FT-INDEX: A Distributed Indexing Scheme for Switch-Centric Cloud Storage System*

Xiaofeng Gao**, Binjie Li, Zongchen Chen, Maofan Yin, Guihai Chen, Yaohui Jin
Shanghai Jiao Tong University, Shanghai, 200240, P.R.China
{gao-xf, gchen}@cs.sjtu.edu.cn, {iamlibinjie, chenzongchen, jinyh}@sjtu.edu.cn, {ted.sybil}@gmail.com

***Abstract*—Nowadays, cloud storage systems may contain tens of thousands of servers and large scale data sets, which significantly require efficient data management scheme and query processing mechanism. To fulfill these requirements in modern data centers, the infrastructure of cloud systems, we propose *FT-Index*, a secondary indexing scheme for cloud system with switch-centric topology. FT-Index has a two-layer design. The upper-layer index, called global index, is distributed across different hosts in the system, while the lower-layer index, named local index, is a B^+-tree for local query. We further adopt the Interval tree to reorganize the global index and propose two versions of FT-Index with different publishing methods to lower the rate of false positives and reduce the cost of forwarding queries. We provide detailed theoretical analysis on the upper bound of false positives, physical hops per query, and the relationship between them. We also conduct abundant experiments to validate the efficiency of FT-Index.**

I. INTRODUCTION

Nowadays, cloud storage systems have been widely used in both industrial fields and academic research to reach scalability, manageability, low latency and satisfy the requirements of data-intensive applications, like GFS [1], Cassandra [2], Dynamo [3], etc. With the emergence of different cloud storage systems, diverse indexing schemes have been proposed to support both large-scale analytical jobs and high concurrent OLTP queries efficiently [4]–[6].

Unfortunately, these indexing frameworks have all been proposed on P2P networks, whereas nowadays majority of cloud systems and applications are deployed on data centers, the system infrastructure with large numbers of servers and switches interconnected by Data Center Networks (DCN's). Unlike P2P Network, whose nodes may scatter widely with unbounded physical hop distance, DCN has organized underlying network topology and efficient routing protocols, resulting high-reliability, cost-efficiency, and scalability. The fundamental difference between P2P network and DCN brings great difficulties for the implementation of indexing frameworks on DCNs. As a result, designing distributed indexing schemes on various DCNs is of great significance.

* This work has been supported in part by the China 973 project (2014CB340303), the National Natural Science Foundation of China (Grant number 61202024, 61472252, 61133006, 61371084, 61422208), Shanghai Pujiang Program 13PJ1403900, Shanghai Educational Development Foundation (Chenguang Grant No.12CG09), Natural Science Foundation of Shanghai (Grant No.12ZR1445000), Jiangsu Future Network Research Project No.BY2013095-1-10, and in part by CCF-Tencent Open Fund.

** X.Gao is the corresponding author.

Correspondingly, in this paper we propose *FT-Index*, a secondary indexing scheme for switch-centric DCN, and take Fat-Tree [7] as an illustration since it is one of the most popular DCN architectures in academia, with high scalability, full interconnection bandwidth, low cost with commodity switches, and backward compatibility with IP/Ethernet. Cisco applies the Fat-Tree topology in its Massively Scalable Data Center (MSDC) [8]. We then discuss the generalization of FT-Index to other switch-centric DCN topologies.

FT-Index consists of a local index layer and a global index layer. We construct a B^+-tree as the local index for each host, and then distribute local indices across hosts as the global index for the whole system. To avoid the bottleneck and single-failure problems, each host only maintains a portion of the global index according to its *potential indexing range*. Other than simply indexing all the data as some traditional proposals or publishing the local tree nodes as [5], we deliberately design two novel publishing methods with the help of gap elimination and Bloom filter to decrease the number of false positives and total communication cost (referred as *FT-Gap* and *FT-Bloom*). We further facilitate the query processing by adopting the Interval tree [9] to reorganize the global index.

We then theoretically analyze the efficiency of FT-Index by calculating the expected number of false positives and physical hops per query. Surprisingly, the two factors only differ by a constant and we investigate this relationship thoroughly on Fat-Tree topology. We also compare the performance of FT-Index with CG-index [5] by mass experiments, and validate the efficiency and scalability of our indexing scheme. The generalization is also involved with more topologies. In all, we design a novel distributed indexing scheme for efficient secondary query processing in switch-centric DCN's.

The rest of this paper is organized as follows: Section II describes the architecture of FT-Index, introduces two publishing methods, and proposes query processing algorithms. Theoretical analysis is shown in Section III. Section IV empirically validates the performance and efficiency of FT-Index. Section V concludes the paper.

