分类号_____ 学号 M201072448

学校代码 10487 密级_____

華中科技大學

硕士学位论文

分布式文件系统名字空间管理

学位申请人：陈云云

学 科 专 业：计算机系统结构

指 导 教 师：曾令仿　副教授

答 辩 日 期：2013.1.22

A Thesis Submitted in Partial Fulfillment of the Requirements
For the Degree of Master of Engineering

The Namespace Management of Distribute File System

Candidate : Chen Yunyun

Major : Computer Architecture

Supervisor : Associate Prof. Zeng Lingfang

Huazhong University of Science and Technology
Wuhan, Hubei 430074, P. R. China
Jan., 2013

独创性声明

本人声明所呈交的学位论文是我个人在导师指导下进行的研究工作及取得的研究成果。尽我所知，除文中已经标明引用的内容外，本论文不包含任何其他个人或集体已经发表或撰写过的研究成果。对本文的研究做出贡献的个人和集体，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。

学位论文作者签名：

日期：　　年　　月　　日

学位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，即：学校有权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。本人授权华中科技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。

本论文属于　保密□，　在_____年解密后适用本授权书。

　　不保密□。

（请在以上方框内打“√”）

学位论文作者签名：　　指导教师签名：

日期：　　年　　月　　日　　日期：　　年　　月　　日

摘 要

随着互联网的普及以及移动互联网的快速发展，人类每年产生的数据越来越多，据统计全球数据信息每年的增长率到达了每年 30%以上。现在，人类每天产生的数据要以 PB 来计算，数据类型从微博、照片、博客、日志等各式各样。近几年增长的数据要更快，这些新增的数据有相当一部分需要被持久化存储到硬盘上。单机文件系统如 ext3、ext4 以及网络文件系统文件系统(NFS)，远远不能满足实际存储需求。这个时候分布式文件系统应运而生，可以存储几十到几百 PB 的数据。分布式文件系统基本上采用的都是典型的三方架构，即由元数据服务器(nameserver)，数据服务器(dataserver)和客户端(client)组成。元数据服务器存储着整个系统的元数据信息，是最关键和复杂的部分，面临 C10K、C100K 问题。如何设计一个高效稳定的元数据服务器对于一个分布式文件系统来说至关重要。

本课题设计并实现了分布式文件系统 RaccoonFS 的元数据服务器，研究并解决了分布式文件系统名字空间的由于锁竞争带来响应速度低下的问题，提出了一种新的名字空间管理方法：元数据服务器在处理数以万计的客户端连接时，往往出现效率低下甚至完全瘫痪，怎么提升元数据服务器在高并发读写下的性能成为了一个非常重要的设计内容。RaccoonFS 采用了 B+树的方式来管理名字空间，并且在 B+树上实现了写时拷贝(Copy on Write)和多版本并发控制(Multi Version Concurrency Control)，从而达到读写分离、写写并发，极大的提高了读写的性能，规避了对 B+树读写操作上锁带来的性能开销。

在测试部分，论文选取了三组测试对象：HDFS(Hadoop Distribute File System)名字空间管理方法、实现了 COW 的 B+树名字空间管理方法、实现了 COW 和 MVCC 的 B+树名字空间管理方法。实验结果表明实现了 COW 的 B+树管理方法在读写的响应速度方面较 HDFS 名字空间管理方法有了 30%以上的提高，实现了 COW 和 MVCC 的 B+树名字空间管理方法在写响应速度方面较只实现了 COW 的 B+树名字空间管理方法有了 10%以上的提高。测试表明，采用 COW 和 MVCC 的 B+名字空间管理方法，可以有效提高名字空间管理效率。

关键词：分布式文件系统，名字空间管理，写时拷贝，多版本并发控制

Abstract

With the rapid development of internet, more and more materials need to be stored in the disk. The amount of global information grow more than 30 percent over a year, and the data that needs to be stored to the disk grow more than 114 percent over a year. Traditional file system like ext2, ext3 and net file system (NFS) can't feed our needs, At this moment, distribute file system come into our sight. Distribute file system can store 1 PB to dozens of PB data, it can feed us well. Traditional file system typically uses 3 party architecture, and nameserver is the brain of all nodes. Nameserver typically faces C10K problem , so how to design the nameserver to adapt it to high concurrency problem becomes more and more important.

This paper design and implement the distribute file system named RaccoonFS. The paper also analyzes and solves the problem that nameserver behaves bad after high concurrency because of the lock, it proposes a new namespace management method. RacoonFS uses B+ tree to manage namespace metadata, and implement copy on write and multiversion concurrency control on the B+ tree, so the read operations and write operations separate from each other and write operations can do concurrently. The things above contribute much to the performance of RaccoonFS.

In the test part, we choose three kinds of object to test : the hadoop distribute file system, the B+ tree namespace management that has implemented copy on write , the B+ tree namespace management that has implemented copy on write and multiversion concurrency control. The result show that the B+ tree namespace management that has implemented copy on write and multiversion concurrency control behaves better than the B+ tree namespace management that has implement copy on write , and the B+ tree namespace management that has implement copy on write behaves better than hadoop distribute file system. The test also shows that copy on write and multiversion concurrency control can contribute much to namespace management.

Key words: Distribute file system, Namespace management, Copy on write, Multiversion concurrency control

目 录

1 绪论

1.1 研究背景

随着互联网的普及以及移动互联网的快速发展，人类每年产生的数据越来越多，据统计全球数据信息每年的增长率到达了每年 30%以上。现在，人类每天产生的数据要以 PB 来计算，数据类型从微博、照片、博客、日志等各式各样。近几年增长的数据要更快，这些新增的数据有相当一部分需要被持久化存储到硬盘上。

新浪微博自 2010 产生，在 2 年多的时间里已经成为大家生活中不可或缺的产品，据统计截至到 2011 年 9 月底，新浪微博注册用户已经超过 2.27 亿。新浪微博用户每天发布的微博数已经从 7500 万增加到了 8600 万。这些用户产生的海量数据该如何高效存储成为一个关系到新浪微博发展的难题，据新浪工程师介绍，新浪微博会首先存储到 redis 和 mysql 的集群中，同时随着时间的增长这些数据会被同步到分布式文件系统中（目前新浪微博使用的分布式文件系统式 Hadoop Distribute File System[1]）。

微信是腾讯公司为了适应移动互联网的发展推出的一款重要的移动通讯工具，通过它我们可以发送语音、图片、文字、视频等信息。据统计，截止到 2013 年 1 月 15 日，微信的注册用户数量已经超过 3 亿。这些用户每天产生的视频和文字信息的信息量超过 PB，如何存储这些信息已经越来越重要。微信目前采用腾讯自研的分布式文件系统 XFS 来存储这些信息，腾讯的 XFS 是仿照 google 在 2003 发表的论文 google file system 的一个开源实现，用来存储 PB 级别的数据。

随着互联网的发展以及移动互联网的普及，传统的网络存储方式(如网络文件系统 NFS)[2]，由于其可以存储的资源有限以及可扩展性比较差，越来越不能满足我们的需求，这个时候分布式文件系统就越来越受到重视。源自雅虎的 Hadoop Distributed File System (HDFS)[1]这样的类 Google File System(GFS)[3]的开源分布式文件系统越来越重要，它们已经成为产业界和学术界不可或缺的基础平台。分布式文件系统由元数据服务器，数据服务器和客户端组成，元数据服务器是整个系统的大脑，管理着数据服务器，而且元数据服务器一般都是由单一的机器提供服务，所以如何提高元

数据管理对于提升整个集群的性能非常重要。

1.2 国内外研究现状

分布式文件系统出现在 20 世纪 80 年代，在经历了 30 多年的发展以后，分布式文件系统不论是在科研学术上，还是在商业应用上都取得了较大进展。分布式文件系统名字空间管理经历了下面几个阶段：集中式名字空间管理，多元数据名字空间管理和无中心名字空间管理。

1984 网络文件系统(NFS)[2]出现，现在已经发展到了第 4 个版本，几乎在所用操作系统系统中都得到了广泛使用。网络文件系统(NFS)采用的是 C/S 模式，而且基本都是分布在局域网中且服务器的数目只有一台，所以网络文件系统采用的名字空间管理方式和传统的 ext2，ext3 是相似的名字空间管理方式。网络文件系统（NFS）以 posix 接口供用户远程访问（linux 下面比较典型的提供了 posix 接口的 samba 服务，windows 下面比较典型的是 CIFS），但是由于网络文件系统的服务器只有一台机器，由于受到单机的计算能力和存储能力的影响，网络文件系统(NFS)更多地关注数据的可靠性和稳定性。

在网络文件系统(NFS)出现以后的几年中，人们对分布式文件系统分的数据访问和元数据访问做了详细的研究，发现元数据访问和数据访问有很大的差异性：在做元数据访问操作时和元数据服务器需要交换的数据包比较小，是计算密集型，消耗的主要是服务器的计算资源（内存和 CPU），而且用户对这一部分数据有较高的实时性要求；在做数据访问操作的时候，用户一般希望系统有高的吞吐率，响应的对响应并没有元数据访问要求那么高，是 I/O 密集型，而且在做数据访问的时候主要消耗服务器的网络带宽。由于这 2 中访问具有比较大的差异性，所以后来人们就把元数据访问和数据访问解耦合，出现了元数据服务器和数据服务器，这个时候为了一致性的考虑，元数据统一放在一台服务能力很强的机器上，数据服务器可以是多台，统一向元数据服务器注册自身信息，这类系统具有很强的可扩展性。这类系统国外典型的研究成果有 google 的 GFS[3]和开源分布式文件系统 HDFS[1]等系统，国内的主要有余庆的 fastdfs[4]，CorsairFS[5]，阿里云的盘古文件系统等。在这种系统架构下，

用户首先调用客户端 API 访问元数据服务器，获得对应的文件的存储的数据服务器的位置，接着直接访问数据服务器进行文件的读写。

随着数据的爆炸式增长，分布式文件系统要存储更多的数据。随着存储的数据的增长，对应的元数据容量也迅速增长，单一的元数据服务器由于存储和计算能力有限，渐渐的出现了瓶颈。所以这个时候出现了多台元数据服务器组成的集群，这些元数据服务器对外作为一个整体提供服务，对内相互协作完成元数据存储和访问的一些任务，这种架构一般采用的是多元数据名字空间管理，将名字空间信息分散到多台元数据服务器上。国外典型的研究成果有 CEPH[6]等系统，国内的主要有华中科技大学信息存储及应用实验室的 capella 文件系统、中科院的蓝鲸分布式文件系统[7,8]等。

随着时间的推移，出现了一类新的文件系统，这类文件系统以 p2p 技术作为支撑，将集群内的本来互不关联的机器组织起来集体对外提供服务。在这类文件系统中没有元数据服务器和数据服务器之分，系统中每个成员都是对等的，相互协作对外提供元数据访问和数据访问，这种架构一般采用的是无中心的名字空间管理方式，每个机器之间的角色是对等的，名字空间信息存在于系统中的每台机器。国外典型的研究成果有 Glusterfs 和 OceanStore[11]等系统，国内的主要有清华大学的 Granary[12]、以及中科大 MDN。这种系统没有元数据服务器和数据服务器之分，每个结点都是一个对等体(Peer)，它们之间直接交换共享的计算、存储等资源和服务。所以此类架构不存在单点故障，但是系统一致性和可靠性实现比较复杂。

1.3 本文的研究内容

本文主要研究了影响分布式文件系统名字空间性能的关键问题，并调研了现有的一些分布式文件系统名字空间管理方法。在现有解决方法的基础上，并结合其它系统（如数据库）的一些相似场景的解决方法，设计了一套可以支持高并发读写的解决方案。并在此基础上，设计并实现了分布式文件系统 RaccoonFS。

1.4 本文的结构

本文由五章组成：

第一章对课题的研究背景、研究内容和研究目标进行了简要的说明，重点介绍了课题的项目背景和技术背景，指出分布式文件系统名字空间管理在国内外研究现状。

第二章介绍了了分布式文件名字空间在高并发情况下由于上锁带来性能损失的问题，然后介绍了几种典型的名字空间管理方法，接下来对这些名字空间管理方法进行了对比。并由此提出了 RaccoonFS 名字空间管理方法。

第三章阐述了 RaccoonFS 系统中名字空间管理方法的详细设计与实现，介绍了系统架构设计、名字空间的数据组织、写时拷贝和多版本并发控制的具体实现。

第四章介绍了对 RaccoonFS 系统名字空间管理方法进行的性能测试，包括测试目的、测试环境、测试过程、对比了不同名字空间管理方法的性能，并对结果进行了详细分析。

第五章总结了 RaccoonFS 名字空间管理的研究成果，并指出了需要进一步改进的地方。

2 名字空间管理方法

元数据服务器是分布式文件系统的大脑，也是整个系统最关键和复杂的部分。如何设计分布式文件系统的元数据服务器是分布式文件系统架构设计中最重要的一个环节。

分布式文件系统的元数据管理主要有下面几个设计点：名字空间与名字解析、副本管理、可用性、一致性。

本篇论文主要针对 RaccoonFS 的名字空间管理，研究了传统的名字空间管理方法在高并发读写情况下的不足，提出了新的解决方案。

分布式文件系统元数据服务器主要用来存储系统文件和目录的名字空间信息以及副本信息，是分布式文件系统中最关键的组件。一般分布式文件系统元数据主要包括三部分的信息：文件名到文件 ID 的映射(名字空间信息)；文件 ID 到块 ID 的映射；块 ID 到数据服务器的映射(副本信息)。名字空间管理负责的是文件名到文件 ID 的映射，文件 ID 到块 ID 的映射这 2 部分的信息。

