A Universal Distributed Indexing Scheme for Data Centers with Tree-Like Topologies

Yuang Liu, 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
liuyuang2012@sjtu.edu.cn, {gchen,gao-xf}@cs.sjtu.edu.cn

Abstract. The indices in the distributed storage systems manage the stored data and support diverse queries efficiently. Secondary index, the index built on the attributes other than the primary key, facilitates a variety of queries for different purposes. In this paper, we propose U^2-Tree, a universal distributed secondary indexing scheme built on cloud storage systems with tree-like topologies. U^2-Tree is composed of two layers, the global index and the local index. We build the local index according to the local data features, and then assign the potential indexing range of the global index for each host. After that, we use several techniques to publish the meta-data about local index to the global index host. The global index is then constructed based on the collected intervals. We take advantage of the topological properties of tree-like topologies, introduce and compare the detailed optimization techniques in the construction of two-layer indexing scheme. Furthermore, we discuss the index updating, index tuning, and the fault tolerance of U^2-Tree. Finally, we propose numerical experiments to evaluate the performance of U^2-Tree. The universal indexing scheme provides a general approach for secondary index on data centers with tree-like topologies. Moreover, many techniques and conclusions can be applied to other DCN topologies.

Keywords: Two-Layer index · Cloud storage system · Data center network

1 Introduction

Nowadays, the unprecedented development of cloud storage systems is drawing attentions from both academia and industry. The efficient queries in distributed

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.

DOI: 10.1007/978-3-319-22849-5_33

cloud systems require the construction of indices. Other than the index based on the primary key in the key-value storage, we need the *secondary indices* on other attributes for various applications. However, a centralized secondary index both consumes huge volume of storage space and causes the issue of access congestion. Therefore, a common design is to distribute the index on servers, and organize them as a *two-layer indexing scheme*.

The two-layer index consists of a global layer and a local layer. The global index collects the information published from the local indices as an overlay. When processing queries, the host will first request the information on the global index hosts, and further forward the request to the corresponding local index host. The two-layer indexing scheme efficiently solve the problem of the secondary index construction. Nonetheless, current researches on the two-layer indexing [4,15,21–23] are mainly concentrated on the *Peer-to-Peer (P2P) network*, whereas nowadays a typical cloud storage system is organized as a data center. *Data Center Networks (DCNs)* [1,9–12,18–20], the backbone of data centers, have the feature of scalability, reliability, and energy efficiency. The DCNs differ from P2P networks due to their specific physical topologies. Hence, we should take advantage of DCN topologies to design an efficient two-layer indexing scheme. However, there are few researches [7,8,21] regarding such design. Besides, each of them focuses on only one specific DCN topology. Consequently, we are motivated to design a general two-layer indexing scheme based on the properties of DCNs.

In this paper, we consider a series of DCNs with *tree-like topologies*. Nowadays, all commercial DCNs adopt the tree-like topologies. Moreover, they provide high bandwidth, satisfying fault tolerance, and regular structures. Therefore, we take advantage of the tree-like topologies to build an efficient secondary index. We first introduce some representative tree-like topologies including Fat Tree [1], Aspen Tree [20], and VL2 [9], and then extend our discussion to general tree-like topologies. Finally, we construct a Universal TWO-layer indexing built on TREE-like topologies named *U^2-Tree*.

We divide the construction of U^2-Tree into 4 steps. The first step is to build the local index on each host depending on the local data features. Next, we assign the potential indexing range of each host based on the characteristic of data distribution. We then publish the information about local index to the corresponding global index host. Finally, the global index is constructed according to the collected information. We explain the process of each step in detail, and introduce several optimization techniques for them. We further compare the performance and applicable conditions of techniques.

Moreover, we discuss the effects of two index update types, and explain the situations of lazy and eager updates. We also compare and analyze different index tuning schemes, and give the applicable scenarios. We then introduce the re-convergence problem after the link failure based on the fault tolerance of topologies. Finally, we validate our universal indexing design and compare different techniques by simulation results.

The contributions of this paper are: We propose a universal indexing scheme U^2-Tree utilizing the advantages of tree-like DCN topologies. We explain and compare the detailed implementations in index construction and maintenance. More importantly, many techniques and conclusions can also be extended to provide a general platform for secondary index construction on DCNs.

The rest of this paper is organized as follows. Section 2 briefly introduces the relate work. Section 3 is an illustration and comparison of tree-like topologies. Section 4 thoroughly explains the construction processes of U^2-Tree. Section 5 is an discussion about index updating, tuning, and fault tolerance. Section 6 proposes the processing approaches for various types of queries. Section 7 evaluates the performance of techniques in U^2-Tree. Finally, Sect. 8 summarizes and concludes the previous contents.

2 Related Work

The cloud storage systems are developing rapidly with the explosive growth of data. Massive data sets are distributed on several nodes, and nodes are connected to supply a fast access to non-relational databases. Google's Bigtable [3], Amazon's Dynamo [5], Apache Cassandra [13] are well-known examples of commercial distributed systems.

