Indexing Technology for Distributed Storage System

Project for Algorithm: Analysis and Theory

Xiaofeng Gao
Department of Computer Science,
Shanghai Jiao Tong University, Shanghai, China

Abstract. This project introduces the basic indexing technology for distributed data storage system. It includes the brief introduction of two layer (level) indexing scheme, the concepts and the definitions regarding index construction and point query processing. Finally, it lists all the requirements and rules for each group. Please read this document carefully and complete the corresponding tasks.

Keywords: Indexing, Distributed Storage System, Key-Value Query Processing

1 Introduction

The cloud is becoming increasingly more important for data management applications, as it can seamlessly handle huge amounts of data. The elastic and vast processing/storage capacity of clouds has facilitated novel applications requiring efficient and scalable access to massive data. Cloud system are usually built upon data centers, which may contain tens of thousands of servers connected by specific network structures. To achieve query efficiency, the system should guarantee less false positives and low routing cost among servers. Various indexing technology and query processing protocols influence the performance of the cloud system, and in this project, we only focus on key-value stores with two-lay indexing strategy for point query process, built upon a Fat-Tree based network topology. Next, we will introduce each concepts in details in the following sections.

2 Two-Layer Indexing Technology

Key-Value Store. Imaging that we have a huge amount of data items (usually in TB/PB level) stored in a distributed storage system with n servers. A datum can be recognized by its primary key, and to simplify the problem, we assume that the keys are numerical values as integers. We refer such type of data storage as "key-value" stores. Each server stores a subset of data items, and such group of data items does not have any special properties like localities or sequential orders. In this project, we assume that the keys of data items are within the range of $(0, B]$.

Point Query. Point query, denoted as $Q(k)$, means a client sends a query requiring the datum with primary key k, and the system will response this query and send the corresponding datum to the client. For a distributed storage system with multiple servers, initially a client will randomly access to a server s_i and send the query. It then checks the local stores of s_i. If the required datum is not located in this server, it has to hop to another server in the system and repeat the searching process. To evaluate the query efficiency of a distributed storage system, we need to accumulate the searching steps on local machines and the routing steps among servers, and such accumulation is defined as "querying cost". The average querying cost is the evaluation criterion for a distributed storage system, and obviously, the smaller the querying cost is, the better the system performs.

Two-Layer Indexing Scheme. Frankly speaking, if a client sends a point query to a distributed storage system without any additional help, it has to check every server until the required datum is

found. To improve the searching efficiency, researchers proposed a two-layer indexing scheme to reduce the searching and hopping redundancies. Firstly, each server can maintain a *local index* to help check the key values of its local stores. Such index can have various data structures, e.g., B^+-Tree, Red-Black Tree, are all the typical used structures. Next, we will construct a *global index* on each server.

The global index corresponds to a *potential indexing range*, indicating what range of data items this server is responsible for processing a coming query. The potential indexing range should be distinct for each server, and the union of the ranges from each server will form the full set $(0, B]$. If we define PIR_i as the potential indexing range of server s_i, then all other server s_j needs to publish the store information of its local data items whose keys are within PIR_i to s_i. A sample publishing information, for instance, may be a node in s_j's local B^+-Tree as $(blk, range, keys, ip)$, where blk is the disk block number of the node, $range$ is the value range of the B^+-Tree node, $keys$ are search keys in the B^+-Tree node and ip is the IP address of the corresponding server. A server also needs to generate the global index information of its local stores. In all, a server should collect the information of global indices from all the n servers in the system, including itself. By organizing the global index information appropriately, clients can find the potential servers which may store datum with key k easily.

According to the two-layer indexing scheme, at this time once a client sends a point query $Q(k)$ to a server s_i, it will firstly check which server is in charge for this key, and hop to the corresponding server s_j. To simplify the problem, in this project we assume each server maintains a function $f : Key \to IP$, such that $f(k)$ will output the IP address of a server s_i where $k \in PIR_i$ with only 1 calculation time. Next, the client will check the global index in s_j to find out a list of potential servers which may have the required datum. Then the client will hop to these servers (in some order) in the list and search their local indices, until it finds the required datum.

False Positive. False positive means a positive report with negative answers. For instance, from the global index of s_j, a client may find a list of servers which may contains datum with key k, but only one of them "does" contain this datum. All others will refer as "false positive" results. An efficient indexing design should reduce the number of false positives during the point query processing.

Please note that "false positive number" has two meanings, one is theoretical based, denoting the number of false positive records in the global index, which has positive report but actually negative results. Another meaning is implementation based, denoting the "actual" number of false positive records during the implementation of query processing. Such number is usually smaller than the theoretical number, since the searching procedure will terminate once the client finds the datum, even though we still have some remaining records in the list.

