上海交通大學

SHANGHAI JIAO TONG UNIVERSITY

学士学位论文

THESIS OF BACHELOR

论文题目 分布式云存储系统的索引结构与设计

学生姓名 李　凡

学生学号 5100309560

指导教师 高晓沨教授

专　　业 计算机专业

学院（系）电子信息与电气工程学院

Submitted in total fulfilment of the requirements for the degree of
Bachelor
in Computer Science

Indexing in Distributed Cloud System

FAN LI

Supervisor
Prof. XIAOFENG GAO

DEPART OF COMPUTER SCIENCE, SCHOOL OF ELECTRONIC INFORMATION AND
ELECTRONIC ENGINEERING
SHANGHAI JIAO TONG UNIVERSITY
SHANGHAI, P.R.CHINA

June. 1st, 2014

上海交通大学
毕业设计（论文）学术诚信声明

本人郑重声明：所呈交的毕业设计（论文），是本人在导师的指导下，独立进行研究工作所取得的成果。除文中已经注明引用的内容外，本论文不包含任何其他个人或集体已经发表或撰写过的作品成果。对本文的研究做出重要贡献的个人和集体，均已在文中以明确方式标明。本人完全意识到本声明的法律结果由本人承担。

作者签名：________________________

日　期：________年 ____月 ____日

上海交通大学
毕业设计（论文）版权使用授权书

本学位论文作者完全了解学校有关保留、使用学位论文的规定，同意学校保留并向国家有关部门或机构送交论文的复印件和电子版，允许论文被查阅和借阅。本人授权上海交通大学可以将本学位论文的全部或部分内容编入有关数据库进行检索，可以采用影印、缩印或扫描等复制手段保存和汇编本学位论文。

保　密 □，在 __ 年解密后适用本授权书。

本学位论文属于

不保密 □。

（请在以上方框内打“✓”）

作者签名：__________________　　　指导教师签名：__________________

日　期：______年 ___月 ___日　　　日　期：______年 ___月 ___日

分布式云存储系统的索引结构与设计

摘　　要

云存储系统，如亚马逊的 Dynamo 以及谷歌的 GFS 文件系统都向我们指出高效的查询以支持各式应用在云计算中的重要性，而这正需要良好的索引设计。因此在本文中，我们提出 RT-HCN，是一种以数据中心为基础的分布式多维数据索引结构，数据中心则是构建云系统的基础。RT-HCN 是一个两层索引方案，它集成了 HCN 的路由协议和 R 树的索引技术，并将索引分布地分发到每个服务器上。在云计算领域，数据中心网络拓扑结构相较于传统的 P2P 网络拥有多种优势，而 R 树是则一种先进的处理如今盛行的多维数据的索引结构。基于 HCN 的特点，我们设计了一个特殊的索引发布规则和查询处理算法，以保证高效地管理整个网络的数据。我们从理论上证明了 RT-HCN 拥有良好的查询效率和空间效率。并且由于每个服务器仅会维持可控量的索引文件，所以可以支持大量用户同时以较低的路由开销进行数据查询和处理。最后，我们的设计将与一个传统 P2P 网络中的类似的设计 RT-CAN [3] 进行比较，因为二者的索引建立有相似之处，具有相应可比性。有关于查询处理开销和索引维护开销的实验都证明了我们提出的方案是高效可行的，因此我们有理由相信数据中心网络中的分布式索引颇具潜力和研究价值。

关键词： 分布式索引; R 树; 数据中心网络

Indexing in Distributed Cloud System

ABSTRACT

Cloud storage system such as Amazon's Dynamo and Google's GFS poses new challenges to the community to support efficient query processing for various applications, which requires good index structure design. In this paper we propose RT-HCN, a distributed indexing scheme for multi-dimensional data query processing in data centers, the infrastructure to build cloud systems. RT-HCN is a two-layer indexing scheme, which integrates HCN-based routing protocol and the R-Tree based indexing technology, and is portionably distributed on every server. Data center network topology has several advantages over traditional P2P network as cloud systems and R-tree is an advanced index structure for multi-dimensional data, which gains great popularity recently. Based on the characteristics of HCN, we design a special index publishing rule and query processing algorithms to guarantee efficient data management for the whole network. We prove theoretically that RT-HCN is both query-efficient and space-efficient, by which each server will only maintain a tolerable number of indices while a large number of users can concurrently process queries with low routing cost. Finally, we compare our design with RT-CAN [3], a similar design for traditional P2P network. Experiments on query processing as well as index maintenance validate the efficiency of our proposed scheme and depict its potential.

KEY WORDS: Distributed Index; R-Tree; Data Center Network

Contents

Tables

Figures

Chapter 1 Introduction

In this chapter, we will first introduce some of the basic concepts about cloud storage system and explain the importance of efficient index structure in a cloud system. Readers without any preliminary knowledge will also be able to understand the work that is going to be discussed later. Moreover, we will then focus on data center network, which is a newly designed network cluster to facilitate cloud computing. Because we believe that data center network topology has many advantages over traditional topology and will be the trend of the future design. Finally, discussions about the key contribution of this work is also included in this chapter and we would compare our work with previous solutions to prove the significance of our design.

1.1 Cloud Storage System

Cloud computing means involving computing power over a network, where an application or a program may run on many connected computing nodes or servers at the same time. Cloud computing is becoming more and more popular in recent years and much work has been done since now. Specifically, cloud computing is based on a hardware computing machine or a group of hardware computing machines commonly referred as several servers connected through a communication network such as an intra-net, a local area network, a wide area network or just the well-known Internet. Any individual user who has permission to access the server can use the server's processing power to run applications, store data, or perform any other computing task. Therefore, instead of using a personal computer every-time to run the application, the individual can now run the application from anywhere in the world, as the server provides the processing power to the application and the server is also connected to a network via internet or other connection platforms to be accessed from anywhere. All this has become possible due to the increasing computer processing power available to humankind as well as the popularity of cloud storage systems. Generally speaking, cloud storage is a model of networked enterprise storage where data is stored in a virtualized pool of storage which form the cloud storage system for cloud computing. Cloud storage systems assist users to make up of as many distributed resources as possible and offer good fault tolerant performance through redundancy and distribution of data. What's more, the whole system is highly durable through the creation and typically offers eventually consistent with regard to data replicas. All of these features make users process their data more easily and gain popularity for cloud storage systems. Cloud storage system has been a significant part of cloud computing, and the improvement of its design is well-worth discussing.

We focus our discussion on cloud computing since cloud storage systems have been continuously drawing attentions from both academia and industry in recently years. From classical systems for general data services, such as Google's GFS [4], Amazon's Dynamo [5], Facebook's Cassandra [6], to newly designed systems with specialities, such as Haystack [7], Megastore [8], Spanner [9], various distributed storage systems were constructed to satisfy the increasing demand of online data-intensive applications that require massive scalability, efficient manageability, reliable availability, and low latency in storage layer. Correspondingly, many works are proposed for designing new indexing scheme and data management system to support large-scale data analytical jobs and high concurrent OLTP queries [3, 10–12]. The key technologies of cloud storage include many facets from servers, networks, clients, and related control measures, that is, availability, reliability, virtualization, feedback, credits' security and so on. Cloud storage system should support automatic management, distributed collaboration, data integration, SLA matching, QoS, certification, access control, authority assignment, audit, etc. Our work focuses on efficient data retrieving in cloud storage system, which is a principal performance of cloud computing.

1.2 Indexing Strategies

The efficiency of index construction is a key design that affects the system performance, since it plays a significant role in query processing and data retrieving. In a cloud DB system, data sets are partitioned among distributed servers, while users may hop among multiple servers to process their query requests. In general, data sets are almost randomly distributed in the whole system. Because if data is allocated according to its range or some specific distribution, it is quite sure that some server may become the bottleneck of performance since some hot data is much more often required than others. This design, then, makes it essential to construct efficient indexes over cloud storage system to provide quick searching process and data retrieval. A simple but naive way is to forward query messages over the network, this is one of the most traditional designs in history, when the P2P network was first introduced to form cloud systems. It is easy to see that this strategy involves huge amount of messages in the network and is not time efficient for query processing or data retrieving. Later, an upgraded method is to invite one server (or maybe a few servers) as a central processing unit to maintain the global information about data allocation in the system. All queries, in this design, are forwarded to this central processing unit in the first step, and then information retrieval is done in some destine server in the system according to the response of central processing unit. Absolutely, this strategy improves query processing or data retrieving efficiency a lot, however, its defect is also sharp. Since all the queries are processed by the central unit in the first step, servers in central processing unit then risk being bottleneck of performance.

Recently, to provide an efficient indexing scheme in cloud storage system, a common feature different from the above literature is to split indices into two categories: global index and local index, and then portioned the global indices to each server according to an overlay network architecture. Following the direction of indices, users route queries among servers based on the underlying routing protocols of the network connecting servers together. To distribute the global index, servers are organized into an overlay network. The global index is built on top of the local indexes. To search the local data efficiently, an multidimensional index should be built for the local database. In the global index, instead of indexing every tuple, only local index nodes are published into the global index. Consequently, the global index in the system plays the role of an overview index, composed of local index nodes from different servers. Indexing nodes reduces the size of the global index and hence lowers maintenance cost. Once we have located an published index node in the global index, we can continue search in the local index, starting from the node registered in the global index. However, All these designs are constructed on P2P networks [12–14], while nowadays cloud systems are usually built on an infrastructure called *data center*, which consists of large number of servers interconnected by a specific *Data Center Network* (DCN) [15, 17–23]. For instance, Cisco applies Fat-Tree topology as its DCN architecture for provably efficient communication [15]. Different from P2P network, DCN is more structured with low equipment cost, high network capacity, and support of incremental expansion. It is natural that such infrastructures bring new challenges for researchers to design efficient indexing scheme to support query processing for various applications.

1.3 Contributions

In this paper, we propose our RT-HCN, a distributed indexing scheme for multi-dimensional query processing in *Hierarchical Irregular Compound Networks* (HC-N) [2], which is the latest designed DCN structure using dual-port servers. HCN has many attractive features including low diameter, high bisection width, large number of node-disjoint paths for pairwise traffic, and supports low overhead and robust routing mechanisms. Additionally, in many online data-intensive services users tend to query data with more than one keys, e.g., in Youtube video system users may want to find videos via both video ids and size ranges. Therefore, designing an indexing scheme for multi-dimensional query processing in HCN is useful and meaningful for real-world cloud applications, which has both theoretical and practical significance in this area. And to search the data efficiently, the R-tree [16] based multi-dimensional index is used in our system. An R-tree is a height-balanced tree similar to a B-tree with index records in its leaf nodes containing pointers to data objects. Nodes correspond to disk pages if the index is disk-resident, and the structure is designed so that a spatial search requires visiting only a small number of nodes. The index is completely dynamic; inserts and deletes can be intermixed with searches and no periodic reorganization is required. RT-HCN integrates HCN-based routing protocol and the R-Tree based indexing technology. Similar as previous works, RT-HCN is a two-layer indexing scheme with a global index layer and a local index layer. Since datasets are distributed among different servers, we can use an R-Tree like indexing structure to index local stored data for each server. Next, RT-HCN portionably distributes these local indices across servers as their global indices. To avoid single master server bottleneck, each server only maintains partial global index for its potential index range. Based on the characteristics of HCN, we design an index publishing rule to guarantee an ”onto” mapping from global index to local stored data. We also propose the corresponding query processing algorithms to achieve query efficiency and load balancing for each node in the network. Finally, we prove theoretically that RT-HCN is both query-efficient and space-efficient, by which servers will not maintain redundant indices while a large number of users can concurrently process queries with low routing cost. We compare our design with RT-CAN [3], a similar design for traditional P2P network. Experiments validate the efficiency of our proposed scheme and depict its potential implementation in data centers.