Typically, a cloud storage system is organized as a data center. Data center network (DCN) interconnects all the resources, such as storage and computational data, of a data center. Therefore, DCN architecture plays a significant role. DCN topologies can be categorized into switch-centric DCNs and server-centric DCNs. High bandwidth and better fault tolerance are the main features of switch-centric DCNs including Fat Tree [1], Aspen Tree [20], and Jellyfish [19], etc. Server-centric DCNs, such as Bcube [10], Dcell [11], and HCN [12], are of high scalability and relatively lower cost.

The design of two-layer index maximizes the topological benefits. Previous two-layer indexing designs [4,15,21–23] are mainly built on P2P networks. The indexing schemes regarding the features of DCNs are rarely referred. The multi-dimensional indices RT-HCN [14] and RB-Index [7] integrated HCN and Bcube topologies and R-tree index. The FT-Index [8] leveraged interval tree and B^+-tree on Fat Tree topology.

3 Data Centers with Tree-Like Topologies

Numerous kinds of DCN architecture designs are proposed in the last ten years. The connection of switches and servers varies among different architectures, and thus they have distinct properties, such as scalability, fault tolerance, energy efficiency, etc.

We mainly focus on tree-like topologies, a category of switch-centric DCNs. In the traditional DCN architecture, the widely used three-tier, multi-rooted tree is an example of tree-like topologies. Usually, the tree-like topologies adopt such multi-rooted structure, and divide the switches into multiple layers. The servers are connected to the bottom layer of switches. The disadvantages of traditional three-tier DCNs include limited bandwidth, poor scalability, and high

Table 1. List of notations

Notation	Definition	Notation	Definition
n	Number of switch layers	c_i	Fault tolerance parameter of L_i
k	Port number of switches	$C = \langle c_2, \cdots c_n \rangle$	Fault tolerance vector
L_i	Tree layer i	$[L, U)$	Boundary of data range
h_i	Host i	pr_i	Potential indexing range of h_i
H	Number of hosts	K	Total number of keys
S	Total number of switches	β	Data density

Table 2. Tree-like topologies

Topology	Structure and features
Fat tree [1]	**Layer:** Three layers of identical k-port switches
	Connection: Half ports of an edge switch connect to servers, and others to aggregation switches. Remaining ports of an aggregation switch connect to core switches. All k ports of a core switch connect to aggregation switches
	Expansion: Can be extended to arbitrary levels adopting such connection rule
	Example: Fig. 1 shows a 3-layer, 4-port Fat Tree topology
Aspen Tree [20]	**Layer:** Arbitrary layers of switches
	Connection: Based on the connection rule of Fat Tree. Adding redundant links between layers to reduce the re-convergence time after link failures
	Diversity: A vector C is used to identify different n-level, k-port Aspen Trees
	Example: Fig. 2 shows two 4-layer, 6-port Aspen Trees with different C
Virtual Layer 2 (VL2) [9]	**Layer:** Three layers of switches
	Connection: Based on the connection rule of Fat Tree
	Specification: A clos topology between D_I-port intermediate switches and D_A-port aggregate switches enables an 1:1 over-subscription
	Example: Fig. 3 shows an example of VL2 architecture
Portland [18]	**Connection:** Similar to Fat Tree
	Specification: A fabric manager for better fault tolerance and multicast

cost. Therefore, there are several novel tree-like topologies presented in recent years aiming to solve these problems. On one hand, the individual specifications enable different topologies to have their own characteristics. On the other hand, we can utilize the regular structure these topologies shared to build our universal two-layer indexing scheme. In this section, we will introduce some representa-

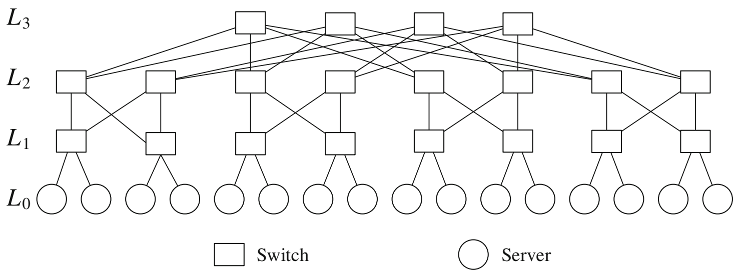
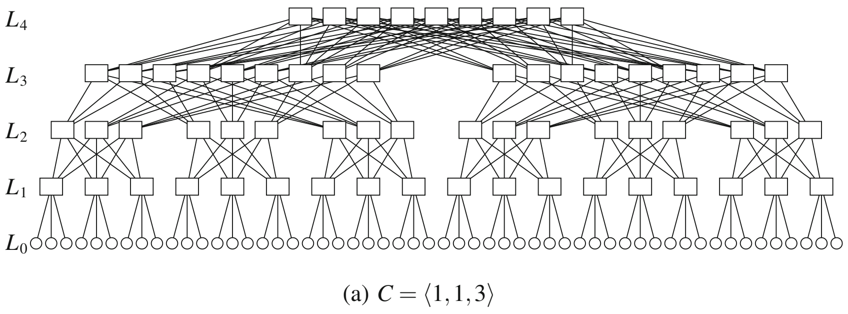


Fig. 1. An example of DCN with a 3-layer, 4-port Fat Tree topology


L_4
L_3
L_2
L_1
L_0
(a) $C = \langle 1,1,3 \rangle$
L_4
L_3
L_2
L_1
L_0
(b) $C = \langle 3,1,1 \rangle$

