

基于 LSM-Tree 结构和 Raft 算法的分布式存储系统

摘 要

随着互联网的发展，网络用户的激增导致互联网服务提供公司需要存储极大规模的数据，而对于一些复杂场景下的数据复制以及分布式系统下数据库的可靠性仍然是一个巨大的挑战。传统的互联网架构时代，单机数据库如 MySQL，Oracle 占领很大的市场份额，而单机数据库有很多缺点。成本高：它们通常比分布式数据库系统更加复杂和成本更高，因为它们需要专门的硬件、存储设备和管理软件。维护困难：由于它们是单独的系统，因此一旦出现问题，修复它们可能需要很长时间和高昂的成本。不易扩展：当需要增加新功能或存储容量时，单机数据库系统可能无法轻松地扩展。数据共享和备份困难：由于它们是单独的系统，数据不能轻松地在它们之间共享或备份。安全性差：由于它们是单独的系统，数据很容易受到黑客攻击和数据泄露。而分布式的数据存储恰如其分地解决了单机数据库的性能瓶颈问题和单机数据存储在实现面向用户系统时的各种痛点。

由于一个可靠的分布式存储系统设计复杂、挑战较大，本文致力于系统的存储策略，数据压缩方法，日志复制同步过程，Raft 算法的可达性分析。本文利用 Golang 编程语言的天然并发的特性，来开发 Raft 共识算法和 LSM-Tree 数据结构以实现分布式数据存储系统。本文的系统通过跨节点集群复制数据并使用 Raft 共识算法确保副本之间的一致性来提供容错性、可扩展性和高可用性。LSM-tree 数据结构通过优化磁盘访问和减少随机查找次数来实现高效的读写。通过精读论文原文，阅读参考 Google 开源 C++ 版本的 leveldb 实现的 LSM-Tree 本文实现了分布式存储系统 RAft Distributed Data Storage，简称为 Radds。本文的评估表明，本文的 Radds 系统在保持强一致性的同时实现了高性能和可扩展性，支持跨多个平台的客户端和多种语言的 API。

关键词： 日志结构归并树，Raft 共识性算法，键值型数据存储，分布式系统

Distributed storage system based on LSM-Tree structure and Raft algorithm

Abstract

With the development of the Internet, the surge of network users has led to the need for Internet service providers to store extremely large-scale data. However, data replication in some complex scenarios and the reliability of databases in distributed systems are still a huge challenge. In the era of traditional Internet architecture, stand-alone databases such as MySQL and Oracle occupy a large market share, but stand-alone databases have many shortcomings. High cost: They are usually more complex and costly than distributed database systems because they require specialized hardware, storage devices, and management software. Difficult to maintain: Since they are separate systems, it can take a long time and be costly to fix if something goes wrong. Not easy to expand: When new functions or storage capacity need to be added, a stand-alone database system may not be easily expanded. Difficulty in data sharing and backup: Since they are separate systems, data cannot be easily shared or backed up between them. Poor security: Since they are separate systems, the data is vulnerable to hacking and data breaches. Distributed data storage properly solves the performance bottleneck of stand-alone databases and various pain points of stand-alone data storage in implementing user-oriented systems.

Since the design of a reliable distributed storage system is complex and challenging, this paper focuses on the system's storage strategy, data compression method, log replication synchronization process, and the reachability analysis of the Raft algorithm. This paper uses the natural concurrency of the Golang programming language to develop the Raft consensus algorithm and the LSM-Tree data structure to implement a distributed data storage system. Our system provides fault tolerance, scalability, and high availability by replicating data across a cluster of nodes and using the Raft consensus algorithm to ensure consistency between replicas. The LSM-tree data structure enables efficient reads and writes by optimizing disk access and reducing the number of random lookups. Through intensive reading of the original text of the paper, read and refer to the LSM-Tree implemented by Google's open source C++ version of leveldb. This paper implements the distributed storage system RAft Distributed Data Storage,

referred to as Radds. Evaluations in this paper show that our Radds system achieves high performance and scalability while maintaining strong consistency, supporting clients across multiple platforms and APIs in multiple languages.

