# Design and Optimization for Distributed Indexing Scheme in Switch-Centric Cloud Storage System

Yuang Liu, Xiaofeng Gao, Guihai Chen
Shanghai Key Laboratory of Scalable Computing and Systems,
Department of Computer Science and Engineering,
Shanghai Jiao Tong University, Shanghai, 200240, China
liuyuang2012@sjtu.edu.cn, {gao-xf, gchen}@cs.sjtu.edu.cn

***Abstract*—The capacity of data management and the query performance are two main metrics for today's cloud storage system. In this paper, we design a two-layer indexing scheme of distributed secondary index on switch-centric data center network (DCN) topologies. We take advantage of the desirable features of switch-centric topologies, such as stability, scalability, and fault tolerance, and successfully improve the query efficiency for cloud storage system. We choose $B^+$-tree as the local layer index to locate data in each local host, while implement segment tree as the global layer index to manage a portion of meta-data published from local hosts. We also optimize the query processing protocols to reduce false positives and network cost. In addition, we propose a top-down index selection method for maintenance. We then analyze the efficiency of query processing theoretically and calculate the expected number of routing cost for each query precisely. Finally we validate the efficiency of our design by experiments and comparison with a previous work.**

***Keywords*—Indexing, Cloud System, Data Center Network**

## I. Introduction

Due to the convenient services and versatile functions, the usage of cloud storage system is becoming a fashion for both individuals and enterprises in recent years. Google's GFS [1] and Bigtable [2], Yahoo!'s Pnuts [3], and Apache Cassandra [4] are the most well-known examples among varieties of systems. On the other hand, researchers and storage providers also meet the challenge of providing powerful, scalable, and efficient data management scheme to organize huge amount of data and support fast query processing. To satisfy these demands, the design of indexing scheme is one method to facilitate effective data analysis and online transactions.

In a key-value distributed storage system, using only the primary key to search for data is not sufficient to support various query processing requirements. Consequently, a common supplement is to consider secondary index. Since such index may distribute randomly among servers, an efficient searching strategy is to construct a global index above the local index, which is known as the *two-layer indexing* [5]–[8]. The global index is responsible for some meta-data information published from servers, while the local index manages the data stored on the host. Each host owns a portion of global index. Typical two-layer indexing examples, including CG-Index [6], RT-CAN [7], and BIDS [8], are based on peer-to-peer (P2P) networks. However, *Data Center Network* (DCN) [9]–[17] overshadows P2P network on scalability, reliability, and security, and gradually becomes a trend in both academia and industry. Therefore, the implementation of powerful indexing schemes on DCN architectures is a new heating topic.

In this paper, we optimize the current two-layer indexing scheme, and design for switch-centric DCN topologies. Switch-centric, especially tree-like topologies, such as Fat Tree [9], VL2 [10], and Aspen Tree [11], are fairly popular designs in research area, and share the features of high bandwidth, cost efficiency, and fault tolerance. We compare and summarize the characteristics of switch-centric architectures and then build our indexing scheme specifically on such topologies to maximize the topological benefits.

In detail, we construct a $B^+$-tree as the local index for each host to index local data, and a global index on top of each local index. Global index maintains the data information globally for all hosts in the DCN, while each host only contain a portion of it due to their responsible data ranges. Based on the features of infrastructure, we use the segment tree, a data structure for storing intervals, as the global index. Compared to traditional table or list, segment tree could reduce the searching time significantly yet remain compatible with key-value storage. False positives in query can be decreased by efficient publishing scheme. Techniques in query can also reduce the effect of false positives and the hops in query consequently.

The maintenance of index is a difficult and seldom referred work. We define the weight of nodes in $B^+$-tree and propose a top-down index selection scheme to reduce the false positives and adapt to the dynamic workload. Besides, the mathematical analysis of point query based on general tree-like topologies is also presented in the paper. To the best of our knowledge, we are the first to estimate the excepted number of hops each point query needs in general switch-centric DCN. It is different from traditional P2P network that we can precisely predict the routing cost per query via physical hop counts, and thus provide strong time guarantee for query efficiency. At last, we offer various numerical experiments to further validate the efficiency of the system and prove the correctness of analysis.

The contributions of this paper are listed as follows:

1. We optimize the two-layer indexing scheme built on switch-centric DCN topologies, including Fat Tree, VL2, etc. We

This work has been supported in part by the National Natural Science Foundation of China (Grant number 61202024, 61472252, 61133006, 61422208), China 973 project (2014CB340303), the Natural Science Foundation of Shanghai (Grant No.12ZR1445000), Shanghai Educational Development Foundation (Chenguang Grant No.12CG09), Shanghai Pujiang Program 13PJ1403900, and in part by Jiangsu Future Network Research Project No. BY2013095-1-10 and CCF-Tencent Open Fund.