分布式文件系统的名字空间管理主要包括以下三种方式：集中式名字空间管理，分布式名字空间管理和无中心名字空间管理。

集中式名字空间管理是将分布式文件系统中所有的元数据放在一台元数据服务器上进行管理。这类名字空间管理比较典型的文件系统有：HDFS，GFS，KFS 等一批学术界和工业界使用的比较多的文件系统。

分布式名字空间管理是针对集中式名字空间管理单元数据服务器存储的数据量有限（内存资源），将原来单个元数据服务器存储的信息分散到多台机器上，从而提升整个集群的存储容量。这类名字空间管理比较典型的文件系统是 CEPH[13]。

无中心名字空间管理方案是随着 p2p 技术发展而提出来的一种管理方案，这种方案的好处是无中心节点和单点故障，系统的可扩展性比较好。但是这样也带来了数据的一致性更加复杂，客户端需要实现更多的功能。目前这类文件系统比较典型的代表是 GlusterFS。

2.1 集中式名字空间管理方法

在分布式文件系统出现的早期，元数据服务器和数据服务器是共用一台机器的，这样的架构元数据管理和数据管理共用一台机器的计算及存储资源，在系统的可扩展性，性能及系统的存储能力等方面出现瓶颈。后来，为了实现系统的高扩展性和高性能，人们仔细研究了分布式文件系统的元数据访问和数据访问特征，发现元数据和数据访问分开存储可以达到更好的性能，所以出现了如所示的集中式名字空间管理方案。在集中式名字管理方案中，整个系统所有与目录和文件的元数据都集中存储在一台机器上，而数据服务器则可以有多个。集中式名字空间管理的系统架构图如图 2.1 集中式名字空间管理架构。

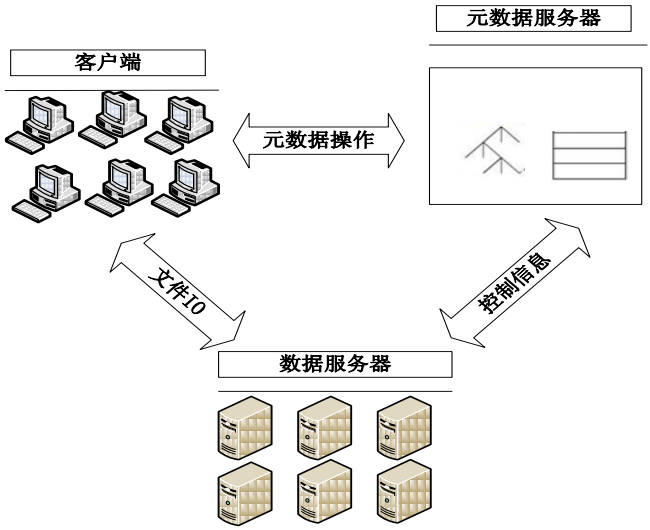


图 2.1 集中式名字空间管理架构

出于简化系统设计和实现的目的，大量的分布式文件系统采用了集中式名字空间管理，这种管理方式通常由 3 部分组成：大量客户端，大量存储服务器，单一的元数据服务器。不同于传统的 DAS 或 NAS 等系统存储访问，集中式名字空间管理中名字空间等元数据信息的访问和数据流的访问不在一台机器上，这样一方面减轻了元数据服务器的负载，另一方面使得控制信息和数据信息得以并发，极大的提高了整个系统的响应时间和吞吐率。

但是集中式名字管理方案也有下面的 2 个问题：性能问题，单点故障。

1. 性能瓶颈

由于集中式名字空间管理一般将名字空间信息全部存放于内存，从图 5 集中式名字空间管理架构可以看出，元数据服务器在整个系统中是一个单点。所以，元数据服务器的内存容量决定了整个系统的存储容量，元数据服务器的计算和处理能力决定了整个系统的处理速度和响应速度。最直接的影响是，一旦元数据服务器出现性能问题（计算，存储等），会直接导致真个系统出现极低的 QPS 和 I/O 吞吐率。

2. 单点故障

一般来说，这个问题没有性能瓶颈那么严格，但是一旦问题出现，对整个系统照成的影响也就是毁灭性的。从集中式名字空间管理的系统架构图中我们可以看到，元数据服务器是整个系统中的一个单点，因此单点故障肯定是存在的，无论是换用更高的硬件或是提高软件的质量，也只能降低单点故障发生的频率。现在针对单点故障，一般采用双机热备（High Availability）的解决方案，根据整个系统高可用的高低，可以使用一个从机或是多个从机提高系统整体的可靠性。

在调研了目前采用集中式名字空间管理的分布式文件系统，它们的实现方法一般有基于全内存的目录树名字空间管理和基于全内存的 Hash 名字空间管理两种管理方式。

2.1.1 基于全内存的目录树名字空间管理方法

HDFS 是这种名字空间管理的典型代表，HDFS 支持传统的层次型文件组织，用户可以像操作本地文件系统一样操作分布式文件系统，用户可以创建删除目录和文件，移动目录和文件，重命名目录和文件等。和上面基于本地文件系统的名字空间管理把元数据存储在磁盘不同，HDFS 名字空间的元数据完全存放在元数据服务器的内存内，它的逻辑视图可以使用如图 2.2 基于全内存的分层设计所示的层次结构来表示。HDFS 的层次结构相当于一棵树，每个结点表示分布式文件系统中的一个目录或文件，一般来说节点的子节点并没有数目的限制，子节点一般使用数组来表示，同时数组的大小也是可以动态调整的，子节点在数组内是按照由小到大的顺序排列的。由于子节点排列有序，所以可以使用二分法进行插入，查找和删除等操作，以

实现比较高的操作效率。

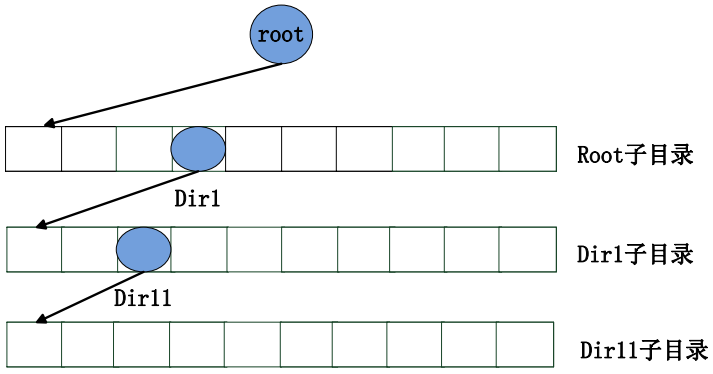


图 2.2 基于全内存的分层设计

由于集中式名字空间管理的元数据服务器是个单点，这样所有的名字空间信息都存放在一台机器上，所以在进行读写操作的时候，肯定会有竞争的产生。为了降低竞争的粒度，有人提出了采用读写锁来代替排它锁以减少竞争的粒度[1, 14-17]，HDFS 的作者也提出在做 snapshot 的时候可以采用 copy on write[18, 19]，以避免 snapshot 操作阻塞其它的写操作，造成系统不可用。但是目录树并不容易实现 copy on write，现在只能在做 snapshot 的时候做一下处理，通过新建临时目录等手段来实现。

2.1.2 基于全内存的 Hash 名字空间管理方法

GFS 是全内存 hash 名字空间管理的典型代表，通过阅读 google 在 2003 发表的论文 google file system，我们可以发现 GFS 的名字空间采用了全内存 hash 设计，同时名字空间也采用了扁平化设计。同时为了加快文件的操作，GFS 还采用了二分查找的方式来进行加速。GFS 的基于全内存 hash 的名字空间管理放法如图 2.3 基于全内存的 hash 设计所示。通过对比可以发现，和基于全内存的目录树名字空间管理方法相比，基于全内存的 Hash 名字空间管理方法在目录查找上性能更优越，比如我们要操作一个文件/a/b/c/d/e.txt，在基于全内存的目录树名字空间管理方法中需要进行 5 次二分操作，而在基于全内存的 hash 名字空间管理方法中，只需首先对/a/b/c/d 进行 hash 和一次二分查找。但是基于全内存 hash 名字空间管理方法也有一个很明显的弊端，由于它的名字空间采用了扁平状设计，所以它不能进行 ls，df，du 等操作，不能很好的满足用户的一些这方面的相关需求。

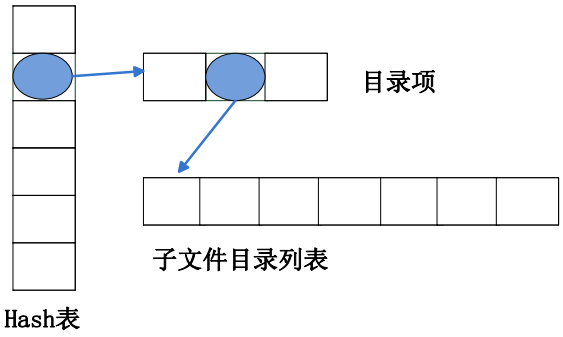


图 2.3 基于全内存的 hash 设计

2.2 多元数据名字空间管理方法

在 2.1 节集中式名字空间管理中，集中式名字空间管理一般有性能问题和单点故障 2 个方面的问题，而且这 2 个问题都是由于元数据服务器只有一台。所以，后来人们提出了将元数据服务器做成集群，相应的名字空间信息也分布到多台机器上，多台元数据服务器协作完成任务，这样我们在访问一个名字空间的时候，锁的粒度降低了很多。多元数据名字空间管理的系统架构如图 2.4 多元数据名字空间管理架构所示：

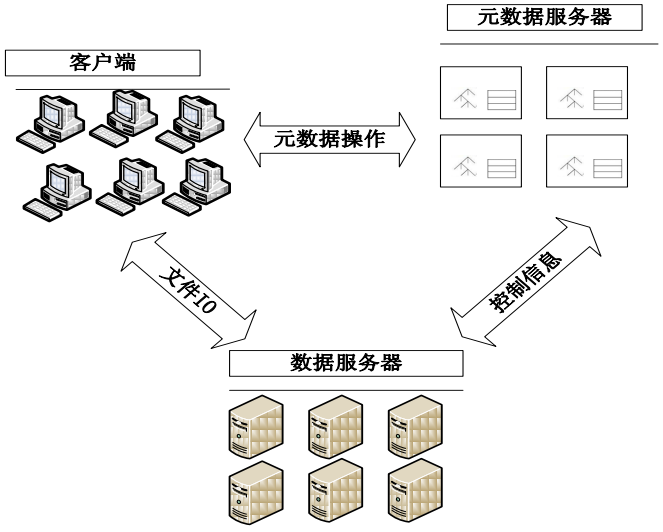


图 2.4 多元数据名字空间管理架构

在多元数据名字空间管理中，元数据服务器起之间的角色有下面 2 种：1.元数据服务器与元数据服务器之间角色对等，相互管理的名字空间信息可能有重叠。2.元数据服务器与元数据服务器之间相互协作，每个元数据服务器负责一部分元数据信息，

元数据服务器与元数据服务器之间没有名字空间信息的重叠。

多元数据名字空间管理解决了集中式名字空间管理的性能和单点故障的问题，但是它同时也引入了性能开销过大和一致性等问题。

1. 性能开销

分布式文件系统名字空间通常会由于节点之间数据同步而造成额外开销，这是因为名字空间在不同的节点之间同步时，会引入各种锁进行同步，以保证数据的一致性。而且这种锁是在不同的节点上进行的，节点与节点之间还会有网络上的开销。

2. 一致性问题

数据一致性是每个系统特别是分布式系统必须面对的一个问题。在多元数据名字空间管理中，元数据服务器与元数据服务器之间有对等或协作两种角色，无论是哪种角色都需要进行数据的同步，这就不可避免的要用到锁和同步等操作，而这些操作都是比较耗时的。

现在多元数据名字空间管理一般用到了静态子树分割和动态子树分割两种方法。

2.2.1 基于静态子树分割名字空间管理方法

静态子树分割是多元数据名字空间管理一种比较简单的实现，它首先将分布式文件系统的名字空间分割成几个相互独立的子树，已经分割好的子树的子目录是不能继续进行分裂的，其中的一个分割示例如表 2.1 目录元数据服务器对照表所示。接着把这些相互独立的子树分配到 N 台元数据服务器上。每台服务器负责这些独立子树名字空间中的一个或多个。

表 2.1 目录元数据服务器对照表

目录	提供服务的 NS
/a	NS0
/b	NS1
/c	NS2
/d	NS3

表 2.1 目录元数据服务器对照表上面的数据分布在 4 台元数据服务器上，示意图如图 2.5 静态子树分割所示。当我们要访问某个名字空间时，首先我们根据表格确

定元数据信息在哪台服务器上，接着直接访问表格上面对应的元数据服务器，所以只需要一次名字空间的访问，不需要元数据服务器之间的访问。

静态子树分割很好的把一台元数据服务器扩展为多台，降低了名字空间在访问时需要上锁的粒度，在一定程度上可以提高系统的可扩展性。但是静态子树分割子树划分的粒度比较粗，很可能热点集中出现在某棵子树上，并不能解决系统的热点问题。