Keywords: Log-Structured Merge-Tree, Raft consensus algorithm, Key-Value Data Storage, Distributed System

目录

1	绪论	1
1.1	课题的背景和意义	1
1.2	分布式存储系统的发展状况	1
1.3	课题研究的主要方法及内容	2
1.4	论文组织结构	3
2	相关背景知识介绍	4
2.1	开发工具和环境	4
2.2	LSM-Tree 存储结构	5
2.3	Raft 共识性算法	6
2.4	涉及的开源库	6
3	Radds 存储系统总体设计	8
3.1	存储系统架构设计	8
3.2	存储层功能总体设计	10
3.2.1	存储引擎设计概述	10
3.2.2	内存可变结构 memtable 总体设计	11
3.2.3	内存不可变结构 immutable memtable 总体设计	11
3.2.4	日志文件结构 journal 总体设计	11
3.2.5	磁盘持久化结构 sstable 总体设计	12
3.2.6	文件元数据 manifest 总体设计	12
3.2.7	版本号 current 总体设计	12
3.3	共识层功能总体设计	13
4	Radds 存储系统详细设计与实现	15
4.1	存储层详细设计与实现	15
4.1.1	日志系统的实现	15
4.1.2	写数据的实现	18

4.1.3	读数据的实现	22
4.1.4	跳表数据结构的实现	24
4.1.5	内存数据库的实现	26
4.1.6	持久化数据存储的实现	27
4.1.7	缓存系统的实现	41
4.1.8	布隆过滤器的实现	45
4.1.9	数据压缩系统的实现	51
4.1.10	版本控制的实现	57
4.2	共识层详细设计与实现	62
4.2.1	共识层角色和状态的实现	62
4.2.2	共识层角色操作的实现	63
4.2.3	共识层多线程的实现	64
5	Radds 存储系统部署、日志分析与测试	66
5.1	存储系统部署	66
5.2	存储系统客户端测试	66
5.3	存储系统日志分析	67
	结论	70
	致 谢	71
	参考文献	72
	附 录	75
	附录 A	75
	附录 B	79

1 绪论

1.1 课题的背景和意义

分布式存储系统是一个重要且不断发展的领域，它涉及到许多学科和技术，如计算机科学、网络安全、数据库管理等。分布式存储系统的研究主要集中在三个方面：分布式存储系统的架构设计、底层协议的研究以及应用场景的拓展。其中，架构设计包括存储系统的硬件架构、软件框架等；底层协议的研究则包括各种存储接口、协议栈等；应用场景的拓展则涵盖了企业级、消费级、个人级等不同类型的存储系统。分布式存储系统的研究具有广泛的应用前景，它可以应用于企业级存储系统、云存储服务和分布式应用等方面。在企业级存储系统中，分布式存储系统可以用于存储大量的数据，提高数据存储的效率和可靠性。在云存储服务中，分布式存储系统可以用于存储云端的数据，提供高效的数据存储和管理服务。在分布式应用中，分布式存储系统可以用于存储和管理大量的数据，提高应用的可靠性和性能。在分布式存储系统的研究中，需要解决的关键问题包括数据一致性、数据持久性、数据备份和恢复等。在解决这些问题的过程中，需要采用一些新的技术和方法，如分布式数据库、分布式文件系统、分布式对象存储等。同时，需要考虑分布式存储系统的安全性和可靠性问题，采用一些新的安全技术和机制，如安全协议、数据加密、访问控制等，以保证分布式存储系统的安全性和可靠性。

1.2 分布式存储系统的发展状况

分布式存储系统的发展历史可以追溯到上世纪 90 年代，当时出现了一些基于局部存储器的分布式存储系统，如 Lustre 和 Xanadu 等。这些系统主要用于文件服务器等领域。随着网络技术的发展，基于网络的分布式存储系统出现了，如 Hadoop 和 HDFS 等。这些系统将数据存储分布在分布式节点上，并通过网络进行数据的访问和管理。近年来，随着云计算的发展，分布式存储系统开始广泛应用于云存储服务中。云存储服务将数据存储在云端，并通过互联网提供数据的访问和管理。这些系统通常采用分布式文件系统，如 AFS、HDFS 和 Gluster 等。随着大数据时代的到来，分布式存储系统的研究和应用也越来越广泛。基于分布式文件系统的分布式存储系统可以存储海量的数据，并支持高效的数据存储和管理。同时，基于块存储的分布式存储系统也得到了广泛的研究和应用，它可以实现数据的高效复制和同步，并支持大规模的数据存储和管理。总之，分布式存储系统的发展历史可以分为三个阶段：基于局部存储器的分布式存储系统、基于网络的

