t^{i'}$ and its radius $in_t^i.r'$. After applying PMF to all the skewed nodes, we can obtain a transformed dataset with uniform distribution. In index mapping stage, the initial server and candidate servers storing in_t^i will be determined by inputting $in_t^i.c'$ and $in_t^i.r'$ into the global index mapping algorithm in Section 3.3.1.

Need to mention that the transformed node $in_t^{i'}$ is only used for global index mapping; initial server and candidate servers still store the original node in_t^i, and the PIRs for each

server stay the same as before. Thus, we can avoid large numbers of transforming calculation during global nodes publishing and querying stage, meanwhile guaranteeing the load balance for each server.

4. QUERY PROCESSING

Since FR-Index has a two-layer structure, query processing in FR-Index is divided into two phases: (1) **Search in the global index**: a query is processed in the global index first, and the responsible servers storing interesting data will be returned. Then, the query will be forwarded to these servers for further searching. (2) **Search in the local index**: These responsible servers receiving the query will search in their local index. Finally, results will be combined and returned to users.

In fact, based on the topology and routing protocol of Fat-tree, a query could be processed concurrently by multiple servers. Suppose that we have deployed an FR-Index system which is composed of several 3D index instances in the data center. Given a multi-dimensional query $q(Ctr)$, where $Ctr = \{ctr_1, \ldots, ctr_u\}$, and ctr_j is typically a key or range criterion along query dimension qd_j $(1 \leq j \leq u)$.

Considering that $u \geq 3$ and index instance I_i in FR-Index comprises only three dimensions, we first prune Ctr into the indexing space $I_i.space$ such that $q(Ctr)$ is converted to $q(Ctr')$, where $I_i.space \subseteq Qd$ (If $u < 3$, we expand Ctr to match the indexing space $I_i.space$ where $Qd \subseteq I_i.space$). Then we employ index instance I_i to process $q(Ctr')$ and retrieve a result set. At last, we prune the result set according to the criteria in Ctr/Ctr'. Generally, we call $q(Ctr')$ a *point query* if the elements in Ctr' are all key criteria. Otherwise, we call $q(Ctr')$ a *range query*.

4.1. Point query

A point query $q(Ctr')$, converted from $q(Ctr)$, is denoted as $q(key)$. Suppose $key = (v_1, v_2, v_3)$ represents a point in a 3D indexing space, and $q(key)$ is first sent to a server S_x randomly chosen by the user. FR-Index processes a point query in the following steps:

(1) The server S_x receiving $q(key)$ first determines a index instance I_{ino} that matches key best. Then according to the indexing space partitioning of I_{ino}, S_x determines the server S_{init} whose PIR pir_{init} contains key and forwards a message $\langle q(key), ino \rangle$ to S_{init}, where $q(key)$ is the query and ino is the number of interesting index instance.
(2) S_{init} generates a hypersphere at the point key with radius R_{thr} (the threshold defined in Section 3.3.1). This hypersphere, denoted as $key.sspace$, is defined as the search space of $q(key)$. S_{init} forwards $q(key)$ to servers whose PIRs overlap with $key.sspace$.
(3) S_{init} and those servers search the global index buffered in their memory to find out which nodes' bounding boxes contain key. Then, $\langle q(key), ino \rangle$ is forwarded to the servers who published these interesting R-tree nodes.
(4) These servers will search $q(key)$ in their local indexes to get the query results. Finally, these results will be pruned according to criteria Ctr/key and returned to user.

Steps 1–3 are global searching phase, and step 4 is local searching phase. Now we concentrate on the search space of key, i.e. a hypersphere with radius R_{thr}. In FR-Index, if we only search $q(key)$ in the global index dwelling in S_{init}, the results may be incomplete.

Figure 6a shows a case of incomplete point query, where four solid boxes represent the PIRs of four servers; dashed boxes denote some global index nodes. The shaded global node, with center contained in S_A, is stored merely in S_A since its radius is shorter than R_{thr}. Consider the point query $q(key)$ represented by a black dot in Fig. 6a. If we search key in S_{init} only, the results contained in the shaded node stored in S_A will be neglected, leading to incomplete query results. Figure 6b depicts the point search strategy in FR-Index. After S_{init} received the $q(key)$, the search space, i.e. a circle with center key and radius R_{thr}, is generated and $q(key)$ will be forwarded to servers whose PIR's overlap with $key.sspace$. In Fig. 6b, S_{init} will forward $q(key)$ to S_A, S_B and S_C for global index searching further.

4.2. Range query

A range query is denoted as $q(range)$, where $range = \{[a_1, b_1], \ldots, [a_{u'}, b_{u'}]\}$. We define

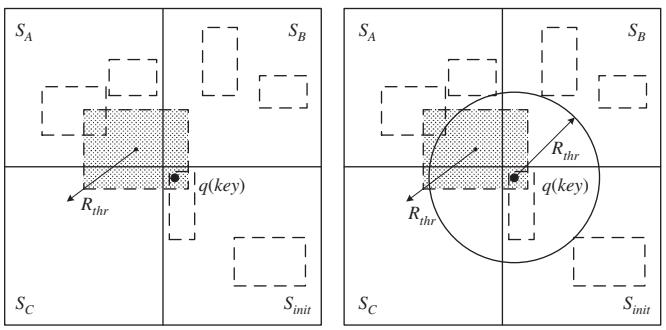

FIGURE 6. An example for point query. (**a**) Search in S_{init} and (**b**) search in $key.searchspace$.

$$\begin{cases} range.c = \left(\dfrac{a_1 + b_1}{2}, \ldots, \dfrac{a_{u'} + b_{u'}}{2}\right) \\ range.r = \dfrac{1}{2}\sqrt{\sum_{i=1}^{u'}(b_i - a_i)^2} \end{cases}. \quad (8)$$

The search space for $q\,(range)$, denoted as *range.sspace*, is a hypersphere with center *range.c* and radius $range.r + R_{thr}$. The processing steps for $q\,(range)$ are similar to point query processing discussed in Section 4.1.

THEOREM 1. *For range query $q\,(range)$, whose search space range.sspace is a hypersphere with center range.c and radius $range.r + R_{thr}$. Searching the global index in servers whose potential index ranges overlap with range.sspace can guarantee the completeness of results.*

Proof. Suppose *range.c* is contained in pir_{t1} of server S_{t1}. in_x^i is a global index node with its radius $in_x^i.r$, center $in_x^i.c$ contained in pir_{t2} of server S_{t2}. Suppose the hypercube denoted by in_x^i overlaps with *range*, i.e. $Distance\,\langle range.c, in_x^i.c\rangle < range.r + in_x^i.r$.

If $in_x^i.r \leq R_{thr}$, since $Distance\,\langle range.c, in_x^i.c\rangle < range.r + in_x^i.r$, then $in_x^i.c$ is contained in a hypersphere search space with center *range.c* and radius $range.r + R_{thr}$, i.e. pir_{t2} overlaps with *range.sspace*. According to the query process strategy, $q\,(range)$ will be forwarded to S_{t2} and the results contained in in_x^i will be retrieved.

If $in_x^i.r > R_{thr}$, denote the overlapping region between in_x^i and *range* as $Region_o$. There exists a server S_y whose pir_y overlaps with $Region_o$, and pir_y satisfies the following two conditions simultaneously:

(1) pir_y overlaps with in_x^i. According to the global index mapping strategy in Section 3.3, server S_y with pir_y has stored in_x^i as the global index node.
(2) pir_y overlaps with *range*. Thus pir_y must overlap with *range.sspace*, which means $q\,(range)$ will be forwarded to S_y.