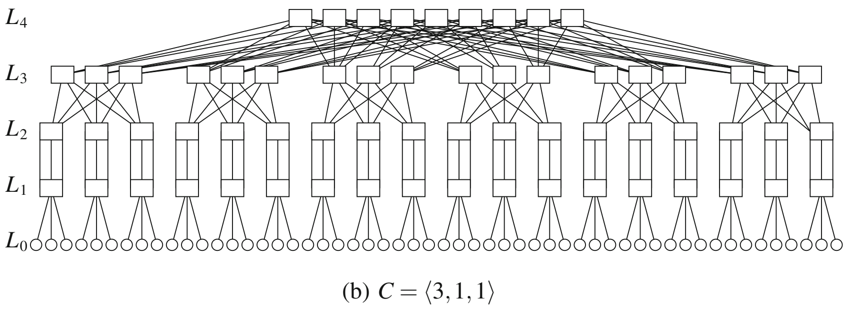


Fig. 2. Examples of DCNs with 4-layer, 6-port Aspen Tree topologies

tive tree-like topologies, including Fat Tree [1], Aspen Tree [20], VL2 [9], and Portland [18], and make a brief comparison among them.

We use k to denote the port number of switches, and n to denote the number of switch layers in tree-like topologies. In this paper, we refer to the switch layers in the three-layer trees as edge layer, aggregate layer, and core layer from bottom to top. Alternatively, we also define them as $L_1, L_2, \cdots, L_n$ in trees with arbitrary layers. Table 1 lists the symbols and their definitions. We will introduce more notations in the following texts. We then give a brief description of popular tree-like topologies in Table 2.

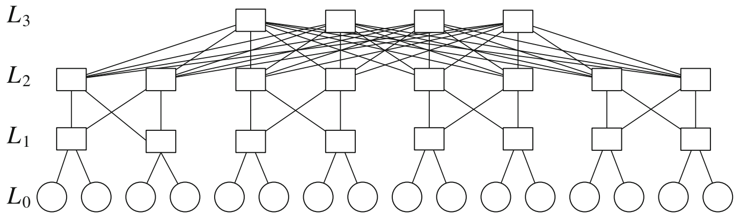


Fig. 3. An example of DCN with a 3-layer VL2 architecture

Table 3. Comparison of tree-like topologies

Topology	Three-tier	Fat tree	VL2	Aspen tree
H	k^n	$\frac{k^n}{2^{n-1}}$	$5k^2$	$\frac{k^n}{2^{n-1} \cdot \prod_{j=2}^{n} c_j}$
S	$\frac{k^n - 1}{k - 1}$	$\left(n - \frac{1}{2}\right) \cdot \frac{k^{n-1}}{2^{n-2}}$	$\frac{k^2 + 6k}{4}$	$\left(n - \frac{1}{2}\right) \cdot \frac{k^{n-1}}{2^{n-2} \cdot \prod_{j=2}^{n} c_j}$
Degree	1	1	1	1
Diameter	$2 \log_2 H$	$2 \log_2 H$	$2 \log_2 H$	$2 \log_2 H$
BiW	1	$\frac{H}{2}$	$\frac{H}{2}$	$\frac{H}{2}$
BoD	$\frac{k-1}{k^2} \cdot H^2$	H	H	H

From previous descriptions we can find that tree-like topologies adopt similar structures. Actually, the Fat Tree is a special instance of Aspen Trees in terms of topology. We define the maximal set of L_i switches connecting to the same set of L_{i-1} switches as a pod [20]. Then we can quantify the fault tolerance of level i c_i as the number of links from an L_i switch s to each L_{i-1} pod that s connects to. The fault tolerance vector $C = \langle c_2, c_3, \cdots, c_n \rangle$ since c_1 is always 1. $C = \langle 1, 1, \cdots, 1 \rangle$ for Fat Tree, while the vector is arbitrary for Aspen Tree.

C affects the fault tolerance of the tree, which can be reflected in the density of links. Fat Tree has a poor fault tolerance compared to Aspen Tree and VL2, while Fat Tree supports the most number of hosts when n and k are determined. However, the topology of VL2 is a "best" choice for both better fault tolerance and more supported hosts [20].

Other than the fault tolerance, we can compare tree-like topologies from several aspects. We first compare the maximum number of supported hosts H, and the total number of switches S in the same scale among topologies. We also compare the degree, the number of links from each host to switches, and the diameter, the longest path length of any pairs of hosts, of each topology. Finally, we compare the bisection bandwidth (BiW), the minimal possible bandwidth between the segmented pair of the network, and the bottleneck degree (BoD), the maximal number of flows over a single link under an all-to-all communication model. The comparison results are shown in Table 3.

4 The U^2-Tree

The universal two-layer indexing scheme U^2-Tree consists of two main parts, the local index and the global index. In the first stage of the index construction, each host builds the local index based on its local data. Then, the whole data range is partitioned into small ranges that each global index is responsible for. After that, the local host publishes some information about the data and local index to the corresponding global index host. Finally, the global index host will collect the received information and construct the global index. Each host in the U^2-Tree will maintain a portion of local index and global index. The outline of the construction process is shown in Algorithm 1. Note that for each stage, the construction of index on all hosts can be executed in parallel.