X. Gao is the corresponding author.

978-1-4673-7194-0/15/$31.00 ©2015 IEEE

use segment tree as global index and B$^+$-tree as local index to handle query efficiently and manage data effectively.
2. We propose a top-down index update scheme based on the system to select B$^+$-tree nodes published to the global index. The selection of index can reduce false positives and adapt to different distributions of data and queries.
3. We give theoretical analysis on the efficiency of point query. We also design experiments to demonstrate the performance of indexing scheme and prove the correctness of our theoretical analysis on various switch-centric topologies.

The rest of the paper is organized as follows: In Sec. II, we introduce some previous related work. In Sec. III, we list and compare switch-centric topologies that can be applied to our system. Section IV explains the global and local index in detail. Section V discusses the index update scheme. The query processing based on the two layer index is given in Sec. VI. In Sec. VII, we provide theoretical analysis on point query performance. The evaluation of efficiency is given in Sec. VIII. Finally we conclude the whole paper in Sec. IX.

## II. Related Work

With the exponential growth of data, the distributed system is becoming a significantly powerful tool for data management. Rather than classic SQL database, non-relational database is especially suitable for distributed systems. Key-value based systems Cassandra [4], Dynamo [18], and Voldemort [19] are eminent commercial NoSQL applications with eventual consistency. The open source systems HDFS, HBase, and HyperTable, implement Google's GFS [1] and BigTable [2]. They provide good platforms for exploration and optimization.

Data center network (DCN) is the backbone and infrastructure of a data center. The distributed resources in servers are interconnected and transmitted via routers, switches, and links. Stability, scalability, and load balance are advantages of the cloud system using DCN. The conventional DCN topologies, including Fat Tree [9], VL2 [10], and Aspen Tree [11], etc, adopt a switch-centric design. Server-centric topologies, such as Bcube [14], Dcell [15], and FiConn [16] also draw some researches' attention due to their regularity and symmetry.

The design and analysis for two-layer indexing scheme on P2P network are discussed thoroughly in [5]–[7], [20]. However, P2P networks, such as BATON [21] and CAN [22], can no longer meet the requirement of today's cloud systems. Optimization for two-layer indexing on DCN is a new topic. Previous similar work RT-HCN [23] concentrates on the multi-dimension data indexing on HCN, while FT-Index [24] focuses on the one-dimension data indexing on Fat Tree.

## III. Switch-Centric Architectures

Switch-centric is one of the main categories in the architecture of DCN. In switch-centric, switches dominate the interconnection of resources and the routing approach. The conventional three-tier data center design met the problem of oversubscription, bad fault tolerance, and high power consumption. Other newly proposed tree-like topologies, including Fat Tree [9], Aspen Tree [11], and Portland [12], solved these issues. Meanwhile, Jellyfish [13] and SPAIN [17] are examples of switch-centric topologies without tree structure.

TABLE I. List of notations

| Notation | Definition |
|---|---|
| $n$ | Number of switch levels |
| $k$ | Number of ports per switch |
| $L_i$ | Level $i$ |
| $h_i$ | Host $i$ |
| $H$ | Number of supported hosts |
| $S$ | Number of switches per level (except $L_n$) |
| $\beta$ | Density of data |
| $S_i$ | Segment tree on $h_i$ |
| $B_i$ | B$^+$-tree on $h_i$ |
| $p_i$ | Number of pods at $L_i$ |
| $m_i$ | Number of switches per $L_i$ pod |
| $r_i$ | Number of $L_{i-1}$ pods to which an $L_i$ switch connects |
| $c_i$ | Number of connections from an $L_i$ switch to $L_{i-1}$ switches per pod |

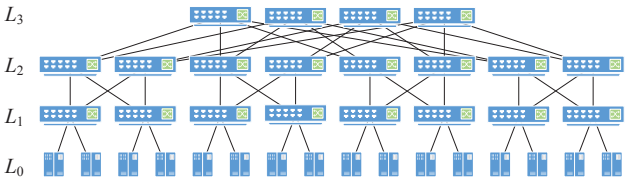



Fig. 1. Fat Tree topology ($n = 3, k = 4$)

We use symbols to denote some common attributes of switch-centric topologies. $k$ indicates the number of ports in each switch, and $n$ represents the number of switch levels. We refer to each switch level from bottom to top as $L_1, L_2, \cdots, L_n$, and host level as $L_0$. We denote number of switches per level (except $L_n$) and supported hosts as $S$ and $H$ respectively. Table I summarizes the notations used in this paper. More notations will be introduced in subsequent texts.