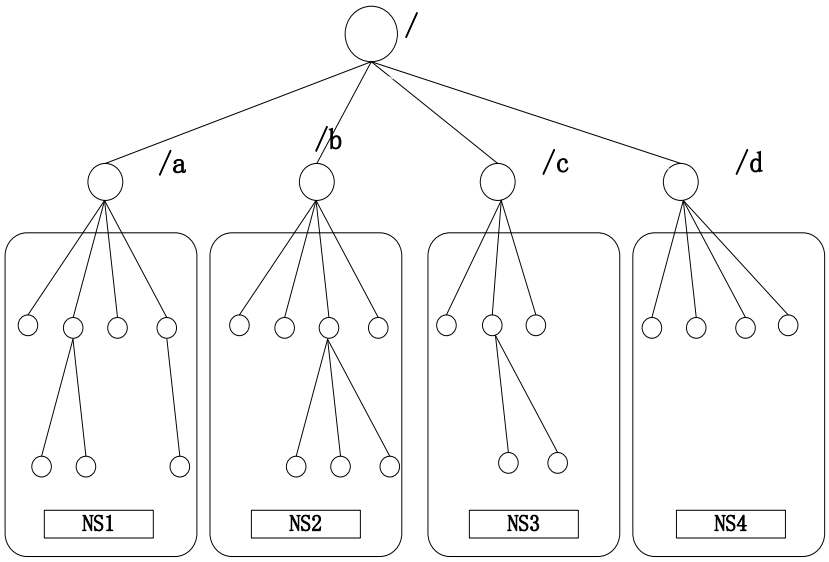


图 2.5 静态子树分割

2.2.2 基于动态子树分割名字空间管理方法

动态子树分割是在静态子树分割的基础上改进而来的。在静态子树分割名字空间管理方案中，已经分配好的子树的子目录是不能继续进行分裂的，由于这种分配方式在系统出现热点信息以后，系统的负载可能会都集中在某一台元数据服务器上。动态子树分割名字空间管理的目录分割的粒度更小更灵活，每个目录的子目录也可以继续分割，在某个子树出现负载较重的情况以后，它可以将子树的某个子树委派给系统中某个负载不重的元数据服务器进行处理，很好的解决了静态子树分割的名字空间的某个子树过热的问题。Ceph[6]是动态子树分割架构的典型代表。

相比于静态子树分割，动态子树分割的粒度更加小，这样在访问文件时，相比于对整个名字空间的上锁操作，动态子树锁的粒度更加小。但是动态子树分割由于要操作多个元数据服务器，所以要保证整个名字空间的一致性就更加复杂。当我们

访问某个文件或目录的名字空间信息的时候，由于文件或目录所在路径上涉及到的目录可能委派给了不同的元数据服务器，因此就需要访问到多个元数据服务器，这就会带来额外的网络开销，降低了系统的响应速度。

2.3 无中心名字空间管理方法

无中心名字空间管理方案是 p2p[9, 10, 20, 21]技术发展的产物，在无中心名字空间管理中每个节点都是对等，没有元数据节点和数据节点之分，它们协作的完成存储任务。无中心名字空间管理方案的系统架构如图 2.6 无中心名字空间管理架构示意图：

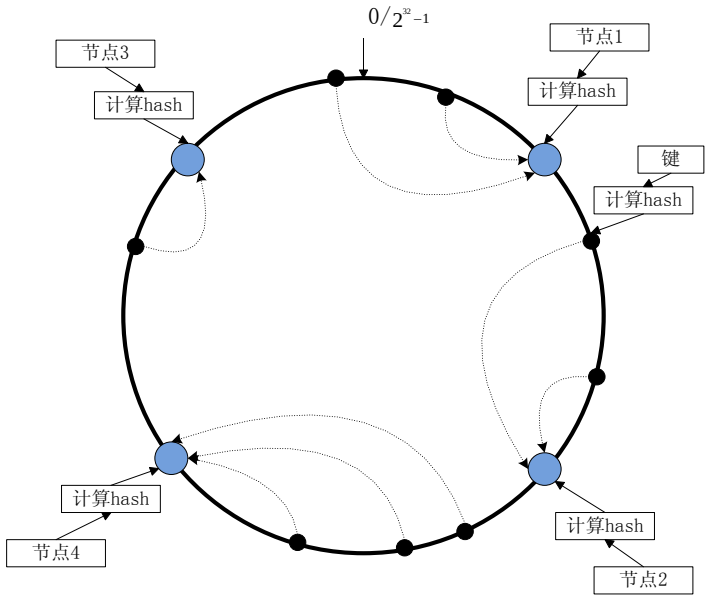


图 2.6 无中心名字空间管理架构示意图

在无中心名字空间管理中，系统的每个节点都是对等的。一般会将整个名字空间分割成很多段，每一段分给系统中的一个或多个节点（为了确保系统的可靠性）进行管理。

无中心名字空间管理，很好的解决了分布式系统中的单点故障以及由单点带来的系统性能瓶颈等方面的问题，提高了系统可靠性，稳定性及可扩展性等。但是这种架构数据一致性问题更加复杂，且缺少对范围查询的支持，产生对目录进行 ls，df，du 等操作效率低下，不能对整个系统很好的进行整体监控等不足之处。

无中心名字空间管理一般采用基于分布式一致性 hash[20, 22-24]来实现。

分布式一致性 hash 是 p2p 技术的发展的产物，近年来它被应用到 BT，视频播放，分布式文件系统及 nosql 等各种系统中进行副本 、名字空间管理。

假设一个系统有 N 台服务器，分布式一致性 hash 将名字空间信息分布到这 N 台机器的工作流程如下：

1. 在进行名字空间的操作之前，我们需要将名字空间通过 hash 函数形成一个 32 位的 hash 值。所以首先要形成一个 hash 环，我们可以把我们的空间想象成一个首(0)尾(2^32 -1)相接的圆环。

2. 把 N 台机器根据 hash 函数计算出的 hash key 放置在 hash 环上。比如我们有 5 台机器，那么它们的分布如图 2.7 hash 分布环所示：

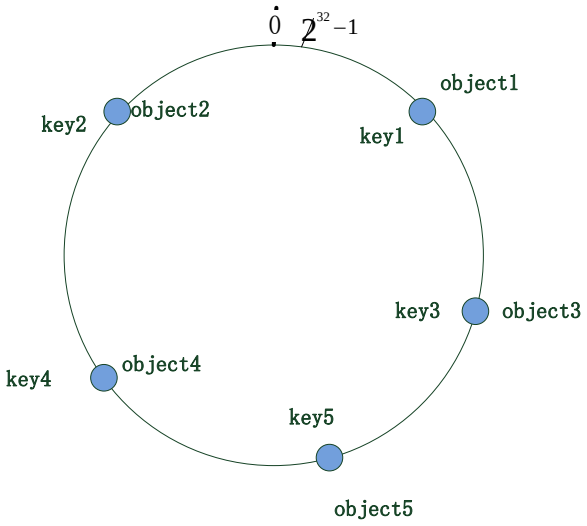


图 2.7 hash 分布环

3. 现在名字空间和 N 台机器都已经通过同一个 hash 算法映射到 hash 环上了，接下来就是要将名字空间信息存储到对应的机器上。在 hash 环形空间中，如果沿着顺时针方向从名字空间的 key 值出发，直到遇见一个节点对象，那么就将该名字空间存储在这台节点上，因为名字空间以及节点对象的 hash 值是固定的，因此必定能唯一确定名字空间和节点的对应关系。

通过上面的操作过程我们可以发现，系统中的每台机器只用负责整个名字空间的一部分，而且分布式一致性 hash 还保证了节点加入或宕机时名字空间信息可以动

态迁移，保证了整个系统的可靠性和可扩展性。

2.4 RaccoonFS 名字空间管理方法

对于分布式文件系统元数据服务器，最主要的就是要保存名字空间信息和副本信息。现在大部分分布式存文件系统使用了集中式或分布式名字空间管理方法，集中式名字空间管理方法可以很好的解决系统一致性的问题，具有一定的可扩展性，但是系统总体的存储能力受限于元数据服务器，存储能力有限。分布式名字空间管理方法可以很好的解决集中式名字空间遇到的问题，但是由于元数据信息分散在不同的机器上，而元数据信息与元数据信息一般具有比较强的关联性，所以为了保证元数据的一致性，需要带来比较大的额外开销，而且编码实现比较复杂。无中心名字空间管理方法在目前出现的分布式文件系统中使用较少，它具有非常好的可扩展性，但是和分布式名字空间管理方法相比，为了保证一致性，它需要付出更大的开销。

在调研了 Google 的 GFS，yahoo 的 HDFS，facebook 的 haystack[25]，腾讯的 XFS，以及 UCSC 的 CEPH 等文件系统以后，充分分析了其名字空间设计特性，及实际应用场景，不难注意到，在主要的 Web 应用运行环境中，普遍存在海量用户相互协同的情况，完全分布式结构虽然可以良好的支持性能扩展，但随着系统规模不断增长，将几乎无法保证一致性，集中式便于全局管控，结合适当方法进行优化，有望在性能与一致性几个方面均获得改进。所以 RaccoonFS 选择了集中式的名字管理方法。在选择了集中式名字空间管理方法以后，我们又分析了基于全内存的目录树管理和基于全内存的 hash 管理方式发现：这 2 种方式都不可避免的要上锁，目录树方式要上全局锁，hash 的方式要对行上锁而且 ls，df，du 等操作非常慢。我们结合上面 2 种设计方式，设计了一种新的目录管理方法：采用了 B+树的方式来管理名字空间，并在 B+树上实现了写时拷贝(COW)和多版本并发控制(MVCC)技术，实现读写分离，以及写写并发，有望提高读写性能，规避上锁带来的性能开销。

2.5 本章小结

本章介绍了了分布式文件名字空间在高并发情况下由于上锁带来性能损失的问

题，然后介绍了几种典型的名字空间管理方法：集中式名字空间管理方法、多元数据名字空间管理方法、无中心名字空间管理方法。接下来对这些名字空间管理方法进行了对比，并由此提出了 RaccoonFS 名字空间管理方法。

3 RaccoonFS 架构及名字空间管理实现

在第二章中讲到 RaccoonFS 采用了集中式的元数据管理，在集中式元数据管理中名字空间管理是个很重要的设计点：随着互联网的迅速发展，越来越多的网络服务开始面临 C10K[26]，C100K 问题，分布式文件系统分的元数据服务器也不例外。元数据服务器在处理数以万计的客户端连接时，往往出现效率低下甚至完全瘫痪，怎么提升元数据服务器在高并发读写下的性能成为了一个非常重要的设计内容。

本章首先介绍 RaccoonFS 分布式文件系统分的架构，接着介绍 RaccoonFS 名字空间管理实现方案。

3.1 RaccoonFS 架构及设计

RaccoonFS 采用了集中式的元数据管理方式，为典型的三方架构。系统的整体架构如图 3.1 RaccoonFS 系统架构所示：

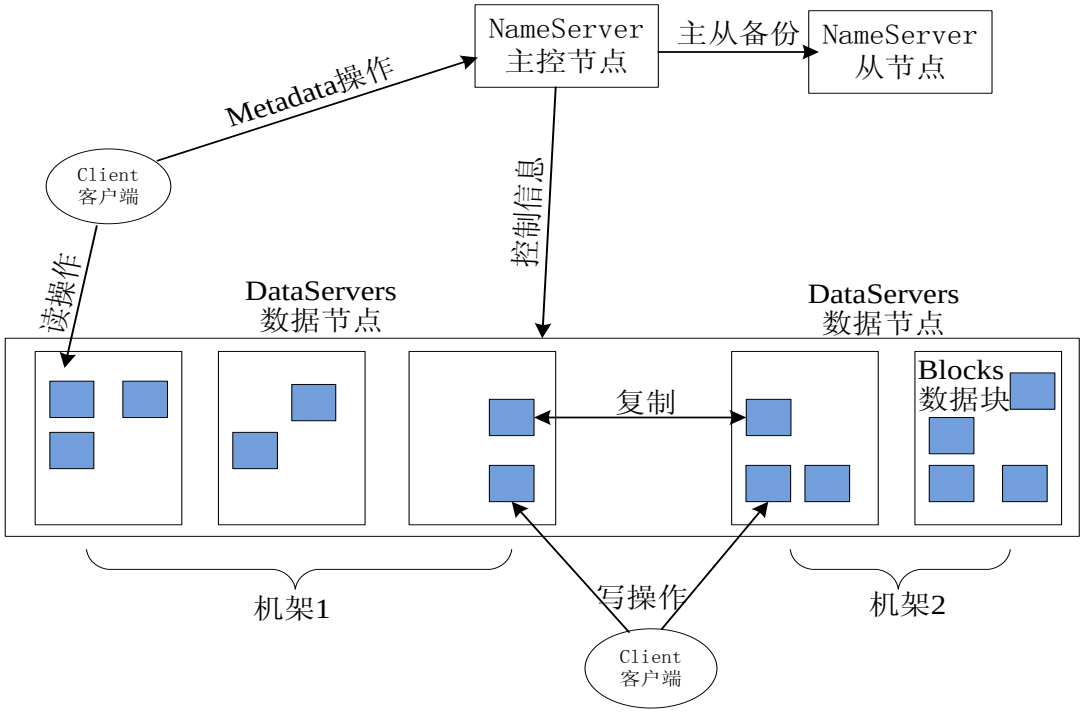


图 3.1 RaccoonFS 系统架构

图 3.1 RaccoonFS 系统架构是系统内部逻辑架构简图。NameServer 是主控节点，存放 DataServer 的节点信息和三级元数据信息，即包括 filename 到 fileid 的映射，fileid 到 Blockid 的映射，Blockid 到 DataServers 的映射；NameServer 有多台机器热备，防止单点故障。由于存储的是元数据信息，数据量很小，可以都存放于内存（数据密集，元数据非密集），并定期更新到磁盘。此外，Client 端与 NameServer 端只有控制流（交互 Block 位置信息），没有数据流，Client 端可以缓存数据信息，并不需要每次都去访问 NameServer，因此对于 NameServer 来说，负载是比较小的。主 NameServer 与从 NameServer 拥有共同的虚拟 IP，通过 heartbeat 管理，主 NameServer 失效后，从 NameServer 能立即获得虚拟 IP，并对外提供服务。