Algorithm 1. U^2-Tree Construction (at h_i side)

```
Input: The data on local host
Output: The universal two-layer index U^2-Tree
1 Construct local index on the given key attribute // Local index construction
2 Scan data and calculate $pr_i.n, pr_i.c, pr_i.l, pr_i.u$ // Potential indexing range
3 Assign $pr'_i \subseteq [L, U)$ by Eqs. 2 and 4
4 Select nodes in the local index // Publishing to global index
5 foreach selected nodes do
6     Publish $(range, ip, pos)$ to the corresponding global index host
7 Collect all information published from $h_j$ // Global index construction
8 Construct the global index based on the information
```

4.1 Local Index Construction

In a cloud system, the data stored on each local host is of extremely large size. Thus, it is necessary to build a local index on each host so as to reduce the searching time and I/O cost. Typically, B-tree and B$^+$-tree, or other search tree structures based on the storage format of data, can be used to build the local index. The nodes in these trees normally have two or more children and thus support efficient query, insertion, and deletion.

The keys we store in the index are non-negative integers, and the integers are unique among hosts, i.e. for each key, it is stored in at most one host. If the keys are not numerical, we can use hash or other techniques to transfer them into integers.

4.2 Potential Indexing Range Assignment

After the construction of local indices, we will determine the *potential indexing range* of each host. The potential indexing range pr_i is the data range that each global index host h_i represents. The pr_i does not intersect with other pr_j $(i \neq j)$, while the union of all the potential indexing ranges $\bigcup_i pr_i$ will be the whole data range $[L, U)$. In this way, each point in the range will correspond to exactly one responsible global index host.

In the index publishing stage, each local host will publish some information about a certain data range to the corresponding global index according to the potential indexing range assignment. Besides, we will also query the data using a portion of global index based on the potential indexing range. Therefore, it is beneficial to assign the potential indexing range in a way that the keys are partitioned evenly on each global index host.

It is quite easy if the data are distributed uniformly on the range. If so, the approach proposed in [8] can be used. We just partition the whole range into H subranges with equal length, and assign them to the corresponding hosts in ascending order, i.e.

$$pr_i = \left[L + i \cdot \frac{U-L}{H}, L + (i+1) \cdot \frac{U-L}{H}\right), \quad 0 \leq i \leq H-1. \tag{1}$$

Otherwise, we need to handle the skewed data distributed on the range. The goal of the assignment is to balance the number of keys published to each global index host. Assume the j-th key distributed on the local host i in ascending order is denoted as k_i^j. Then the keys on host h_i are $K_i = \{k_i^j \mid k_i^0 < k_i^1 < \ldots < k_i^{|K_i|}\}$. If we sort the keys on all hosts and get $k_{i_0}^{j_0} < k_{i_1}^{j_1} < \ldots k_{i_{K-1}}^{j_{K-1}}$, where $K = |\bigcup_i K_i|$, we can assign

$$pr'_i = \begin{cases} [L, b_1)\,, & i = 0, \\ [b_i, b_{i+1})\,, & 1 \leq i \leq H-2, \\ [b_{H-1}, U)\,, & i = H-1, \end{cases} \tag{2}$$

where $b_i = k_{i_{\lfloor iK/H \rfloor}}^{j_{\lfloor iK/H \rfloor}}$. In this way, the number of keys published to each host is almost the same. However, the ideal method is not practical in that we must collect all the keys on hosts and sort them in ascending order. Now that the data stored in a single host are already massive, it is both storage and time intolerable to achieve such kind of task.

Nevertheless, we can use an approximate method to balance the keys. Zhang et al. [24] offered a *Piecewise Mapping Function (PMF)* that can solve the problem. We first use potential indexing range described in Eq. (1) to divide the whole range $[L, U)$ into H subranges, and count the number of keys distributed on each subranges. Moreover, we record the maximum key $pr_i.u$ and the minimum key $pr_i.l$ in each subrange. Note that this step can be done in parallel, and costs only linear time. We denote the count on each subrange as $pr_i.n$, and the cumulative count, i.e. the number of keys smaller than the right boundary of each subrange, as $pr_i.c$. Clearly, $pr_i.c = \sum_{j \leq i} pr_j.n$. Thus we can get an approximate mapping from $x \in pr_i \subseteq [L, U)$ to the key $k_{i_m}^{j_m}$ where

$$m = \begin{cases} pr_{i-1}.c, & x < pr_i.l, \\ \dfrac{pr_i.n}{pr_i.u - pr_i.l}\,(x - pr_i.l) + pr_{i-1}.c, & pr_i.l \leq x \leq pr_i.u, \\ pr_i.c, & x > pr_i.u. \end{cases} \tag{3}$$

Conversely, the potential indexing range assignment can be given by Eq. (2), where

$$b_i = \begin{cases} \dfrac{pr_j.u - pr_j.l}{pr_j.n}\left(\dfrac{iK}{H} - pr_{j-1}.c\right) + pr_j.l, & pr_{j-1}.c < \dfrac{iK}{H} < pr_j.c, \\ \dfrac{1}{2}\left(pr_j.u + pr_{j+n+1}.l\right), & \dfrac{i}{H} = pr_j.c = \ldots = pr_{j+n}.c. \end{cases} \tag{4}$$

As an example, we consider the 15 prime keys $\{2, 3, 5, \ldots, 43, 47\}$ distributed on $[0, 50)$. If we divide them into 5 subranges using Eq. (1), the count of keys in each subranges will be $4, 4, 2, 2, 3$, with a variance of 0.8. However, Eqs. (2) and (4) can balance the counts perfectly, as shown in Table 4.