### A. Examples of Switch-Centric Topologies

A typical $k$-ary Fat Tree proposed in [9] consists of hosts at the bottom and three levels of switches. Each $k$-port switch is directly connected to $k/2$ switches or hosts in the lower level. The remaining $k/2$ ports are connected to $k/2$ ports in the upper layer of the hierarchy. However, all $k$ ports of an $L_3$ switch connect to $L_2$. Figure 1 shows an example of Fat Tree.

Aspen Tree [11] is also a multi-root tree switch-centric topology. The general connection rule of links is similar to Fat Tree, while there are redundant links between switches in some adjacent level. Each level of the tree except $L_n$ consists of $S$ switches, while there are only $S/2$ switches in $L_n$. Figure 2 shows an instance of 3-level, 6-port Aspen Tree.

Virtual Layer 2 (VL2) [10] applies a similar tree-like topology to the design. The topology in VL2 is very similar to Fat Tree, except that it adds redundant links between $L_n$ and $L_{n-1}$. Thus VL2 performs a better load balance and fault tolerance. Figure 3 shows an $n = 3, k = 4$ VL2 architecture.

Jellyfish [13] is a representative of switch-centric topology without tree structure. Each switch in Jellyfish is connected to a certain number of servers and other switches, and the switch network is interconnected as a random regular graph.

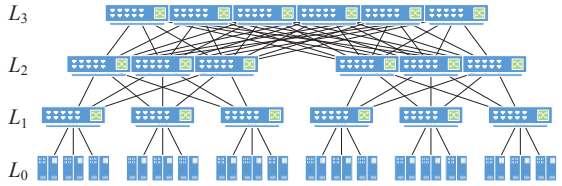



Fig. 2. Aspen Tree topology ($n = 3, k = 6, C = \langle 1, 3\rangle$)

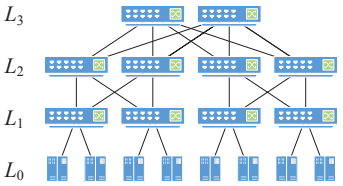



Fig. 3. VL2 topology ($n = 3, k = 4$)

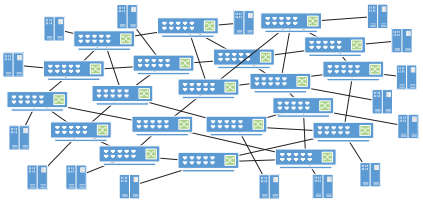


Fig. 4. Jellyfish topology

The advantages of Jellyfish are flexibility, scalability, and high bandwidth. Figure 4 shows a sample Jellyfish architecture.

### B. Comparison of Topologies

The topology determines the connection of switches and routing scheme. We can define some parameters to identify the type of trees and describe the property of them accurately. The comparison of topologies will facilitate following discussion on query processing and performance analysis.

A pod in tree-like topology includes the maximal set of $L_i$ switches that all connect to the same set of $L_{i-1}$ switches [11]. The switches combined together in Fig. 2 are grouped into a pod. Let pod division parameters $p_i$ and $m_i$ describe the number of pods at $L_i$, and the number of switches per $L_i$ pod, respectively. $r_i$ is the number of $L_{i-1}$ pods to which each $L_i$ switch connects, while $c_i$ denotes the number of connections from an $L_i$ switch $s$ to each of the $L_{i-1}$ pods that $s$ neighbors. $r_i$ and $c_i$ define the fault tolerance of each level.

Now, we can use $n$, $k$, and $C = \langle c_2, c_3, \cdots, c_n\rangle$ to identify and differentiate each type of tree. For Fat Tree, $c_2 = c_3 = \cdots = c_n = 1$, while for the topology used in VL2, $c_2 = c_3 = \cdots = c_{n-1} = 1$ and $c_n > 1$. Aspen Tree is a more general form, the restriction is only $c_2 \cdot c_3 \cdot \cdots \cdot c_n > 1$.

Based on the generation algorithm in [11], we quantify the general structure of tree-like topology in Table II. For each level $L_i$ from $L_1$ to $L_{n-1}$, $m_i$ multiplies with a factor of $r_i$ while the trend of $p_i$ reverses. Therefore, the switch number in these level remains a constant $S$. The maximum number of hosts supported by the tree is $H = p_1 \cdot \frac{k}{2} = \frac{k^n}{2^{n-1} \cdot \prod_{j=2}^{n} c_j}$. Note that $k$ should have enough divisors to make $H$ be an integer.