DataServer 是真正存储数据的地方，将大文件切割后，按照一定负载均衡策略存放在磁盘。DataServer 是以 Block 为最小单位存储（64M），每份数据存储 3 个备份（可配置）。写入时，只有流水线中所有的 DataServer 均写成功之后才返回成功，保证强一致性；读取时，3 台均可作为读取服务器。

Client 提供给用户使用，提供 read、write，delete 等几个简单的操作接口，让后面的分布式集群对用户透明。Client 与 NameServer 通信，获得文件分配在哪几个 Block 以及相应 Block 到机器 ip 的映射关系，然后根据获得的信息直接与该 DataServer 交互完成相应的读、写删除操作以及文件的切割合并操作。

RaccoonFS 元数据服务器采用集中式的架构，只有一台元数据服务器，但是为了提高可靠性有多台机器做副本热备（一主一备或一主多备），对外提供唯一的 VIP，绑定在主服务器上。主服务器负责提供所有对外服务，并且实时将数据同步到备机，并保证数据的顺序和一致性。主服务不可用时候，HA 将 VIP 漂移到备机，这时备机将变为主机并对外提供服务。

客户端所有的更新请求，都将由元数据服务器处理，元数据服务器将所有的更新记录写入 commit log 并同步到备机后再写入内存中。元数据服务器 Slave 启动后首先和 Master 同步，同步到一致状态后 Master 通过 Lease 机制给 Slave 授权，此时 Slave 可以切换为 Master。当 Master 宕机后，HA 将 VIP 漂移到备机，备机检测到

VIP 和本机 IP 相同且持有的 Lease 有效则自动切换为 Master 对外服务。

元数据服务器模块及功能介绍：

RaccoonFS 元数据服务器按照功能将模块划分如图 3.2 系统模块设计，各个模块说明如下：

1. 网络管理模块

模块描述：

服务器端，处理流程如下：从套接字读取请求；解析请求（反串行化）；处理请求；发送响应。网络框架主要处理网络数据的接收，发送；套接字管理；接收与发送数据缓冲区内存管理。选取网络框架同时也选取了服务器的线程模型。

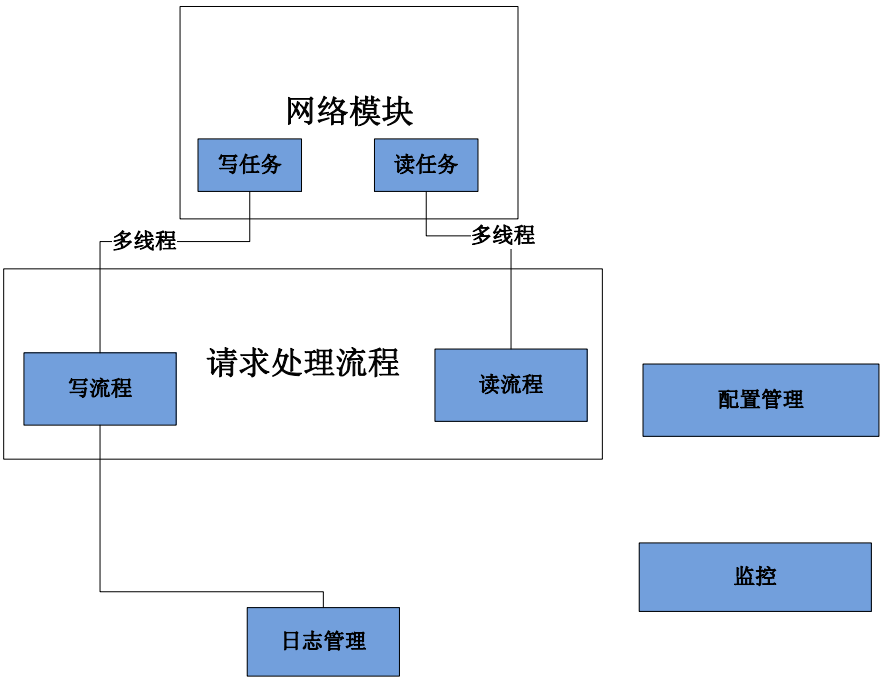


图 3.2 系统模块设计

网络处理线程不断的从套接字中读取 request 并解序列化包头，根据包的类型（是读请求还是写请求）分别 push 到读队列或者写队列。对于读队列和写队列，每个队列都有多个工作线程来不断的处理这些请求，工作线程不断的从任务队列中 pop 出 request，并根据每个任务的具体类型对系统的名字空间（B+树）和副本（BlocksMap）进行相应的操作，在任务处理完成以后，发送 resonse 给客户端，在发送 response 的

时候，网络线程需要知道 response 应该发送给哪个套接字，为了解决这个问题为每个套接字分配一个全局唯一的 64 位 id(channel id)，请求和响应消息中都必须包含该 64 位 id。

2. 日志管理模块

模块描述：

Commit Log（操作日志）是用来记录客户端（client）和数据服务器(dataserver)所有对名字空间（B+树）以及副本（BlocksMap）的写（修改）操作。nameserver 按照 60M 对 commit log 进行切分，日志文件的名称在系统中直接采用一个无符号的 64 位整数来命名，每当当前日志文件的大小大于等于 60M 时，日志文件发生切换，新的日志文件的名字在当前日志文件名字的基础上加一，同时新的日志文件变成当前文件。

nameserver 采用主从热备的模式进行工作，主从热备可以采用一主一备或一主多备。Nameserver 在启动时，首先检查虚拟 ip（vip）是否在自己身上，如果在自己身上则以 master 的身份启动，否者以 slave 的身份启动。如果是以 master 的身份启动，需要在 master 机器上开启读请求处理线程和写请求处理线程。如果是以 slave 身份启动，则需要开启日志接受线程和日志回放(replay)线程。当有写请求到达 master 机器时，master 首先生成 commit log ，同时发送给 slave，在收到 slave 响应时，首先将 commit log 写入日志文件同时对名字空间（namespace）和副本（BlocksMap）进行响应的修改操作。Slave 的日志接受线程在收到主机发来的 commit log 时，直接将 commit log 写入日志文件，slave 的另外的日志回放线程负责读取日志接受线程收到的 commit log，并进行回放相应的操作，追赶 master 的操作。

3. 统计监控模块

监控系统采用 ganglia 来监控分布式文件系统的整体运行状态，为运维、整个系统的 bug 定位以及系统瓶颈的发现提供支持。监控的内容主要包括 ip 地址信息，日志同步信息，client 端信息，nameserver 端信息，dataserver 端信息等。

监控模块监控的信息如表 3.1 监控信息列表：

表 3.1 监控信息列表

Ip 地址	系统的虚拟 ip 地址（HA 使用） slave，master 的自身的 ip 地址
日志同步	Master 和 slave 超时重试次数 每天新产生的日志数量，日志所占磁盘空间大小
Client 端监控内容	每天从 client 发来的请求中，读请求次数，写请求次数。 每天从 client 发来的请求中，不同错误类型的请求个数（如请求的文件路径不存在等）
Dataserver 端监控内容	每天从 client 发来的请求数，请求处理成功和失败的数目 每天从 nameserver 端发出的请求数，请求处理成功和失败的数目
Nameserver 端监控内容	内存使用率情况，CPU 利用率情况，磁盘利用率 读任务队列任务统计，已经处理的任务数目以及队列里未处理的任务数目 写任务队列队列任务统计，已经处理的任务数目以及队列里未处理的任务数目

3.2 RaccoonFS 名字空间管理方法

在 HDFS 的名字空间管理中采用了目录式的管理方式，这种管理方式有下面 2 个方面的不足：

1. 如果目录分布比较均匀，目录的深度没有特别的差异，目录式的管理方式可以很好的处理读写请求。但是如果出现目录深度相差特别大，目录分布不均匀的这种情况，目录式的管理方式就会出现对深度比较深的目录和文件的处理特别耗时，从而影响整体的性能。

2. 在高并发读写(C10K，C100K)场景中，由于读写操作都要对名字空间进行操作。为了保证数据的正确性和一致性，就必须对名字空间上锁，而且这个锁的粒度是名字空间整体，锁的粒度很大，这样就必定会影响文件系统的整体性能。

针对 HDFS 目录式组织方式的第一点不足，RaccoonFS 采用了 B+树的方式来管理名字空间。针对其第二点不足，RaccoonFS 在 B+树上实现了写时拷贝(COW)和多版本并发控制(MVCC)技术。

3.3 RaccoonFS 名字空间数据组织

RaccoonFS 使用的 B+树[27, 28]定义如下(m 阶)：

1. 每个节点最多可以有 m 个元素；

2. 除了根节点外，每个节点最少有 (m/2) 个元素；

3. 如果根节点不是叶节点，那么它最少有 2 个孩子节点；

4. 一个有 k 个孩子节点的非叶子节点有 (k-1) 个元素，按升序排列；

5. 所有的叶子节点都在同一层；

6. 某个元素的左子树中的元素都比它小，右子树的元素都大于或等于它；

7. 非叶子节点只存放关键字和指向下一个孩子节点的索引，记录只存放在叶子节点中；

B+树用图形表示如图 3.3 B+树示意图:

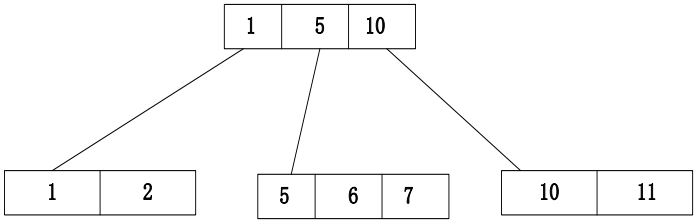


图 3.3 B+树示意图

在 B+树中可以定义 K 是键，K 在 B+树种是可以比较大小的，可以自己定义比较函数。在 B+树中定义 L 是标示，来表示节点是否是叶子节点。节点内的键值对按照键值由小到大排列。

为了保存文件名到文件 ID，文件 ID 到 Block ID 的映射以及 Block 的一些属性。B+ 树上面的节点有以下几种属性(META_INTERNAL， META_DENTRY，META_FATTR，META_BLOCKINFO)。

META_INTERNAL：用于标示非叶子节点。

META_DENTRY ：用于标示叶子节点里面的（保存 filename-fileid 的映射关系）的节点。

META_FATTR：用于标示叶子节点里面(保存 file 的属性以及 fileid 到 Blockid 映射)的节点。

META_BLOCKINFO：用于标示叶子节点里面（保存信息）的节点。

在 B+树中，只有叶子节点保存数据，非叶子节点中保存的是索引信息，用来加快

查找。在系统中存在三类叶子节点，分别是文件目录节点(MetaDentry)，文件属性节点(MetaFattr)，数据块节点(MetaBlockInfo)，节点与节点之间是可以比较大小的，其大小决定了其在 B+树叶子节点上的位置，其中目录节点<文件属性节点<数据块节点。

1. 基类节点

```
//基类节点
class MetaNode
{
        MetaType type;//节点属性
       int64_t fid;      //节点所属的文件或目录id
};
enum MetaType
{
       META_INTERNAL，          //内部节点
       META_DENTRY，            //文件或目录的dentry
      META_FATTR，             //文件或目录属性
       META_BLOCKINFO，        //文件block信息
};
class Key
{
       MetaType kind;    //第一个比较要素(META_DENTRY < META_FATTR < META_BLOCKINFO)
       KeyData kdata1; //第二个比较要素
       KeyData kdata2; //第三个比较要素
       int compare(const Key &test) const;//比较函数确定节点在B+树上的放置位置
};
```

2. 文件目录节点

```
//保存系统中文件或者目录的dentry基本信息
Class MetaDentry : public class MetaNode //节点MetaType为META_DENTRY
{
      int64_t pid; //父目录的ID
      string    name; //文件或目录的名字
      const Key key() const
      {
```

```
            return Key(BLADE_DENTRY，  pid，  fid);
        }
};
```

3. 文件属性节点

```
//保存文件或目录的一些基本属性，相当于inode节点
Class MetaFattr : public class MetaNode//节点MetaType为META_FATTR
{
    FileType fileType;          //标识为文件或目录
    int16_t numReplicas;        //需求副本数目
    struct timeval mtime;       //修改时间
    struct timeval ctime;       //属性修改时间
    struct timeval crtime;      //创建时间
    int64_t chunkcount;         //数据块数目
    off_t filesize;     //文件大小
    const Key key() const
    {
        return Key(BLADE_FATTR，  fid);
    }
};
```

4. 数据块节点

```
//保存Block的一些基本信息
classMetaChunkInfo : public class MetaNode//节点MetaType为
META_BLOCKINFO
{
    int64_t offset;          //数据块在文件内的偏移量
    int64_t blockId;         //数据块ID
    int32_t blockVersion;        //数据块版本号
     int16_t num_replicas_;//副本数
    const Key key() const
    {
        return Key(BLADE_CHUNKINFO，  fid，  offset);
    }
};
```