Our contribution of this paper is threefold: (1) to the best of our knowledge, we are the first one to propose a distributed multi-dimensional indexing scheme for special DCN structure to improve query efficiency and system QoS; (2) noticing and taking advantage of the topology of HCN, we present a specialized mapping technique to distribute global index proportionally among the network, resulting query-efficiency and load-balancing for the scalable cloud system; and (3)we theoretically prove the efficiency of our RT-HCN design, and compare our model numerically with RT-CAN [3], an indexing scheme for P2P network. Simulation results show that our scheme costs less index space for each node while provides faster query processing speed with higher bandwidth.

The rest of this paper is organized as follows. Chapter 2 summarizes the related work and the background knowledge in this research area. Chapter 3 introduces the overview of our system design, including coding of HCN and meta-servers. In Chapter 4 we illustrate the two-layer index construction with global index publishing rules. In Chapter 5 we depict the query processing algorithms with corresponding performance analysis theoretically. Chapter 6 discusses the maintenance and improvement of RT-HCN, some of these work remain future implementation. Chapter 7 compares our design with the latest work RT-CAN in [3] and proves the efficiency of our construction since our design is physical topology aware. Finally, Chapter 8 gives a conclusion of this paper.

Chapter 2 Related Works

2.1 Existing Cloud Storage Systems

Nowadays, there are lots of distributed storage systems which assist to manage big data for cloud applications. Among them, we have excellent commercial implementations like key-value based system Amazon's Dynamo [5], Google's Google File System [4] (GFS) and BigTable [24] which aim at dealing with large scales of data sets. Dynamo is a key-value storage system, which is a very highly available implementation. The key design goal of Dynamo is to provide users with an experience named always-on. We have to mention here that Dynamo sacrifices consistency in their system design under a certain failure scenario in order to achieve their desired level of availability. That is to say, Dynamo design makes sure users have access to their data at any time because there are many duplicates in the system, even one server is shut down, the data is also available. However, this design does not make sure the data is the most updated one and do not apply good consistent performance. Speaking of scalable distributed file system, the one we definitely think of may be the Google File System (GFS), which is aimed at dealing with serving for applications that based on huge distributed data sets. The best features about GFS is that it runs basically on inexpensive commodity hardware while providing good fault tolerance performance. Moreover, even interacting concurrently with a huge number of clients, it also delivers high aggregation performance. GFS keeps duplications of data in their system and provides good query performance. But this design is a kind of central processing design since it involves a centralized manage unit. Bigtable is another one of distributed storage system designed for processing very huge scale size data on top of hundreds and thousands of commodity computing nodes. The key idea of Bigtable's design is to offer good scalability and big data processing. Meanwhile, some open source systems such as Hadoop Distributed File System (HDFS), HBase and HyperTable also provided a good platform for research use. HDFS is a distributed file system that is designed to run on commodity hardware. It is highly fault-tolerant and is designed to be deployed on low-cost hardware. HDFS provides high throughput access to application data and is suitable for applications that have large data sets. It is actually seen as an open source design of Google File System, and that's why it is very popular in both academy and industry. HBase is written in Java and provides BigTable like capabilities on top of Hadoop. It is a solution to big data processing in Hadoop design. Hypertable is developed in C++ and is compatible with multiple distributed file systems. Cassandra [6] is one non-rational database that combines features of BigTable and Dynamo. It is a distributed storage system for managing very large amounts of structured data spread out across many commodity servers, while providing highly available service with no single point of failure. Some other systems such as Ceph [25] Sinfonia [26] are designed to provide high performance in objects retrieval. Ceph is a distributed file system that provides excellent performance, reliability, and scalability. Ceph maximizes the separation between data and metadata management by replacing allocation tables with a pseudo-random data distribution function designed for heterogeneous and dynamic clusters of unreliable object storage devices. Sinfonia is another design that keeps data from applications on a subset of computing nodes, each of the servers exporting an address space linearly. A novel mini-transaction primitive is in the core of Sinfonia to enable efficient and consistent access to the data stored in the system, while hiding the complexities that arise from dealing with concurrency problems and failures. Most of these systems organize data into data chunks or data blocks, and invite a central server to manage tasks like data partitioning as well as query processing. However, we want to propose a more scalable design and avoid the use of central server since it risks of being the bottleneck.

2.2 Two-level Index Strategy

Different from most of existing designs, our design is to build a second level overview index and our work follows the framework proposed in [10]. It offers an idea to build a two level index in cloud system for data retrieval on top of a physical layer. Moreover, an efficient and extensible framework for index in P2P based cloud system was put forward in [12]. This strategy is to firstly build a highly efficient index locally in each server in the cloud system. Then, to process queries, local index nodes are published to some remote servers and form global index in the cloud according to some mapping function. In this way, even though data sets are still randomly allocated in the system, however, the index nodes that are responsible for data retrieving are actually dispersed in cloud according to some distribution given by the mapping function. With this design of global index, query processing is more efficient taking advantage of this information about where to find the responsible index nodes. However, other than P2P network, more specialized topology has been designed to meet the requirement of today's cloud system and that's why we want to apply the two-level index design to specific data center network and discuss its improvement. There are three design goals for data center network. First, the network infrastructure must be scalable to a large number of servers and allow for incremental expansion. Second, DCN must be fault tolerant against various types of server failures, link outages, or server-rack failures. Third, DCN must be able to provide high network capacity to better support bandwidth hungry services. These are advantages of data center network over traditional P2P network as cloud storage systems. So designing efficient cloud storage system based on data center network as physical topology is worth discussing. Additionally, as the topology of data center network is known, we can guaranty the processing time by calculating out the physical hops needed for a given query. While in P2P network only logical hops of overlay network can be estimated. When we are designing the mapping function to publish local index nodes to cloud system, we can also take the physical topology into account to facilitate the system performance. These features make data center network more propitious to serve for cloud storage systems.

2.3 R-tree Based Multi-dimensional Searching

Nowadays, multi-dimensional data is becoming more and more popular. Also known as spatial data, multi-dimensional objects often cover areas in multi-dimensional spaces and are not well represented by point locations. For example, map objects like counties, census tracts occupy regions of non-zero size in two dimensions. Searching for all objects in a given area is a common operation on spatial data sets. One of the most simple and popular example is to find all of the counties that have a land space within some mile of a particular point on the map. These kind of spatial searches occur very frequently in cloud storage systems and the upper level applications, this is why we believe it is of great significance to retrieve items based on their spatial features with high efficiency. The index structure chosen in our design is R-tree [16], which is a height-balanced tree very similar to a B-tree. It is a well-known multi-dimensional index structure with high efficiency. In this section, we'll give some brief introduction to R-tree, so that readers can have a general idea about R-tree based index format.

All the index records of an R-tree are stored in its leaf nodes, which contains pointers that redirect to data objects. Some nodes are also correspond to pages in disk if and only if the index is also disk-resident while construction. This structure is designed in order to reduce the number of index nodes that need to be searched while processing a spatial query. Index of R-tree is totally dynamic, this means inserting and deleting operation can be inter-mixed with queries and do not require extra periodic organizations. Tuples which represent spatial items are stored in spatial databases in the format of collections. Each tuple may be assigned with an unique identifier that is used in data retrieving process. The index record entries that are maintained by all the leaf nodes in an R-tree are denoted as $(I, tupte-identifier)$, where $tuple-identifier$ refers to a tuple of object in the database and I is an n-dimensional rectangle which stands for the bounding box of the spatial object that is indeed indexed as $I = (I_0, I_1, \cdots, I_{n-1})$. n is the number of dimensions here and I_i is a closed bounded interval $[a, b]$, which describes the extent value of the item along the dimension i. Noticing that I_i may alternatively have one or both of its endpoints equal to infinity, which indicates that the item extends outward indefinitely in dimension i. Similarly, the entries of none-leaf nodes are in the format of $(I, child-pointer)$ where $child-pointer$ is the pointer address of a lower level node in the R-tree and I is a bigger rectangle that covers all rectangles of lower node's entries. There are also two parameters M and m that $m \leq \frac{M}{2}$ stands, which specify the maximum number and minimum number of entries that can fit in a single node of R-tree. In general, R-tree index structure satisfies the following properties:

- For each leaf node, it contains at least m and at most M index records unless the leaf node itself is the tree root.
- Every index record which has the format of $(I, tuple-identifier)$ in a leaf node,

I must be the smallest rectangle that fully covers the n-dimensional data object spatially. Here, the data object is the one represented by the $tuple - identifier$ given.

- Similarly, for each non-leaf node, it also contains at least m and at most M index records unless the leaf node itself is the tree root.
- Similarly, every index record which has the format of $(I, child - pointer)$ in a non-leaf node, I must be the smallest rectangle that fully covers rectangles of its child nodes spatially. Here, the child nodes are the ones represented by the $child - pointer$ given.
- For the root node, it must contains at least two children unless the root node is a leaf node itself.
- Finally, all the leaves must appear on the same level for balancing.

Figure 2.1 gives a general idea about what a R-tree index looks like. It is easy to understand that how a R-tree index can be used to support efficient spatial queries, since all of its index nodes cover a multi-dimensional range. To process query searching, the algorithm descends the index nodes in R-tree from the root in a way similar to searching process of a B-tree. However, in R-tree searching, it is common that more than one subtree need to be visited under a single node, that is to say, guaranteeing good worst-case performance in R-tree searching is not quite possible. Nevertheless, it is great to notice that most kinds of data maintained in a form that may allow the search algorithm to eliminate several non-relevant regions. This process may take place in update algorithm and thus only some data near the search region is visited without much extra cost.