TABLE II. STRUCTURE OF GENERAL TREE-LIKE TOPOLOGIES

| $L_i$ | $L_1$ | $\cdots$ | $L_i$ | $\cdots$ | $L_{n-1}$ | $L_n$ |
|---|---|---|---|---|---|---|
| $p_i$ | $\frac{k^{n-1}}{2^{n-2} \cdot \prod_{j=2}^{n} c_j}$ | $\cdots$ | $\frac{k^{n-i}}{2^{n-i-1} \cdot \prod_{j=i+1}^{n} c_j}$ | $\cdots$ | $\frac{k}{c_n}$ | 1 |
| $m_i$ | 1 | $\cdots$ | $\frac{k^{i-1}}{2^{i-1} \cdot \prod_{j=2}^{i} c_j}$ | $\cdots$ | $\frac{k^{n-2}}{2^{n-2} \cdot \prod_{j=2}^{n-1} c_j}$ | $\frac{k^{n-1}}{2^{n-1} \cdot \prod_{j=2}^{n} c_j}$ |
| $r_i$ | $\frac{k}{2}$ | $\cdots$ | $\frac{k}{2c_i}$ | $\cdots$ | $\frac{k}{2c_{n-1}}$ | $\frac{k}{c_n}$ |
| $c_i$ | 1 | $\cdots$ | $c_i$ | $\cdots$ | $c_{n-1}$ | $c_n$ |

We do not discuss much about non-tree-like DCN architectures in further sections, since they are not as popular as tree-like structures and use arbitrary topologies. However, we argue that two-layer indexing can also be applied to these switch-centric topologies with proper routing scheme given.

## IV. TWO LAYER INDEXING

Since the local data on hosts are distributed in random, we need to build secondary index on the system. With the two layer indexing, we can locate all the possible hosts whose index ranges intersect with the queried key, and then do the local search on these hosts to fetch the data. In this way, we divide the query processing into two phases. We build the global index to accomplish the first phase, and use local index to search on one or more hosts in the second phase.

### A. Global Index and Local Index

We first assign the range of data that each host is responsible for, i.e. the potential indexing range. Here we use a simply scheme [24]. Assume the boundary of the dataset is denoted as $\boldsymbol{D} \subseteq [L, U)$, and we number hosts as $h_1, h_2, \cdots, h_H$ from left to right. Then the potential indexing range of each host

$$pr_i = [L + (i-1) \cdot \tfrac{U-L}{H}, L + i \cdot \tfrac{U-L}{H}), \quad 1 \le i \le H. \quad (1)$$

Equation (1) is only a simple distribution way. The potential range may be arbitrarily assigned in practice as long as hosts can find out the server responsible for the query range.

For searching data on servers efficiently, we build local indexes for data. Here we use B$^+$-tree to index data, which can speed up the local search and reduce I/O cost. In the B$^+$-tree, key-value pairs are pointed to by the leaf nodes, and other inner nodes store keys and pointers. To manage the intervals published by local B$^+$-trees, we apply another tree structure, segment tree [25], as the global index. The data we store and query in segment tree are intervals.

In the pre-processing stage of the system, each B$^+$-tree publish some nodes to the corresponding global index. The information published by the local index will include the key $range$ in the node, and some meta-data, for example local index $ip$ and $address$ of the node in the B$^+$-tree.

### B. Segment Tree

Segment tree [25] is a binary tree structure for storing intervals. In segment tree, the whole range is divided into several elementary intervals, and each of them is represented by a leaf node. The range of an inner node is the union of its children. Evidently, the root represents the whole range. When storing the intervals, we will choose the nodes whose ranges are included in the interval range, while their parents are not.

After collecting the intervals published from each local host, the construction of the segment tree will be executed. The insertion of intervals is recursive. The build of the segment tree is in $O(n \log n)$ time and consumes $O(n \log n)$ storage,

where $n$ denotes the total number of intervals stored and $k$ denotes the number of reported intervals. Similarly, we can use a recursive method to do the point query. From the root node, we determine whether the point intersects with its left child or right child, and then use its child as new root until we meet leaf node. All intervals saved in the nodes on the searching path should be returned. Using the top-down method, the efficiency is in $O(\log n + k)$ time. Compared to tradition lists or tables, the segment tree can reduce searching time and I/O cost remarkably. In the range query, a simple idea is to do point query using the two boundary of the range, and then return all intervals stored in the nodes between the results of two point query in one level.

To store the segment tree effectively, [26] proposed two-tier Endpoints Index (EPI) and MRSegmentTree (MRST) implementation for key-value storage on cloud systems. Moreover, Updates Index (UI) will facilitate the operations of insertion and deletion on segment tree before the reconstruction, although segment tree is a static data structure in theory. We adopt the idea of two-tier segment tree to the global index.