In summary, S_y, storing in_x^i like S_{t2}, will be searched and the results contained in in_x^i will be retrieved. □

In general, the range query strategy in FR-Index can guarantee the completeness of query results.

4.3. Query on skewed datasets

In this section, we discuss the query processing of skewed dataset, corresponding to the skewed data mapping scheme in Section 3.4. For a point query $q\,(key)$ with $key = (v_1, v_2, v_3)$, we perform the following steps:

(1) The server S_x receiving $q\,(key)$ first check if the indexing instance I_{ino} matching $q\,(key)$ is constructed on skewed dataset. If so, S_x will transform *key* to $key' = (v_1', v_2', v_3')$:

$$v_j' = cdf_j(v_j) \times |b_j| + l_j \quad (j = 1, 2, 3), \quad (9)$$

where $b_j = [l_j, u_j]$ denotes the data range on dimension d_j.
(2) According to indexing space partitioning scheme, S_x determines the server S_{init} whose PIR pir_{init} contains key' and forwards a message $\langle q\,(key), key', ino\rangle$ to S_{init}.
(3) S_{init} first generates a hypersphere at point key' with radius R_{thr} and then forwards $\langle q\,(key), key', ino\rangle$ to the candidate servers.
(4) S_{init} and candidate servers search *key* in global nodes and forward $\langle q\,(key), ino\rangle$ to servers that published the global nodes containing $q\,(key)$. The final results of $q\,(key)$ will be fetched from the local indexes of these servers.

Similarly, for a range query $q\,(range)$, we first transform *range* to *range'*, then perform the similar query steps in Section 4.2.

4.4. Discussion on R_{thr}

Up to now, we can discuss the impact of R_{thr} in FR-Index system. According to Section 3.3, a smaller R_{thr} incurs more index node replicas in multiple servers, which increases the maintenance cost. On the other hand, according to the query processing strategy, a larger R_{thr} implies a larger search space, which means we must search more servers to retrieve complete results for a query, reducing the efficiency of query processing.

In FR-Index, we calculate a specific R_{thr} for an index instance. **Collector** is used to sample R-tree nodes in local index. After several trials of FR-Index implementation, we found that when the R_{thr} in an index instance is slightly larger than the average radius of 70% sample nodes, the trade-off between query efficiency and index maintenance would be optimal.

5. INDEX MAINTENANCE AND UPDATING

In reality, query pattern to the cluster could significantly alter in some scenarios, especially in data-intensive applications. Therefore, dynamic index updating is a crucial component in FR-Index. The update of FR-Index is classified into two categories:

Update in index instances: at first, we have no knowledge about the query pattern and data updating pattern. Therefore,

in an h-level local R-tree, we publish the index nodes in $h-1$ level (the h level nodes is leaf nodes). After a period of time, we adopt a cost model to update some published index nodes to reduce the overhead of index maintenance. More details will be discussed in Section 5.1.

Update to index instances: another updating requirement for our system is to update the index instances we have constructed. We propose a simple and efficient strategy to deal with this requirement. Each server stores a histogram to maintain the accessing status for every index instance. The FR-Index collector will first request the histograms from all the servers, then adopt least recently used algorithm to delete obsolete index instances and add new index instances, which will be discussed in Section 5.2 further.

5.1. Update in index instance

As mentioned before, each server S_t in the cluster needs to select a portion of local index to publish as the global index. However, which local index nodes should be selected is a critical question. In this section, we introduce a cost model for nodes selection and index dynamic update.

5.1.1. *Cost model*

Suppose a server S_t selects an index node set $IN_t = \{in_t^1, \ldots, in_t^k\}$ from its local R-tree as the global index, satisfying index completeness and uniqueness. In FR-Index, the cost $C(in)$ of index node in is composed of index maintenance cost $C_m(in)$ and query processing cost $C_q(in)$:

$$C(in) = C_m(in) + C_q(in). \tag{10}$$

Index maintenance cost $C_m(in)$ represents the cost of index node update. If a published local index node is split or merged locally because of data update, the corresponding global index node needs to be updated as well. When a published local index node is split, the server storing this node should publish three index update messages, i.e. one message for deleting obsolete node and two messages for publishing two new nodes. Similarly, if two published local index nodes are merged, two messages for deleting two obsolete nodes and one message for publishing the new node need to be published. For a node in, suppose the probabilities of splitting and merging in are $p_{\text{split}}(in)$ and $p_{\text{merge}}(in)$, respectively, then:

$$C_m(in) = 3(p_{\text{split}}(in) + p_{\text{merge}}(in)). \tag{11}$$

Now the values of $p_{\text{split}}(in)$ and $p_{\text{merge}}(in)$ need to be specified. A two-state Markov chain model in [14] was used to calculate the probabilities of splitting or merging each R-tree node. To improve precision, this method needs a histogram for recording historical update pattern and considerable calculation for update prediction. In FR-Index, the method is employed to calculate $p_{\text{split}}(in)$ and $p_{\text{merge}}(in)$ of leaf nodes in R-tree. As for non-leaf nodes, we employ the method in [16] for probability calculation.

Suppose the probability of inserting a key to node in is $p_1(in)$, and the probability of deleting a key from in is $p_2(in)$, then we have:

$$p_{\text{split}} = \frac{\left(\frac{p_2}{p_1}\right)^{\frac{3m}{2}} - \left(\frac{p_2}{p_1}\right)^{m}}{\left(\frac{p_2}{p_1}\right)^{2m} - \left(\frac{p_2}{p_1}\right)^{m}}, \tag{12}$$

$$p_{\text{merge}} = \frac{\left(\frac{p_2}{p_1}\right)^{2m} - \left(\frac{p_2}{p_1}\right)^{\frac{3m}{2}}}{\left(\frac{p_2}{p_1}\right)^{2m} - \left(\frac{p_2}{p_1}\right)^{m}}, \tag{13}$$

where m is an R-tree parameter indicating the minimum number of subtrees of a parent node.

For any non-leaf node in, suppose its child node set is $\{c_1, c_2, \ldots, c_i\}$, then p_1 and p_2 can be calculated:

$$p_1(in) = \prod_{j=1}^{i} (1 - p_{\text{split}}(c_j)), \tag{14}$$

$$p_2(in) = \prod_{j=1}^{i} (1 - p_{\text{merge}}(c_j)). \tag{15}$$

According to Equations (12)–(15), if we obtain p_{split} and p_{merge} of leaf nodes by Markov model, the probability of non-leaf nodes update can be calculated subsequently.

Query processing cost $C_q(in)$ consists of an essential cost and a false positive cost p_{fp}. In FR-Index, we are mainly focused on p_{fp} since the essential cost cannot be further reduced.

To better understand a *false positive* query, suppose two published global R-tree nodes in_{ig}^1 and in_{jg}^2 have an intersect region reg_{isc}, and corresponding local index nodes in_{il}^1 and in_{jl}^2 are stored in server S_i and S_j, respectively. Suppose a point key for query $q(key)$ is covered by reg_{isc}, but key is actually stored in S_i instead of S_j. When a user sends $q(key)$ to FR-Index, this query will be forwarded to S_i and S_j simultaneously, thus the query to S_j is a false positive query. Suppose $B(in)$ is the bounding box of node in, and $D(in)$ represents the actual data range of in, then the false positive probability p_{fp} is:

$$p_{fp} = \frac{B(in)/D(in)}{B(in)}. \tag{16}$$

And we have $C_q(in) = p_{fp}(in)$. After obtaining the index maintenance cost $C_m(in)$ and query processing cost $C_q(in)$, we can calculate the cost of node in:

$$C(in) = 3p_{\text{split}}(in) + 3p_{\text{merge}}(in) + p_{fp}(in) \quad (17)$$

In addition, according to the index mapping and publishing strategy in Section 3, an index node *in* could be forwarded to multiple servers as the global index when its radius $in.r > R_{thr}$, which means the communication cost for index update to *in* could be significantly influenced by the number of servers storing *in*. We further optimize the cost model:

$$C(in) = \begin{cases} \frac{ins+1}{ins}[3p_{\text{split}}(in) + 3p_{\text{merge}}(in) + p_{fp}(in)], & in.r > R_{thr} \\ 3p_{\text{split}}(in) + 3p_{\text{merge}}(in) + p_{fp}(in), & \text{otherwise} \end{cases} \quad (18)$$

where *ins* is the number of PIR's which intersect with the bounding box of *in*.