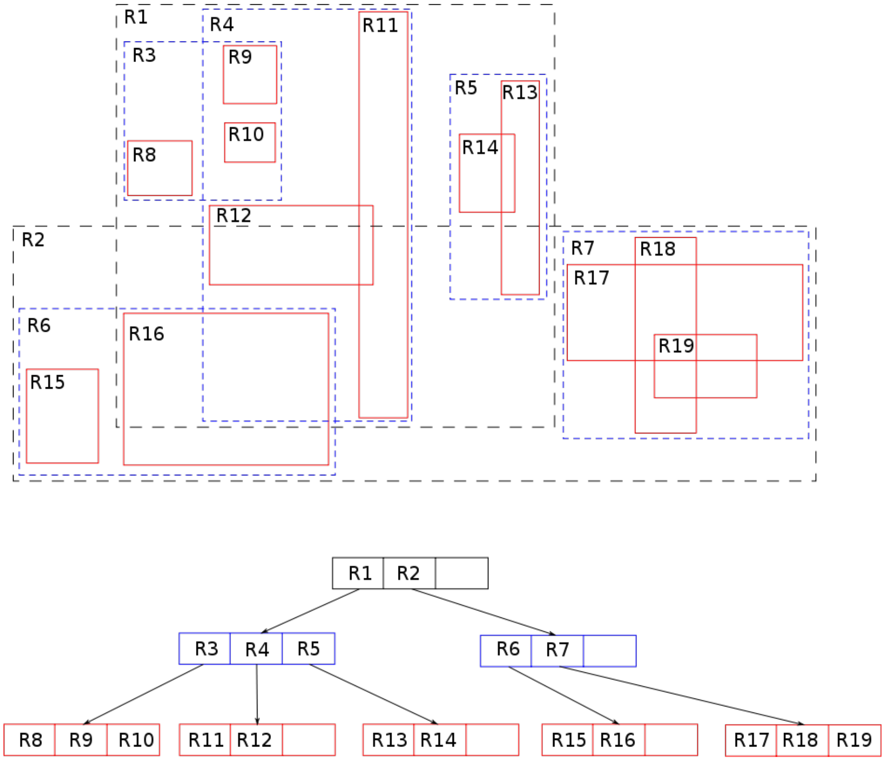


Figure 2.1 Illustration of R-tree Index

2.4 Brief Introduction To DCN

Data center network (DCN) is the network infrastructure inside a data center, which connects a large number of servers via high-speed links and switches. Compared to traditional cloud system which is usually based on P2P network, special and carefully designed DCN topologies fulfill the requirements with low-cost, high scalability, low configuration overhead, robustness and energy-saving. DCN structures can be roughly divided into two categories, one is switch centric such as VL2 [17] and Fat-Tree [15]. VL2 is a practical network architecture that scales to support huge data centers with uniform high capacity between servers, performance isolation between services, and Ethernet layer-semantics. Fat-Tree is a new class of universal routing networks, which might be used to interconnect the processors of a general-purpose parallel supercomputer. The other is server centric like BCube [19], DCell [18], FiConn [20, 21], MDCube [22] and uFix [23]. BCube is a new network architecture specially designed for shipping-container based, modular data centers, where servers with multiple network ports connect to multiple layers of commodity off-the-shelf mini-switches. DCell is a novel network structure that has many desirable features for data center networking. DCell is a recursively defined structure. This means a high-level DCell topology is actually constructed from many low-level DCell topologies and DCells at the same level are fully connected with one another. FiConn is a topology built on top of servers with only two-ports and commodity switches of very low cost. It is a scalable solution that is also recursively designed, which means a upper-level FiConn is also constructed by several of the lower level FiConns. MDCube is a high performance interconnection structure to scale BCube-based containers to mega-data centers. MDCube uses the high-speed uplink interfaces of the commodity switches in BCube containers to build the inter-container structure, reducing the cabling complexity greatly. The last design of uFix is a scalable, flexible, and modularized network architecture to interconnect heterogeneous data center containers. Server centric designs usually have more advantages than the former design since server centric designs put the interconnection intelligence on servers rather than switches. The main reason is that servers are more programmable than switches and provide larger throughput and better fault-tolerant ability. HCN [2], the topology chosen in our system falls into the server centric topology. It is a well-designed network for data center and offers a high degree of regularity, scalability, and symmetry. Different from traditional P2P network, we are quite aware of the physical topology when we are discussing DCN and that is why we can improve the mapping technique for distributing global index to fix a given network.

Chapter 3 System Overview

We want to invite the two-level index system from P2P network to data center network. Data can be randomly located in different servers and this design fully maintains the scalability of the network and avoids any server being the bottleneck of performance. As is known to all, the data in a storage system may not be completely randomly distributed, if the data is allocated according to its range, it is sure that some server may be quite busy while others are idle. In a two-level indexing strategy, each server, besides its own local index, each server also keeps part of the global index so that different servers will keep the full knowledge of the data in different range. To process a query, we first find out the desired range and then forward the query to the responsible servers. After looking up the global index, the responsible server forwards the query to the servers where the data located and then data is retrieved. Since queries various in range, queries will be handled by different servers thus lead to a good performance of load balance.

In this chapter, we'll first introduce the topology of HCN which is the DCN mainly concerned in this paper, and gives some basic features of it. And then explain some preliminary definitions for further illustration of index construction strategy.

3.1 Hierarchical Irregular Compound Network

HCN (Hierarchical irregular Compound Network) is a well-designed network for data center and offers a high degree of regularity, scalability, and symmetry. A level-h HCN with n servers in every single unit is denoted as HCN(n, h). HCN is a recursively defined structure. This means for any high-level HCN(n, h), it actually employs a lower level HCN$(n, h - 1)$ as its construction unit cluster. The means to connect these unit cluster in HCN is in a way of a complete graph. That is, HCN$(n, 0)$ is then the smallest module or known as the basic construction unit. It is consists of an n-port mini-switch and n dual-port servers. For each server in HCN, the first one port is connected to the mini-switch in the single unit while the other one is employed by the HCN as interconnection with other HCNs. In this way, different smallest modules of HCN can finally constitute a larger network. In general, HCN(n, i) for $i \geq 0$ is formed by n HCN$(n, i - 1)$s and has n available servers each in one HCN$(n, i - 1)$ for further expansion. Figure 3.1 illustrates an example of HCN with $n = 4$ and $h = 2$, which consists of 64 servers. Each server is labeled according to a coding process, which will be introduced in Sec. 3.2.

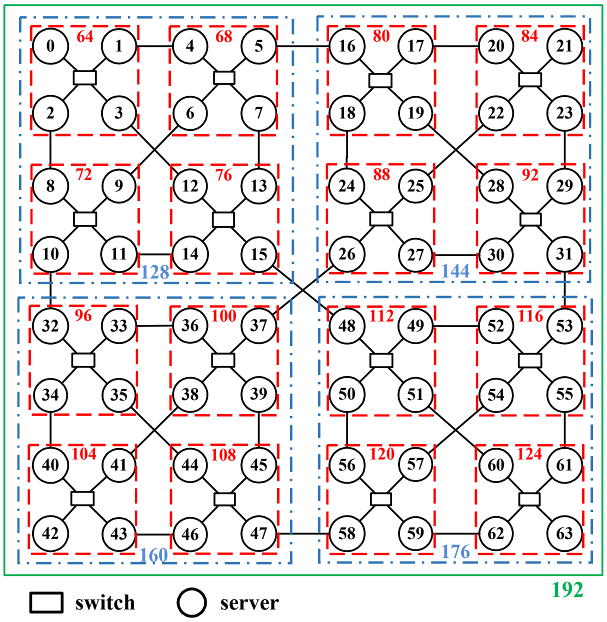


Figure 3.1 HCN(4,2) with Coding of Meta-Server

It is easy to see that there is always multiple routes between any two servers in

HCN and this is called multi-path routing which provides good features like high bandwidth, good balancing and error tolerance. And this is the main aspect we want to concern during index construction and also becomes a main reason why special designed index scheme for specific network is well worth being discussed.

For clarity, we summarize the symbols with their meanings in Table 3.1. Some of them will be described in the following sections. We use n to be the coding of both servers S_n and meta-servers M_n in the system. The representatives of meta-server are denoted as R_n^i and the local index nodes to be published are denoted as N_n^i.

Table 3.1 Symbol Description

Sym	**Description**	**Sym**	**Description**
n	Code for server and meta-server	h	The highest level of HCN
S_n	The server with code n	**B**	Data boundary
M_n	The meta-server with code n	$\mathbf{B}_n$	Potential index range for M_n
$\mathbf{R}_n$	Representatives for M_n	R_n^i	The i^{th} representative for M_n
$\mathbf{N}_n$	S_n's node set for publishing.	N_n^i	The i^{th} publishing node for S_n

3.2 Meta-server And Coding Strategy

Given an HCN$(4, h)$, there are 4^{h+1} servers in total and are coded by n ranging from 0 to $4^{h+1}-1$. Thus we use S_n to denote the n^{th} server in the HCN. We first explain the coding strategy of n. In [2], the connection between servers in HCN are illustrated as: Each server in HCN(n, h) is assigned a label $x_h \cdots x_1 x_0$, where $1 \leq x_i \leq n$ for $0 \leq i \leq h$. Two servers $x_h \cdots x_1 x_0$ and $x_h \cdots x_{j+1} x_{j-1} x_j^j$ are connected only if $x_j \neq x_{j-1}, x_{j-1} = x_{j-2} = \cdots = x_1 = x_0$ for some $1 \leq j \leq h$, where $1 \leq x_0 \leq n$ and x_j^j represents j consecutive x_js. In addition, n servers are reserved for further expansion only if $x_h = x_{h-1} = \cdots = x_0$ for any $1 \leq x_0 \leq n$. Taking advantage of the label $x_h \cdots x_1 x_0$ provided by the author of HCN, we explain the coding strategy of our design. We regard each label explain $x_h \cdots x_1 x_0$ of a server as a $h+1$ digit n-dimensional number, by transformation, we convert the number into decimals and let that decimal number be the coding n or that server.

Moreover, since HCN itself is a recursively defined structure, there are also 4^{h-l} different HCN$(4,l)$s $(0 \leq l \leq h)$ in the same HCN$(4,h)$. We consider each S_n and HCN$(4,l)$ $(0 \leq l \leq h)$ as a meta-server in our system.

Definition 3.1. Meta-server is a single server or an HCN$(4,l)$ $(0 \leq l \leq h)$ considered entirety.

Meta-servers together constitute the overlay network and facilitate global queries, this is going to be explained in detail later. Meta-servers are also coded by n and denoted as M_n. The coding strategy of M_n is given by:

$$M_n = \begin{cases} S_n, & 0 \leq n < 4^{h+1} \\ \text{HCN}(4, q-1) \text{ consists of } S_r, S_{r+1}, \cdots, S_{r+4^q-1}, & n \geq 4^{h+1} \end{cases}, \quad (3.1)$$

where q and r from the above equation represent the quotient and the reminder when the given n is divided by 4^{h+1}. From the function we know that M_n is equivalent to S_n when $n < 4^{h+1}$, and when $n \geq 4^{h+1}$ M_n represents a specific HCN$(4,l)$ $(0 \leq l \leq h)$. For example, Fig. 3.1 shows that in an HCN$(4,2)$, 64 meta-servers formed by single server are coded with the number ranging from 0 to 63, 16 meta-servers formed by HCN$(4,0)$ are coded with 64, 68, $\cdots$, 124 (shown as red squares), 4 bigger meta-servers formed by HCN$(4,1)$ are coded with number 128, 144, 160, 176 (shown as blue squares), while the biggest green square formed by the whole HCN$(4,2)$ is coded with 192.