### C. False Positive

When we publish B$^+$-tree nodes to the global index, we will inevitably introduce false positives in query. For example, if the elements in a B$^+$-tree node are {2,4,19,33}, then we publish [2,33] to the global index. However, there are actually only 4 keys in this node, while we cannot fetch the result if we search other keys (3,5,6,...), even if they are in [2,33]. False positive will cause redundant intervals in global index and extra hops in query. [24] proposed two alternatives, FT-Gap and FT-Bloom, to solve the problem. In FT-Gap, the $g$ biggest gap in the intervals published from local host to global index will be eliminated and $g+1$ segments will be published. In FT-Bloom, we use Bloom filter to examine whether a $key$ is in the local B$^+$-tree or not. Both FT-Gap and FT-Bloom can be applied to our system to decrease the false positive ratio. Less false positives can reduce traffic load effectively in either parallel or sequential visit. More details about the two strategies can be referred in our another work [24].

## V. Update and Maintenance

In the pre-processing stage of the system, we publish local B$^+$-tree nodes to the global index. However, we can select different nodes to publish. If the published nodes are closer to leaf nodes, less false positives will generate, yet there will be more global index nodes and update costs. The selection of published nodes is correct if and only if the index is complete and unique [7]. In other words, there must be one and only one node published on the path from each leaf nodes to the root of B$^+$-tree. Among these correct selections, we use a top-down approach to publish a portion of nodes to reduce false positives. When the system has been running for some time, the selection procedure can be reran to update the global index.

We define the weight of a leaf node as the number of times it has been visited during queries. The weight of an inner node is the sum of the weights of its children. Then we can calculate the weight of each node in the B$^+$-tree recursively. The weight of node indicates the frequency of visits. If the weight of a node is relatively high, it implies either some keys in the node (or in its children) are frequently queried or the range of the node is so wide that causes many false positives. Whatever the case is, we should select its descendant nodes as closer to the leaf nodes as possible in order to reduce searching costs or false positives, if the weight of a node is high enough.

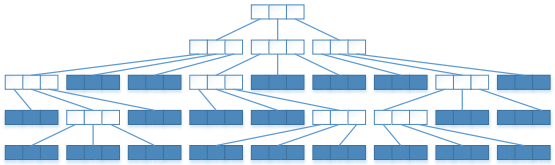


Fig. 5. An example of published nodes in B$^+$-tree ($l = 1$, $s = 1$)

**Algorithm 1:** NodeSelect($root, level$)

```
   Input: The root of a subtree root and current tree level
1  C = {c_1, c_2, ..., c_|C|} ← child(root)
2  if isleaf(root) then
3  |  Publish(root)
4  else if level ≤ l then
5  |  for ∀c_i ∈ C do
6  |  |  NodeSelect (c_i, level + 1)
7  else
8  |  sort C by weight
9  |  for ∀c_i ∈ C ∧ i ≤ s do
10 |  |  NodeSelect(c_i, level + 1)
11 |  for ∀c_i ∈ C ∧ i > s do
12 |  |  Publish(c_i)
```

Based on the idea of weight, we can select index completely and uniquely in a top-down approach. Here we use two parameters $l$ and $s$. All the published nodes are below level $l$ in the B$^+$-tree (root node as level 0). After level $l$, we will publish all but $s$ biggest weight nodes among siblings. For those $s$ nodes, we will recursively publish all but $s$ biggest weight nodes among their descendants, until we reach leaf nodes. Figure 5 is an example of published nodes (dark nodes).

The selection algorithm is shown in Alg. 1. At the beginning, we calculate the weight of each node in the B$^+$-tree. From root to level $l$, we descend to the lower level recursively (Line 6). After level $l+1$, we select the $s$ largest weight nodes to descend the same as before (Line 10), while publish other nodes directly (Line 12). If the node is already a leaf node in this process, then we simply publish it (Line 3).

## VI. Query Processing

The query processing is similar to most two layer indexing schemes. However, to reduce the hops needed in query, we sort the possible hosts and visit them in sequence in the second phase. In point query, the query host is to retrieve the corresponding value of the key $k$. In the first phase, the responsible host for $k$ would be found according to the potential indexing range at first. Then we do point search on the segment tree so that all possible hosts that store the value of the key could be known. Actually, there is no more than one host stores the value among these, while others are false positive hosts. Thus we visit hosts one by one, rather

than forward the query in parallel. In order to diminish the hops taken in this process, we first sort these intervals by their hosts' $ip$ and then we are able to visit the nearest host every time. In the second phase, we forward the query to hosts in consequence, and search the local B$^+$-tree. If the result value is returned from the search, we simply end up here and return the value to the query host. Otherwise we continue to search the next host until there are no more possible hosts.

In range query, given the query range $[l, u]$, all values whose keys intersect with the range are to be returned. Note that the point query is just a special range query when $l = u$. In the first phase, we would find one or more hosts whose potential indexing ranges intersect with the query range. After collecting all possible intervals, we sort these intervals similar to the point query. The second phase visits possible hosts one by one and add values to the result set continuously. Finally we return the set of values to the query host.