上面 3 类叶子节点之间是可以比较大小的，比较函数实现如下：

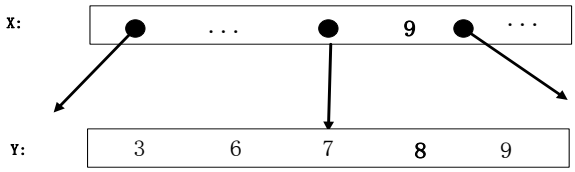
```
//Key的比较函数，决定key在B+树上的排列位置
int Key::compare(const Key &test) const
{
	int kdiff = kind - test.kind;
	int d = 0;

	if (kdiff != 0)
	{
		d = kdiff;
	}
	else if (kdata1 != test.kdata1)
	{
		d = (kdata1 < test.kdata1) ? -1 : 1;
	}
	else if (kdata2 != test.kdata2 && kdata2 != MATCH_ANY && test.kdata2 != MATCH_ANY)
	{
		d = (kdata2 < test.kdata2) ? -1 : 1;
	}
	return d;
}
```

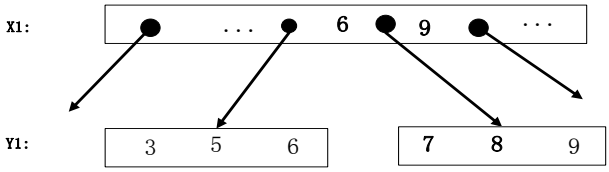
通过上面的比较函数可以发现，具有相同父节点的文件或目录在 B+树上连续排列，一个文件的 block 在文件中的位置连续排列。这种结构不但可以规避传统目录结构分布不均匀时带来的性能开销，而且通过上面比较函数的设计可以很好的实现 ls，df，du 等操作。

3.4 RaccoonFS 写时拷贝(COW)

RaccoonFS 的元数据服务器面临的是高并发的读写。下面来看 RaccoonFS 名字空间的访问的一个实际场景：我们用 T 来代表客户端对元数据服务器的具体请求，T 可以是读请求也可以是写请求，用 N 来代表 B+树，用来存储分布式文件系统的名字空间信息。B+树的并发操作比较复杂，下面以一个实例来说明对 5 阶 B+树的并发操作并不正确（并发读写情况下的读写冲突），示意如图 3.4 并发读写冲突。



(a)B+树插入节点5分裂前



(b)B+树插入节点5分裂后

图 3.4 并发读写冲突

现在有 2 个线程，一个读线程(T1)，读取 K 值为 6 的节点的值；一个写线程(T2)，向 B+树中写入节点 5.

考虑下面的表 3.2 操作流程：

表 3.2 操作流程

时序	T1(find(6))	T2(insert(5))
1	N=read(6)	
2		N=read(5)
3	检查指向 Y 的指针	
4		检查指向 Y 的指针
5		N=read(Y)
6		把 5 插入到 N 中，Y 分裂为 N，N1
7		put(N1，Y1)
8		Put(N，Y)
9		把 Y1 加入到 X 中
10	C=read(Y)	
11	未发现 6(错误)	

上面的结果之所以会错误，是因为在读写并发操作中，第一个线程在读到 6 所属的数据块以后，第二个操作抢占了 CPU 的使用权，并把 6 所属的数据块一分为 2，所以结果是错误的。解决读写竞争造成的错误主要有 2 中方案：

解决方案 1：基于锁的事务

为了让 T 的处理可串行化（执行结果和各个业务的执行顺序无关），最直接的方法就是为存储名字空间的 B+树上排它锁，不过为了提升性能也可以上读写锁，这种类型的锁读读之间互不影响，读写之间互相排斥，读请求或者写请求 T 在执行具体操作之前上锁，在操作完成以后释放自己占用的锁。这种方案实现简单，但是为了名字空间 N 在并发情况下数据的一致性，需要对整个名字空间上锁，锁的粒度比较大，对性能的影响比较明显。

解决方案 2：写时拷贝

Copy-on-write[28-33]技术在互联网公司使用比较多，这时因为大多数应用的读写比例接近 9: 1，Copy-on-write 读操作不用加锁，读写之间互不影响，极大地提高了读的效率，特别是现在服务器一般都有 8 个或者 16 个核。

RacconnFS 采用树形结构来管理名字空间，一方面可以避免目录分布不均匀带来的操作开销，另外一方面是因为 Copy-on-write 技术在树形结构中比较容易实现。假如实现一个支持 Copy-on-write 的 B+树，基本可以用来作为大多数管理结构的内部数据结构，比如 RaccoonFS 的文件名管理，Block 管理。Copy-on-write 的示意图如图 3.5 COW 示意图：

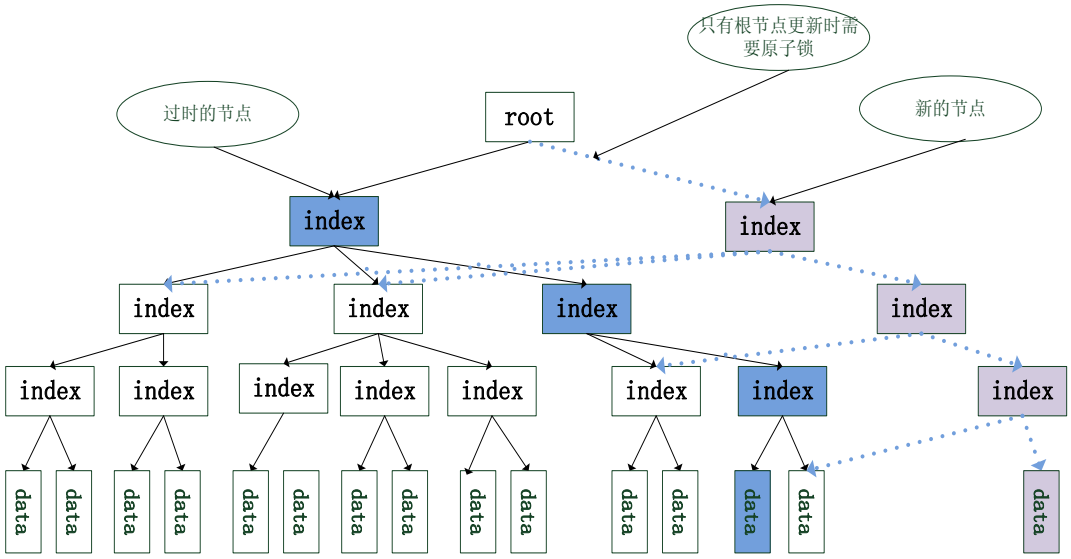


图 3.5 COW 示意图

如图 3.5 COW 示意图所示，B+树每次执行更新操作时，将从叶子到根节点路径上的所有节点先拷贝出来，并在拷贝的节点上执行修改，更新操作通过原子地切换

根节点的指针来提交。由于使用了 Copy-on-write 技术，如果读取操作发生在写操作生效前将读取老的数据，否则读取新的数据，不需要加锁。Copy-on-write 技术需要利用到引用计数，当节点没有被引用，也就是引用计数减为 0 时被释放，引用计数极大地增加了数据结构的复杂度。

Copy-on-write 技术还带来了一个好处，那就是 Snapshot 的时候无需中断，而 Snapshot 功能对于分布式文件系统非常重要。

如果需要对 B+树的某一个子树进行 Snapshot 操作，首先需要对子树的根节点增加引用计数，后续的读取操作都将读取执行 Snapshot 操作时的数据。

为了实现 copy on write，需要对 B+树做如下几个方面的修改:

1. 自顶向下进行更新
2. 去掉了 B+树叶子节点与叶子节点之间链接
3. 采取了一种 lazy 方式的引用计数，实现被拷贝节点的自动释放

下面用 3 阶的 B+树来进行查询，插入，更新，删除的说明，图 3.6 B+初始化是一个已经初始化好的深度为 2 的一个 3 阶 B+树。

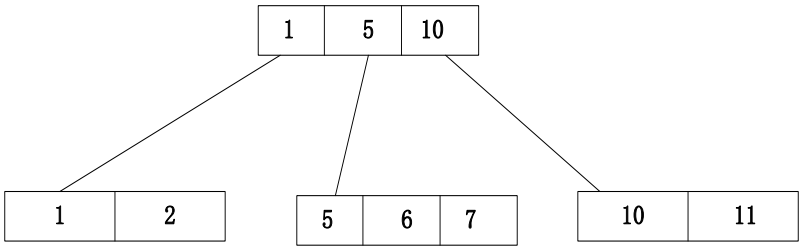
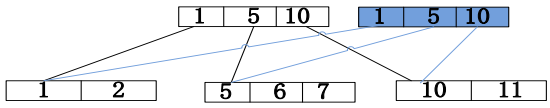


图 3.6 B+初始化

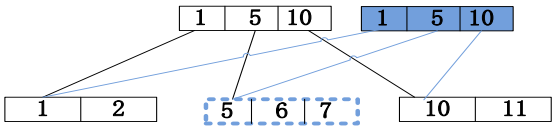
B+树（COW）的插入过程如下：

1. 复制 B+树最新的根节点
2. 找到要插入的节点从根节点到叶子节点的路径
3. 复制第 2 步中找到的节点上的所有节点
4. 进行插入（以及一些必要的分裂操作）
5. 把最新的根节点设为自己(原子操作)

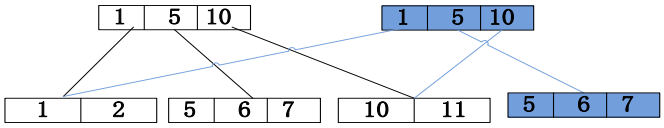
下面以对图 3.6 B+初始化所示的 B+树插入 8 为例说明，插入的过程示意图见图 3.7 节点插入过程-1 和图 3.8 节点插入过程-2。



1. 复制根节点

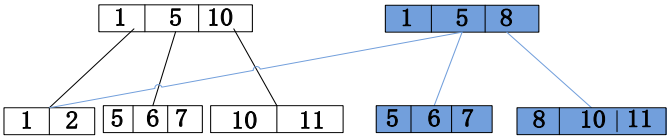


2. 找路径

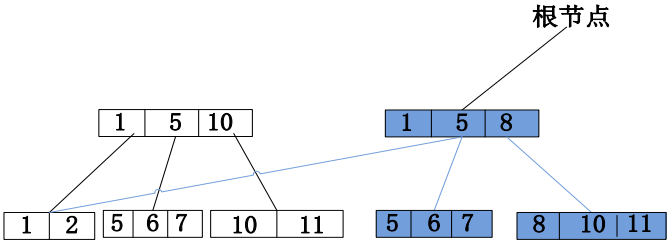


3. 复制路径节点

图 3.7 节点插入过程-1



4. 插入节点8



5. 更新根节点

图 3.8 节点插入过程-2

B+树（COW）的删除过程如下：

1. 复制 B+树最新的根节点
2. 找到要插入的节点从根节点到叶子节点的路径
3. 复制第 2 步中找到的节点上的所有节点
4. 进行删除（以及一些必要的分裂操作）
5. 把最新的根节点设为自己(原子操作)

下面以对图3.6 B+初始化所示的B+树删除节点7进行说明，删除过程见图3.9 节点删除过程-1 和图 3.10 节点删除过程-2。

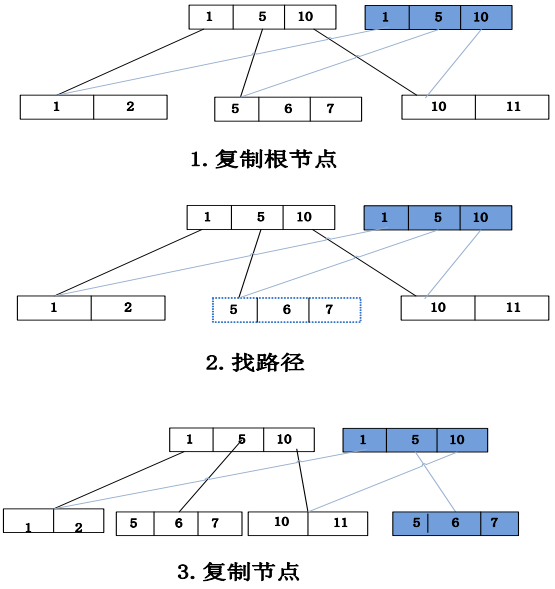


图 3.9 节点删除过程-1

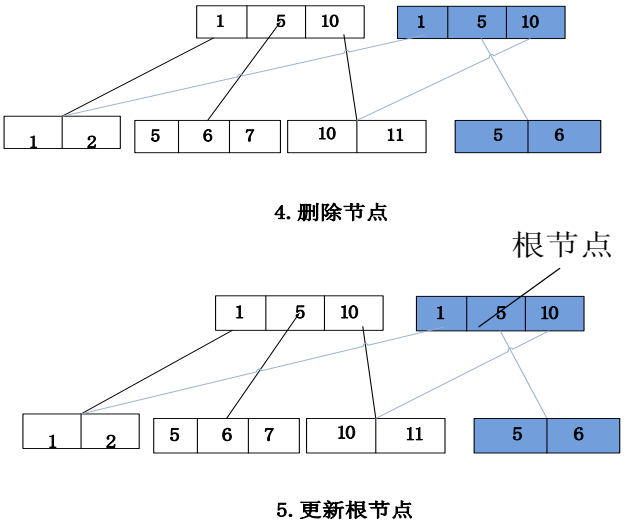


图 3.10 节点删除过程-2

由于实现 copy on write 的时候需要对现有节点进行拷贝，在操作完成以后需要将被拷贝的节点空间进行释放。为了实现节点的高效管理（如被拷贝节点空间自动释放），有人提出了对节点加上引用计数进行管理（包括叶子节点和非叶子节点）[28, 34]，但是这种做法实现比较复杂。

RaccoonFS 在实现的时候借鉴了这种思想，只需对 B+树的根节点进行引用计数，降低了实现的复杂度，在对根节点加引用计数的同时，RaccoonFS 还引入了复制节点队列来实现对被拷贝节点的自动释放，实现过程如图 3.11 引用计数实现过程-1 和图 3.12 引用计数实现过程-2 所示：