II. THE DESIGN OF FT-INDEX

There are briefly two types of DCN structures: switch-centric topology such as Fat-Tree [7], VL2 [10], ElasticTree [11], and Aspen Tree [12], where switches take charge of majority interconnection and routing works; and server-centric topology such as BCube [13], DCell [14], FiConn [15] and

978-1-4673-6432-4/15/$31.00 ©2015 IEEE

HCN [16], where servers enable the functions of interconnection and routing while the switches only provide easy crossbar function. Majority of commercial data centers adopt tree-like topologies, which are usually switch-centric. Thus we focus on switch-centric DCN structure and take Fat-Tree as an example to illustrate our design.

A. Fat-Tree Topology

A k-ary Fat-Tree consists of three layer of switches (namely *core* layer, *aggregation* layer and *edge* layer) and at the bottom of the hierarchy are hosts. There are k pods, each with $\frac{k}{2}$ edge switches and $\frac{k}{2}$ aggregation switches, forming a complete bipartite connection betweenness. Every edge switch connects $\frac{k}{2}$ hosts, while every aggregation switch connects $\frac{k}{2}$ core switches. The i^{th} port of a core switch is connected to pod i. In general, a k-ary Fat-Tree can support $\frac{k^3}{4}$ hosts. Fig. 1 is an example Fat-Tree topology with $k = 4$. A two level routing table is introduced to evenly distribute the communication load in the system and achieve full bisection bandwidth at each layer. For more details, please refer [7].

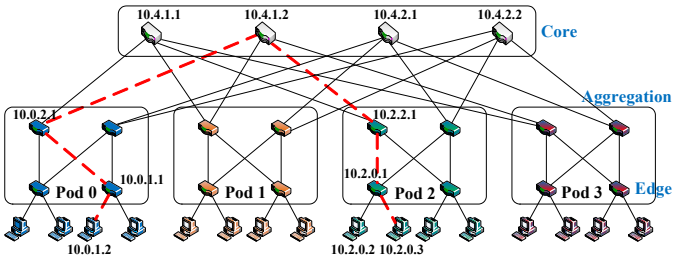


Fig. 1. Fat-Tree Topology ($k = 4$)

B. Potential Indexing Range

Assume there are $n = \frac{k^3}{4}$ hosts in a Fat-Tree with k pods. Denote h_i as the i^{th} host (from left to right, $i \in [1, \frac{k^3}{4}]$). In order to achieve efficiency query processing and load balancing among hosts, we distribute global indices evenly according to the number of hosts and the range of overall data. By assigning different range to each host, given any query, we can figure out the responsible host easily. Assume data in the system are bounded by $\boldsymbol{D} = [L, U)$. Then an example potential indexing range of h_i can be expressed as follows:

$$pr_i = \big[\lceil L + (i-1)(U-L)/n \rceil,\ \lfloor L + i(U-L)/n \rfloor\big)$$

Note that we can assign different potential indexing range for h_i according to the distribution feature of the data.

C. Two-Layer Indexing Architecture

In traditional query processing scheme, queries will be broadcast to all the hosts in the system, where local search is performed in parallel. This strategy is simple but neither efficient nor scalable. Another direct method is to save all the data partitioning information in a centralized server, which needs to handle all the queries in the system, consequently forming the bottleneck and bringing load-balance issue.

In our system, each host h_i holds a B$^+$-tree B_i as its local index to manage its local data. The leaf nodes of the B$^+$-tree are linked together left-to-right to allow efficient range query. Next, each host maintains a global index responsible for data in its pr_i. Thus when given different queries, they will be handled by different hosts with corresponding ranges, eliminating the bottleneck and achieving good query efficiency. Figure 2 illustrates the architecture of our two-layer index.

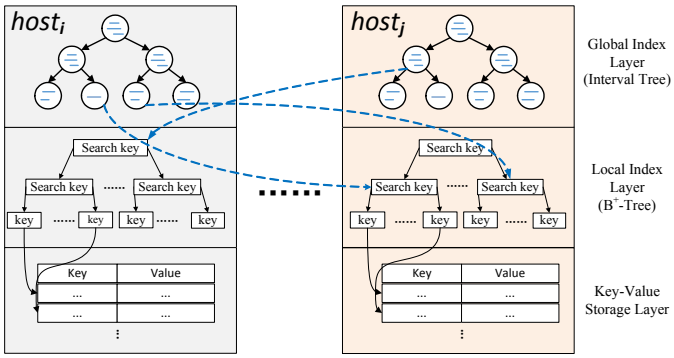


Fig. 2. The two-layer architecture of FT-INDEX

To construct the global index, hosts should publish some local information, such as the ranges of local data, to the hosts with corresponding global index range, so that queries can be forwarded to the correct host containing the required datum after checking the global index. To reach higher query efficiency, we adopt the Interval tree [9], denoted as I_i, to serve as a global index to organize the ranges (segments in Fig. 2) published from other hosts. An Interval tree for a set of n intervals uses $O(n)$ storage and can be built in $O(n \log n)$ time. Thus with Interval tree I_i we can report all intervals intersecting with the queried $range$ (key) in $O(\log n + k)$ time, where k is the number of reported intervals.