Table 4. An example of potential indexing range assignment

i	pr_i	$pr_i.n$	$pr_i.c$	$pr_i.l$	$pr_i.u$	b_i	pr'_i	$pr'_i.c$
0	[0,10)	4	4	2	7	-	[0,5.75)	3
1	[10,20)	4	8	11	19	5.75	[5.75,15)	3
2	[20,30)	2	10	23	29	15	[15,26)	3
3	[30,40)	2	12	31	37	26	[26,39)	3
4	[40,50)	3	15	41	47	39	[39,50)	3

In practice, we assign range for several times to achieve a better performance. Algorithm 2 shows the procedure of potential indexing range assignment for multiple times.

Algorithm 2. Potential Indexing Range Assignment

```
Input: Original potential indexing range assignment $pr_i$ and assignment $times$
Output: New assignment $pr'_i$
1 foreach $t \in [1, times]$ do
2     foreach $i \in [0, H)$ do                    // Estimate data distribution
3         Calculate $pr_i.n, pr_i.c, pr_i.l, pr_i.u$
4     foreach $i \in [1, H)$ do                    // Calculate range boundaries
5         Calculate $b_i$ by Eq. 4
6     foreach $i \in [0, H)$ do                    // Update potential indexing range
7         Calculate $pr'_i$ by Eq. 2
8         $pr_i \leftarrow pr'_i$
9 return $pr'_i$
```

4.3 Publishing Scheme

After assigning the potential indexing range, each host will publish some information about the local index to corresponding global index hosts. Since the nodes in the local index typically store keys as the subtree separation values, we will publish the intervals of keys which indicate the possible keys that exist in a subtree of the node. Moreover, we will also publish the ip address of the host and the position of the node stored in the host. Therefore, a tuple of $(range, ip, pos)$ can be published to the global index host to locate the data on the local host if the range of query intersects with the range of node.

However, publishing in such way is plausible but not efficient enough because of the false positives. On one hand, there is no false negative for query, since all possible keys in the subtrees of the nodes are included in the published intervals. On the other hand, there may be many false positives for query since we only publish the minimum and maximum boundaries of the keys. The publishing scheme significantly reduces the size of published information, while it causes the problem of false positives. The false positives will directly increase the hops needed in queries.

Gap Elimination (FT-Gap) and *Bloom Filter (FT-Bloom)*, two methods proposed in [8], can efficiently solve the problem. The false positives can be considered as the gaps in the intervals. For example, if the keys stored in a node are $\{4, 7, 9, 15, 17\}$, we will publish an interval [4,17]. The queries in the "gaps", for instance, [10,14] will cause false positives. Gap elimination will remove several biggest gaps in the interval and publish the remaining segments. Bloom filter uses hash functions to map the keys in the intervals into a bit array with all 0's and set the corresponding positions to 1. In the query phase, we will hash the query key to check whether the bits are 1. Both methods guarantee no false negative and moderate false positives with tolerable additional space.

4.4 Global Index Construction

The global index is on top of the local index logically. The global index collects the meta-data information published by the local hosts, and will arrange them sensibly to facilitate efficient query. Wu et al. [22] used conventional table or list to store the information. Instead, we can also construct more efficient tree data structures to index the interval ranges. Interval tree, segment tree, and priority search tree are common data structures for storing intervals and supporting various queries.

Interval tree [6,16] uses the median of the endpoints of the intervals to separate the intervals into three sets: intervals intersecting with the median, those lying in the left of it, and those lying in the right. The subtrees are built recursively on the last two sets.

Segment tree [2] decides the atomic intervals based on the endpoints of the intervals. Each node corresponds to an atomic interval or the union of some atomic intervals. An interval is stored in the nodes whose union is exactly the range of the interval.

Priority search tree [17] stores two-dimension data in the nodes. The tree is a heap for one dimension and also a binary search tree for the other dimension. The two dimensions correspond to the lower bound and the upper bound for storing intervals.

For n intervals in total, interval tree and priority search tree will cost $O(n)$ storage, while segment tree will cost $O(n \log n)$ storage. The construction times of the three trees are all $O(n \log n)$. Moreover, the point query times of them are all in $O(\log n + k)$, where k is the number of reported intervals. Compared with scanning the lists, these structures can significantly reduce the searching time on global index. Typically, the interval tree and priority search tree are preferred to segment tree, while the segment tree can be modified to support multi-dimensional query which is not applicable to the others.

5 Update and Maintenance

5.1 Index Updating

The insertion and deletion of data after the index construction will cause the updating of index. The updates in the local index should be executed immediately in order to guarantee the correctness of index. However, since sending the updates to the global index causes additional network cost, a common method is to divide the updates into lazy ones and eager ones [22]. The lazy updates are those that may increase false positives while not affect the correctness. The eager updates will cause false negatives and thus the index fail to provide correct results. Therefore, we forward the eager updates immediately to the global index host while do a batch update for lazy updates.