Notice that the cost model (Equation (18)) is based on the premise that the switches in fat-tree are two-layer devices, which means data packages can be forwarded in linear speed. We define the routing cost for forwarding one package in two-layer switches is 1. If the switches in data center are three-layer switches, the store-and-forward function could take effect. We define the routing cost of one package forwarding in three-layer switches as the number of hops passed by this package.

A k-pod Fat-tree data center has $k^3/4$ servers. For any server S, the hops needed during communications between S and other servers are probably 1, 3 or 5. Suppose the probabilities that S communicates with any other servers are equal, then we get the expected hops:

$$E(k) = \frac{(k/2-1) + 3(k/2-1)k/2 + 5(k-1)k^2/4}{k^3/4} = 5 - \frac{2}{k} - \frac{4}{k^2} - \frac{4}{k^3}. \quad (19)$$

Therefore, when three-layer switches are employed in fat-tree, the cost of indexing node *in* is:

$$C(in)' = E(k)C(in). \quad (20)$$

5.1.2. *Indexing nodes selection and update*

In our proposal, each server chooses some index nodes from local R-tree index and publishes them into global index. A published higher-level R-tree node may incur less update cost while generate more false positives. Besides, its bounding box may overlap with more servers' PIRs, increasing storage cost and query processing complexity. Therefore, it is crucial to choose 'proper' local index nodes to publish.

According to Equation (17), if a global index node is updated frequently but scarcely queried, it has relatively greater maintenance cost and should be replaced by its parent node; if a node is queried frequently but scarcely updated, then it has great query processing cost and should be replaced by its child nodes. The indexing node set, selected from server S_t and published as global index, should not only satisfy index completeness and uniqueness but also has a minimum cost summation.

Ref. [14] introduces a dynamic programming algorithm. We adapted this algorithm for index node selection in FR-Index, which is shown in Algorithm 2. The time complexity of Algorithm 2 is $O|V|$, where $|V|$ denotes the number of nodes in the R-tree.

5.2. Update to index instances

A case in reality is that the query pattern to FR-Index could alter significantly under some scenarios, such as hot events in social applications. Two problems exist in this situation: the first is that most queries cannot match with any indexing instance, which could degrade query efficiency greatly; the second is that some indexing instances are scarcely queried but need to be updated periodically, which could waste storage and network resources. Therefore, it is necessary to dynamically update indexing instances for variations of workloads.

In FR-index, **Collector** will periodically (assume the time window is T) collect query logs of all servers in a stratified random sampling manner. Based on the analysis of these logs, **Collector** will determine the queries mismatching all the indexing instances. If in one T, the percentage of queries matching indexing instances is less than P_{thr} (see Section 3.1), **Collector** will execute the following steps:

Algorithm 2 Index node selection

Input : A local index node in
Output: A set of local index nodes IN, total cost C

```
1  IN = {in};
2  C = C(in);
3  if in is a leaf node then
4  |  return IN and C;
5  else
6  |  IN_temp = ∅;
7  |  C_temp = 0;
8  |  for each child node in_i of in do
9  |  |  {IN', C'} = IndexNodeSelect(in_i);
10 |  |  IN_temp = IN_temp ∪ IN';
11 |  |  C_temp = C_temp + C';
12 |  if C_temp < C(n) then
13 |  |  return IN_temp and C_temp;
14 |  else
15 |  |  return IN and C;
```

(1) Suppose the query criteria set of all unmatched queries is Q_{un}. Using the indexing space construction strategy in Section 3.1, collector calculates a new data dimension collection D_{new}. D_{new} should cover $P_{thr} \times 100\%$ items in Q_{un}.
(2) For each set D_{c_i} in D_{new}, collector first calculates the percentage P_i of queries covered by D_{c_i}, then sorts D_{c_i} by corresponding P_i in descending order.
(3) For each set D_{c_j} in current indexing space collection D_{ans}, collector calculates the P_j for each D_{c_j}, then sorts D_{c_j} by corresponding P_j in descending order.
(4) Collector substitutes the latter items in D_{ans} with the ahead several items in D_{new} such that the summation of corresponding P_j in D_{ans} is larger than P_{thr}. These new indexing spaces in D_{ans} will be used for constructing new indexing instances, and the obsolete indexing instances will be deleted.

To ensure the availability of FR-Index, collector ought to first construct new indexing instances, then substitute the obsolete indexing instances one by one during non-peak periods.

6. PERFORMANCE EVALUATION

We evaluated the proposed indexing scheme on Amazon's EC2 platform. Each instance has a 2.5 GHz DualCore Intel *Xeon* processor and 4 GB memory. We organized EC2 computing units into a simulative data center with Fat-tree topology. Table 3 lists some common experiment settings.

TABLE 3. Common experiment settings.

Configuration items	Setting
Size of data center	4-pod(16 servers) or 6-pod(54 servers)
Number of stored data items	20k, 40k, 60k, 80k or 100k on each server
Datasets	YearPredictionMSD, Uniform_3d, Zipfian_3d
Number of queries	1k, 2k, 3k, 4k or 5k
R_{thr}	larger than the radius of 70% of published index nodes

Datasets. One real dataset and two synthetic datasets are used in our experiments. The real dataset is *YearPredictionMSD* [31] with size 199.5GB, which comprises 54 metadata and audio features from hundreds of thousands of songs. Table 4 shows the details of the second to sixth dimensions of YearPredictionMSD. Accordingly, we synthesized two datasets based on the 7digitalid, latitude and longitude dimensions of YearPredictionMSD: *Uniform_3d*, generated following uniform distribution; *Zipfian_3d*, generated strictly following zipfian distribution of skewness factor 0.8. For the two synthetic datasets, we generated 600 000 data points.

Index construction. For each dataset, we randomly partitioned and distributed the entire dataset over the cluster, roughly making each server maintain the same number of data points. Then, we constructed a local R-tree in each node, and published the upper layer nodes of leaf nodes in each R-tree as the global index. Before query experiments, we randomly extracted 50 000 data items to generate a query sample covering all of the indexing dimensions of FR-Index, and played the query sample to fat-tree cluster, enabling the updating of global index.

Query generation. For point query, we randomly selected a certain number of data items to generate the query sets. This method could roughly simulate the data distribution of target datasets, and the dense portion of the original datasets could be queried more frequently. For range query, the query sets are generated according to different selectivity, i.e. the different percentage of indexing space. Generally, one range query accounts for 0.1% of the searching space. The performance comparison between range queries with different selectivity will be conducted in Section 6.3.

6.1. Evaluation on index construction

Initially, to construct an FR-Index instance, we selected the second, third and fourth dimensions of YearPredictionMSD, namely the 7digitalid, familiarity and hotness as the index space. Each server needs to construct a local index, i.e. a R-tree, for its data in the index space. Subsequently, the upper layer nodes of leaf nodes in R-tree were selected as the global index and were published in the cluster according to the mapping and publishing rules in Section 3. The global index nodes were cached in memory to accelerate query process.