Since publishing the whole subtree causes too much space and traffic load, host h_i selects and publishes some nodes in its local tree to h_j based on pr_j [5]. An intermediate node in a local tree only contains range information, so a search query would be forwarded to all the hosts whose published ranges intersect with the queried keys. However, such solution would cause many false positives. For example, a published local node with range [5,20] has local keys {5,6,9,10,17,18,20}, and any query in the range of [11,16] would be falsely forwarded to this host since it intersects with the range [5,20]. To decrease the number of false positives, we design two publishing methods by gap elimination and Bloom filter, and construct FT-GAP and FT-Bloom respectively.

D. FT-Gap: FT-Index with Gap Elimination

The main idea of FT-Gap is to eliminate g biggest gaps when publishing the global index. Originally, host h_i should send some nodes of its local tree corresponding to pr_j to host h_j. In FT-Gap, for the data in the range of pr_j stored on h_i, we choose $g+1$ segments (denoted as $S_{ij} = \{s_{ij}^1, \ldots, s_{ij}^{g+1}\}$) to be published to the global index I_j of host h_j by eliminating the biggest g gaps in pr_j. As the example mentioned above, the existing keys in a potential range of [5,20] are {5,6,9,10,17,18,20}. In FT-GAP, if $g = 2$ then the published ranges would be [5,6], [9,10], and [17,20] instead of [5,20].

Since each segment does not have physical information, we need to find a representative node on the local search tree as a substitute. Define the format of published index items as $(range, ip, addr)$, where $range$ represents the range of each segment in S_{ij} and is expressed by two integers; ip represents the IP address of the local host; and $addr$ is the physical address of the representative local node on B_i at h_i. To find the most suitable $addr$ for each segment in S_{ij}, we can select a local node from h_i's B$^+$-tree in a bottom-up fashion from the leaf with minimum range to its lowest ancestor whose range can fully cover the segment. Algorithm 1 describes the process of finding the $addr$ for a given range.

Algorithm 1: $nodeChoosing(range)$

```
Select a b_i in B+-Tree B_i according to range.minimum;
while (b_i.maximum < range.maximum) do
    b_i = b_i.ancestor;
return b_i.address;
```

For each h_i, it first figures out the actual $range$ according to pr_j. Next it eliminates g gaps, finds $g+1$ representative local nodes, and then publishes the global index information to h_j. At h_j's side, it will reorganize the received information by Interval tree I_j according to the attribute $range$. Algorithm 2 describes the publishing method in FT-GAP at h_i side.

Algorithm 2: $indexPublishing$

```
for j = 1 to n do // h_i publish its local information to each h_j
    range_j = the exact range of the data in pr_j of h_i;
    Eliminate biggest g gaps in range_j as S_ij =
    {s_ij^1,...,s_ij^{g+1}};
    for k = 1 to g + 1 do
        addr_k = nodeChoosing(s_ij^k);
        h_i publishes (s_ij^k, h_i.ip, addr_k) to I_j;
```

E. FT-Bloom: FT-Index with Bloom Filter

A Bloom filter can represent a set $A = \{a_1, a_2, \ldots, a_t\}$ of t elements by using a bit array of s bits, initially all set to 0. l independent hash functions are used, denoted as $f_1, f_2, \ldots, f_l$, each producing an integer in the range $[1, s]$. For each element $a_i \in A$, the bits at positions $f_1(a_i), f_2(a_i), \ldots, f_l(a_i)$ in the array of s bits are set to 1. To check if an element e is in A, we check whether all $h_i(e)$ for $1 \leq i \leq l$ are set to 1. In FT-Bloom, by tuning the parameters s and l appropriately, the probability of false positives can be low enough and the space cost is more efficient than many other methods like hash coding or tree-based data structures.

Bloom filters can be used as a summary shared among hosts to demonstrate each host's existing resources. Different from FT-Gap, we can add a Bloom filter array instead of eliminating g biggest gaps. By doing so, we can clearly inform other hosts that whether the key exists (though with a low probability of false positives). Compared to Alg. 2, the publishing rule in FT-Bloom differs in two main respects. Firstly, the format of published content has changed to $(range, filter, ip, addr)$ where $filter$ represents the corresponding Bloom filter to the $range$. Secondly, for each pr_i we publish one single segment instead of $g+1$ segments. Fig. 3 depicts two different publishing methods when publishing the data in pr_i of [5, 20].