The merges and splits of local index nodes that change previous *pos* of the nodes are considered to be eager updates. Any insertions that enlarge the ranges of local index nodes or lie in the eliminated gaps for FT-Gap will also trigger the eager updates. Moreover, for FT-Bloom, the insertions that change the bit array of bloom filter are also eager updates. On the contrary, the changes that do not affect the published information are lazy updates that can be issued in a batch way.

Frequent updates in trees for global index will cause the unbalance problem. Therefore, we can maintain an additional *updates index* to store the updated intervals. After a certain time period or when the size of the updates index is too large, we destroy the original tree and reconstruct a new one.

5.2 Index Tuning

When we select the intervals (nodes) in the local index to build the global index, we have multiple choices. For each leaf node, we should select at least one node in the path from the root to it. Yet it is enough that we select only one node in this path. Moreover, we can reduce the update cost by selecting only one node. The two properties, index completeness and unique index, guarantee the correctness of the index selection. However, even among these right selections, we still have different choices. For example, we can select the nodes near the leaves in order to reduce false positives. On the contrary, the nodes near the root are unlikely to be split or merged, and the update cost is reduced. There are some index tuning approaches to balance the false positives and update cost.

Top-Down Approach. The top-down approaches focus on false positives of index. For example, we only select the nodes that are under a certain level (namely l) to publish. For sibling nodes under l, we select a portion to publish directly, while recursively select the descendants of other s frequently accessed nodes to publish. The access times can be counted incrementally, while the tuning is updated in batch. The top-down approaches can effectively reduce false positives, with a limited size of published nodes. It is especially suitable for the indices that support skewed query, but are seldom updated.

Bottom-Up Approach. The more sophisticated bottom-up approaches consider cost models. The cost model for each node consists of the cost of query processing and index maintenance. Wang and Wu et al. [21–23] gave different cost models based on the type of data and the network feature. The goal of the index tuning is to select a set of nodes with the least total cost. Therefore, we calculate the cost of a node and its children, and select the one(s) with less cost. Recursively doing the same job until we meet the root in a bottom-up approach, we can get the optimal indexing set. The calculation of the cost and the selection of nodes are both executed in a batch way.

5.3 Fault Tolerance

In Sect. 3, we mentioned that VL2 and Aspen Tree introduce additional links between some layers. These links can help the system react to link failures more conveniently and rapidly. Walraed-Sullivan et al. [20] proposed the Aspen Reaction and Notification Protocol (ANP). The notifications are sent upwards to ancestors located near to a failure with ANP, rather than a global re-convergence. Suppose there is a failed link between L_i and L_{i+1}, the global re-convergence can be avoided so that the convergence time is shortened as long as the failed link occurs along the upward segment, or $c_j > 1 (j \geq i)$ along the downward segment. For an n-level tree, the supported hosts are decreased by half if $c_n = 2$ while the convergence time can speed up $70 \sim 80\,\%$ compared to Fat Tree. VL2 just utilizes the property to gain a better fault tolerance and scalability.

6 Query Processing

U^2-Tree can process kinds of queries, such as point query, range query, and k-NN query.

The range query in U^2-Tree is similar to most other two-layer indexing schemes. The query host first find out the global index hosts with the potential indexing range intersecting with the queried range. The query is then forwarded to search on the global indices. After that, possible hosts storing the data are returned. Since the hosts with more similar ip are connected within less hops, we sort the possible local index hosts by ip and visit them one by one in sequence. Finally, we collect all the data intersecting with the range on these local indices and forward the results to the query host.

The point query is a special case of range query when the bounds of range are equal. The difference are that (1) we only search on one global index host, (2) we can halt the search on the local index hosts if we have retrieved the data.

The k-nearest neighbours (k-NN) query returns the top-k nearest results to the *key* given the query (key, k). We define the density of data β as

$$\beta = \frac{K}{U - L}, \tag{5}$$

where K is the total number of keys on all hosts. Actually, we have already counted K in the potential indexing range assignment and $K = pr_{H-1}.c$. Therefore, we can use β to estimate the ranges of k nearest results and do the range query. We will first query the range $[key - \gamma k/\beta, key + \gamma k/\beta)$, where γ is a scaling parameter typically slightly large than 0.5. If we can find out more than k results in this range, we simply select the k nearest ones and return. Otherwise, we do range query on $[key - (i+1)\gamma k/\beta, key - i\gamma k/\beta)$ and $[key + i\gamma k/\beta, key + (i+1)\gamma k/\beta)$ continuously with increasing i by one for each iteration until the number of results is greater or equal to k.

7 Performance Evaluation

We simulated the U^2-Tree on different tree-like topologies in C++. We generated the non-negative integers as keys randomly among the range $[0, U)$. The keys in the range are unique, while each possible key in the range does not necessarily exist due to the data density (or existence probability) $\beta = K/U$. Table 5 shows the experiment settings.