## VII. Theoretical Analysis

For the cloud system, the performance of query is mainly determined by the network connection efficiency. Thus, the dominant metric of performance is the hops for each query, rather than efficiency of search on global or local index. Therefore, a good query processing can significantly reduce the hops each query takes. However, the network cost of traditional P2P overlay network is unable to be predicted because logic hops are not necessarily equivalent to physical hops. The desirable characteristics of DCN network make it possible to estimate the hops in a query. Based on the query processing scheme and the uniform distribution of data, we predict the performance of point query in switch-centric topologies.

**Theorem 1.** *The expected number of hops point query takes is expressed by*

$$\begin{aligned}\mathbb{E}[pq] =&(2-\beta)\cdot 2\sum_{i=1}^{n-1}\frac{k^{n-i}}{2^{n-i}\cdot\prod_{j=i+1}^{n}c_j}+4n+\beta-\\ &2\sum_{i=1}^{n}\frac{2^{n-i+1}\cdot\prod_{j=i}^{n}c_j}{k^{n-i+1}}+(2-\beta)\,\mathbb{E}[fp],\end{aligned}\tag{2}$$

*where $\beta$ denotes the density of data, $\beta \in [0, 1]$, and $\mathbb{E}[pq]$ and $\mathbb{E}[fp]$ denote the expected number of hops in point query and false positive, respectively.*

*Proof:* In the first phase, the query will be forwarded to the host which is responsible for the key. If the responsible host is just the query host, then it takes 0 hops. Otherwise, there are $\prod_{j=1}^{i} r_j - \prod_{j=1}^{i-1} r_j$ possible hosts to which it takes $2i$ hops to forward the query. Since the potential ranges of each host are disjoint, there must be only one responsible host. Assume the query is distributed evenly in the range, then the expected number of hops in the first phase is

$$\sum_{i=2}^{n}\frac{\prod_{j=1}^{i} r_j-\prod_{j=1}^{i-1} r_j}{H}\cdot 2i = 2n-\sum_{i=1}^{n}\frac{2^{n-i+1}\cdot\prod_{j=i}^{n}c_j}{k^{n-i+1}}.$$

Next, the query will be forwarded to the possible hosts so as to do local search. The query will be forwarded to the nearest host every time, and total number of these hosts is $\mathbb{E}[fp] + 1$. The intra-pod hops of adjacent hosts and the additional inter-pod hops will both be taken into consideration. So the expected number of hops in the second phase is

$$(\mathbb{E}[fp]+1)+\sum_{i=1}^{n-1}p_i = 2\sum_{i=1}^{n-1}\frac{k^{n-i}}{2^{n-i}\cdot\prod_{j=i+1}^{n}c_j}+\mathbb{E}[fp]+1.$$

Additionally, note that the key is not certainly in any of the hosts. If the key is not in the data set, then the final number of average hops will be twice of the formula above. Denote the number of all keys stored in the system as $\alpha$, then data density $\beta = \alpha/(U - L)$. So the expected number of hops in the second phase after modification is

$$(2-\beta)\cdot\left(2\sum_{i=1}^{n-1}\frac{k^{n-i}}{2^{n-i}\cdot\prod_{j=i+1}^{n}c_j}+\mathbb{E}[fp]\right)+\beta.$$

Finally, the fetched value would be returned back to the initial query host. The cost is the same as the first phase. Combining the three intermediate results together, we can get the conclusion in Eq. (2). ■

## VIII. Performance Evaluation

In the experiment, we simulate the performance of the system on different switch-centric topologies. We store the data randomly across all the hosts and index them by the key. We use non-negative integers in a decided range as keys. Average keys per host ($kph$) vary from 2,000 to 20,000, and the whole dataset $\boldsymbol{D} \subseteq [0, H \cdot kph)$. Notice that not all keys in the range are in the dataset, since we defined data density $\beta$, the number of keys $|\boldsymbol{D}| = \beta \cdot H \cdot kph$.

To show the advantage of segment tree, we compare the number of visited intervals with and without segment tree. Figure 6 shows the number of visited intervals with segment tree and only with traditional list (used in CG-Index [6]). We learn that segment tree can significantly reduce the searching time in query. Moreover, as the published intervals increase, the visited intervals also go up for the list. However, the change is only slight for segment tree.