3.3 Representatives In Meta-server

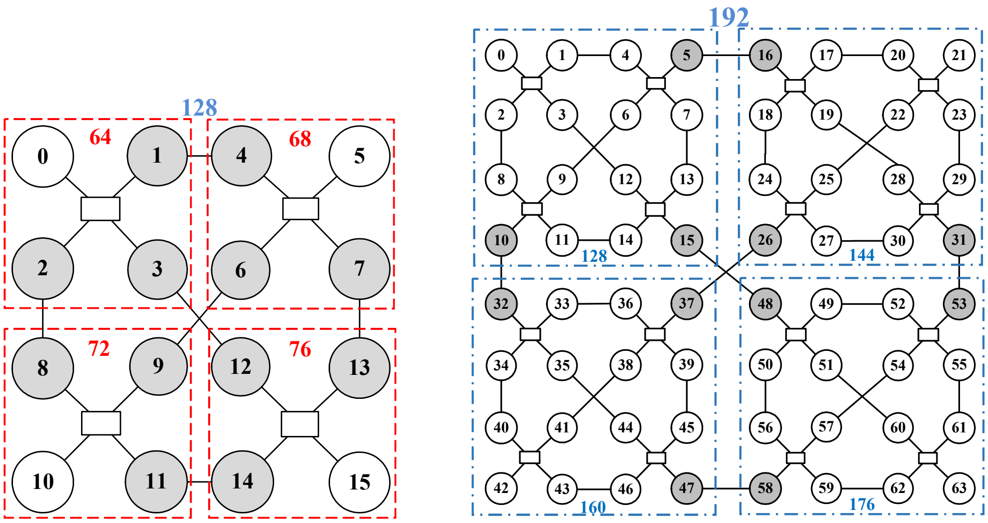
As is already mentioned, meta-servers form a higher-level overlay network and assist query processing in the network. However, as meta-servers are merely an abstract concept, we need to pick up several physical servers to be in charge of queries that are sent to corresponding meta-server from the overlay network.

Definition 3.2. Representatives of a given meta-server are several carefully picked servers that physically deals with queries that should be processed by the meta-server seen from the overlay network.

We now provide our strategy for choosing representatives of a given meta-server M_n and there are two cases:

1. If $n < 4^{h+1}$, we chose S_n as the representative of M_n since $M_n = S_n$.
2. If $n \geq 4^{h+1}$, M_n, now, is actually an HCN$(4, l)$ $(0 \leq l \leq h)$, and we provide a special bit manipulation to find out its representatives. First we calculate the quaternary form of n, denoted as $q_0q_1 \cdots q_m$. Here $m > h$ stands since $n \geq 4^{h+1}$. Secondly, we pick up the first $m - h$ bits as shown in Eqn. (3.2) and calculate the decimal number b. Then we replace each of the last b bits with q_*, where $q_* \neq q_{m-b}$ and will get several newly formed number. Finally we calculate the decimal form of the last $h + 1$ bits of the new numbers and servers coded with those numbers are the representatives for this M_n.

$$\overbrace{\underset{=b\text{ (decimal)}}{\underline{q_0q_1\cdots q_{m-h-1}}}}^{m-h\text{ bits}} \;\Bigg|\; \overbrace{q_{m-h}\cdots q_{m-b}\underset{\text{replace part}}{\underline{q_{m-b+1}\cdots q_m}}}^{h+1\text{ bits}} \tag{3.2}$$



(a). $l = 0$ (b). $l = 1$

Figure 3.2 The Representatives of Meta-servers in HCN(4,2)

For example, the grey nodes in Fig. 3.2 (a), (b) illustrates the representatives for corresponding meta-server HCN$(4, l)$ when l equals to 0 and 1. It is obviously that these representatives actually takes the advantages of HCN topology and offer good connectivity which will facilitate query process.

We denote the representatives of M_n as set $\mathbf{R}_n$ and $\mathbf{R}_n$ has different number of entities in different situation. In case 1, $\mathbf{R}_n = \{R_n^1\}$ while in case 2, $\mathbf{R}_n = \{R_n^1, R_n^2, R_n^3\}$. Here R_n^i stands for the server which is the i^{th} representative of meta-server M_n, and R_n^i is called an l^{th}-level representative if and only if M_n is a meta-server formed by an HCN$(4, l)$ $(0 \leq l \leq h)$.

Now we give some explanation on choosing representatives. It is obviously to choose S_n as representatives for M_n when $n < 4^{h+1}$ because they are exactly the same. So we focus our discussion on case 2 here. According to the topology of HCN, it is not hard to find that all of the representatives we choose for M_n $(n \geq 4^{h+1})$ are servers that are more closely connected to other HCN$(4, l)$s in the same level. Thus, our strategy cut the cost of queries forwarding and since there are three representatives for a single M_n, it also offers flexibility to choose the closet representative or the dullest one. And this strategy also fit quite well with the multi-path routing of HCN. What's more, the following theorem shows that our strategy also offers good scalability and balancing property.

Theorem 3.1. *Each server S_n will be the representative for exactly two different meta-servers.*

Proof: Firstly, it is obviously that each server should be chosen as a representative at least once according to the case when $n < 4^{h+1}$. Secondly, each meta-server M_n ($n \geq 4^{h+1}$) has exactly three representatives according to the bit replacement. There are four different bits in quaternary number and our strategy replace the bits with a same bit different with q_{m-b}, so there are three cases. The biggest $M_{(h+1)*4^{h+1}}$ has four representatives since the bit q_{m-b} does not exist in this case. Thus it is easy to calculate that there are $2 \cdot 4^{h+1}$ representatives in total. Thirdly, no server can be chosen for three times. Since the quaternary form of the coding for a given server is fixed, there exist only one way for it to be split according to the form given by Eqn. (3.2). Combining the above discussion, we can easily draw a conclusion to our proof according to the Pigeonhole Principle. □

We mention here that, if we choose the outermost servers as the representatives for meta-server HCN$(4, h)$, according to the above strategy, we can fully maintain the scalability and balancing of HCN. However, it is essential to notice that HCN is actually not totally symmetric, in which inner servers are indeed assigned with higher level of connectivity. In order to avoid the waste of cost in index publishing and query processing, we may also alternatively choose the $h - 1^{th}$ level representatives to serve as representatives for HCN$(4, h)$ in real implementation.

Chapter 4 Index Construction

In this section, we first introduce the vital component of the higher level overlay network, which is essential for constructing the global index. Then we introduce our two-layer index construction in detail with index publishing rules.

4.1 Potential Indexing Range

Before we begin our discussion of index construction, we illustrate another essential concept about our meta-servers. In order to construct global index for multi-dimensional data in our overlay network, we have assigned each meta-server a potential indexing range. Thus, for any given queries, we can figure out which meta-server is responsible for it and then the query is processed by the representatives of that meta-server.

The multi-dimensional data forms a data boundary denoted as **B**, which is a k-dimensional rectangle as the bounding box of the spatial data objects:

$$\mathbf{B} = (B_0, B_1, B_2, \cdots, B_k) \ . \tag{4.1}$$

Here k is the number of dimensions and each B_i is a closed bounded interval $[l_i, u_i]$ describing the extent of the data along dimension i. To better illustrate our idea, the following discussion focuses on an example situation when $k = 2$.

Definition 4.1. Potential indexing range is an abstract attribute assigned to meta-server and indicates which meta-server (indeed the subordinate representatives) is (are) responsible for processing a coming query.

We now discuss our strategy for assigning potential indexing range to each meta-server. We suppose our data is bounded by $\mathbf{B} = (B_0, B_1)$, where B_0 is $[l_0, u_0]$ and B_1 is $[l_1, u_1]$. We calculate a quaternary number Q_n for each meta-server M_n and use it to help figure out the potential index range $\mathbf{B}_n$ for M_n. Suppose $q_0q_1\cdots q_m$ is the corresponding quaternary form for n, then Q_n is calculated according to the following two cases:

1. If $m \leq h$, we add $m - h$ consecutive 0s to the front of $q_0q_1\cdots q_m$ and construct an $h+1$ bit quaternary number $Q_n = 00\cdots0q_0q_1\cdots q_m$.
2. If $m > h$, $q_0q_1\cdots q_m$ can be split as the form explained in Eqn. (3.2). In this situation, we pick out the last $h+1$ bits and delete the replace part to get $Q_n = q_{m-h}q_{m-h+1}\cdots q_{m-b}$.

Now, we use the following iteratively defined function to calculate $\mathbf{B}_n$.

$$\mathbf{B}_n = pir(\mathbf{B}, Q_n) = \begin{cases} pir(([l_0, \frac{l_0+u_0}{2}], [l_1, \frac{l_1+u_1}{2}]), Q'_n), & q_0 = 0 \\ pir(([\frac{l_0+u_0}{2}, u_0], [l_1, \frac{l_1+u_1}{2}]), Q'_n), & q_0 = 1 \\ pir(([l_0, \frac{l_0+u_0}{2}], [\frac{l_1+u_1}{2}, u_1]), Q'_n), & q_0 = 2 \\ pir(([\frac{l_0+u_0}{2}, u_0], [\frac{l_1+u_1}{2}, u_1]), Q'_n), & q_0 = 3 \\ ([l_0, u_0], [l_1, u_1]) & Q_n = \varnothing \end{cases}, \qquad (4.2)$$

where Q_n is a quaternary number denoted as $q_0q_1\cdots q_i$ and Q'_n is a newly constructed quaternary number $q_1q_2\cdots q_i$.

For example in Fig. 4.1, two axes B_0 and B_1 denote the range of data in two dimensional space (w.l.o.g., we assume it generates a square area, rather than a rectangle). Then the potential indexing range for M_{80} (denoted as red square) should be

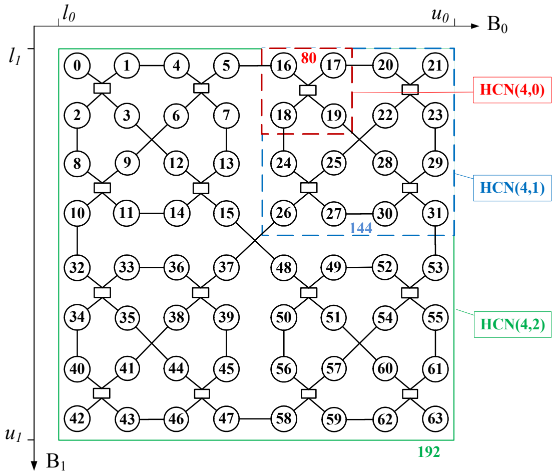


Figure 4.1 Potential Index Range

$([\frac{l_0+u_0}{2}, \frac{l_0+3u_0}{4}]$,$[l_1, \frac{3l_1+u_1}{4}])$; the *pir* for M_{144} (denoted as blue square) is($[\frac{l_0+u_0}{2}, u_0]$, $[l_1, \frac{l_1+u_1}{2}]$), while the *pir* for M_{192} is ($[l_0, u_0]$, $[l_1, u_1]$).

4.2 Publishing Rule

We choose a node set from each local R-tree and publish the index nodes to the system. An intuitive way is to choose the last but one level index nodes from a given R-tree to be published since these nodes are usually not frequently updated and do not introduce too many false positives either. However, in the later section, we will introduce a cost model to facilitate node selecting procedure. The format of the published R-tree nodes is (n, mbr), where n indicates the origin server for storing the data and mbr is the minimal bounding range of the published R-tree node. After receiving the published nodes, representatives buffers the index in memory. In this way, the global index composes of several R-tree nodes from the local indexes and is distributed over the data center.

The global index can be considered as a secondary index on top of the local R-trees. This design splits the processing of a query into two phases. In the first phase, the query is done by the response server S_t of the meta-server that is responsible for the query and the representatives look up the global index, search the buffered R-tree nodes and return the entries that satisfy the query. In the second phase, based on the received index entries, the query is forwarded to the corresponding physical server S_t and retrieve the results via the local R-tree.

In previous work, the index node is published to the server whose potential index range overlaps with the range of the index node itself. Even though some optimization is taken, this strategy cannot overcome the shortage of bringing many duplications of published index nodes. To eliminate duplications, our strategy is to find the meta-server with the minimum potential index range that covers the range of the index nodes and then publish the index node to the representatives of that meta-server. Alg. 1 describes the indexing publishing policy for each server in HCN.