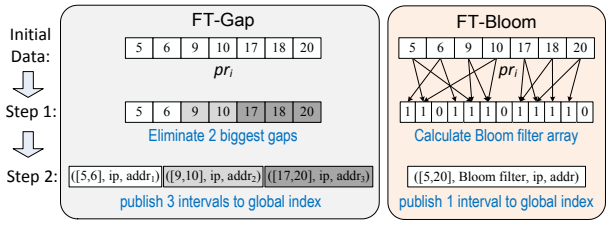


Fig. 3. A comparison of FT-Gap ($g = 2$) and FT-Bloom

F. Query Processing Algorithms

Point Query Processing: We take FT-Gap as an example. When any host receives a point query, it can find out which host's global index is responsible for the query by the potential indexing range. Then the corresponding global index would return all the ranges that cover the key. Since the key exists in only one host, many of them are false positives. To reduce the cost of physical hops, we sort the result set and visit them in order. As shown in Fig. 1, to make the best use of the Fat-Tree topology, we sort the result set with their ip in ascending order (left to right in Fig. 1) such that the relative distance between two consecutively visited hosts is the shortest. The same process works for FT-Bloom as well.

Range Query Processing: For the range query, we also take FT-Gap as an example. We denote a range query as $Q(range)$, where $range = [q_l, q_u]$. To handle a range query, we first find out which hosts have intersection with the queried $range$ according to their potential indexing ranges. After that there exists two conditions: (1) Considering a host h_i whose pr_i fully covers $range$, to assure the completeness of results, we simply take its whole global index set into account and add them to the result set $\boldsymbol{R}$. (2) host h_j has its pr_j partly intersects the $range$. As a generalization from the point query, we need to find all the intervals have intersections with the $range$ and add them to the result set $\boldsymbol{R}$. Then we have similar process of sorting and forwarding.

III. Theoretical Analysis

In this section, we theoretically evaluate the efficiency of FT-Index. Section III-A gives the main metrics of our evaluation and the relationship between them. Section III-B and Section III-C respectively analyzes the performance of FT-Gap and FT-Bloom, taking point query as an example.

A. Holistic Analysis

It is necessary to consider the average hop count per query and try to minimize this metric. We use hop to represent the number of hops for one point query. Another important parameter is the number of false positives the processor generates during the searching process. The goal of FT-Index is to reduce the number of false positives, denoted by fp, and consequently the number of hops hop as much as possible.

Theorem 1 reveals the quantitative relationship between the expected value of hop and fp.

Theorem 1. *In a Fat-Tree topology, the expected numbers of hops and false positives differ only by a constant c, say,*

$$\mathbb{E}[hop] = \mathbb{E}[fp] + c,$$

where k is the number of pods in the Fat-Tree topology and $c = \frac{k^2}{2} + k + 13 - \frac{4}{k} - \frac{8}{k^2} - \frac{16}{k^3}$ is a constant.

Proof: First, the query will be forwarded to the corresponding host. For this host, $\frac{k}{2} - 1$ other hosts are connected to the same edge switches of it and it takes 2 hops to forward the query to them. $\frac{k^2}{4} - \frac{k}{2}$ hosts are in the same pod but belong to different edge switches, and it takes 4 hops for the query to get there. Other $n - \frac{k^2}{4}$ switches pertain to different pods and takes 6 hops. Thus, the expected number of hops is

$$\frac{\frac{k}{2}-1}{n} \cdot 2 + \frac{\frac{k^2}{4} - \frac{k}{2}}{n} \cdot 4 + \frac{n - \frac{k^2}{4}}{n} \cdot 6 = 6 - \frac{2}{k} - \frac{4}{k^2} - \frac{8}{k^3},$$

where we utilize the equation $n = \frac{k^3}{4}$.

After find all positive hosts, the query will be forwarded to all these hosts successively, every time to the nearest host from the current host. The process will not stop until the exact host that stores the key is found. On average, the number of hosts that local search is processed is half of those positive hosts, that is $\frac{1}{2}(\mathbb{E}[fp] + 1)$. Hence, the hops it will take to forward query to such number of hosts is

$$\frac{1}{2}(\mathbb{E}[fp] + 1) \cdot 2 + \frac{1}{2}\frac{k^2}{2} \cdot 2 + \frac{1}{2}k \cdot 2 = \mathbb{E}[fp] + \frac{k^2}{2} + k + 1.$$

Finally, the corresponding information will return to the host that receives the query. The expected number of hops is the same as that in step one, which is $6 - \frac{2}{k} - \frac{4}{k^2} - \frac{8}{k^3}$.