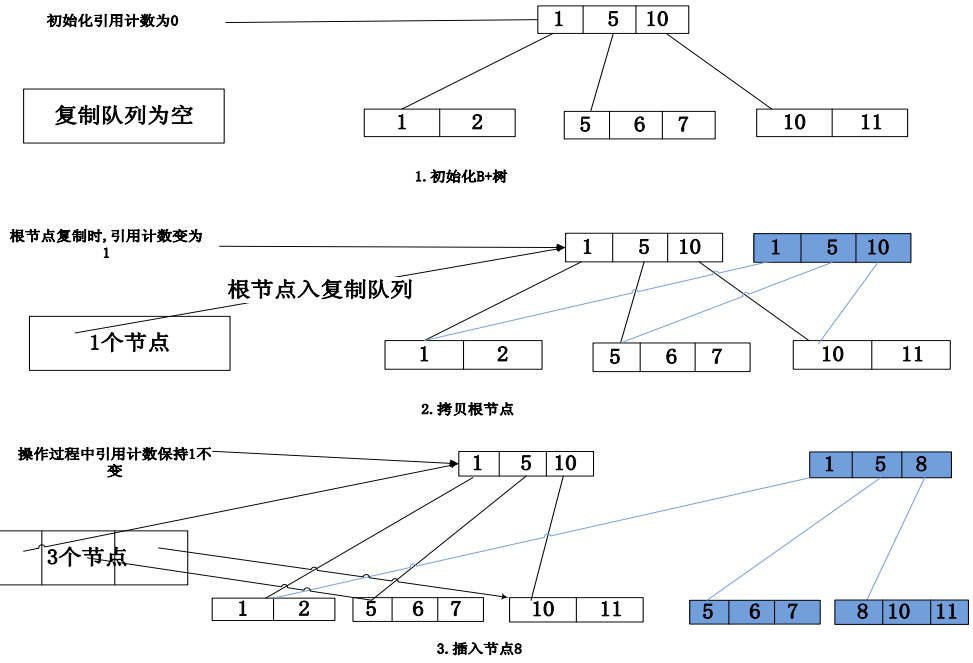


图 3.11 引用计数实现过程-1

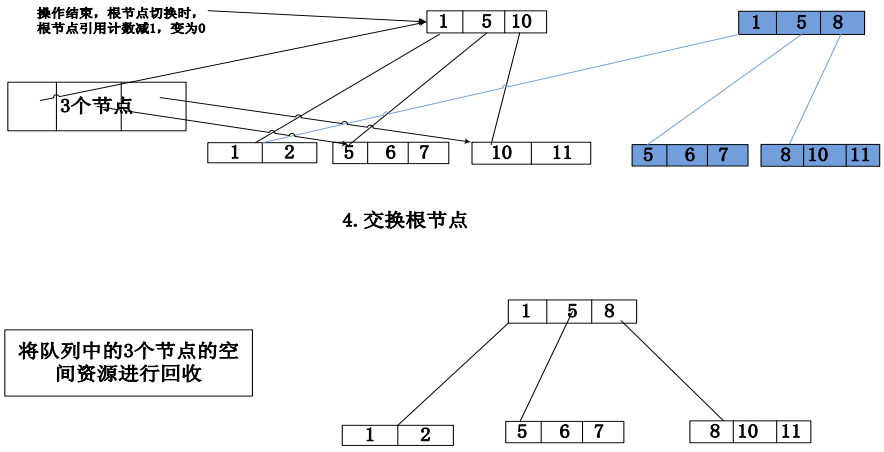


图 3.12 引用计数实现过程-2

3.5 RaccoonFS 多版本并发控制

RaccoonFS 通过写时拷贝解决了读写竞争照成的错误，那么对于写写操作出现竞争如何解决。

RaccoonFS 使用了多版本并发控制（MVCC）[35-39]，和传统的两阶段锁（2PL）[40]相比，它既保证了读写、写写互不阻塞，又保证了多用户并发操作情况下数据的一致性。

一般来说，解决写写并发一般有基于锁和基于多版本并发控制(MVCC)2 种方法。但是基于锁的方案需要对整个名字空间上锁，写写之间在临界区相互阻塞，对性能影响很大，元数据访问延迟比较大。不同于基于锁的解决方案，多版本并发控制的解决方案(MVCC)写写之间可以在时间和空间上做到真正的并发，元数据访问延迟比较低，性能比较好。

RaccoonFS 为了对 B+[31]树中节点的多个版本进行有效管理，设计了版本链 VL(version list)的辅助结构。版本链主要由时间戳，版本指针项，以及连接版本指针项的双向链表组成，在版本链中版本指针项根据时间戳由小到大排列。

对于一个读任务或者写任务 T 用一个全局时间戳来记录它的创建时间，读任务在任务开始的时候记录时间戳，写任务在 commit 的时候记录时间戳。如果数据链表只有一项那么直接指向数据，如果有多项的话那么指向版本链。结构如图 3.13 多版本并发控制所示。

从图 3.13 多版本并发控制中我们可以看到，如果一个数据的版本项过多而且长期不回收，会浪费大量的空间。所以在读事务不需要相关版本的时候，需要回收相关版本项，以减轻系统的负载。满足下面条件的版本项可以认为是可以回收的：如果系统中所有的读事务的时间戳大于或者等于所要检查的版本的时间戳，那么这个版本是可以回收的。

有多版本并发控制同时有写时拷贝的 B+树的读，插入及修改，删除流程如下：

读流程：

首先从名字空间的根节点开始查找，当查找的节点索引节点时（非叶子节点），所要查找的下一级节点 P 满足下列条件：Ki<k<Kj，在 Ki 和 Kj 之间的 P 节点就是我

们要查找的下一级的节点，k 在这里表示要查找的值。如果 k 高于节点最右端的值表示我们要查找的值不存在，直接退出，否者可以一直按照上面的流程一直找到叶子节点对应的键值对。键值对应的索引指向数据项或者指向版本链，如果是数据项那么查询终止，直接返回结果；如果是版本链，那么就根据读事务在启动时获得的时间戳在版本链中找到对应的数据项。

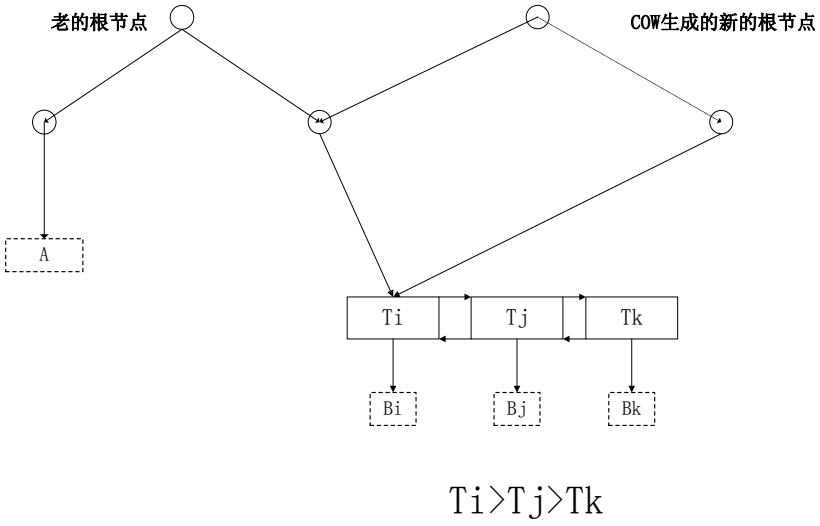


图 3.13 多版本并发控制

对于读流程，不论是在数据项中查找还是在版本链中查找，都不需要上锁，因此查询的性能和查询事务的吞吐量都很高。同时B+树对于范围查询支持的也非常好，因为所有相邻的节点都聚集在一起。

插入及修改流程：

插入流程需要首先查询数据在 B+树中的位置，这个在前面读流程中已经介绍过了，但是和查询流程不同的是，我们需要对查询路径上的节点做一次拷贝（写时拷贝 COW）。由于插入过程是新插入一个数据，不存在历史版本，所以不需要新建版本链。

对于修改流程第一步和上面的插入流程类似，如果键值对指向数据项，则需要新建版本链，同时新建 2 个版本项，第一个版本项指向历史数据，第二个版本项指向修改后的数据，版本与版本之间用双向链表进行连接。同时将老版本和新版本的

指针都指向版本链。

删除流程：

当我们进行一个删除流程，在流程结束后，所有在这个流程以后的数据都是看不见这个数据了，但是之前的流程要看到正确版本的数据。为了保证上面这一点，我们不能直接删除版本链或者对应的数据项。在这个时候，我们需要新建一个特别的版本项(带有删除标记)，其指向的数据为空，并将它插入到版本链中。

版本链空间回收：

当一个版本项不在需要时，我们需要回收它的空间，将版本项以及它所对应的数据都从系统中删除。我们可以运行一个优先级较低的后台线程，定期的扫描所有的版本项，然后将不需要的版本项从系统中回收。当版本链中只剩下最后一个版本项时，我们需要回收版本链，并将键值直接指向数据项。

通过上面的操作流程可见多版本并发控制(MVCC)避免了大粒度和长时间的锁定，更好地适应对读的响应速度和并发性要求高的场景。

3.6 本章小结

本章详细介绍了 RaccoonFS 系统中名字空间管理方法的详细设计与实现，介绍了系统架构设计、名字空间的数据组织、写时拷贝和多版本并发控制的具体实现，包括如何用 B+树来支持目录操作，如何利用写时拷贝（COW）来规避上锁带来的性能开销，如何利用多版本并发控制（MVCC）来实现写写并发，如何利用引用计数来实现拷贝空间的自动回收。

4 系统测试与分析

现在的系统无论是单机还是分布式，都会尽可能追求高性能。对于不同的系统，不同的应用服务，关注的性能也大不相同。常见的性能指标有：系统的吞吐能力，指系统在某一时间可以处理的数据总量，通常可以用系统每秒处理的总的数据量来衡量；系统的响应延迟，指系统完成某一功能需要使用的时间；系统的并发能力，指系统可以同时完成某一功能的能力，通常也用 QPS(Query Per Second)来衡量。不同于 IO 密集型的数据服务器（主要指标是系统的吞吐能力），分布式文件系统法的元数据服务器是计算密集型的，目标是追求低延迟，高响应，所以一般来说分布式文件系统元数据服务器的 QPS 越高表示分布式文件系统元数据服务器的性能越优越。所以 RaccoonFS 测试主要的关注点是 QPS。

4.1 测试环境

测试系统的硬件配置如下：

CPU：Intel(R) Xeon(R) CPU 2.27GHz * 8

内存：DDR16GB

硬盘：SATA 500GB

网卡：1000 Network Connection * 2

操作系统：Redhat 5.4 2.6.18-164.el5 x86_64 GNU/Linux

测试系统由 120 个客户端(6 台实体机，每台机器开 20 个进程虚拟出 120 台客户端)，1 台元数据服务器，6 台数据服务器组成，系统的拓扑结构如图 4.1 测试集群网络拓扑所示：

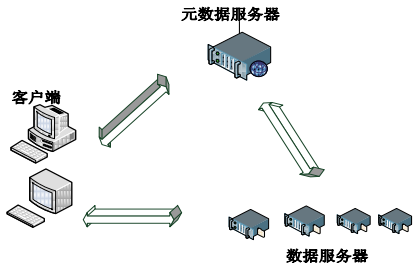


图 4.1 测试集群网络拓扑

4.2 测试过程及结果

分布式文件系统的元数据服务器是典型的计算密集型的应用，它追求的目标是低延迟，高响应，所以我们重点关注 RaccoonFS 的 QPS。为了说明 RaccoonFS 在使用了新的名字空间实现方法(具有写时拷贝以及多版本并发控制的 B+树)后，元数据服务器有了更好的服务性能，我们选取了 3 组测试对象：HDFS 名字空间管理方法、实现了 COW 的 B+树名字空间管理方法、实现了 COW 和 MVCC 的 B+树名字空间管理方法，分别对这 3 组测试对象进行读、写、混合读写压力测试，并测试在同等压力下这 3 组对象的 QPS。通过测试结果来说明，哪一种名字空间管理方法在高并发的情况下具有更好的响应速度。

分布式文件系统元数据服务器端的操作包括读操作和写操作，为了使测试结果具有说服力，我们从 3 个方面进行了测试：读并发响应速度测试、写并发响应速度测试、读写混合响应速度测试。

4.2.1 读响应速度测试

在读操作方面，RaccoonFS 和 HDFS 相似，我们选取部分客户端对元数据服务器的操作进行说明，具体内容如表 4.1 读函数列表：

表 4.1 读函数列表

接口函数	返回值	参数说明
vector<string> GetListing(const string& dirname);	vector非空，得到目录下的文件信息； 为空，操作失败，或该目录为空	dirname：要列出内容的目录路径 对每个需要列出目录项的文件，向NameServer请求数据，并返回给用户。
int Open(const string& filename， int oflag = O_RDONLY)	非负，操作成功；负数，操作失败 操作成功时，返回值即为需要获取的文件ID	filename：要打开的文件路径 oflag：打开方式，暂只支持O_RDONLY（0，只读打开）对于每个需要读取的文件，通过读取缓存(若未命中调用对NameServer交互的getBlockLocations接口，更新缓存)，取文件ID。
FileInfo GetFileInfo(const string& filename);	FineInfo 中 file_id 大于0，操作成功； 小于等于0，操作失败	filename：要得到属性的文件路径 对每个需要读取的文件，向NameServer请求获取其信息。