TABLE III. Network cost on switch-centric topologies

| $\beta$ | 1 | 0.9 | 0.5 | 1 | 0.9 | 0.5 |
|---|---|---|---|---|---|---|
| $kph$ | 4-level, 6-port, Fat Tree $C=\langle 1,1,1\rangle$ | | | 4-level, 6-port, Aspen Tree $C=\langle 3,1,1\rangle$ | | |
| 2000 | 95.76 | 103.59 | 132.23 | 59.52 | 63.52 | 79.72 |
| 5000 | 96.17 | 103.01 | 133.15 | 59.67 | 63.59 | 79.67 |
| 10000 | 94.43 | 102.82 | 133.66 | 59.49 | 63.57 | 79.34 |
| 20000 | 95.84 | 103.41 | 133.72 | 59.13 | 63.69 | 79.63 |
| predict | 95 | 102.7 | 133.5 | 59 | 63.1 | 79.5 |
| $kph$ | 4-level, 6-port, VL2 $C=\langle 1,1,3\rangle$ | | | 3-level, 8-port, Fat Tree $C=\langle 1,1\rangle$ | | |
| 2000 | 41.81 | 44.23 | 53.72 | 53.54 | 57.56 | 72.65 |
| 5000 | 41.63 | 43.97 | 53.68 | 53.93 | 57.75 | 72.95 |
| 10000 | 41.46 | 43.92 | 53.49 | 53.86 | 57.54 | 73.27 |
| 20000 | 41.75 | 43.83 | 53.45 | 53.51 | 57.92 | 72.52 |
| predict | 41 | 43.5 | 53.5 | 53 | 56.9 | 72.5 |

Next, we simulate the performance of point query to verify Eq. (2). The queries generated in this experiment follow the

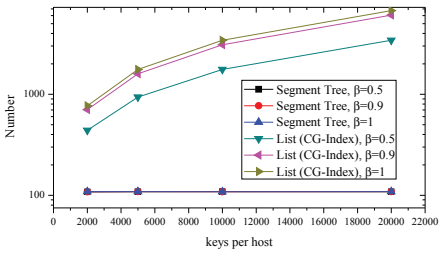
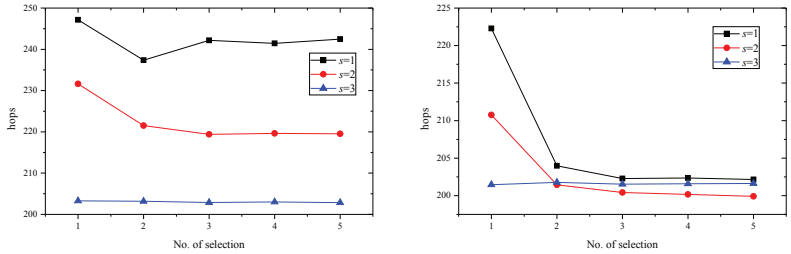



Fig. 6. Number of visited intervals for segment tree compared with CG-Index (in logarithm scale)




(a) $l = 1$ (b) $l = 3$

Fig. 7. Performance of index update scheme on 4-level, 6-port Fat Tree (Zipf)

uniform distribution. We record the average hops of each query under different $kph$ and $\beta$. To compare with our theoretical analysis, we use $pq - (2 - \beta)fp$ as the metric. Table III demonstrates the result on four trees. The prediction value is the expected hops in Eq. (2). From the result we know that our analysis fits the practical situation very well.

To examine the performance of the indexing update scheme, we use Zipfian distribution query to record the average hops in point query under different $l$ and $s$. For each group of experiment, we do 10,000 point queries and run the index selection procedure for 5 times. From Fig. 7 we learn after the second selection, the average hops reduce greatly, although it may fluctuate after more update. We also learn the influence of $s$ on the false positives is less significant if $l$ is higher. In practical, we had better test different combination of $l$ and $s$ in advance based on the distribution of data and query, and also the capacity of global index to achieve a better performance.

## IX. Conclusion

In this paper, we optimize the two-layer indexing in distributed storage system on switch-centric DCN topologies. The local index, $B^+$-tree, and the global index, segment tree, are adopted to manage data. Segment trees are responsible for the meta-data of a portion of intervals published by local $B^+$-tree. Various types of queries can be supported by the two-layer indexing. We also design the top-down scheme of index update. Moreover, we give theoretical analysis on the performance of point query. Finally we simulate the system and use experiments to validate the efficiency of our work.

## References


[1] S. Ghemawat, H. Gobioff, and S.-T. Leung, "The Google file system," *ACM SIGOPS Operating Systems Review*, vol. 37, no. 5, pp. 29–43, 2003.

[2] F. Chang, J. Dean, S. Ghemawat, W. C. Hsieh, D. A. Wallach, M. Burrows, T. Chandra, A. Fikes, and R. E. Gruber, "Bigtable: A distributed storage system for structured data," *ACM Transactions on Computer Systems*, vol. 26, no. 2, p. 4, 2008.

[3] B. F. Cooper, R. Ramakrishnan, U. Srivastava, A. Silberstein, P. Bohannon, H.-A. Jacobsen, N. Puz, D. Weaver, and R. Yerneni, "Pnuts: Yahoo!'s hosted data serving platform," in *VLDB*, vol. 1, no. 2, 2008, pp. 1277–1288.