All in all, add up the number of hops in these three steps and we obtain the expected value of hop:

$$\mathbb{E}[hop] = 2(6 - \frac{2}{k} - \frac{4}{k^2} - \frac{8}{k^3}) + \mathbb{E}[fp] + \frac{k^2}{2} + k + 1 = \mathbb{E}[fp] + c.$$

The theorem is thus proved. ■

B. Performance of FT-Gap

Suppose the boundary of data is $[L, U)$. The length of potential indexing range each host takes charge of equals $d = \frac{U-L}{n}$. Theorem 2 estimates the average number of false positives during one point query when employing FT-Gap.

Theorem 2. *When employing FT-Gap, the expected number of false positives is formulated approximately as follows:*

$$\mathbb{E}[fp] = n - 1 - n[1 - (1 - \alpha)^g] \ln N,$$

where n = total no. of hosts, $N = \frac{d}{n} - 1$, $\alpha = (1 - \frac{1}{2e})/N$.

Proof: During the process of point query, the processor finds all the hosts whose ranges cover the key, and regards them as positive. Among those hosts, only one does possess the key, namely true positive, while the others, so-called false positives, does not have the key although their global index covers it. More specifically, we have

$$\mathbb{E}[fp] = n - \mathbb{E}[tp] - \mathbb{E}[neg] = n - 1 - \mathbb{E}[neg],$$

where $fp(tp) = \text{False(True) Positive}$, $neg = \text{Negative}$.

Our next objective is to calculate the expected number of hosts whose ranges does not intersect with the key (i.e., negative hosts). By summing up the number of negatives for each datum in one pr in two ways, we have:

$$\begin{aligned}\mathbb{E}[neg] &= \frac{1}{|pr|} \sum_{\text{datum} \in pr} \mathbb{E}[neg(\text{key} = \text{datum})] \\ &= \frac{1}{d} \sum_{i=1}^{n} \mathbb{E}[\text{No. of data; } host_i \text{ is Neg when key} = \text{datum}] \\ &= \frac{n}{d}\ \mathbb{E}[\text{No. of data; one host does not cover}]\end{aligned}$$

In FT-Gap, data that are not included in the ranges of one certain host can be divided into three categories:

(1). Data smaller than the minimum in ranges. The possibility that each datum emerges in one particular host is $p = \frac{1}{n}$. Without loss of generality, we assume that the potential indexing range is $[1, d]$. Then the expected minimum datum that is distributed to one certain host is $\frac{1}{p} = n$. Consequently, the expected number of data that are smaller than the minimum datum in one host is $n - 1$.

(2). Data larger than the maximum in ranges. Identically, the expected number of such data is also $n - 1$.

(3). Data in the gaps that are obligated We need to estimate the total length of gaps that are eliminated, i.e. the gross length of g largest gaps in the original range. We use the notation l_i to represent the length of the ith largest gap.

First of all, we formulate the expectation of the maximal gap length l_1 as a function of d. As stated, each datum appears in one particular host with possibility $p = \frac{1}{n}$. Thus, the probability that the maximal gap length l_1 is no more than j, namely the length of all gaps is no more than j, is

$$\Pr(l_1 \le j) \approx \left[1 - (1-p)^{j+1}\right]^{\mathbb{E}[\text{No. of gaps}]} = \left(1 - e^{-(j+1)/n}\right)^N$$

where $N = \mathbb{E}[\text{No. of gaps}] = \frac{d}{n} - 1$.

Then we can obtain the expected value of l_1 according to the property of convex sequences.

$$\begin{aligned}\mathbb{E}[l_1] &= -\sum_{j=0}^{d-1} \Pr(l_1 \le j) + \Pr(l_1 \le d) \cdot d \\ &= \sum_{j=0}^{d-1} \left[1 - \left(1 - e^{-(j+1)/n}\right)^N\right] \\ &\approx \frac{1}{2}\left[1 - \left(1 - e^{-1/n}\right)^N + 1 - \left(1 - e^{-\ln N}\right)^N\right] n \ln N \\ &= (1 - \frac{1}{2e}) n \ln N.\end{aligned}$$

The length of the ith largest gap can be considered as the largest gap after we obliterate $i-1$ largest gap. Therefore, we could derive the iteration as follows:

$$\begin{aligned}\mathbb{E}[l_1] &= \left(1 - \frac{1}{2e}\right) n \ln \left(\frac{d - \mathbb{E}[l_1] - \cdots - \mathbb{E}[l_{i-1}]}{n} - 1\right) \\ &= \left(1 - \frac{1}{2e}\right) n \left(\ln N - \frac{\mathbb{E}[l_1] + \cdots + \mathbb{E}[l_{i-1}]}{nN}\right) \\ &= (1 - \alpha)\mathbb{E}[l_1] - \alpha(\mathbb{E}[l_2] + \cdots + \mathbb{E}[l_{i-1}]),\end{aligned}$$

where $\alpha = (1 - \frac{1}{2e})/N$.