TABLE 4. Details of YearPredictionMSD.

Dimension	Field name	Type	Description	Range	Standard deviation
2	Artist 7digitalid	int	ID from 7digital.com	[4, 809205]	142 161.827
3	Artist familiarity	float	Algorithmic estimation	[0.0, 1.0]	0.160
4	Artist hotness	float	Algorithmic estimation	[0.0, 1.083]	0.144
5	Artist latitude	float	Latitude	[-41.281, 69.651]	15.596
6	Artist longitude	float	Longitude	[-162.437, 174.767]	50.501

We respectively constructed FR-Index instances in a 4-pod Fat-tree with 16 servers and a 6-pod Fat-tree with 54 servers. The data items stored in each server were increased from 20,000 to 100 000 gradually. Figure 7 shows the size of local index and global index in 4-pod Fat-tree (a) and 6-pod Fat-tree (b).

A key observation in Fig. 7 is that the size of global index is almost 10× smaller than that of local indexes under the same setting, which indicates a lightweight feature for FR-Index. Moreover, this advantage can be retained when the data center becomes larger, which verifies that FR-Index is scalable. In Fig. 7, we constructed a 3D indexing instance based on at most 100 000 data items (roughly 19GB) in each server, bring about the size of global index less than 40MB. In reality, however, the data volume stored in data centers could be much larger than the data volume in our experiments. FR-Index can reuse data servers as global index containers such that data center administrators need no more new machines to store global index, greatly reducing deployment cost.

Note that the local index in data servers, though consuming much storage space, is necessary for data retrieval, while the global index introduced by FR-Index, occupying lesser storage, could be placed in memory for rapidly query processing.

6.2. Evaluation on load balancing

In Section 3.4, we introduced PMF to balance the workloads in Fat-tree cluster, and gave the corresponding query algorithm in Section 4.3. To evaluate the validity, we first employed PMF to preprocess datasets *YearPredictionMSD* and *Zipfian_3d*, then randomly selected 20 000 × 16 data items to be distributed in 4-pod Fat-tree, with each server owning 20 000 items. We constructed an FR-Index instance with dimensions (2, 3, 4) and randomly generated 5, 000 point queries with dimensions (2, 3, 4, 5, 6) to be played to FR-Index. For comparison, we also conducted query experiments on the original *YearPredictionMSD* and *Zipfian_3d*, along with *Uniform_3d*. The visiting times of each server are recorded and presented in Fig. 8.

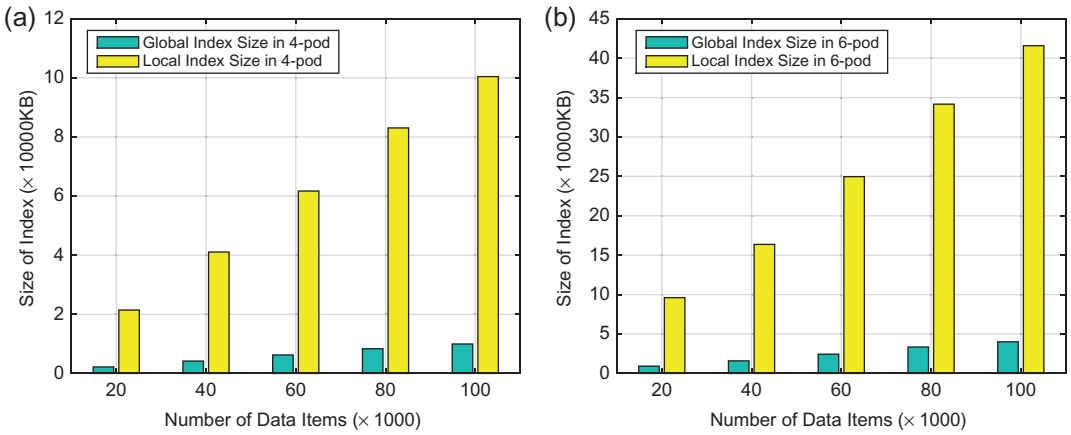

FIGURE 7. Size of global index and local index. (**a**) In 4-pod fat-tree and (**b**) in 6-pod fat-tree.

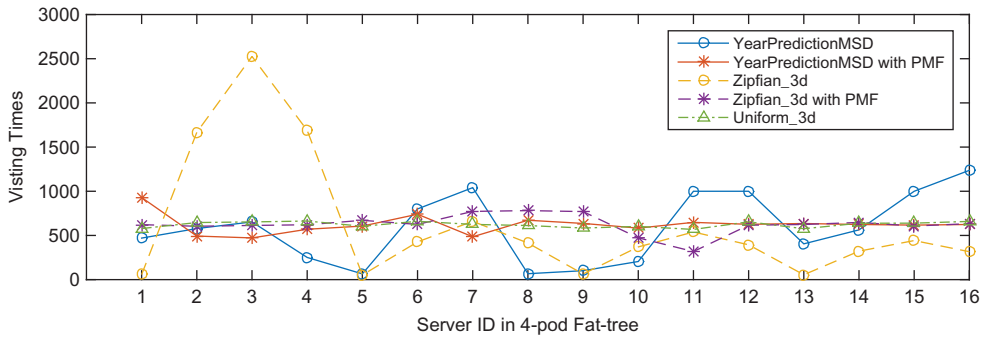

FIGURE 8. Visiting frequency for each server in 4-pod fat-tree.

We observe that under *Uniform_3d*, the visiting frequencies of each server are roughly the same, while the visiting frequencies are greatly different for each server under *Zipfian_3d* and *YearPredictionMSD*. Another observation is that the effect of *PMF* is considerable, especially for *YearPredictionMSD*. Note that the PMF algorithm is typically employed to balance the query workload in global layer. In the following experiments, we employ PMF to preprocess all the datasets to obtain a nearly uniform distribution among servers.

6.3. Evaluation on query processing

To evaluate the query performance of FR-Index, we first placed 20 000 data items in each server randomly. Then a 3D FR-Index instance is built to facilitate query processing. As mentioned in Section 6.1, the index space was composed of the second, third and fourth dimensions of YearPredictionMSD.

6.3.1. *Comparison with scanning strategy*

After the index construction, we randomly generated a certain number of 5D point queries and range queries (searching 0.1% range of the entire indexing space), where the query dimensions consist of the second, third, fourth, fifth and sixth features from YearPredictionMSD. The number of queries was increased from 1000 to 5000 for each evaluation.

Additionally, we set two query strategies: (1) **FR-index strategy**: process queries with the assistance of FR-Index instances. (2) **Scanning strategy**: broadcast queries to all servers and each server processes queries locally. Such strategy is similar to traditional data retrieval in distributed file systems.

Note that for FR-Index, two rounds of queries were sent to the cluster in order. The first round was used to update indexing instances, and the second round was used to evaluate the query performance.

Suppose that it costs T_1 time with the first strategy and T_2 time with the second strategy to process a same query set. We define $(T_2 - T_1)/T_2 \times 100\%$ as *time saving ratio* (TSR). If the performance of FR-Index is better than the scanning strategy, $TSR > 0$; else, $TSR < 0$.

Figure 9a and b shows the *TSR* in point/range query processing respectively. In the 4-pod data center, due to the extra cost for query forwarding and storage accessing, FR-Index behaved not very well. In the 6-pod fat-tree topology, however, our proposal can reduce nearly 15% time cost for point queries, and even nearly 20% time cost for range queries. Such observation implies that as the scale of data center increases, FR-Index can significantly accelerate the process of data retrieval compared to the traditional scanning strategy.