Algorithm 1: Index Publishing (For S_n)

```
$\mathbf{N}_n$ =getSelectedRTreeNode($S_n$)
for each $N_n^i \in \mathbf{N}_n$ do
    Find the least $n'$ s.t. $\mathbf{B}_{n'}$ fully covers $N_n^i.mbr$
    Get the representatives $\mathbf{R}_{n'}$ for $M_{n'}$
    for each $S_k \in \mathbf{R}_{n'}$ do
        $S_k$ inserts $(n, N_n^i.mbr)$ into its global index set
    end
end
```

We can see that even some meta-servers are assigned with a relatively larger potential index range, however, since we only choose the meta-server with the minimum potential index range (the smallest n') that covers the range of a published node, meta-servers with larger potential index range are assigned with relatively larger index nodes, but not definitely more index nodes. This means in our system, nearly every server are responsible for approximately the same amount of queries, since we suppose queries have random ranges and our global index is distributed with a high load balancing property.

The average routing cost for one-to-one traffic in HCN is $O(2^{h+1})$ [2], and the cost for information transmission between representatives of the same meta-server is $o(2^{h+1})$, thus the cost to publish an index node should be $O(2^{h+1})$ on average, and it equals to $O(\sqrt{N})$ when $n = 4$, where N is the total number of servers in the given HCN. For more general situation, the cost is given by $N^{-\log_2 n}$ and can be reduced as n get larger. This cost can be further reduced as long as the parameter n for HCN increase. And also we want to mention here that the cost shown above are exactly physical hops between servers while previous works claims that it takes only $O(\log N)$ to publish index in P2P network but they are only discussing hops in the overlay network. And since the physical connection of P2P network is unclear, the physical hops can be hard to exam and constrain. Another improved feature for index publishing in our system is that under this design we can make sure that each index node is published to no more than tree servers in the system. However, the strategy used in [3] publish the nodes to other servers as long as the range of the index node overlaps with the potential index range of the given servers. Then, the amount of index nodes published in the system is hard to control.

4.3 Index Node Selection

As is discussed above, each data server S_n has built a R-tree index for its local data to facilitate multi-dimensional search. Then, S_n adaptively selects a set of index nodes $\mathbf{N}_n = \{N_n^1, N_n^2, \cdots, N_n^{d_n}\}$ from its local R-tree and publishes each N_n^i to the representatives of a specific meta-server whose potential indexing range just covers the minimal bounding range of N_n^i. We discuss how we choose the index nodes from a given local R-tree in this subsection.

And in [3], authors provided a way to choose index nodes set with two properties: ***Completeness*** and ***Uniqueness***. This means that we can choose a set of index nodes that fully covers all local data without any redundancy by maintaining these two features. The former property makes sure the correctness of query processing and the latter one removes redundancy during index node maintenance. Although the definition of this two properties is not included in our work, we believe readers can easily interpret from the literal that this two properties together require the node set that is chosen to be published covers the local R-tree precisely. It indicates that exactly one of a given node or a set of all its children will be chosen. Thus we can build a function to calculate the cost for index publishing in two cases and decide which nodes to be published based on historical queries.

Then, this task is actually about giving a cost model to decide which node set is more cost-effective. Index maintenance cost and query processing cost are two types of cost that affects system performance mostly. Index maintenance cost occurs when an index node is updated, which is often triggered by the local R-tree's node splitting or merging. Query processing cost means the cost of searching for false positives, which is induced by the feature of R-tree. The essential cost of retrieving the required data is not under discussion since it cannot be further reduced. Then, given a R-tree index node N^i, the cost function is given by:

$$C(N^i) = \alpha \cdot C_M(N^i) + C_Q(N^i) \tag{4.3}$$

where $C_M(N^i)$ is the maintenance cost, $C_Q(N^i)$ is the query processing cost and α is a balancing coefficient which represents the updating frequency of the node. In order to estimate $C_M(N^i)$ and $C_Q(N^i)$, we need to figure out both the probability and the cost for a single update or query. The probability is well estimated by features of index nodes, however, the cost is simply evaluated by the maximum routing cost of the whole network. Taking advantage of the known physical network connection, we can provide a more accurate way of calculating the cost.

Given an index node N^i, the destination server for its publishing is clear by the publishing rule, and since the topology of the whole network is exactly known, the cost of querying or maintaining process that involves N^i is then indicated by $r(N^i)$, which is the routing cost between the server that builds index node N^i and the destine server for N^i to be published. Node splitting and merging are the two primary index maintenance operations. When node N^i split into two nodes N^j, N^k, it costs $r(N^i)$ to delete N^i, and it seems that publishing nodes N^j and N^k takes another $r(N^j) + r(N^k)$. However, according to the rule of R-tree index node splitting and our index node publishing rule, routing from the destine server to publish N^i and the destine server to publish N^j or N^k is actually $o(r(N^i))$. This procedure is similar for node merging, the only difference is that the cost of republishing a merged index node is $O(r(N^i))$. So, $C_M(N^i)$ ca be computed as follows:

$$C_M(N^i) = O(r(N^i)) \cdot P_{split}(N^i) + 2 \cdot O(r(N^i)) \cdot P_{merge}(N^i) \tag{4.4}$$

where $P_{split}(N^i)$ and $P_{merge}(N^i)$ are the probabilities of node N^i's splitting and merging process.

The essential cost of a querying process cannot be reduced since the our propose is to retrieve the data. We focus our attention on false positives when we are discussing cost of querying. Intuitively, $P_{fp}(N^i)$, the probability of false positives can be simply defined as:

$$P_{fp}(N^i) = \frac{N^i.range \cap data.range}{N^i.range} \tag{4.5}$$

where $data.range$ represents the real data range in the system. And then $C_Q(N^i)$ is given by:

$$C_Q(N^i) = O(r(N^i)) \cdot P_{fp}(N^i) \tag{4.6}$$

And notice that we are considering a node set when we apply our cost model, so the total cost of index node is estimated as:

$$C(\mathbf{N}) = \Sigma_{N \in \mathbf{N}}(C(N)) \tag{4.7}$$

4.4 Index Maintenance

Since cloud computing aims at offering data processing service for upper level applications, updating is a common operation in the system. When data is updated in the system, the index for searching the updated data should also be update, otherwise, queries can not be efficiently processed. In our system, R-tree is used to index multi-dimensional data, and we distributed index nodes separately in different servers, so we focus our discussion on updating process of R-tree in distributed design.

We first explain the updating process generally. When data in the cloud storage system need to be updated, the local R-tree is updated to search the new data. When the index nodes of local R-tree is updated, we should republish the updated index nodes to guarantee the correctness of global index. For a local R-tree, if a data tuple is updated so that its covering rectangle is changed, its index record must be deleted, updated, and then re-inserted, so that it will find its way to the right place in the tree. This process is very similar in distributed situation. The published index node should be deleted, updated, and then re-published, so that it will find its way to the right server for its publishing.

From the above discussion, it is easy to find out the index nodes that are published as global index are actually mutable. To maintain good query performance, the global index nodes are always stored in memory and are available at any time. However, in previous works, the index nodes are simply managed in a list in memory, which restricts its efficiency of query processing and updating. In our design, we think it is better to form a global R-tree as well to manage the published global index nodes. This process is feasible since all the published index nodes are part of some local R-tree index with the same format. Experiments about this design is not finished, by the time of this article, however, we believe this strategy will improve the performance of the whole system a lot.

Chapter 5 Query Processing

In this section we show how our global index can be applied to process queries and reduce the cost of physical hop counts in an HCN. To start with, we first give a brief introduction about query processing in cloud storage system. Basically speaking, there are three types of mostly used queries in cloud computing, that is point query, range query, and KNN query. Point query, maybe the most simply and fundamental type of query processing is to retrieve the data according to a given value point. In traditional key value based database, this process is done by simply matching the value given by the query with the keys stored in the system. However, it is a little bit different with traditional strategy when we are considering multi-dimensional data. Since multi-dimensional data is indeed data with a range, so given a value, which is a point in multi-dimensional space, the point query requires all the data whose range covers this point. Similarly, a range query for multi-dimensional data is given a value or range and the target is to retrieve all the data whose range overlaps with the given range. KNN query, which is short for k nearest neighbors, is another common query processing in cloud computing. Different from point query or range query, KNN query is one of similar queries. Since we are not sure data of what size should be searched to get the right result, a approximation algorithm or a heuristic strategy is essential for high efficient.

5.1 Point Query

A point query is denoted as $Q(value)$ where $value = (v_0, v_1)$, indicating a multi-dimensional data point. Given an indexed R-tree node N_i^n, N_i^n should be searched to retrieve possible results if and only if $N_i^n.mbr$ covers the query point (v_0, v_1) just as what we have discussed above. Such index node N_i^n, however, can be published to several different representatives in our system, since our publishing strategy aims at avoiding any server being bottleneck of performance. According to our publishing rule, however, we can generate the subset of servers that may store such index nodes. The following theorem shows that how we figure out the set of servers whose global index we have to search to get the full results.

Theorem 5.1. *The published index node that may contain results for a given value (v_0, v_1) are in representatives of $h + 2$ meta-servers. And exactly $h + 1$ servers need to be searched in total to get the full results.*

Proof: Suppose an indexed R-tree node N_i^n contains the search value (v_0, v_1), it is certain that the destination for publishing this index node must be the representative of some meta-servers whose potential index range covers point (v_0, v_1), other wise it will contradict with our publishing rule. Therefore, we can easily figure out there are $h + 2$ different meta-servers that meet the above condition. They respectively consist of a single server, an HCN$(4, 0)$, an HCN$(4, 1)$, $\cdots$, and the biggest HCN$(4, h)$. Since the meta-server formed by a single server takes itself as a representative and is also one of the representatives for the meta-server formed by HCN$(4, l)$ that meet the above condition for some l, the total number of servers to search is then $h + 1$. It is also good to notice that the $h + 1$ servers are actually representatives of different levels that is from 0^{th} level to h^{th} level, this will facilitate our explanation for query forwarding. □

Although it seems that $h+1$ is a quite big number since the total number of servers are only 4^{h+1}, we provide our strategy for forwarding the query based on a greedy algorithm and show that the cost of forwarding is really small even when there are $h+1$

servers to be searched. As we have discussed above, there are $h+1$ representatives to search and they range from level-0 representative to level-h representative. To process a query $Q(v_0, v_1)$, we first forward the query to the nearest h^{th} level representative S_{n_h} which is responsible for the given query. S_{n_h} searches its buffered global index and returns matched results to the user. Then, it forwards the query to the nearest $h-1^{th}$ level representative $S_{n_{h-1}}$, respectively. Until a 0^{th} level representative has been searched, we can finally get the full results. The head of the routing message follows the format of $\{l, value\}$, where l is the level for representative and $value$ is the querying point. Alg. 2 describes such process in detail.