3 Fat-Tree Topology

An Overview. Fat-Tree topology is used to connect servers by a multi-root tree structure. A Fat-Tree topology consists of three layer of switches, namely *core* layer switches, *aggregation* layer switches, and *edge* layer switches. Each switch has k ports, meaning it can connect k nodes in the network. In the bottom of the hierarchy are servers.

There are k pods for a k-ary Fat-Tree, each containing $k/2$ edge switches and $k/2$ aggregation switches (k should be an even number). These switches form a complete bipartite graph in each pod. Next, since every edge switch still has $k/2$ free ports, it will directly connect to $k/2$ servers. Similarly, every aggregation switch also has $k/2$ free ports, and it will connect to $k/2$ core switches. There are $(k/2)^2$ k-port core switches. The i^{th} port of any core switch is connected to some aggregation switch at

the i^{th} pod, in consecutive orders. In general, a k-ary Fat-Tree network can support $k^3/4$ hosts. Figure 1 is an example of a Fat-Tree topology with $k = 4$.

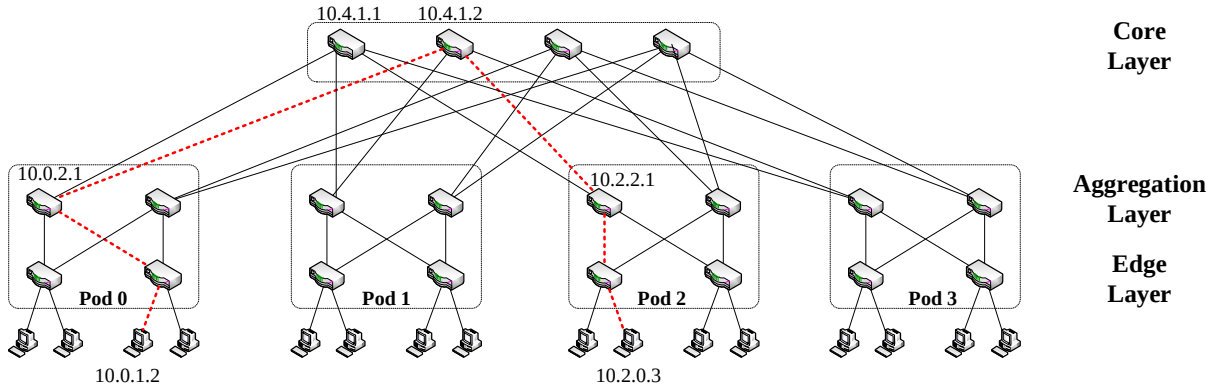

Fig. 1. An Example of a Fat-Tree Topology ($k = 4$)

Addressing Scheme. Fat-Tree topology is associated with specific private IP addressing scheme to help simplify the routing process. In this network, we have different addressing rule to switches and hosts. For pod switches, the form 10.*pod.switch.*1 acts as their IP addresses, where *pod* denotes the pod number (in [0,k-1]), and *switch* means the position of that switch in the pod (in [0, k-1], starting from left to right, bottom to top). Core switches have address of the form 10.$k.i.j$, where (i, j) denotes the coordinates of that switch in $(k/2)^2$ core switch grid (each in [1,$k/2$], starting from top-left to top-right). The address of a server follows the edge switch it is connected to, with the form of 10.*pod.switch.ID*, where ID is the host's position in that subnet (in [2,$k/2$-1], starting from left to right). From Fig. 1, we can also see some examples of private IP address for switches and servers.

Routing Scheme. We have a simple routing scheme for Fat-Tree topology to evenly distribute the communication load and achieve full bisection bandwidth at each level. To do this, we modify the routing tables to allow two-level prefix lookup. Correspondingly, in each routing table at a switch we will have a main table of $(prefix, port)$ and a secondary table of $(suffix, port)$, where $prefix$ is responsible for intra-pod traffic and the $suffix$ is for inter-node traffic. The main table will potentially have an additional pointer to the secondary table. A first-level prefix is terminating if it does not contain any second-level suffixes, and a secondary table may be pointed to by more than one first-level prefix. Whereas entries in the primary table are left-handed (i.e., $/m$ $prefix$ masks of the form 1^m0^{32-m}), entries in the secondary tables are right-handed (i.e. $/m$ $suffix$ masks of the form $0^{32-m}1^m$). If the longest-matching prefix search yields a non-terminating prefix, then the longest-matching suffix in the secondary table is found and used.