6.3.2. *Evaluation on different range selectivity*

For range query, we defined range selectivity as the percentage of indexing space. To evaluate the impact of different selectivity on range query performance, we generated 1000 range queries with selectivity 0.1% and 1%, respectively, then sent these queries to 4-pod and 6-pod Fat-tree networks. Figure 10 shows the query performance in different fat-tree scales.

For selectivity 0.1%, when Fat-tree scales from 16 to 54, we observed a 10% speedup; while for selectivity 1%, the speedup is 18%. According to the indexing space partitioning strategy in Section 3.2, under the same dataset, the indexing space designated to each server in 4-pod Fat-tree is larger than that in 6-pod Fat-tree. It indicates that for a range query, more servers would be involved for query processing in 6-pod Fat-tree than in 4-pod Fat-tree, thus the parallelism contributes to the whole query performance to some extent. When range selectivity increases, the initial server receiving queries could be the bottleneck, because it needs to forward queries to more candidate servers, and false positive cases could greatly increase as well, which causes performance degradation of the whole system.

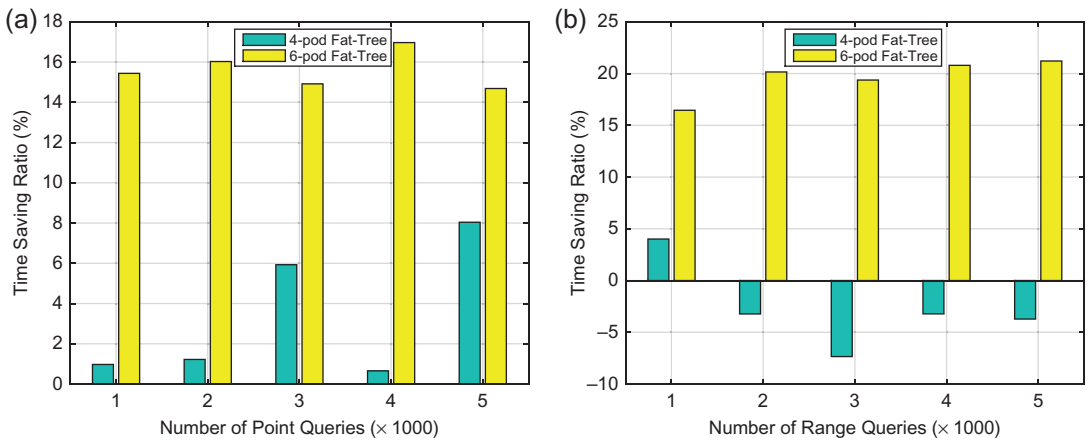

FIGURE 9. *Time Saving Ratio* in query processing. (**a**) Point Queries and (**b**) range queries.

6.4. Comparisons with RT-CAN and RB-index

In this section, we present the performance comparisons between FR-Index and RT-CAN/RB-Index. RT-CAN employs R-tree based indexing structure and CAN based P2P routing protocol to support efficient multi-dimensional data retrieval. RB-Index is another two-layer indexing system designed for modular data centers based on Bcube overlay network. We respectively followed [4] and [9] to implement RT-CAN and RB-Index. The R_{thr} in RT-CAN and RB-Index was set to be slightly larger than the average radius of 70% published index nodes, similar to the R_{thr} in FR-Index. In addition, each server stores 20 000 data items.

We built a 3d FR-Index instance on 4-pod Fat-tree with 16 servers and built an RB-Index on 4-port $Bcube_1$ with 16 servers. We also organized similar number of servers to simulate the CAN topology. In addition, the searching space of CAN was divided and randomly designated among the cluster. To construct a 3D RB-Index system, we built one 2D main global index in $Bcube_1$ and four 1D subsidiary global index in $Bcube_0$.

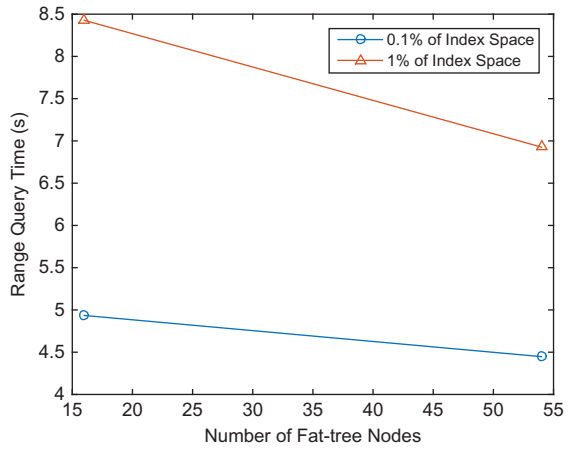

FIGURE 10. Range queries with different selectivities.

6.4.1. *Time cost of index construction*

Figure 11a shows the index construction time of these three schemes. Results imply that the time consumption of FR-Index construction is 18–30% less than that of RT-CAN under the same data volume and server performance. The reason is that as a type of P2P network, the routing protocol in CAN is relatively inefficient, which could increase the time consumption of global index publishing.

Besides, the construction time of RB-Index is more than the construction time of FR-Index and RT-CAN. For FR-Index and RT-CAN, local index construction and global index publishing are performed only in one 3D indexing space. For RB-Index, however, one 2D R-tree and four B-trees need to be constructed in each server to cover all the indexing dimensions, which could greatly increase the overhead of index construction and storage maintenance.

6.4.2. *Time cost of query processing*

We randomly generated 2000 5D point queries and played them to these three indexing systems respectively. Figure 11b and c presents the performance comparisons of point query and range query. Results show that FR-Index saved roughly 20% of query time compared to RT-CAN. Additionally, the performance difference between RT-CAN and FR-Index becomes larger as the scale of data center increases.

For range query, RB-Index preferentially employs its 2d main global index to process queries, then prune the query data to get the final results, which is slower than the 3d FR-Index, as shown in Fig. 11c. For point query, however, the query performance of RB-Index seems equal to the performance of FR-Index. In fact, RB-index employs bloom filters to reduce false positive cases, which could accelerate the query performance.

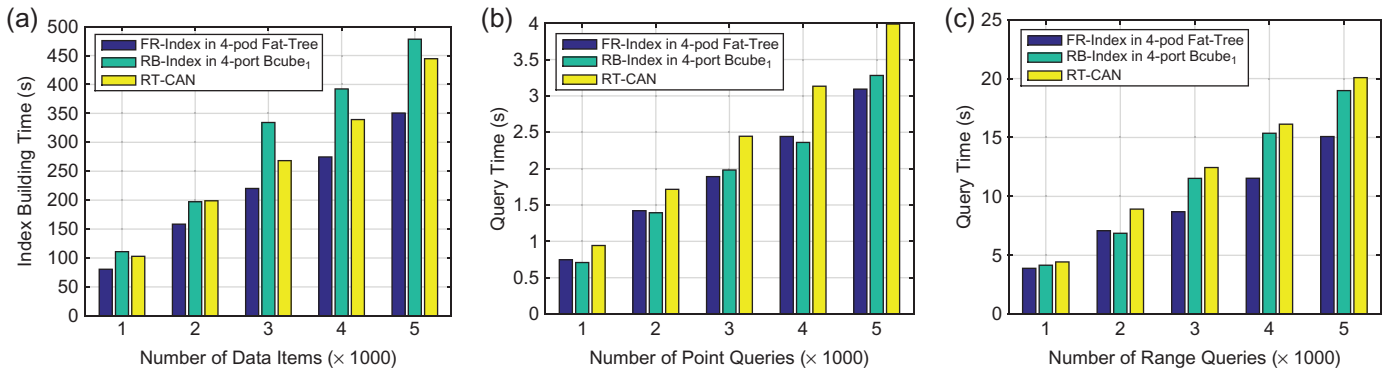

FIGURE 11. Performance comparison between FR-index and RT-CAN/RB-index. (**a**) Indexing construction, (**b**) point queries and (**c**) range queries.