By induction, we obtain $\mathbb{E}[l_i] = (1 - \alpha)^{i-1}\mathbb{E}[l_1]$. Hence, the

expected number of data that are obligated is

$$\sum_{i=1}^{g}\mathbb{E}[l_i] = \frac{1-(1-\alpha)^g}{\alpha}\mathbb{E}[l_1] = [1-(1-\alpha)^g](d-n)\ln N.$$

Finally we acquire our conclusion.

$$\begin{aligned}\mathbb{E}[fp] &= n-1-\frac{n}{d}\{2(n-1)+[1-(1-\alpha)^g](d-n)\ln N\}\\ &= (1-\frac{2n}{d})(n-1)-\frac{n(d-n)}{d}[1-(1-\alpha)^g]\ln N\\ &\approx n-1-n[1-(1-\alpha)^g]\ln N.\end{aligned}$$

The definitions of N and α have been mentioned above. ■

With the help of Theorem 1, we can directly obtain the expected number of hops one query will take.

Theorem 3. *The expected number of hops one point query process takes can be formulated approximately as follows when we employ FT-Gap:*

$$\mathbb{E}[hop] = n-1-n[1-(1-\alpha)^g]\ln N + c.$$

C. Performance of FT-Bloom

Firstly, we estimate the expected amount of false positives during one point query when adopting FT-Bloom.

Theorem 4. *When employing FT-Bloom, the expected number of false positives is formulated approximately as:*

$$\mathbb{E}[fp] = (n-1)\left(1-e^{-lt/s}\right)^l.$$

Proof: A Bloom filter [17] uses an s-bit array to represent a t-element set $A = \{a_1, a_2, \ldots, a_t\}$. We assume that the l hash functions used are random and independent of each other.

The probability that a bit is still 0 after all the elements of A are hashed into the Bloom filter is $(1-\frac{1}{s})^{lt} \approx e^{-lt/s}$. Thus, the probability that a host is wrongly considered as positive could be expressed as $[1-(1-\frac{1}{s})^{lt}]^l \approx (1-e^{-lt/s})^l$.

Since only one host stores the key while all the other $n-1$ do not, the formula above then leads to the conclusion. ■

Theorem 1 and Theorem 4 yield the expected number of hops one point query takes when adopting FT-Bloom.

Theorem 5. *The expected number of hops one point query process takes can be formulated approximately as follows when we employ FT-Bloom:*

$$\mathbb{E}[hop] = (n-1)(1-e^{-lt/s})^l + c.$$

Theorem 2-5 show that both FT-Gap and FT-Bloom generate far smaller number of false positives and consequently require fewer hops and less traffic for point queries.

IV. Performance Evaluation

In this section, we evaluate FT-INDEX by simulating a DCN with $n = 128$ hosts. We consider the secondary attributes, which are stored randomly across hosts and indexed by the unique integer keys in a decided range. We set $k = 8$ and gradually increase the number of keys per host, denoted as kph, from 8000 to 880000, so the total range of keys in $\boldsymbol{D}$ is equal to $[0, n \cdot kph)$. We set the number of existing keys as $m = \beta \cdot |\boldsymbol{D}|$, where β is the filling factor. In the following experiments, β is set as 0.8 by default. The queries generated in our experiments follow two different distribution: Uniform distribution and Zipf distribution. We calculate the byte size of global index set on each host (without the maintenance cost of the Interval-tree) as the index size of both FT-Gap and FT-Bloom to compare their performance.