[4] A. Lakshman and P. Malik, "Cassandra: A decentralized structured storage system," *ACM SIGOPS Operating Systems Review*, vol. 44, no. 2, pp. 35–40, 2010.

[5] S. Wu and K.-L. Wu, "An indexing framework for efficient retrieval on the cloud," *IEEE Data Engineering Bulletin*, vol. 32, no. 1, pp. 75–82, 2009.

[6] S. Wu, D. Jiang, B. C. Ooi, and K.-L. Wu, "Efficient B-tree based indexing for cloud data processing," in *VLDB*, vol. 3, no. 1-2, 2010, pp. 1207–1218.

[7] J. Wang, S. Wu, H. Gao, J. Li, and B. C. Ooi, "Indexing multi-dimensional data in a cloud system," in *SIGMOD*, 2010, pp. 591–602.

[8] P. Lu, S. Wu, L. Shou, and K.-L. Tan, "An efficient and compact indexing scheme for large-scale data store," in *ICDE*, 2013, pp. 326–337.

[9] M. Al-Fares, A. Loukissas, and A. Vahdat, "A scalable, commodity data center network architecture," *ACM SIGCOMM Computer Communication Review*, vol. 38, no. 4, pp. 63–74, 2008.

[10] 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," *ACM SIGCOMM Computer Communication Review*, vol. 39, no. 4, pp. 51–62, 2009.

[11] M. Walraed-Sullivan, A. Vahdat, and K. Marzullo, "Aspen trees: Balancing data center fault tolerance, scalability and cost," in *CoNEXT*, 2013, pp. 85–96.

[12] R. Niranjan Mysore, A. Pamboris, N. Farrington, N. Huang, P. Miri, S. Radhakrishnan, V. Subramanya, and A. Vahdat, "Portland: A scalable fault-tolerant layer 2 data center network fabric," *ACM SIGCOMM Computer Communication Review*, vol. 39, no. 4, pp. 39–50, 2009.

[13] A. Singla, C.-Y. Hong, L. Popa, and P. B. Godfrey, "Jellyfish: Networking data centers randomly," in *NSDI*, vol. 12, 2012, p. 17.

[14] 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," *ACM SIGCOMM Computer Communication Review*, vol. 39, no. 4, pp. 63–74, 2009.

[15] 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 Computer Communication Review*, vol. 38, no. 4, pp. 75–86, 2008.

[16] D. Li, C. Guo, H. Wu, K. Tan, Y. Zhang, and S. Lu, "FiConn: Using backup port for server interconnection in data centers," in *INFOCOM*, 2009, pp. 2276–2285.

[17] J. Mudigonda, P. Yalagandula, M. Al-Fares, and J. C. Mogul, "SPAIN: COTS data-center Ethernet for multipathing over arbitrary topologies," in *NSDI*, 2010, pp. 265–280.

[18] 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 Operating Systems Review*, vol. 41, no. 6, pp. 205–220, 2007.

[19] R. Sumbaly, J. Kreps, L. Gao, A. Feinberg, C. Soman, and S. Shah, "Serving large-scale batch computed data with project Voldemort," in *FAST*, 2012, p. 18.

[20] G. Chen, H. T. Vo, S. Wu, B. C. Ooi, and M. T. Özsu, "A framework for supporting DBMS-like indexes in the cloud," in *VLDB*, vol. 4, no. 11, 2011, pp. 702–713.

[21] H. V. Jagadish, B. C. Ooi, and Q. H. Vu, "BATON: A balanced tree structure for peer-to-peer networks," in *VLDB*, 2005, pp. 661–672.

[22] S. Ratnasamy, P. Francis, M. Handley, R. Karp, and S. Shenker, "A scalable content-addressable network," in *SIGCOMM*, vol. 31, no. 4, 2001.

[23] F. Li, W. Liang, X. Gao, B. Yao, and G. Chen, "Efficient R-tree based indexing for cloud storage system with dual-port servers," in *DEXA*, 2014, pp. 375–391.

[24] X. Gao, B. Li, Z. Chen, M. Yin, G. Chen, and Y. Jin, "FT-INDEX: A distributed indexing scheme for switch-centric cloud storage system," in *ICC*, 2015.

[25] J. L. Bentley, "Solutions to Klee's rectangle problems," Technical report, Carnegie-Mellon Univ., Pittsburgh, PA, Tech. Rep., 1977.

[26] G. Sfakianakis, I. Patlakas, N. Ntarmos, and P. Triantafillou, "Interval indexing and querying on key-value cloud stores," in *ICDE*, 2013, pp. 805–816.