6.5. The scalability of indexing instances

FR-Index is designed to accelerate multi-dimensional data retrieval. With one instance in charge of three dimensions, users can construct any number of indexing instances to satisfy specific indexing requirements. In this section, we compare the performance of index construction and query processing between FR-Index systems with different instances.

For FR-Index$_1$, we constructed an indexing instance with dimensions (2, 3, 4) in Table 4. For FR-Index$_2$, we constructed two instances with dimensions (2, 3, 4) and (2, 5, 6). Figure 12a presents the index construction time of FR-Index$_1$ and FR-Index$_2$. Note that for each instance of FR-Index, each server should build a R-tree and publish selected nodes among the cluster as global index. But we observe that the construction time of FR-Index$_2$ is less than twice of the construction time of FR-Index$_1$. The reason is that when constructing the first instance, we prefetch all the related data for subsequent instances construction. Thus, the disk I/O could be performed only once disregarding of the number of instances built.

To compare the query performance between FR-Index$_1$ and FR-Index$_2$, we generated two types of point query sets: one set Q_{5d} consists of queries with dimensions (2, 3, 4, 5, 6), and the other set Q_{4d} consists of queries with dimensions (2, 3, 5, 6). Figure 12b shows the query performance of FR-Index$_1$ and FR-Index$_2$ under Q_{5d} and Q_{4d}. One observation is that under Q_{5d}, the query performance of FR-Index$_1$ roughly equals the performance of FR-Index$_2$; while under Q_{4d}, FR-Index$_2$ greatly outperforms FR-Index$_1$.

According to the query strategy in Section 4, FR-Index generally assigns the instance whose dimensions best match query set to process the incoming queries. For Q_{5d} with dimensions (2, 3, 4, 5, 6), FR-Index$_1$ would assign instance$_{(2,3,4)}$ to process queries, while FR-Index$_2$ would assign instance$_{(2,3,4)}$ or instance$_{(2,5,6)}$ to finish the job. With the similar processing steps, the query performance of FR-Index$_1$ and FR-Index$_2$ are also very similar. For Q_{4d} with dimensions (2, 3, 5, 6), however, only two indexing dimensions in instance$_{(2,3,4)}$ of FR-Index$_1$ are useful for query processing, which indicates more pruning cost and more false positive cases; while FR-Index$_2$ can simply handle the queries with instance$_{(2,5,6)}$.

To summarize, more indexing instances in FR-Index could involve more construction time and storage cost, while the performance of query processing could be also improved.

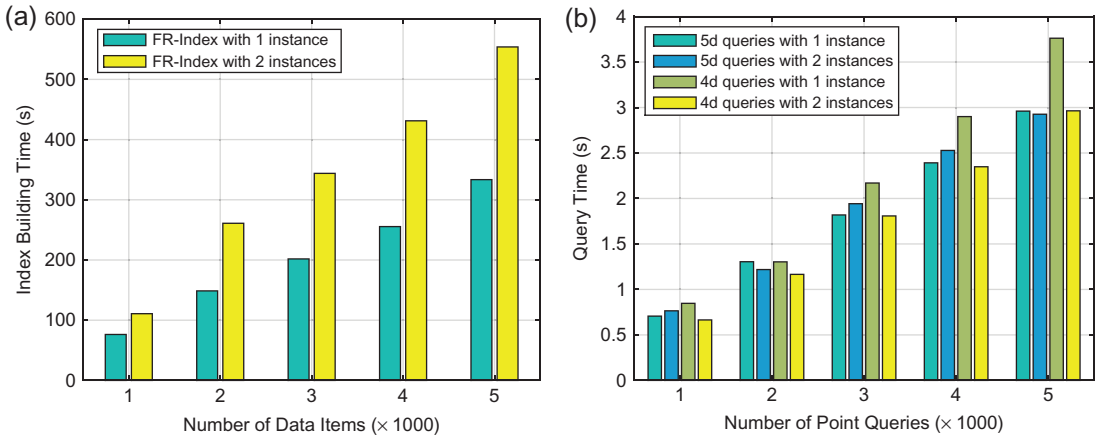

FIGURE 12. Comparisons between FR-Index with different instances. (**a**) Index construction and (**b**) query performance.

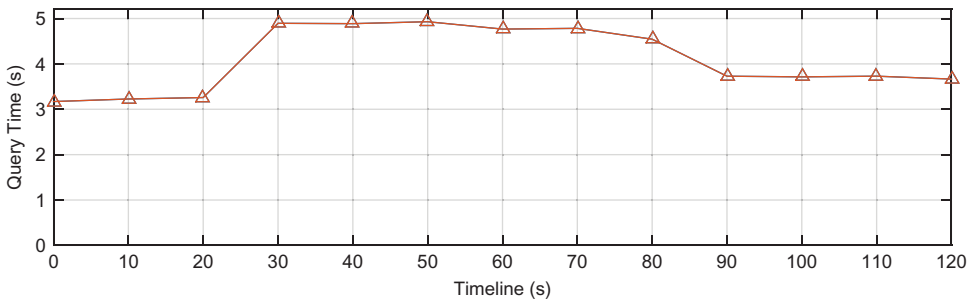

FIGURE 13. Performance of updates.

6.6. Evaluation on updates

As discussed in Section 5, the updates in FR-Index are divided into two categories: update in index instances and update to index instances. Update to index instances is typically performed by the collector during non-peak periods, while the update in index instances, employing a cost model to dynamically adjust published global nodes, could contribute more to query performance in a short period of time. In this section, we evaluate the performance of global indexing update in a FR-Index instance constructed on 4-port Fat-tree.

We generated two types of workloads with 5000 queries from YearPredictionMSD: one is produced by randomly extracting 5000 items from the target dataset, and the other is produced for querying items exclusively stored in four neighboring servers. In addition, we set the updating period of global index as 30 seconds. In this experiment, we first played the first type of queries for three times, then played the second type of queries, i.e. skewed workload, for ten times. The playback period of workloads is 10 seconds. Figure 13 shows the variation of query performance over time.

We observe that at 30 seconds, the skewed workload targeted to four servers was played to FR-Index for the first time. Since the global index in these four servers was still adapted for the first type of workload, the query performance degraded greatly with more false positive cases. At 60 seconds, the four target servers started to adjust the published global nodes to reduce false positive queries based on the cost model discussed in Section 5.1.1. While not until experiencing another 30 seconds, the query latency caused by workload skewness was relieved to some extent. Note that even after the updating, the system performance is not as good as the performance in the first 30 seconds, the reason could be that the four target servers were still overloaded and thus becoming the system bottleneck. This problem can be solved by the underlying file system with techniques like cache mechanism.