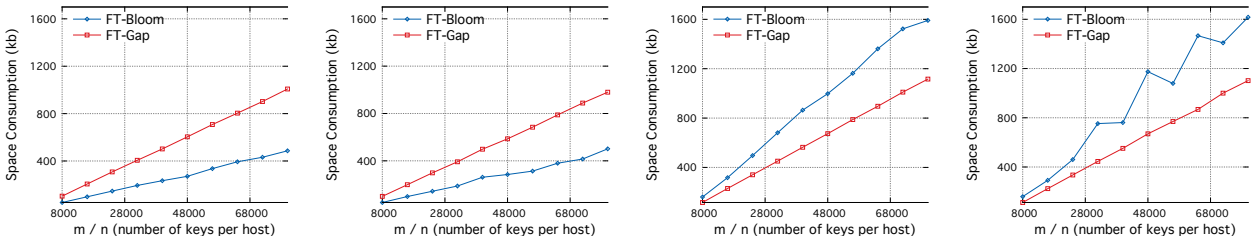
In FT-GAP, the format of index is $(range, ip, addr)$ and there are at most $(g+1)n$ such elements. In FT-Bloom, the index is formed as $(range, ip, bf, addr)$ and for any host there are at most n items. We further assume that $range$ consists of two integers each with 4 bytes. ip and $addr$ each occupy 4 bytes too. We use the Murmur hash similar as Cassandra [18] to generate Bloom filters, which is faster than SHA-based approach and provides as-good collision resistance. To achieve a better performance, we choose the optimal number of hash functions as in Section III, which is $\max(1, \lfloor m/n \cdot \ln 2 \rfloor)$ because we need at least one hash function for implementation.

The performance of our scheme depends on two main metrics: false positive rate fpr and number of physical hops hop during query processing. The fpr is defined as the rate of the number of hosts visited before finding the result. Different from fp in Section III, $\mathbb{E}[fpr] \approx 0.5 \cdot \mathbb{E}[fp]/n$ because we sort the result set before forwarding the query. Moreover, we define the unit of hop as the forwarding cost between two hosts which are directly linked in the underlying physical network and choose the number of hop one query takes as another main metric to evaluate the efficiency and scalability of our system. All experiments are executed for at least 1000 times (1000 for Fig. 4(a)-(d) and 10000 for Fig. 5(a)-(d)) and we exhibit the average values in each figure.

A. Performance of Point Query

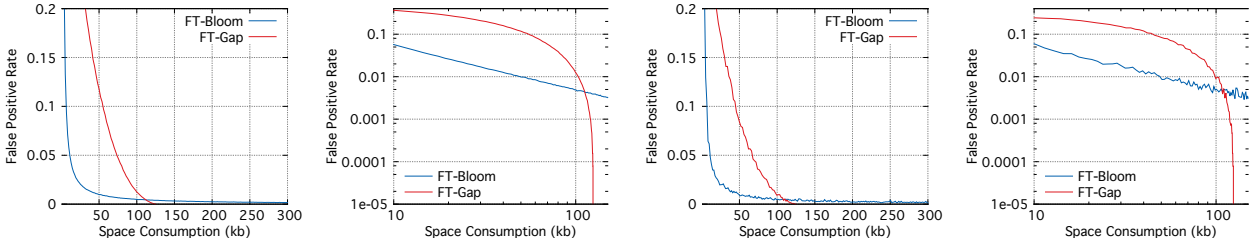
Firstly we consider the relationship between the number of data items and the size of the indices when we fix a false positive rate. As shown in Fig. 4(a), we set $fpr < 1\%$ and gradually increase $\frac{m}{n}$. As data number increases, the size of index of both methods increases almost linearly to keep fpr, resulting good space-efficiency and scalability with the growth of data. We can also tell that when $fpr < 1\%$, FT-Bloom shows better space-efficiency than FT-Gap. Fig. 4(b) exhibits a similar situation when queries follow the Zipf distribution. While in Fig. 4(c)(d), FT-Gap performs better than FT-Bloom when $fpr < 0.3\%$ since the size of Bloom array increases speedily if we raise the accuracy requirement.

Figure 5(a)(b) illustrate the relationship between the size of index and the fpr. We use the uniformed byte size described before and fix $\frac{m}{n} = 10000$. Figure 5(a) shows that with both methods, the false positive rate decreases exponentially with the size of index. However, when we further check Fig. 5(b) in *logarithmic* scale, we find out that the false positive rate decreases more rapidly with FT-Gap. When the $fpr \approx 0.05$, the index size of both versions is the same. If we give a more strict requirement on fpr, FT-Gap would act better. Moreover, we can slightly increase the size of index (by increasing g or the size of Bloom array) to achieve sufficiently low false positive rate and efficient query processing. Similar performance



(a) $fpr < 1\%$ (Uniform) (b) $fpr < 1\%$ (Zipf) (c) $fpr < 0.3\%$ (Uniform) (d) $fpr < 0.3\%$ (Zipf)

Fig. 4. Comparisons of space consumption w.r.t. the average size per host



(a) Uniform (ordinary view) (b) Uniform (logarithmic scale) (c) Zipf (ordinary view) (d) Zipf (logarithmic scale)

Fig. 5. Effect of false positive rate (fpr) w.r.t. space consumption (kb)

can be seen under the Zipf distribution in Fig. 5(c)(d). Due to the nature of Zipf distribution there exist small fluctuations of the fpr when we increase the index size.