Such a two-level lookup can be implemented using a special CAM, called Ternary CAM (TACM) to make the table lookup faster. Furthermore, the routing algorithm used can prevent intra-node traffic from leaving the pod by matching the $prefix$. It also assures that the inter-node traffic evenly spreads upward among the outgoing links to the *core* switches and thus avoids the reordering for subsequent packets. Table 1 is an example routing table at switch 10.2.2.1. An incoming packet with destination IP address 10.2.1.2 is forwarded on port 1, whereas a packet with destination IP address 10.3.0.3 is forwarded on port 3. In addition, Fig. 1 illustrates an example routing path (as red dotted lines) from the source server with address 10.0.1.2 to the destination server with address 10.2.0.3.

Prefix	Output Port
10.2.0.0/24	0
10.2.1.0/24	1
0.0.0.0/0	

⟶

Suffix	Output Port
0.0.0.2/8	2
0.0.0.3/8	3

Table 1. An Example Two-level Routing Table

4 Tasks and Requirements

In this project, you are required to finish two tasks, a design task and an implementation task.

4.1 Design Task:

Given the range B, the server number n with pod number k (where $n = k^3/4$), please design the structure of the local index and the global index for each server s_i, such that on average, your index structure can significantly reduce the searching steps and hopping numbers for point queries, and reduce the number of false positives. Additionally, the size of your indices should be relatively small.

Please also provide theoretical analysis on the size of your indices, the number of false positives, the searching steps locally, and the hopping numbers among servers.

You may consider several questions to improve the performance. For instance,

1. What types of local index is the most appropriate? You can refer B$^+$-Tree, Red-Black Tree, Multiway Tree or any design for searching by comparison of keys.
2. What information at s_j should be published to the server s_i with PIR_i? The information for a global index should contain the IP address of the server s_j, the physical address of the local index on s_j, and the range or other information reflecting the local storage situations. Publishing the whole local tree to a server is impractical, since it will occupy too much space to form a relative small-size indices. Thus, you have to remove some local information and only publish the most important ones. Correspondingly, how can you reduce the number of false positive records? You can refer Bloom Filter, Hash function, digital search, or any possible design which takes small space but contains much searching information.
3. How can you organize the global index to better improve the system performance? A list of global indices with some order will be the first try. However, such organization will result a linear search with $O(n)$ to check the key. Can you provide some better ideas to reduce the searching complexity? You can refer Ordered table, Interval Tree, Segment Tree, and many other designs to organize the global index.
4. Let us consider the routing efficiency. According to the querying procedure, the client will follow the list of potential servers to find its required datum. At this moment the order of the list becomes significant to reduce the routing steps, since servers have fixed logical positions in the network with locality property. The paths between servers connecting to the same edge switch, or within the same pod, are definitely shorter than the paths for servers located in different pods. Correspondingly, an proper order correlated to the private IP address of servers might be helpful for routing efficiency.

4.2 Implementation Task

Test the efficiency of your design by simulations, where B is set to be 1280000, k is set to be 8, and the number of data items stored on each server are almost the same. We should test two types of data distributions for each server, a uniform distribution and a Zipf distribution. After constructing your indices, please generate at least $10,000 \sim 100,000$ queries to evaluate the performance of your system. You need to provide numerical results for the average search steps (both on global indices and on local indices), and the average hopping numbers among servers. Illustrations on false positive numbers (both theoretical and practical results), and the size of your indices (also both global and local indices) should also be involved.

4.3 Report Requirements

You need to provide a report for this project, with the following requirements:

1. Your report should have the title, the author names, IDs, email addresses, the page header, the page numbers, figure for your simulations, tables for discussions and comparisons, with the corresponding figure titles and table titles.
2. Your report is English only, with a clear structure, divided by sections, and may contain organizational architecture like itemizations, definitions, or theorems and proofs.
3. Please define your variables clearly. If needed, a symbol table is strongly recommended to help readers catch your design.
4. Please also include your latex source and simulation codes upon submission.

To better understand the Fat-Tree topology, you can refer [1]. Reference [2] provides more explanations for two-layer global indexing scheme. For B^+-Tree, Red-Black Tree, Interval Tree, and other advanced data structures, please refer CLRS Algorithm Book from MIT [3]. The dictionary [4] is a good reference for many sorting and searching based techniques.

References

1. Al-Fares, M., Loukissas, A., Vahdat, A.: A scalable, commodity data center network architecture. SIGCOMM (2008) 63–74
2. Wu, S., Jiang, D., Ooi, B.C., Wu, K.L.: Efficient b-tree based indexing for cloud data processing. VLDB **3**(1-2) (2010) 1207–1218
3. Cormen, T.H., Leiserson, C.E., Clifford Stein, R.L.R.: Introduction to Algorithms. MIT Press (2009)
4. Knuth, D.E.: The Art of Computer Programming, Vol 3: Sorting and Searching. Addison-Wesley Professional (1998)