我们选取里面比较典型的操作 GetFileInfo 进行对比测试，测试的具体步骤如下：

1. 开启一台元数据服务器和 6 台数据服务器，元数据服务器和数据服务器位于同一个局域网，机器的配置见测试环境。

2. 用 10 个客户端连接元数据服务器，分别进行 1000 次 GetFileInfo 操作，记录下所需要的时间，那么 QPS 可以表示为：QPS=(操作次数*客户端数目/所需要的时间)

3. 按照 20，40，60，80，100 的顺序增加客户端的次数，让每个客户端重复进行第二步的操作，记录下完成整个操作所需要的时间，那么 QPS 可以表示为：QPS=(操作次数*客户端数目)/(所需要的时间)

RaccoonFS 和 HDFS 读响应速度测试的伪代码如下：

```
//读响应速度伪代码
int    start_time = now();
latch = new countdownlatch(n) //这里的n是线程的数目
For(int i = 0; i < n; i++)
{
        New Thread(thread_fun,  ops_num,  latch).start();//这里的ops_num是每个客户端每次操作的次数
}
latch.wait_done();
int end_time = now();
QPS = (ops_num * n) / (end_time – start_time)

////////////////下面是具体的函数实现部分//////////////

void thread_func(ops_num,  latch)
{
        for(int i = 0; i < ops_num; i++)
        {
                do_op();//对应的具体的读操作函数
        }
        latch.count_down();
}
```

对 HDFS 名字空间管理方法、实现了 COW 的 B+树名字空间管理方法、实现了 COW 和 MVCC 的名字空间管理方法的读速度测试结果如图 4.2 读 QPS 测试结果所示

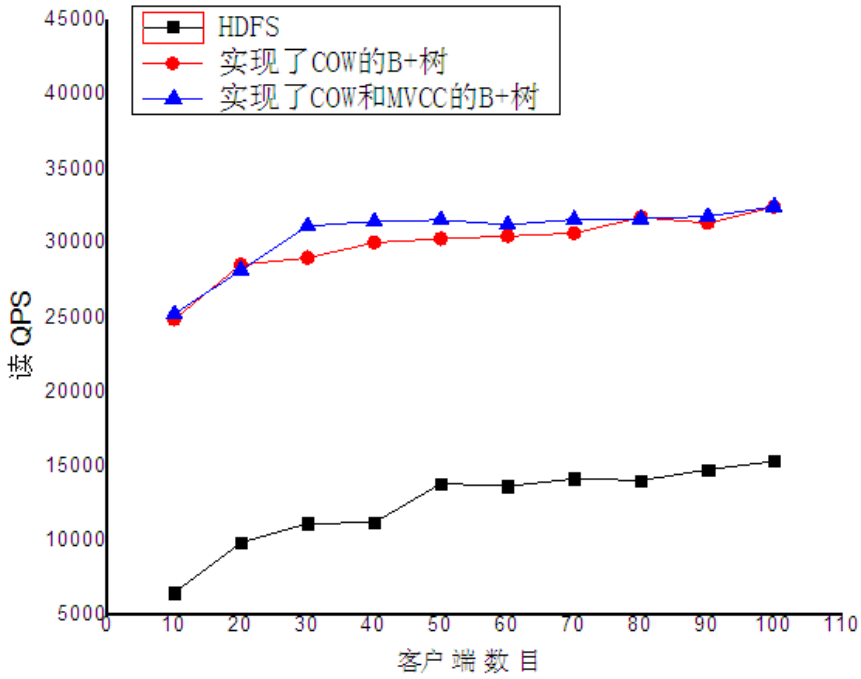


图 4.2 读 QPS 测试结果

从图 4.2 读 QPS 测试结果可以看出实现了 COW 的 B+树名字空间管理方法和实现了 COW 和 MVCC 的名字空间管理方法读的 QPS 大致相同，这是因为 2 者的不同之处在于后者实现了 MVCC，而 MVCC 主要是用来优化写，提高写写并发，所以读的 QPS 大致相同。从图中还可以看出实现了 COW 的 B+树名字空间管理方法和实现了 COW 和 MVCC 的 B+树名字空间管理方法的读 QPS 要高于 HDFS 的读的 QPS，这是因为前两者为了提高读的响应速度采用了多线程进行处理，而且前 2 者是使用 C++实现，在网络库上和读处理流程上做了大量优化。

4.2.2 写响应速度测试

在写操作方面，RaccoonFS 与 HDFS 相似，我们选取部分客户端对元数据服务器的操作进行说明，具体内容如表 4.2 写函数列表。

表 4.2 写函数列表

接口函数	返回值	参数说明
int Create(const string& filename， bool oflag = false，int replication);	非负，操作成功；负数，操作失败 操作成功时，返回值即为文件ID	filename：要打开的文件路径 oflag：创建方式，创建普通写文件或安全写文件 replication：设置副本数 对于每个写入的文件，向NameServer发送消息，得到新创建文件的ID，更新缓存。
int Delete(const string& filename);	0，操作成功； 非0，操作失败	filename：需要删除的文件路径 对每个需要删除的文件，向NameServer交互执行操作。
LocatedBlock AddBlock(int64_t file_id)	LocatedBlock中获取的DS数目大于0，操作成功；否则，操作失败	File_id:需要添加Block的文件的ID 在执行写入操作时，向NameServer添加一个Block。

我们选取里面比较典型的操作 create 进行对比测试，测试的具体步骤如下:

1. 开启一台元数据服务器和 6 台数据服务器，元数据服务器和数据服务器位于同一个局域网，机器的配置见测试环境。

2. 用 10 个客户端连接元数据服务器，分别进行 1000 次 create 操作，记录下所需要的时间，那么 QPS 可以表示为：QPS=(操作次数*客户端数目/所需要的时间)

3. 按照 20，40，60，80，100 的顺序增加客户端的次数，让每个客户端重复进行第二步的操作，记录下完成整个操作所需要的时间，那么 QPS 可以表示为：QPS=(操作次数*客户端数目)/(所需要的时间)

RaccoonFS 和 HDFS 写响应速度测试的伪代码如下：

```
//写响应速度伪代码
int    start_time = now();
latch = new countdownlatch(n) //这里的n是线程的数目
For(int i = 0; i < n; i++)
{
        New Thread(thread_fun， ops_num， latch).start();//这里的ops_num是每个客户端每次操作的次数
}
latch.wait_done();
```

```
int end_time = now();
QPS = (ops_num * n) / (end_time – start_time)

//////////////下面是具体的函数实现部分/////////////

void thread_func(ops_num，  latch)
{
        for(int i = 0; i < ops_num; i++)
        {
                do_op();//对应的具体的写操作函数
        }
        latch.count_down();
}
```

对 HDFS 名字空间管理方法、实现了 COW 的 B+树名字空间管理方法、实现了 COW 和 MVCC 的名字空间管理方法的写速度测试结果如图 4.3 写 QPS 测试结果所示。

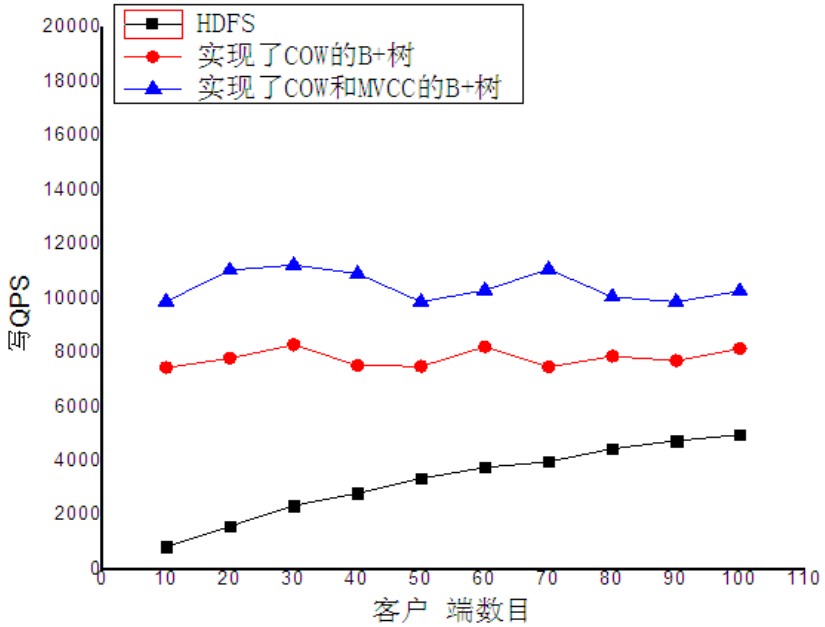


图 4.3 写 QPS 测试结果

从图 4.3 写 QPS 测试结果可以看出实现了 COW 和 MVCC 的 B+树名字空间管理方法的写的 QPS 要高于实现了 COW 的 B+名字空间管理方法，这是因为前者在写操作上面使用了“乐观锁”的 MVCC 操作，从而使得写写可以并发。而实现了 COW 的 B+名字空间管理方法的写 QPS 比 HDFS 高出不了多少，这是因为 2 者机制相同，

都使用了“悲观锁”，写写并不能并发，操作流程也大致相同。

4.2.3 读写混合响应速度测试

我们选取 4.2.1 里面的 GetFileInfo 读操作和 4.2.2 里面的 create 写操作进行读写混合测试，测试的具体步骤如下:

1. 开启一台元数据服务器和 6 台数据服务器，元数据服务器和数据服务器位于同一个局域网，机器的配置见测试环境。

2. 用十个客户端连接元数据服务器，按照 9:1 的比例分配读写客户端(这里选用 9:1 的比例是因为研究表明在分布式文件系统中读写比例是 9:1)，即其中的 9 个客户端进行 1000 次 GetFileInfo 操作，其中的 1 个客户端进行 create 操作，记录下所需要的时间，那么 QPS 可以表示为：QPS=(操作次数*客户端数目/所需要的时间)

3. 按照 20，40，60，80，100 的顺序增加客户端的次数，按照 9:1 的比例分配读写客户端重复进行第二步的操作，记录下完成整个操作所需要的时间，那么 QPS 可以表示为：QPS=(操作次数*客户端数目)/(所需要的时间)

RaccoonFS 和 HDFS 读写混合响应速度测试的伪代码如下：

```
//读写混合响应速度伪代码
int    start_time = now();
latch = new countdownlatch(n) //这里的n是线程总的数目
readthread_num = n * percent //percent是读写混合比例
writethread_num = n – readthread_num
For(int i = 0; i < readthread_num; i++)
{
        new Thread(read_thread_fun，  ops_num，  latch).start();//这里的ops_num
是每个客户端每次操作的次数
}
For(int i = 0; i < writethread_num; i++)
{
        new Thread(write_thread_fun，  ops_num，  latch).start();//这里的ops_num
是每个客户端每次操作的次数
}
latch.wait_done();
int end_time = now();
QPS = (ops_num * n) / (end_time – start_time)
```

```
//////////////下面是具体的函数实现部分/////////////
void read_thread_func(ops_num，  latch)
{
        for(int i = 0; i < ops_num; i++)
        {
                do_op();//对应的具体的读操作函数
        }
        latch.count_down();
}

void write_thread_func(ops_num，  latch)
{
        for(int i = 0; i < ops_num; i++)
        {
                do_op();//对应的具体的写操作函数
        }
        latch.count_down();
}
```

对 HDFS 名字空间管理方法、实现了 COW 的 B+树名字空间管理方法、实现了 COW 和 MVCC 的名字空间管理方法的写速度测试结果如图 4.4 读写混合 QPS 测试结果所示

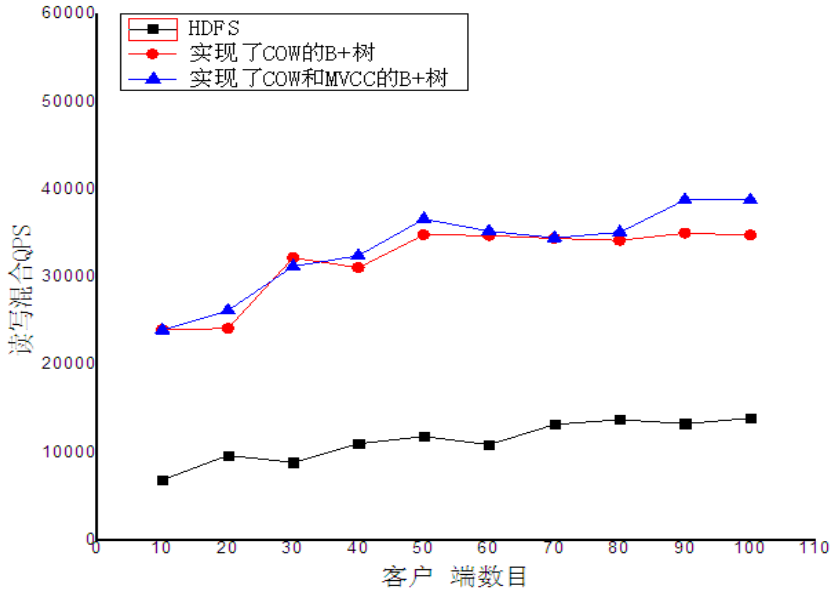


图 4.4 读写混合 QPS 测试结果

综合 4.2.1 和 4.2.2 的测试结果，从图 4.4 读写混合 QPS 测试结果可以看出实现了 COW 的 B+树名字空间管理方法和实现了 COW 和 MVCC 的名字空间管理方法的混合读写 QPS 基本是读 QPS 和写 QPS 的加和，而 HDFS 的读写混合 QPS 要元低于读 QPS 和写 QPS 的加和，这是因为前 2 者使用了 COW，从而达到了读写分离，读写互不影响，而 HDFS 的读写操作是排斥的，需要上排它锁。