B. Performance of Range Query

Here we use FT-Gap as an example to perform range query. Similarly, we set $\frac{m}{n} = 10000$, $k = 8$, $g = 50$, and generate range queries with the uniform distribution. The length of queried ranges is enlarged to evaluate our scheme and obviously a bigger range would result in more searching data and traffic cost. Tab. I shows the performance of our scheme with different range sizes. The experiment shows that our schemes performs better when the range is relatively small, especially when handling point queries. Meanwhile, when the queried range becomes larger, the cost increases almost linearly with the increase of ranges and keeps in an acceptable order of magnitude compared to the range size.

TABLE I
RANGE QUERY PERFORMANCE

$size$	1	128	256	512	640	1024
hop	20	219	285	325	331	335
$size(10^3)$	2	15	30	102	204	640
hop	336	339	343	363	390	504

V. Conclusion

In this paper we propose a distributed indexing scheme (named FT-Index) for cloud storage systems with switch-centric topology. FT-Index consists of two-layers to provide good scalability and load balance. We adopt indexing technologies such as the B$^+$-tree and the Interval tree to organize data set both locally and globally. We also design suitable global index publishing methods with the use of gap elimination and Bloom filters to decrease the probability of false positives and achieve space-efficiency. We then theoretically analyze the expected performance of FT-Index and execute various experiments to validate its efficiency and effectiveness.

References

[1] S. Ghemawat, H. Gobioff, and S.-T. Leung, "The google file system." in *SOSP*, 2003, pp. 29–43.
[2] A. Lakshman and P. Malik, "Cassandra: a decentralized structured storage system." *Operating Systems Review*, pp. 35–40, 2010.
[3] 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." in *SOSP*, 2007, pp. 205–220.
[4] S. Wu and K.-L. Wu, "An indexing framework for efficient retrieval on the cloud." *IEEE Data Eng. Bull.*, pp. 75–82, 2009.
[5] S. Wu, D. Jiang, B. C. Ooi, and K.-L. Wu, "Efficient b-tree based indexing for cloud data processing," in *VLDB*, 2010, pp. 1207–1218.
[6] 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.
[7] M. Al-Fares, A. Loukissas, and A. Vahdat, "A scalable, commodity data center network architecture." in *SIGCOMM*, 2008, pp. 63–74.
[8] "Cisco's massively scalable data center," http://www.cisco.com/c/dam/en/us/td/docs/solutions/Enterprise/Data_Center/MSDC/1-0/MSDC_AAG_1.pdf.
[9] M. De Berg, M. Van Kreveld, M. Overmars, and O. C. Schwarzkopf, *Computational geometry, 3rd edition*. Springer, 2008.
[10] A. Greenberg, J. Hamilton, N. Jain, S. Kandula, C. Kim, P. Lahiri, D. Maltz, P. Patel, and S. Sengupta, "VL2: a scalable and flexible data center network." in *SIGCOMM*, 2009, pp. 51–62.
[11] B. Heller, S. Seetharaman, P. Mahadevan, Y. Yiakoumis, P. Sharma, S. Banerjee, and N. McKeown, "Elastictree: Saving energy in data center networks." in *NSDI*, 2010, pp. 249–264.
[12] M. Walraed-Sullivan, A. Vahdat, and K. Marzullo, "Aspen trees: balancing data center fault tolerance, scalability and cost," in *CoNEXT*, 2013.
[13] C. Guo, G. Lu, D. Li, H. Wu, X. Zhang, Y. Shi, C. Tian, Y. Zhang, and S. Lu, "Bcube: a high performance, server-centric network architecture for modular data centers," in *SIGCOMM*, 2009, pp. 63–74.
[14] C. Guo, H. Wu, K. Tan, L. Shi, Y. Zhang, and S. Lu, "Dcell: a scalable and fault-tolerant network structure for data centers." in *SIGCOMM*, 2008, pp. 75–86.
[15] D. Li, C. Guo, H. Wu, K. Tan, and S. Lu, "Ficonn: Using backup port for server interconnection in data centers." in *INFOCOM*, 2009.
[16] D. Guo, T. Chen, D. Li, M. Li, Y. Liu, and G. Chen, "Expandable and cost-effective network structures for data centers using dual-port servers." *IEEE Trans. Computers*, pp. 1303–1317, 2013.
[17] B. H. Bloom, "Space/time trade-offs in hash coding with allowable errors," *Communications of the ACM*, pp. 422–426, 1970.
[18] https://git-wip-us.apache.org/repos/asf?p=cassandra.git;a=summary.