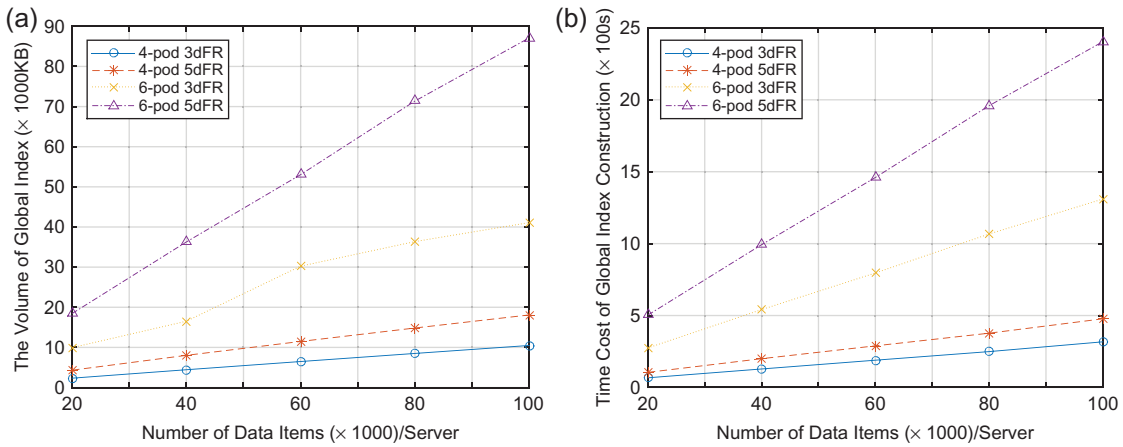

FIGURE 14. Index construction comparison between 3dFR and 5dFR. (**a**) Volume of global index and (**b**) time cost for construction.

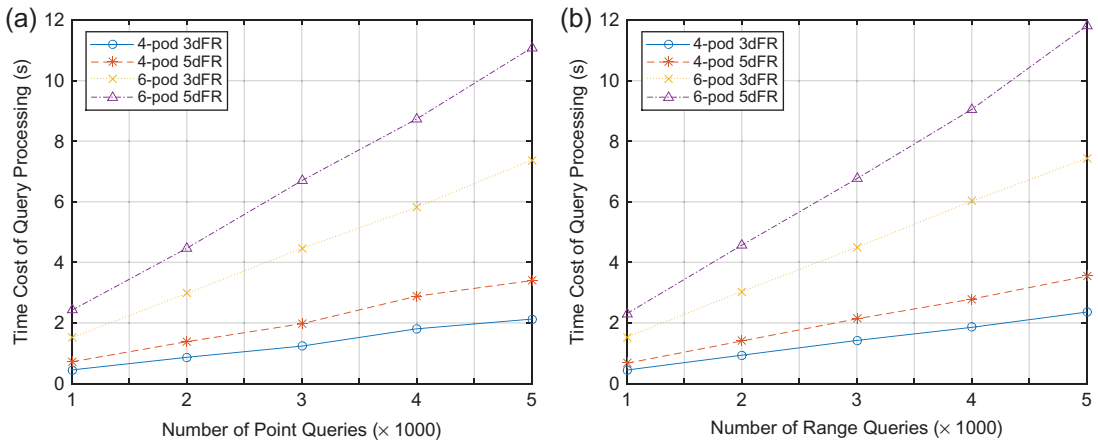

FIGURE 15. Query performance comparison between 3dFR and 5dFR. (**a**) Point queries and (**b**) range queries.

6.7. The validity of dimension selection

The indexing space selection strategy in Section 3.1 constricts the number of dimensions in a index instance to three. To evaluate the performance of indexing space selection in FR-Index, we built two different index instances: one is a 3D index instance, whose index dimensions are the 2th to 4th features in the dataset, called 3*dFR*; the other is a 5D index instance, whose dimensions are the 2th to 6th features in Table 4, called 5*dFR*. The indexing space partitioning and global nodes mapping of 5*dFR* were based on the first three dimensions. Construction comparisons between 3dFR and 5dFR are shown in Fig. 14. One observation is that the global index size and construction time of 3dFR are superior to that of 5dFR, whether in 4-pod Fat-tree or in 6-pod Fat-tree. It is not surprising since the global index of 5dFR involved more data dimensions, which could incur more disk access and consume more storage space.

Based on the index instances of 3dFR and 5dFR, we generated different 5D query sets, produced from the $(2-6)$th features in the dataset, and sent these queries to 3dFR and 5dFR, respectively. The number of data items stored in each server was 20, 000. Figure 15a and b, respectively, shows the time cost of point queries and range queries. Notice that for 5dFR, the results returned by local index need no pruning, while the searching results in 3dFR need to be pruned. Nevertheless, Fig. 15 shows that the time cost of point queries and range queries of 3dFR was better than that of 5dFR under the same workloads. Besides, with the data volume increasing, the increment of query time in 3dFR is less than that of 5dFR.

Since the query efficiency in global index of 5dFR is fairly similar to the searching efficiency of 3dFR, the reason why 3dFR outperformed 5dFR lies in the process of local index searching. As mentioned before, the global index of FR-Index dwells in memory, while the local index, much bigger than global index in size, is stored on disk. Searching in the local index files of 5dFR could incur much more disk access than 3dFR. Therefore, the query time of 3dFR is faster than the query time of 5dFR, which verifies the efficiency and validity of indexing space selection strategy in our proposal.

7. CONCLUSION

This paper presents a distributed multi-dimensional indexing framework for switch-centric data centers with tree-like topology. We design FR-Index, a two-layer multi-dimensional indexing system, and corresponding query processing strategy to accelerate data retrieval in Fat-tree topology. A cost model based on Markov model and Fat-tree routing protocol is proposed for dynamic index selection and updating. We evaluated the performance of FR-Index on Amazon EC2 platform with real dataset and compared FR-Index with RT-CAN/RB-Index. Experiments validate that our proposal is scalable, efficient and lightweight, which can behave better on switch-centric data center.

FUNDING

This work was supported by the National Key R&D Program of China [2018YFB1004703]; the National Natural Science Foundation of China [61872238, 61672353]; the Shanghai Science and Technology Fund [17510740200]; the Huawei Innovation Research Program [HO2018085286]; and the State Key Laboratory of Air Traffic Management System and Technology [SKLATM20180X].

ACKNOWLEDGMENTS

The authors would like to thank Yatao Zhang for his contribution on the early versions of this paper.

REFERENCES

[1] Ghemawat, S., Gobioff, H. and Leung, S. (2003) The Google File System. *Proc. 19th ACM Symp. Operating Systems Principles 2003*, Bolton Landing, NY, USA, 19–22 October, pp. 29–43. ACM, New York, USA.

[2] Lakshman, A. and Malik, P. (2010) Cassandra: a decentralized structured storage system. *Oper. Syst. Rev.*, **44**, 35–40.

[3] DeCandia, G., Hastorun, D., Jampani, M., Kakulapati, G., Lakshman, A., Pilchin, A., Sivasubramanian, S., Vosshall, P. and Vogels, W. (2007) Dynamo: Amazon's Highly Available Key-Value Store. *Proc. 21st ACM Symp. Operating Systems Principles 2007*, Stevenson, Washington, USA, 14–17 October, pp. 205–220. ACM, New York, USA.

[4] Wang, J., Wu, S., Gao, H., Li, J. and Ooi, B.C. (2010) Indexing Multi-Dimensional Data in a Cloud System. *Proc. ACM SIGMOD Int. Conf. Management of Data*, Indianapolis, Indiana, USA, 6–10 June, pp. 591–602. ACM, New York, USA.

[5] Wu, S., Jiang, D., Ooi, B.C. and Wu, K. (2010) Efficient B-tree based indexing for cloud data processing. *PVLDB*, **3**, 1207–1218.

[6] Wu, S. and Wu, K. (2009) An indexing framework for efficient retrieval on the cloud. *IEEE Data Eng. Bull.*, **32**, 75–82.