Algorithm 2: Point Qery Processing

```
Input  : $\{l, value\}$
Output: N(the result set of indexed R-tree nodes)
N = ∅
for $\forall N \in S.globalIndex$ do
    if $N.mbr$ $covers$ $value$ then
        Add $N$ into N
    end
end
if $l > 1$ then
    $S_{next}$ = the nearest $l-1^{th}$ level representative for $value$
    Forward query message $\{l-1, value\}$ to $S_{next}$
end
return N
```

We can see from the above algorithm that the shortest query path for a given query is $O(\log N)$ and if we want to maintain the multi-path routing of HCN, our index publishing scheme do bring out some extra cost for specific queries. But since the cost of forwarding queries from any $l+1^{th}$ level representative to the nearest l^{th} level representative will never exceed the maximum cost of point-to-point routing in a HCN$(4, l)$ which is $O(2^{l+1})$, and noticing that under this design only with a probability of $\frac{1}{4}$ will a query pass result in redundant routing, thus, the average cost of query in our system will only be $\frac{5}{4}$ times of the cost of point-to-point routing in a HCN$(4, h)$ and then is still constrained by a logarithmic scale. What's more, with a similar analysis, it is easy to figure out that the parameter $\frac{5}{4}$ will decrease to $\frac{n+1}{n}$ when considering a HCN(n, h). And we mention again that the cost we were discussing above are all physical connections rather than hops in overlay network that are discussed in P2P context.

5.2 Range Query

A range query is denoted as $Q(range)$, where $range = ([l_0, u_0], [l_1, u_1])$ is a multi-dimensional hypercube. And if a R-tree node's boundary $N_i^n.mbr$ overlaps with $range$, it should be searched to retrieve the possible query results. Let's think about an easy case first, if the $range$ is small enough and can be fully covered by some $\mathbf{B}_n$, where n is smaller than 4^{h+1}, then $Q(range)$ actually goes in the same way we do for point query in the last section. Inspired by this special case, we can figure out that range query is an variation of point query, with only a small modification.

We can first find out the smallest HCN$(4, l)$ that fully covers the queried range, then just as what we did in point query, we first forward the query to the nearest h^{th} level representative S_{n_h}, which is responsible for the given range. Then, S_{n_h} forwards the query to the nearest $h - 1^{th}$ level representative $S_{n_{h-1}}$, respectively, until an l^{th} level representative has been searched. Then, different from what we did in point query, from the l^{th} level on, the representatives will forward the query to all the lower level representatives, whose potential index range overlaps with $range$, to guarantee the correctness and completeness of results. Since routing cost in HCN$(4, l)$ where $l < h$ is $o(\log N)$, with similar analysis for point query, it is easy to figure out that the cost for a range query is still $O(\log N)$, and the detail of the range query algorithm is very similar to point query and the details for range query is given in Alg. 3

Algorithm 3: Range Qery Processing

```
Input  : $\{l, range\}$
Output: N(the result set of indexed R-tree nodes)
N $= \varnothing$
for $\forall N \in S.globalIndex$ do
    if $N.mbr$ covers $value$ then
        Add $N$ into N
    end
end
Find out the smallest HCN$(4, k)$ that covers $range$
if $l > k$ then
    $S_{next}$ = the nearest $l - 1^{th}$ level representative for $value$
else
    $S_{next} = l - 1^{th}$ level HCN whose range overlaps with $value$
end
Forward query message $\{l - 1, value\}$ to $S_{next}$
return N
```

We are glad to notice that our algorithm for range query processing also makes a good use of the multiple routing strategy of HCN. This design provides good scalability and saves a lot bandwidth. Additionally, the cost of range query in RT-HCN is $O(\log N)$ physically, while former P2P architecture design usually needs $O(\log N)$ hops of overlay network.

5.3 KNN Query

In order to process k-nearest neighbors queries more efficiently, most of special designed algorithms involve training phase. This train process is very similar with machine learning and data mining process. A commonly used distance metric for continuous variables is Euclidean distance. For discrete variables, such as for text classification, another metric can be used, such as the overlap metric (or Hamming distance). Often, the classification accuracy of KNN can be improved significantly if the distance metric is learned with specialized algorithms such as Large Margin Nearest Neighbor or Neighbourhood components analysis. However, multi-label classification tasks are ubiquitous in real-world problems. For example, in text categorization, each document may belong to several predefined topics; in bioinformatics, one protein may have many effects on a cell when predicting its functional classes. In either case, instances in the training set are each associated with a set of labels, and the task is to output the label set for the unseen instance whose set size is not available. This situation is the same with multi-dimensional data. In some algorithm, for high-dimensional data, dimension reduction is usually performed prior to applying the KNN algorithm in order to avoid the effects of the curse of dimensionality. And in this article, since we are not designing a special processing strategy for this problem, we offer a simple but efficient way to solve KNN queries.

We can use the idea of range query to achieve KNN query processing. Since KNN query require the results of k-nearest neighbors of a given value, the format of KNN query is given by $Q(k, value)$ where k stands for the number of expected results and $value = (v_0, v_1)$, indicating a multi-dimensional data point. It is sure that these k results for a given query is in a range that covers the point (v_0, v_1) by the definition of KNN query. Inspired by this property, we can estimate a range by the given value and do range query $Q(range)$ first, and then find out the top-k related outputs as the results of KNN query $Q(k, value)$. Of course, if the number of outputs are less than k, we can estimate a new range and process a range query again. The algorithm for KNN query is given by Alg. [**?**].

The cost of KNN query, in this design, is surely bounded by the cost of range

Algorithm 4: KNN Qery Processing

```
   Input  : $\{l, k, value\}$
   Output: N(the result set of indexed R-tree nodes)
1  N $= \varnothing$
2  for $\forall N \in S.globalIndex$ do
3      if $N.mbr\ covers\ value$ then
4          Add $N$ into N
5      end
6  end
7  $range$ = Estimate($value$)
8  Find out the smallest HCN$(4, t)$ that covers $range$
9  if $l > k$ then
10     $S_{next}$ = the nearest $l - 1^{th}$ level representative for $value$
11 else
12     $S_{next} = l - 1^{th}$ level HCN whose range overlaps with $value$
13 end
14 Forward query message $\{l - 1, k, value\}$ to $S_{next}$
15 if N.size()$\geq k$ then
16     return N
17 else
18     $range$ = ReEstimate($value$)
19     goto Line 8
20 end
```

query. Although it is sure that estimation of range from value also takes time and a poor estimation may also cause multiple range query processes. However, similar with the procedure that KNN queries are processed, the estimation function is also a function of machine learning or a function with architecture intelligence. This means that the estimation process can be much more precise as the system handles more KNN queries. That is why we can believe that the cost of KNN query is also in a logarithmic scale and is cost efficiency.

Chapter 6 Further Discussion

So far, we have already introduced the system architecture and our index construction strategy. Moreover, we have already discussed that how our index strategy can facilitate query processing in distributed cloud storage systems. The main idea of this article is already illustrated. However, there are still several aspects that may be improved and in this section, we discuss some further optimization strategy that may facilitate the whole system to be more efficiency.

6.1 Threshold Strategy In Publishing

We have already discussed in Chap. 5 that to process queries by our existing design, $h + 1$ servers need to be searched in total to get the full results. Even though this number is $O(\log N)$ where N is the total number of servers in the system and we have also provided a searching chain to save routing cost, the number of servers and the cost for routing may be further reduced by a k-level threshold design. This design is an improvement of the existing publishing rule. The key idea of our k-level threshold design is to make a compromise between routing cost and index maintenance cost.

Different from our previous publishing rule, given a index node, we first find the meta-server with the minimum potential index range that covers the range of the index nodes and let the meta-server found be a HCN$(4, i)$ for some $0 \leq i \leq h$. Then, instead of directly publish the index node to the representatives of that meta-server, we have to check whether $i \leq k$ stands or not. If $i \leq k$, then the index node is published to the representatives of the meta-server we found just as what we did before without

$k-$level threshold strategy. Otherwise, which means $i > k$, the index node is then published to the representatives of all the HCN$(4, k)$s whose range overlaps with the node's range. In other words, this design can be seen as a combination of traditional publishing rule(overlap publishing design) and our former naive publishing rule.

Intuitively, this strategy is trying to achieve a tradeoff between query cost and index duplications. On one hind, any query processing can end searching with some level-k representatives rather than a level-1 representative, improving the performance of query processing by reducing routing cost. On the other hand, since we publish the index nodes whose range can not be fully covered by a meta-server HCN$(4, i)$ where $i \leq k$, there are certainly more duplications of these kind of index nodes.

However, as what have been discussed above, this k-level threshold design is a combination of two strategies and achieves a tradeoff between cost efficiency and index maintenance cost. Then, the level k, which is the threshold choosing in this design, is an essential parameter and definitely affects system performance deeply. Unfortunately, we didn't figure out a authoritative cost function to decide the value of k, but we still provide some heuristic method of choosing the value of k. When we run our simulation program, we find out that index nodes with huge range is actually not common in global index, while the last but two level index nodes of local R-tree are more often published as global index in the system. This phenomenon shows that the index node's range overlaps with many meta-servers only because of the range covers the boundary of ranges of different meta-servers rather than the range of index node is huge. Then, the duplications of index nodes caused by this k-level threshold strategy can be estimated by the side length or the perimeter of HCN$(4, k)$. On the other hand, estimating the saved routing cost of this new design is much easier. The number of servers we need to search for a given query is about h, using the original publishing rule. When a k-level threshold design is in use, the number reduced approximatively to $h - k$ since we end our searching process at level-k representatives. Thus the save of routing cost is about $\frac{k}{h} \cdot O(\log N)$. With other coefficients to indicate the weight of cost efficiency and index maintenance, we can easily figure out a best value for k.

6.2 Network Expansion

We only use HCN$(4, 2)$ and 2-dimensional data for discussion in the above sections, this does not mean these parameters should be fixed. For other values, we can carefully chose functions to map high dimensional space to uni-dimension and then build the system similarly.

When considering the implementation of a HCN(n, h), the function to map each meta-server a different potential index range should be changed since n is different. Noticing that h is a parameter that indicates the scalability of the whole system, it does not affect much the design of the mapping function because the system is a recursively designed architecture. Although in situation where $n \neq 4$, the topology of HCN(n, h) is not like a square, which means it is not easy for us to map a range to a meta-server intuitively. However, since HCN is indeed a recursively designed topology, which also stands with the case HCN(n, h), there exists something unchanged as n changes. One simple design is to keep mapping the range according to the coding of servers or meta-servers. Then the range of meta-servers is also given by a range mapping function which is quite similar with the mapping function given in previous discussion. The only thing different is that we can not draw a picture to show the range mapping intuitively. However, this does not cause much trouble. The parameter h stands for the scalability of HCN network and the range mapping process does not change even though h changes a lot. However, when the value of h changes, it is sure that the range of a single server or a meta-server also changes. Although this does not affects the system design acutely, however, some experiential parameter may be different such as the value k we talked about in the last section. This means extra experiments are required for new parameter values.

6.3 Higher Dimensional Data

For higher dimensional data, we can take advantage of the space-filling curve [28] to reduce dimension. There are famous curves like Peano's space-filling curve, which is the first space filling curve in history proposed by Peano. Hilbert's space-filling curve is one of the most widely used space filling curve nowadays, and in [29] authors even offers a strategy to improve R-tree performance with Hilbert curve. Taking advantage of these space filling curves, it is possible to generate one-to-one mappings between spaces of different dimensions, that is, we can apply our strategy similarly to higher-dimension data.