4.3 本章小结

在上面的测试中选取了三组测试对象：HDFS(hadoop distribute file system)名字空间管理方法、实现了 COW 的 B+树名字空间管理方法、实现了 COW 和 MVCC 的 B+树名字空间管理方法。实验结果表明实现了 COW 的 B+树管理方法在读写的响应速度方面高于 HDFS 名字空间管理方法，实现了 COW 和 MVCC 的 B+树名字空间管理方法在写响应速度方面要高于只实现了 COW 的 B+树名字空间管理方法。测试表明，采用 COW 和 MVCC 的 B+名字空间管理方法，可以有效提高名字空间管理效率。

5 总结与展望

分布式文件系统元数据服务器在处理数以万计的客户端连接时，往往出现效率低下甚至完全瘫痪，怎么提升元数据服务器在高并发读写下的性能成为了一个非常重要的设计内容。论文主要解决了下面几个方面的任务：

1. 论文研究当前一些分布式文件系统元数据服务器（特别是名字空间部分）的解决方案，研究并解决了分布式文件系统名字空间的由于锁竞争带来响应速度低下的问题，并在这个方案的基础上实现了分布式文件系统 RaccoonFS。

2. RaccoonFS 采用了 B+树的方式来管理名字空间，同时 RaccoonFS 在 B+树上实现了写时拷贝(COW)和多版本并发控制(MVCC)技术，实现读写分离，以及写写并发，极大的提高了读写的性能，规避了上锁带来的性能开销。在读压力测试，写压力测试和读写混合压力测试中，性能均优于传统的名字空间管理方式（HDFS 的目录式管理）。

未来的工作有几个方面可以展望：

1. 为了增强元数据服务器的性能，网络库方面需要重新设计，可以采用 Leader-Follower 等各种经典的架构，在 epoll 处理读写请求这一块需要改为多线程。

2. RaccoonFS 在集群存储文件的规模达到 PB 级别的时候，元数据服务器也达到了 GB 的级别，如何降低 RaccoonFS 的元数据的量同时不降低元数据的管理效率也变得非常重要。名字空间的元数据量并不容易减少，我们可以对副本管理进行改进，可以用对照表来管理副本以达到大大减少副本的元数据量，从而减少整体的元数据量。

3. RaccoonFS 为了实现元数据服务器的高可用，使用了 hearbeat 的虚拟 IP 来进行主备选举，但是 hearbeat 对于脑裂等情况并不能解决，后续可以使用 zookeeper 开替换这一块。

4. RaccoonFS 的主备日志同步采用了 2 阶段锁，这种同步方式在有一台备机的时候有不错的效果，但是当备机的数目增长时系统的性能会急剧下降，后续可以考虑采用分布式消息队列来替换这一块。

致 谢

时间如白驹过隙，不知不觉两年半的时间已经过去。回首这两年半的研究生学习和生活，感受颇多，有苦有甜，有喜有悲，但也收获满满。这两年半的研究生学习给我指明了以后学习的方向，为我以后的工作打下了坚实的专业基础，更为重要的是培养了我独立思考和学习的能力。

首先我要感谢我的导师曾老师，曾老师深厚的学术造诣、严谨的治学态度给了我很深的印象。记得大四时和大四暑假，曾老师经常为了科研工作忙到很晚，曾老师论文阅读之多，知识面之广，让我看到了一个真正的学者，这在接下来的 2 年半时间里深深的影响了我。曾老师在我学习上给了很多的指导，这让我少走了很多的弯路，真的很谢谢你。

感谢我的指导老师施老师，一想到施老师，首先想到的就是施老师爽朗的笑声。施老师是实验室最为耀眼的正能量，是我们这群学生的良师益友。记得大四时被施老师的学术研究和个人魅力深深吸引进入了实验室，在接下来的 3 年多时间里，施老师耐心指导我的学习，让我摸清了学习的方向，少走了不少弯路。特别是在毕业论文开题时，不论是在大的方向还是小的细节上，耐心的指导让我找到了前进的方向。你真的是我们这一辈子的朋友和老师。

感谢实验室的冯丹老师、王芳老师、李洁琼老师、谭支鹏老师、刘景玲老师、华宇老师等等各位老师。没有各位老师的辛苦工作与管理，实验室不会有如此优秀的学习和研究平台，实验室也不会有如此优秀的发展。

感谢 F309 共同学习和做项目的姚莹莹、王敬轩和叶为民同学，一起合作，一起做项目的日子很愉快；感谢已经毕业的贺燕、赵恒、刘小军、郑颖、王霁欣，感谢你们在我刚来实验室时的指导；感谢柳青、李勇、李宁、焦田峰博士的在学习上给的指导和帮助；感谢王艳萍、付宁、桂莅、赵千、方波同学，虽然相处只有 1 年多的时间，但是很开心；感谢李白、黄立、韩江、张成文、郭鹏飞同学；因为有了大家，我们 F309 才是一个完整的集体，可以一起面对生活和学习上的困难，一起享受问题解决后的喜悦。

最后要感谢我的家人，我的父母，我的姐姐和弟弟，感谢你们在背后给我莫大的支持！感谢我的女朋友，感谢这么多年来你对我信任和支持，从本科走到现在，我真的很幸福，感谢有你。

参考文献

[1] Shvachko K V. HDFS Scalability: The limits to growth. Usenix, 2010, 35(2): 6-16

[2] Pawlowski B, Juszczak C, Staubach P, et al. NFS Version 3 design and implementation. in Proceedings of the Summer 1994 USENIX Conference, Berkeley, CA, USA: USENIX Assoc, 1994: 137-152

[3] Ghemawat S, Gobioff H, Leung S. The Google file system. in Proceedings of the nineteenth ACM symposium on Operating systems principles, Bolton Landing, NY, USA: ACM, 2003: 29-43

[4] 余庆. 分布式文件系统 FastDFS 架构剖析. 程序员, 2010(011): 63-65

[5] 刘立坤, 武永卫, 徐鹏志等. CorsairFS: 一种面向校园网的分布式文件系统. 西安交通大学学报, 2009, 43(008): 43-47

[6] Weil S A, Brandt S A, Miller E L, et al. Ceph: a scalable, high-performance distributed file system. in Proceedings of the 7th USENIX Symposium on Operating Systems Design and Implementation (OSDI'06), Berkeley, CA, USA: USENIX Assoc, 2006: 307-320

[7] 黄华, 张建刚, 许鲁. 蓝鲸分布式文件系统的分布式分层资源管理模型. 计算机研究与发展, 2005, 42(6): 1034-1038

[8] 杨德志, 黄华, 张建刚等. 大容量, 高性能, 高扩展能力的蓝鲸分布式文件系统. 计算机研究与发展, 2005, 42(6): 1028-1033

[9] Lai H, Wang P. P2P traffic control based on ternary content addressable memory. Computer Engineering, 2010, 36(9): 120-122

[10] Kempe D, Dobra A, Gehrke J. Gossip-based computation of aggregate information. in Proceedings 44th Annual IEEE Symposium on Foundations of Computer Science - FOCS 2003, Los Alamitos, CA, USA: IEEE Computer. Soc, 2003: 482-491

[11] Kubiatowicz J, Bindel D, Chen Y, et al. Oceanstore: An architecture for global-scale persistent storage. ACM Sigplan Notices, 2000, 35(11): 190-201

[12] Weimin Z, Jinfeng H, Ming L. Granary: architecture of object oriented Internet storage service. in Proceedings of the IEEE International Conference on

E-Commerce Technology for Dynamic E-Business, Los Alamitos, CA, USA: IEEE Comput. Soc, 2004: 294-297

[13] Weil S A, Brandt S A, Miller E L, et al. Ceph: a scalable, high-performance distributed file system. in Proceedings of the 7th USENIX Symposium on Operating Systems Design and Implementation (OSDI'06), 6-8 Nov. 2006, Berkeley, CA, USA: USENIX Assoc, 2006: 307-320

[14] Shvachko K, Hairong K, Radia S, et al. The Hadoop Distributed File System. in 2010 IEEE 26th Symposium on Mass Storage Systems and Technologies (MSST 2010), Piscataway, NJ, USA: IEEE, 2010: 10

[15] Singh H J, Singh V P. High Scalability of HDFS using Distributed Namespace. International Journal of Computer Applications, 2012, 52(17): 30-37

[16] Liu X, Tu C. Distributed metadata management scheme in HDFS. Advances in Information Sciences and Service Sciences, 2012, 4(22): 456-463

[17] Chandrasekar A, Chandrasekar K, Ramasatagopan H, et al. Classification Based Metadata Management for HDFS. in 2012 IEEE 14th International Conference on High Performance Computing and Communication (HPCC) and 2012 IEEE 9th International Conference on Embedded Software and Systems (ICESS), Los Alamitos, CA, USA: IEEE Computer Society, 2012: 1021-1026

[18] Redesign the dfs namespace datastructures to be copy on write https: //issues. apache. org/jira/browse/HADOOP-334. 2010

[19] Support for RW/RO snapshots in HDFS https: //issues. apache. org/jira/browse/HDFS-2802. 2012

[20] Morris R, Kaashoek M F, Karger D, et al. Chord: A scalable peer-to-peer look-up protocol for internet applications. IEEE/ACM TRANSACTIONS ON NETWORKING, 2003, 11(1): 1-13

[21] El-Ansary S, Alima L O, Brand P, et al. Efficient broadcast in structured P2P networks. in Peer-to-Peer Systems II. Second International Workshop, IPTPS 2003. Revised Papers (Lecture Notes in Comput. Sci. Vol. 2735), Berlin, Germany: Springer-Verlag, 2003: 304-314

[22] Martins V, Pacitti E. Dynamic and distributed reconciliation in P2P-DHT networks. in Euro-Par 2006 Parallel Processing. 12th International Euro-Par Conference.

Proceedings (Lecture Notes in Computer Science Vol. 4128), Berlin, Germany: Springer-Verlag, 2006: 337-349

[23] Karger D, Sherman A, Berkheimer A, et al. Web caching with consistent hashing. Computer Networks, 1999, 31(11): 1203-1213

[24] Decandia G, Hastorun D, Jampani M, et al. Dynamo: Amazon's highly available key-value store. Operating Systems Review, 2007, 41(6): 205-220

[25] Beaver D, Kumar S, Li H C, et al. Finding a needle in Haystack: facebook's photo storage. in Proceedings of the 9th USENIX conference on Operating systems design and implementation, Vancouver, BC, Canada: USENIX Association, 2010: 1-8

[26] Liu D, Deters R. The Reverse C10K Problem for Server-Side Mashups. in Service-Oriented Computing–ICSOC 2008 Workshops, Springer, 2009: 166-177

[27] Toptsis A A. B**-tree: a data organization method for high storage utilization. in Proceedings of ICCI'93: 5th International Conference on Computing and Information, Los Alamitos, CA, USA: IEEE Comput. Soc. Press, 1993: 277-281

[28] Rodeh O. B-trees, shadowing, and clones. Trans. Storage, 2008, 3(4): 1-27

[29] Braginsky A, Petrank E. A lock-free B+tree. in Proceedinbgs of the 24th ACM symposium on Parallelism in algorithms and architectures, Pittsburgh, Pennsylvania, USA: ACM, 2012: 58-67

[30] Costa V S. COWL: Copy-On-Write for logic programs. in Proceedings 13th International Parallel Processing Symposium and 10th Symposium on Parallel and Distributed Processing. IPPS/SPDP 1999, Los Alamitos, CA, USA: IEEE Comput. Soc, 1999: 720-727

[31] Wu S, Jiang D, Ooi B C, et al. Efficient b-tree based indexing for cloud data processing. Proceedings of the VLDB Endowment, 2010, 3(1-2): 1207-1218

[32] Graefe G. Modern B-Tree Techniques. Foundations and Trends in Databases, 2010, 3(4): 203-402

[33] Shrira L, Hao X. Thresher: an efficient storage manager for copy-on-write snapshots. in Proceedings of the 2006 USENIX Annual Technical Conference, Berkeley, CA, USA: USENIX Association, 2006: 57-70

[34] Nelson M, Ousterhout J. Copy-on-write for Sprite [operating system]. in Proceedings of the Summer 1988 USENIX Conference, Berkeley, CA, USA:

USENIX Assoc, 1988: 187-201

[35] Bernstein P A, Goodman N. Multiversion concurrency control—theory and algorithms. ACM Transactions on Database Systems (TODS), 1983, 8(4): 465-483

[36] Wang D, Li M. Research and realization of nesting transaction based on MVCC. Jisuanji Gongcheng/Computer Engineering, 2005, 31(3): 88-89

[37] Liu T, Zhu H, Chang G, et al. The design and implementation of zero-copy for Linux. in 2008 Eighth International Conference on Intelligent Systems Design and Applications, Piscataway, NJ, USA: IEEE, 2008: 121-126

[38] Carey M J, Muhanna W A. The performance of multiversion concurrency control algorithms. ACM Transactions on Computer Systems (TOCS), 1986, 4(4): 338-378

[39] Seifert A, Scholl M H. Processing read-only transactions in hybrid data delivery environments with consistency and currency guarantees. Mobile Networks and Applications, 2003, 8(4): 327-342

[40] Son S H, David R. Design and analysis of a secure two-phase locking protocol. in Proceedings of the 18th Annual International Computer Software & Applications Conference (COMPSAC 94), Taipei, Taiwan: IEEE, 1994: 374-379