[7] Hong, Y., Tang, Q., Gao, X., Yao, B., Chen, G. and Tang, S. (2016) Efficient r-tree based indexing scheme for server-centric cloud storage system. *IEEE Trans. Knowl. Data Eng.*, **28**, 1503–1517.

[8] Al-Fares, M., Loukissas, A. and Vahdat, A. (2008) A Scalable, Commodity Data Center Network Architecture. *Proc. ACM SIGCOMM Conf. Applications, Technologies, Architectures, and Protocols for Computer Communications*, Seattle, WA, USA, 17–22 August, pp. 63–74. ACM, New York, USA.

[9] Gao, L., Zhang, Y., Gao, X. and Chen, G. (2015) Indexing Multi-Dimensional Data in Modular Data Centers. *Proc. 26th*

Int. Conf. Database and Expert Systems Applications, Valencia, Spain, 1–4 September, pp. 304–319. Springer, Berlin, Germany.

[10] Gao, X., Li, B., Chen, Z., Yin, M., Chen, G. and Jin, Y. (2015) FT-INDEX: A Distributed Indexing Scheme for Switch-Centric Cloud Storage System. *2015 IEEE Int. Conf. Communications*, London, UK, 8–12 June, pp. 301–306. IEEE, New York, USA.

[11] Liu, Y., Gao, X. and Chen, G. (2015) A Universal Distributed Indexing Scheme for Data Centers with Tree-Like Topologies. *Proc. 26th Int. Conf. Database and Expert Systems Applications*, Valencia, Spain, 1–4 September, pp. 481–496. Springer, Berlin, Germany.

[12] Chen, T., Gao, X. and Chen, G. (2016) The features, hardware, and architectures of data center networks: a survey. *J. Parallel Distrib. Comput.*, **96**, 45–74.

[13] Jagadish, H.V., Ooi, B.C. and Vu, Q.H. (2005) BATON: A Balanced Tree Structure for Peer-to-Peer Networks. *Proc. 31st Int. Conf. Very Large Data Bases*, Trondheim, Norway, 30 August–2 September, pp. 661–672. ACM, New York, USA.

[14] Wang, J., Wu, S., Gao, H., Li, J. and Ooi, B.C. (2010) Indexing Multi-Dimensional Data in a Cloud System. *Proc. ACM SIGMOD Int. Conf. Management of Data*, Indianapolis, Indiana, USA, 6–10 June, pp. 591–602. ACM, New York, USA.

[15] Ratnasamy, S., Francis, P., Handley, M., Karp, R.M. and Shenker, S. (2001) A Scalable Content-Addressable Network. *Proc. 2001 Conf. Applications, Technologies, Architectures, and Protocols for Computer Communications*, San Diego, CA, USA, October, pp. 161–172. ACM, New York, USA.

[16] Ding, L., Qiao, B., Wang, G. and Chen, C. (2011) An Efficient Quad-Tree Based Index Structure for Cloud Data Management. *Proc. 12th Int. Conf. Web-Age Information Management*, Wuhan, China, 14–16 September, pp. 238–250. Springer, Berlin, Germany.

[17] Stoica, I., Morris, R.T., Liben-Nowell, D., Karger, D.R., Kaashoek, M.F., Dabek, F. and Balakrishnan, H. (2003) Chord: a scalable peer-to-peer lookup protocol for internet applications. *IEEE/ACM Trans. Netw.*, **11**, 17–32.

[18] Bin, H. and Yu-Xing, P. (2011) An Efficient Two-Level Bitmap Index for Cloud Data Management. *Proc. Third IEEE Int. Conf. Communication Software and Networks*, Xi'an, China, 27–29 May, pp. 509–513. IEEE, New York, USA.

[19] Guo, C., Lu, G., Li, D., Wu, H., Zhang, X., Shi, Y., Tian, C., Zhang, Y. and Lu, S. (2009) Bcube: A High Performance, Server-Centric Network Architecture for Modular Data Centers. *Proc. ACM SIGCOMM 2009 Conf. Applications, Technologies, Architectures, and Protocols for Computer Communications*, Barcelona, Spain, 16–21 August, pp. 63–74. ACM, New York, USA.

[20] Guo, D., Chen, T., Li, D., Li, M., Liu, Y. and Chen, G. (2013) Expandable and cost-effective network structures for data centers using dual-port servers. *IEEE Trans. Comput.*, **62**, 1303–1317.

[21] Sioutas, S., Sourla, E., Tsichlas, K. and Zaroliagis, C.D. (2015) ART $++$: A Fault-Tolerant Decentralized Tree Structure with Ultimate Sub-logarithmic Efficiency. *Proc. First Int. Workshop on Algorithmic Aspects of Cloud Computing*, Patras, Greece, 14–15 September, pp. 126–137.

[22] Sioutas, S., Triantafillou, P., Papaloukopoulos, G., Sakkopoulos, E., Tsichlas, K. and Manolopoulos, Y. (2013) ART: sub-logarithmic decentralized range query processing with probabilistic guarantees. *Distrib. Parallel Databases*, **31**, 71–109.

[23] Sioutas, S., Sourla, E., Tsichlas, K. and Zaroliagis, C.D. (2015) D3-tree: A Dynamic Deterministic Decentralized Structure. *Proc. 23rd Annual European Symposium on Algorithms*, Patras, Greece, 14–16 September, pp. 989–1000. Springer, Berlin, Germany.

[24] Brodal, G.S., Sioutas, S., Tsichlas, K. and Zaroliagis, C.D. (2015) D^2-tree: a new overlay with deterministic bounds. *Algorithmica*, **72**, 860–883.

[25] Kokotinis, I., Kendea, M., Nodarakis, N., Rapti, A., Sioutas, S., Tsakalidis, A.K., Tsolis, D. and Panagis, Y. (2016) NSM-Tree: Efficient Indexing on Top of NoSQL Databases. *Proc. Second Int. Workshop on Algorithmic Aspects of Cloud Computing*, Aarhus, Denmark, 22 August, pp. 3–14. Springer, Berlin, Germany.

[26] Ciaccia, P., Patella, M. and Zezula, P. (1997) M-Tree: An Efficient Access Method for Similarity Search in Metric Spaces. *Proc. 23rd Int. Conf. Very Large Data Bases*, Athens, Greece, 25–29 August, pp. 426–435. Morgan Kaufmann, MA, USA.

[27] Schmidt, C. and Parashar, M. (2008) Squid: enabling search in DHT-based systems. *J. Parallel Distrib. Comput.*, **68**, 962–975.

[28] Lee, J., Lee, H., Kang, S., Kim, S.M. and Song, J. (2007) CISS: an efficient object clustering framework for dht-based peer-to-peer applications. *Comput. Netw.*, **51**, 1072–1094.

[29] Ganesan, P., Yang, B. and Garcia-Molina, H. (2004) One Torus to Rule Them All: Multi-dimensional Queries in p2p Systems. *Proc. 7th Int. Workshop on the Web and Databases: Colocated with ACM SIGMOD/PODS*, Paris, France, 17–18 June, pp. 19–24. ACM, New York, USA.

[30] Zhang, R., Qi, J., Stradling, M. and Huang, J. (2014) Towards a painless index for spatial objects. *ACM Trans. Database Syst.*, **39**, 19:1–19:42.

[31] Bertin-Mahieux, T., Ellis, D.P.W., Whitman, B. and Lamere, P. (2011) The Million Song Dataset. *Proc. 12th Int. Society for Music Information Retrieval Conference*, Miami, FL, USA, 24–28 October, pp. 591–596. University of Miami, Florida, USA.