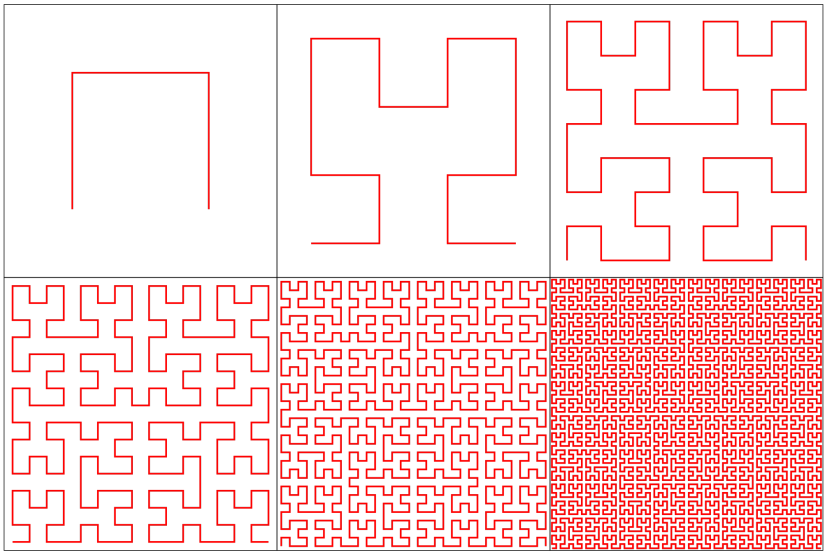


Figure 6.1 Example of Space-filling Curves

If a curve is not injective, then one can find two intersecting subcurves of the curve, each obtained by considering the images of two disjoint segments from the curve's domain. The two subcurves intersect if the intersection of the two images is non-empty. One might be tempted to think that the meaning of curves intersecting is that they necessarily cross each other, like the intersection point of two non-parallel lines, from one side to the other. However, two curves or two subcurves of one curve may contact one another without crossing, as, for example, a line tangent to a circle does. For the classic Peano and Hilbert space-filling curves, where two subcurves intersect, in the technical sense, there is self-contact without self-crossing. A space-filling curve can be everywhere self-crossing if its approximation curves are self-crossing. A space-filling curve's approximations can be self-avoiding, as Figure. 6.1 illustrates. In 3 dimensions, self-avoiding approximation curves can even contain knots. Approximation curves remain within a bounded portion of n-dimensional space, but their lengths increase without bound. That is, space-filling curves are special cases of fractal constructions. No differentiable space-filling curve can exist. Roughly speaking, differentiability puts a bound on how fast the curve can turn. That is, with the help of space-filling curve, we can reduce the data dimension and then apply the strategy what we discussed with 2-dimensional data.

6.4 Data Allocation

We also offers an optimization for data distribution. It is sure that the system will be more balanced if the data with close features are allocated closer. It is also good to keep the locality coherence of data for retrieval, especially for multi-dimensional data. Locality sensitive functions offered by [27] can facilitate data distribution and offer good performance of query processing. If data is allocated according to its range or locality, range queries as well as KNN queries will be much more efficient. This is because, when we are processing a range query, we want to retrieve the data in a small range, and if they are stored in a same server or servers that are relatively closer to each other, the cost of routing together with cost of I/O are reduced. This situation is exactly the same with KNN queries. However, there is an obvious disadvantage of locality sensitive data allocation. This design may cause the system to be less scalable and less balancing. Especially when there are hot data sets, which are more frequently queried than other data sets, in the system. Since locality sensitive data allocation is in use at this moment, the servers storing hot data will be much busier than other servers in the cloud. Moreover, most of the query messages are forwarded to these servers, breaking the balance of the whole network.

That's why we also want to discuss another significant improvement that can be done, which is about queries with specific frequency. Even though the data is not allocated according to some given distribution in the cloud, it is a common scene in real world that some data are more hot and queried more frequently than others. So despite without using the locality sensitive data allocation, in this case, it is sure that some servers which are responsible for the hot data are more busy and are in the risk of being bottleneck for the system. A heuristic strategy for this problem is that we may want to assign a dynamic potential index range for each server and discuss the cost of maintenance and the improvement of performance. This strategy is somewhat alike the meta-data maintenance in some distributed database systems. For example, in our system, we can implement a dynamic range function to change the potential index range of meta-servers according to the query frequency. In this way, the range is mapped to different representatives by the number of queries other than the number of data sets. This implementation, however, causes extra index maintenance cost because the index range of different meta-servers change along with time. Taking advantage of the connectivity of HCN, cost of messages sent in this process is small.

Chapter 7 Performance Evaluation

In this section, we evaluate the performance of RT-HCN. We simulate a HCN$(4, 2)$, generate the multi-dimensional data and randomly allocate them in different servers. We consider randomly generated data together with data following Zipf distribution to prove the scalability of RT-HCN. The queries including both point queries and range queries are randomly generated from any server. Since the cost of query processing is proved to takes only $N^{-\log_2 n}$ physical hops for any given query, performance of our system discussed in this section focus mainly on two metrics: Heat of each server and Index nodes in each server, both of which reflects the high scalability and balance of the system. Table 7.1 is the parameters used in our experiments.

Table 7.1 Parameters Used in Our Experiments

Parameter	Value	Meaning
D	$[320k, 1280k]$	Number of data items in total
h	2	Highest level of HCN
N	64	Total number of servers

7.1 Index Size

First, we estimate the number of published index nodes in different servers in the whole system. We compare the results of our publishing strategy with the one used in RT-CAN to show that we provide a more scalable and balanced publishing rule. Our evaluation is first carried out with randomly generated data and then with data that obey centralized distribution. Fig. 7.1 and Fig. 7.2 show the data distribution in the experiments.

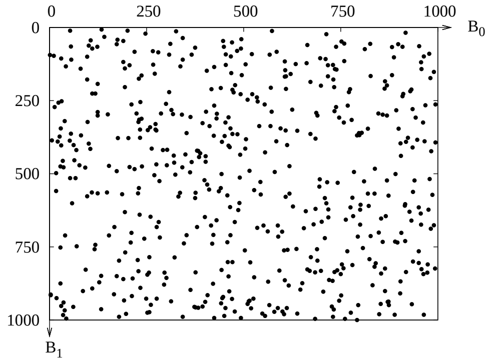


Figure 7.1 Uniform Data Distribution

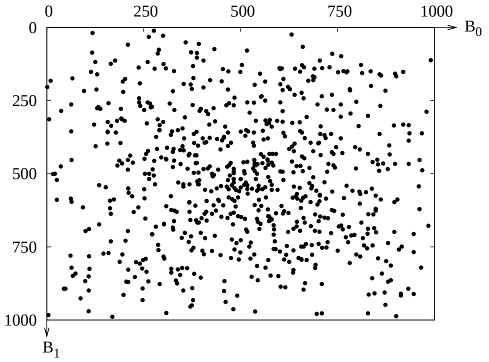


Figure 7.2 Zipf Data Distribution

We then vary the number of data items distributed to each server and investigate the changes in number of published index nodes. Fig. 7.3, 7.4, 7.5 shows the number of global indices on each server according to RT-HCN and RT-CAN publishing rules, where number of data item varies from 320000 to 1280000. From the results we can see that our system maintains less index entries at each server compared to RT-CAN under uniform data distribution. Similarly, results in Fig. 7.6, 7.7, 7.8 show that when data are not randomly distributed, our design still provides a better result with tolerate number of indices.

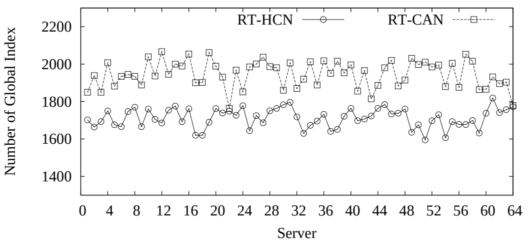


Figure 7.3 D=320,000

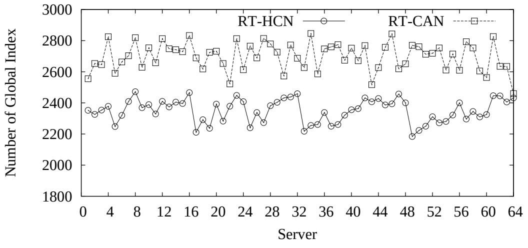


Figure 7.4 D=640,000

7.2 Scalability

Then, we estimate the visiting frequency of different servers in the network (denoted as a heatmap) when a given number of randomly generated queries are processed.

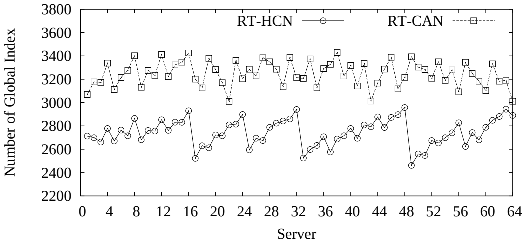


Figure 7.5 D=1,280,000

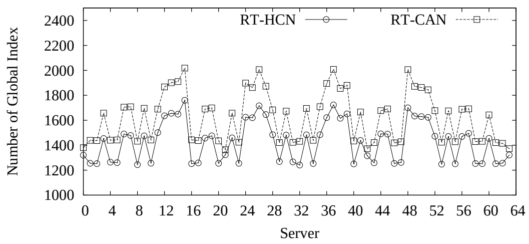


Figure 7.6 D=320,000

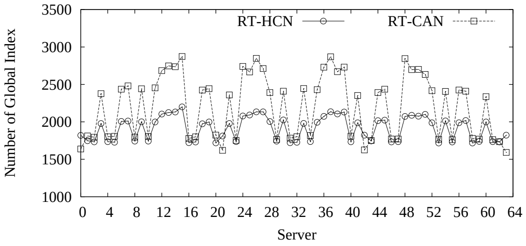


Figure 7.7 D=640,000

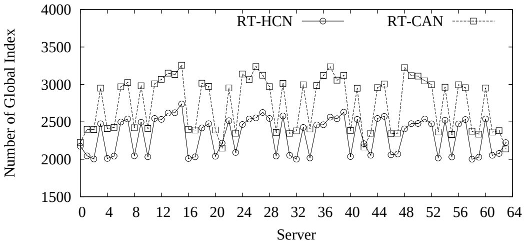


Figure 7.8 D=1,280,000

We compare the results of using shortest one-to-one routing in HCN with using fault-tolerant routing in HCN. The results are shown in Figure 7.9 and Figure 7.10. In this experiment, we randomly generate a group of queries, including all kinds of query processing. The queries are processed by servers in the system according to the query processing algorithms. Meanwhile, each server employs a counter to calculate the number of query processing the server involves. Finally, we all the queries are processed in parallel, the count of each counter is used to indicate whether the server is busy or not. The result is given in form of heatmap, which gives intuitive feelings. We can see that shortest path routing offers a not bad performance since most servers are almost as busy as each other and when we apply multipaths for one-to-one traffic we get a even more balanced performance since our publishing strategy is not locality based.