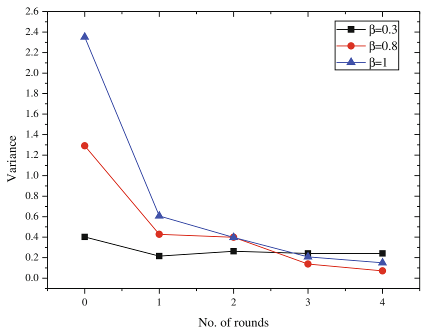
Table 5. Experiment settings

Parameter	Value
Data density (β)	0.3, 0.8, 1
Upper bound of data (U)	500 K, 1 M, 2 M, 3 M
Host number (H)	Depends on topology
Data distribution	Uniform, Zipfian

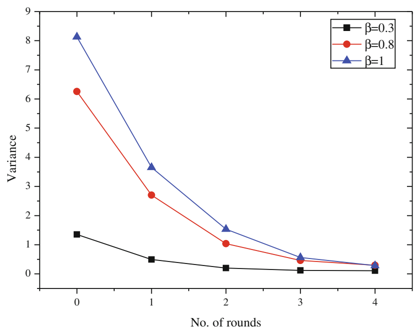
We first evaluate the performance of the potential indexing range assignment algorithm. The inconsistency of keys distributed on global index host is reflected by the variance of the counts. The variance is defined as

$$V = \frac{1}{H} \sum_{i} pr_i.n. \tag{6}$$

Therefore, we use Zipfian distribution data and compare the variance of counts. In Fig. 4, the number of rounds means the result after such times of assignment. From the figure we know that the variance significantly reduces after the assignment. Moreover, the performance is better after more times of assignment.



(a) $U = 500\text{K}$



(b) $U = 3\text{M}$

Fig. 4. Performance of potential indexing range assignment algorithm

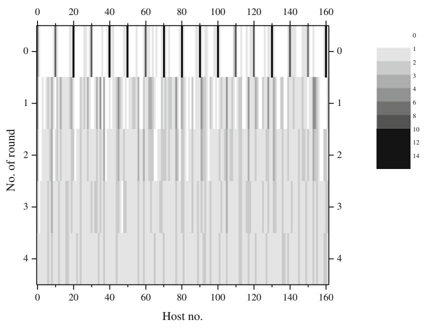


Fig. 5. Heatmap of keys counts on hosts

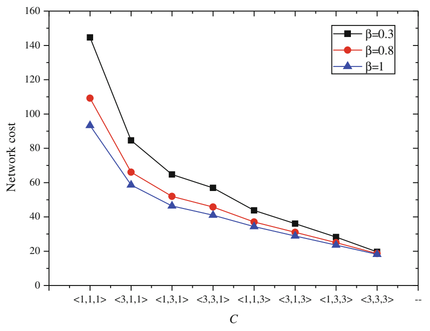
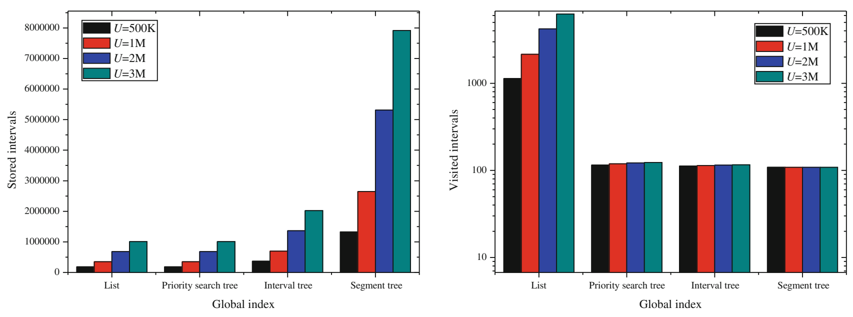


Fig. 6. Network cost on tree-like topologies

The data with different data densities will result in similar variances eventually. Figure 5 shows the key counts on each host after each time of assignment when $U = 3\text{M}$. The darkness of color shows the number of keys on corresponding hosts. We can find that initially the distribution is extremely unbalanced, while the counts vary a little among hosts after the assignments.

Figure 6 shows the average network cost on different 4-level, 6-port tree-like topologies. The network cost is defined as the hop counts on networks for each point query. The topology with $C = \langle 1, 1, 1\rangle$ is a Fat Tree, while others are various Aspen Trees. We can learn from the figure that the average hops decreases as the fault tolerance increases. However, we must point out that it does not mean that Aspen Trees are superior, since the number of hosts also reduces as C increases. The data density β also has an effect on the network cost, because if the queries key does not exist in the data range, the query will be forwarded to all possible local index host, instead of halting halfway.

Finally, we compare the performance of different global indices. We record the number of stored and visited intervals in index construction and query processing. The result is shown in Fig. 7. From Fig. 7(a), the number of stored intervals



(a) Number of stored intervals (b) Number of visited intervals (Log)

Fig. 7. Comparison of global index structures

for priority search tree and interval tree is similar to the traditional list, while segment tree consumes much more space. However, the segment tree can be built faster in practice. Figure 7(b) shows the average number of visited intervals in logarithm scale. The visited intervals of list grow linearly with the total number of keys. On the contrary, the performances of three trees are similarly fine and quite stable with the increasing of stored intervals.

8 Conclusion

In this paper, we proposed the U^2-Tree, a universal distributed secondary index scheme with tree-like DCN topologies. We took the topological benefits to build a two-layer indexing. We explained and compared the techniques in the index construction in detail. We also discussed the index maintenance problems including updating, tuning, and fault tolerance. The U^2-Tree can support several types of query processing efficiently. The experiment evaluated the performance of U^2-Tree, and provided further conclusions about different techniques. In a broad sense, the universal indexing scheme can even be applied to other DCN topologies with proper modifications.

References

1. Al-Fares, M., Loukissas, A., Vahdat, A.: A scalable, commodity data center network architecture. ACM SIGCOMM Comput. Commun. Rev. **38**(4), 63–74 (2008)
2. Bentley, J.L.: Solutions to klee's rectangle problems. Technical report, Carnegie-Mellon University, Pittsburgh (1977)
3. Chang, F., Dean, J., Ghemawat, S., Hsieh, W.C., Wallach, D.A., Burrows, M., Chandra, T., Fikes, A., Gruber, R.E.: Bigtable: a distributed storage system for structured data. ACM Trans. Comput. Syst. **26**(2), 4 (2008)
4. Chen, G., Vo, H.T., Wu, S., Ooi, B.C., Özsu, M.T.: A framework for supporting DBMS-like indexes in the cloud. VLDB. **4**, 702–713 (2011)

5. DeCandia, G., Hastorun, D., Jampani, M., Kakulapati, G., Lakshman, A., Pilchin, A., Sivasubramanian, S., Vosshall, P., Vogels, W.: Dynamo: amazon's highly available key-value store. ACM SIGOPS Operating Syst. Rev. **41**(6), 205–220 (2007)
6. Edelsbrunner, H.: Dynamic data structures for orthogonal intersection queries. Technical report, TU Graz (1980)
7. Gao, L., Zhang, Y., Gao, X., Chen, G.: Indexing multi-dimension data in modular data centers. In: DEXA (2015)
8. Gao, X., Li, B., Chen, Z., Yin, M., Chen, G., Jin, Y.: FT-INDEX: A distributed indexing scheme for switch-centric cloud storage system. In: ICC (2015)
9. Greenberg, A., Hamilton, J.R., Jain, N., Kandula, S., Kim, C., Lahiri, P., Maltz, D.A., Patel, P., Sengupta, S.: VL2: a scalable and flexible data center network. ACM SIGCOMM Comput. Commun. Rev. **39**(4), 51–62 (2009)
10. Guo, C., Lu, G., Li, D., Wu, H., Zhang, X., Shi, Y., Tian, C., Zhang, Y., Lu, S.: Bcube: a high performance, server-centric network architecture for modular data centers. ACM SIGCOMM Comput. Commun. Rev. **39**(4), 63–74 (2009)
11. Guo, C., Wu, H., Tan, K., Shi, L., Zhang, Y., Lu, S.: Dcell: a scalable and fault-tolerant network structure for data centers. ACM SIGCOMM Comput. Commun. Rev. **38**(4), 75–86 (2008)
12. Guo, D., Chen, T., Li, D., Liu, Y., Liu, X., Chen, G.: BCN: Expansible network structures for data centers using hierarchical compound graphs. In: INFOCOM, pp. 61–65. IEEE (2011)
13. Lakshman, A., Malik, P.: Cassandra: a decentralized structured storage system. ACM SIGOPS Operating Syst. Rev. **44**(2), 35–40 (2010)
14. Li, F., Liang, W., Gao, X., Yao, B., Chen, G.: Efficient R-tree based indexing for cloud storage system with dual-port servers. In: DEXA, pp. 375–391 (2014)
15. Lu, P., Wu, S., Shou, L., Tan, K.L.: An efficient and compact indexing scheme for large-scale data store. In: ICDE, pp. 326–337 (2013)
16. McCreight, E.M.: Efficient algorithms for enumerating intersection intervals and rectangles. Technical report, Xerox Paolo Alto Reserach Center (1980)
17. McCreight, E.M.: Priority search trees. SIAM J. Comput. **14**(2), 257–276 (1985)
18. Mysore, R.N., Pamboris, A., Farrington, N., Huang, N., Miri, P., Radhakrishnan, S., Subramanya, V., Vahdat, A.: Portland: a scalable fault-tolerant layer 2 data center network fabric. ACM SIGCOMM Comput. Commun. Rev. **39**(4), 39–50 (2009)
19. Singla, A., Hong, C.Y., Popa, L., Godfrey, P.B.: Jellyfish: networking data centers randomly. In: NSDI. vol. 12, p. 17 (2012)
20. Walraed-Sullivan, M., Vahdat, A., Marzullo, K.: Aspen trees: balancing data center fault tolerance, scalability and cost. In: CoNEXT, pp. 85–96 (2013)
21. Wang, J., Wu, S., Gao, H., Li, J., Ooi, B.C.: Indexing multi-dimensional data in a cloud system. In: SIGMOD, pp. 591–602 (2010)
22. Wu, S., Jiang, D., Ooi, B.C., Wu, K.L.: Efficient B-tree based indexing for cloud data processing. VLDB **3**, 1207–1218 (2010)
23. Wu, S., Wu, K.L.: An indexing framework for efficient retrieval on the cloud. IEEE Data Eng. Bull. **32**(1), 75–82 (2009)
24. Zhang, R., Qi, J., Stradling, M., Huang, J.: Towards a painless index for spatial objects. ACM Trans. Database Syst. **39**(3), 19 (2014)