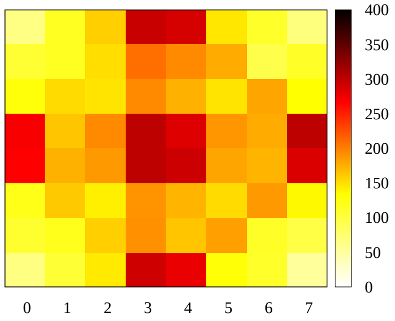


Figure 7.9 Heatmap of Shortest-path

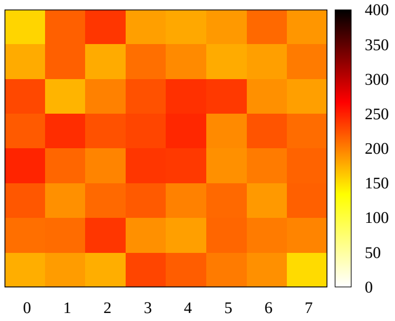


Figure 7.10 Heatmap of Multiple-paths

7.3 Other Characters

Due to several reasons, some experiments are not accomplished by the time when we end this article and the results about these features remain unclear. However, we also discuss some strategies about how we run our experiments. To start with, even though the efficiency of query processing is given by a constant value theoretically, this is merely about point query. When range queries or KNN queries are in consideration, things may be a little different since these two queries also induce an extra work to be done. This means that experiments on efficiency about different kinds of queries is feasible and can validate the implementation of our system.

Moreover, since the strategy of building a new cost function for index choosing, we did not have the time to do experiments on this characteristic. The cost function is used to choose a set of index nodes from any local R-trees, and the main purpose of this design is to maintain less index nodes in the system and mean while reduce query processing cost. So, we can compare the global index nodes' size and query efficiency in situation of whether the index cost function is in use or which cost function is in use since there is an old design.

Another experiment can be done about data dimension problem. We have already discussed that space-filling curves can be used to achieve multi-dimensional data processing in our system. We may implement the design of mult-dimensional data storage in the feature and then we can compare the efficiency of querying for multi-dimensional data with 2-dimensional data. Since higher dimensional data often induce less efficiency in cloud computing, this experiment can help us understand whether space-filling curve is a good strategy in solving multi-dimensional data mapping.

Chapter 8 Conclusion

We gives a conclusion of our article in this chapter. We are discussing how to build an efficient index structure in cloud storage system, this is well worth researching because indexing strategy is an essential implementation in cloud computing for both query processing and data retrieving. Moreover, P2P network is a overlay network which can be used as physical topology of cloud storage systems. However, this is a traditional design and nowadays, data center network is gaining its popularity for cloud computing systems due to its advantages in features like scalability, network balancing, high bandwidth and so on. That is why, our focus is on how to build efficient index structure on data center network to facilitate cloud computing. HCN, is the topology that we choose because it is one of the latest design with great performance. Another key idea is about the indexing construction, since it may be difficult to create novel but authoritative way for index construction, we simply transplant the way used in traditional P2P network and made it more suitable for our data center network topology by some improvements.

Generally speaking, in this paper, we proposed an indexing scheme named RT-HCN for multi-dimensional query processing in data centers, which are the infrastructures for building cloud storage systems and are interconnected using a specific data center network (DCN). RT-HCN is a two-layer indexing scheme, which integrates HCN [2]-based routing protocol and the R-Tree based indexing technology, and is portionably distributed on every server. Based on the characteristics of HCN, we design a special index publishing rule and query processing algorithms to guarantee efficient data management for the whole network. We prove theoretically that RT-HCN is both query-efficient and space-efficient, by which each server will only maintain a constrained number of indices while a large number of users can concurrently process queries with low routing cost. We compare our design with RT-CAN [3], a similar design for traditional P2P network. Experiments validate the efficiency of our proposed scheme and depict its potential implementation in data centers.

REFERENCES

[1] Guo, Deke and Chen, Tao and Li, Dan and et.al. ”BCN: expansible network structures for data centers using hierarchical compound graphs” *IEEE International Conference on Computer Communications*, 61-65, 2011.

[2] Guo, Deke and Chen, Tao and Li, Dan and et.al. ”Expandable and cost-effective network structures for data centers using dual-port servers” *IEEE Transactions on Computers*, 1303-1317, 2013.

[3] Wang, Jinbao and Wu, Sai and Gao, Hong and et.al. ”Indexing multi-dimensional data in a cloud system” *ACM International Conference on Management of Data*, 591-602, 2010.

[4] Ghemawat, Sanjay and Gobioff, Howard and Leung, Shun-Tak ”The Google file system” *ACM SIGOPS Operating Systems Review*, 29-43, 2003.

[5] DeCandia, Giuseppe and Hastorun, Deniz and Jampani, Madan and et.al. ”Dynamo: amazon’s highly available key-value store” *ACM SIGOPS Operating Systems Review*, 205-220, 2007.

[6] Lakshman, Avinash and Malik, Prashant ”Cassandra: a decentralized structured storage system” *ACM SIGOPS Operating Systems Review*, 35-40, 2010.

[7] Beaver, Doug and Kumar, Sanjeev and Li, Harry C and et.al. ”Finding a Needle in Haystack: Facebook’s Photo Storage” *USENIX Symposium on Operating Systems Design and Implementation*, 47-60, 2010.

[8] Baker, Jason and Bond, Chris and Corbett and et.al. ”Megastore: Providing Scalable, Highly Available Storage for Interactive Services” *ACM Conference on Innovative Data Systems Research*, 223-234, 2011.

[9] Corbett, James C and Dean, Jeffrey and Epstein, Michael and et.al. ”Spanner: Google’s globally-distributed database” *ACM Transactions on Computer Systems*, 8, 2013.

[10] Wu, Sai and Wu, Kun-Lung ”An Indexing Framework for Efficient Retrieval on the Cloud” *Bulletin of TCDE of the IEEE Computer Society*, 75-82, 2009.

[11] Wu, Sai and Jiang, Dawei and Ooi, Beng Chin and Wu, Kun-Lung ”Efficient b-tree based indexing for cloud data processing” *ACM International Conference on Very Large Data Bases*, 1207-1218, 2010.

[12] Chen, Gang and Vo, Hoang Tam and Wu, Sai and et.al. ”A Framework for Supporting DBMS-like Indexes in the Cloud” *ACM International Conference on Very Large Data Bases*, 702-713, 2011.

[13] Jagadish, Hosagrahar V and Ooi, Beng Chin and Vu, Quang Hieu ”Baton: A balanced tree structure for peer-to-peer networks” *ACM International Conference on Very Large Data Bases*, 661-672, 2005.

[14] Ratnasamy, Sylvia and Francis, Paul and Handley, Mark and et.al. ”A scalable content-addressable network” *ACM Special Interest Group on Data Communication*, 161-172, 2001.

[15] Al-Fares, Mohammad and Loukissas, Alexander and Vahdat, Amin ”A scalable, commodity data center network architecture” *ACM Special Interest Group on Data Communication*, 63-74, 2008.

[16] Guttman, Antonin ”R-trees: A dynamic index structure for spatial searching” *ACM Special Interest Group on Management of Data*, 47-57, 1984.

[17] Greenberg, Albert and Hamilton, James R and Jain, Navendu and et.al. ”VL2: a scalable and flexible data center network” *ACM Special Interest Group on Data Communication*, 51-62, 2009.

[18] Guo, Chuanxiong and Wu, Haitao and Tan, Kun and et.al. ”Dcell: a scalable and fault-tolerant network structure for data centers” *ACM Special Interest Group on Data Communication*, 75-86, 2008.

[19] Guo, Chuanxiong and Lu, Guohan and Li, Dan and et.al. ”BCube: a high performance, server-centric network architecture for modular data centers” *ACM Special Interest Group on Data Communication*, 63-74, 2009.

[20] Li, Dan and Guo, Chuanxiong and Wu, Haitao and et.al. ”FiConn: Using backup port for server interconnection in data centers” *IEEE International Conference on Computer Communications*, 2276-2285, 2009.

[21] Li, Dan and Guo, Chuanxiong and Wu, Haitao and et.al. ”Scalable and cost-effective interconnection of data-center servers using dual server ports” *IEEE/ACM Transactions on Networking*, 102-114, 2011.

[22] Wu, Haitao and Lu, Guohan and Li, Dan and et.al. ”MDCube: a high performance network structure for modular data center interconnection” *ACM Conference on emerging Networking EXperiments and Technologies*, 25-36, 2009.

[23] Li, Dan and Xu, Mingwei and Zhao, Hongze and Fu, Xiaoming ”Building mega data center from heterogeneous containers” *IEEE International Conference on Network Protocols*, 256-265, 2011.

[24] Chang, Fay and Dean, Jeffrey and Ghemawat, Sanjay and et.al. ”Bigtable: A distributed storage system for structured data” *ACM Transactions on Computer Systems*, 4, 2008.

[25] Weil, Sage A and Brandt, Scott A and Miller, Ethan L and et.al. ”Ceph: A scalable, high-performance distributed file system” *USENIX Symposium on Operating Systems Design and Implementation*, 307-320, 2006.

[26] Aguilera, Marcos K and Merchant, Arif and Shah, Mehul and et.al. ”Sinfonia: a new paradigm for building scalable distributed systems” *ACM SIGOPS Operating Systems Review*, 159-174, 2007.

[27] Charikar, Moses S ”Similarity estimation techniques from rounding algorithms” *ACM Symposium on Theory of Computing*, 380-388, 2002.

[28] Sagan, Hans ”Space-filling curves” *Springer-Verlag New York*, 1994.

[29] Kamel, Ibrahim and Faloutsos, Christos ”Hilbert R-tree: An improved R-tree using fractals” *ACM International Conference on Very Large Data Bases*, 500-509, 1994.

ACKNOWLEDGMENTS

Here, I am glad to express my heartfelt gratitude to all those who helped me finish this thesis during the past few months. Their selfless and enthusiastic assisted me a lot in both background knowledge studying and experiments evaluation.

The deepest of my gratitude goes foremost and first to my supervisor, Professor Xiaofeng Gao, for her useful suggestions and instructive advice on my thesis. I am deeply grateful of her selfless assistance in the completion of this paper. From the very beginning, she has walked me through all stages of the writing of this paper. She provided me with not only the preliminary knowledge of cloud computing but also inspiring ideas on improvement implementation. Without her illuminating and consistent instruction, this thesis could not have reached its present form. She also guide me a lot during the publication of the conference version of this article. With her incessant help, the conference version of this paper was finally accepted by 25^{th} International Conference on Database and Expert Systems. I would like to express my thanks to Professor Gao from the bottom of my heat again.

Then, high tribute should be paid to Mr. Wanchao Liang, whose profound knowledge of programming and simulation work assist me a lot in evaluating the performance and efficiency of this design. Without his effort, I am not able to compare my work with previous design and prove the feasibility of this work. I appreciate it a lot for his accompany during the publication of the conference version. I really cherish those memories that we fought together every day and night.

I am also deeply indebted to all the other tutors and teachers in Advanced Network

Laboratory for their direct and indirect help to me. I've been in ANL for about two years, and I learned a lot about research work and laboratory life here. It is grateful to stay with all of the members here for me.

Special thanks should definitely go to my friends and schoolmates who have put considerable time and effort into their comments on the draft. It is them who cheered me up when I felt blue and down. Some of them also inspired me a lot when we are discussing together, which facilitate the completion of this article. I also owe my sincere gratitude to my friends and my fellow classmates who gave me their help and time in listening to me and helping me work out my problems during the difficult course of the thesis.

I'd also like to express my thanks to my schoolmates in Computer Science Department, with whom I fought together all these four years. Some of them also helped me solving those layout problems during this paper.

Finally, my thanks would go to my beloved family members, for their loving considerations and great confidence in me all through these years. They are really considerable for me with my everyday life and they support me in finance to help me go through my studies in college.