分布式存储系统和基于块存储的分布式存储系统。随着大数据时代的到来，分布式存储系统的研究和应用也将越来越广泛和深入。

1.3 课题研究的主要方法及内容

分布式存储系统是一种基于分布式数据库的存储方案，它将数据存储多个节点上，并通过一种分布式协议实现数据的同步和一致性。**Raft** 是一个用于分布式存储系统的协议，它可以保证数据的持久性和一致性，并支持高可用性和可扩展性。本文将介绍一种基于 **LSMTree** 和 **Raft** 的分布式存储系统的实现方案。系统架构本文提出的系统架构包括以下几个部分：存储层：存储层是分布式存储系统的核心，它负责将数据存储多个节点上，并实现数据的同步和一致性。存储层使用 **LSMTree** 数据结构来实现分布式存储，**LSMTree** 是一种高效的数据结构，可以在多个节点之间实现数据的高效复制和同步。数据层：数据层是分布式存储系统的基础，它负责将存储层中的数据提取出来，并提供数据的读写操作。数据层使用 **Raft** 协议来管理数据的持久性和一致性，**Raft** 协议可以保证数据的持久性和一致性。服务层：服务层是分布式存储系统的中间层，它负责提供数据的读写操作，并与数据层进行交互。服务层使用 **LSM** 树数据结构来实现分布式存储，**LSM** 树可以在多个节点之间实现数据的高效复制和同步。应用层：应用层是分布式存储系统的最底层，它负责与数据层进行交互，并实现数据的读写操作。应用层使用 **CLI** 命令行接口开发，并使用 **RPC** 通信方式来与存储层进行交互。系统实现方面，本文提出了以下几个关键点：数据分片：本文提出的系统采用 **LSMTree** 数据结构来实现数据的分布式存储，**LSMTree** 数据结构可以在多个节点之间实现数据的高效复制和同步。在数据分片时，本文将数据分为多个块，每个块都存储在一个节点上，并使用链接器将多个块链接起来。数据同步：在数据存储过程中，本文需要保证数据的一致性和持久性。为了实现数据的同步，本文采用 **Raft** 协议来管理数据的持久性和一致性。在 **Raft** 协议中，每个节点都有一个票数，每次写操作都会将票数增加，写操作完成后，票数会减少。当票数为 0 时，表示所有节点都认为数据已经同步，数据不再发生变化。数据一致性：为了保证数据的一致性，本文需要在写操作之前进行一致性校验。一致性校验可以通过多个节点之间的同步来实现，保证不同节点上的数据一致。在一致性校验中，本文将数据分为多个块，每个块都存储在一个节点上，并使用链接器将多个块链接起来。然后，每个节点都会对自己的数据进行一致性校验，并当写入数据到本地后，节点会将数据提交到主节点，由主节点完成数据的同步和一致性校验。最后，

数据会按照一定的顺序写入到主节点上，保证数据的一致性和完整性。数据持久化：为了保证数据的可靠性，本文需要将数据持久化到磁盘中。在数据持久化过程中，本文需要将数据的元数据（如数据类型、数据大小等）保存到磁盘中，并将数据的数据部分保存在内存中。当需要读取数据时，本文只需要读取内存中的数据即可。数据备份和恢复：在分布式存储系统中，数据备份和恢复是非常重要的。本文提出的系统支持数据备份和恢复，备份和恢复采用冷备份和热备份两种方式。冷备份是将数据备份到磁带或者光盘上，当磁带或者光盘损坏时，可以进行快速的恢复。热备份是将数据备份到内存中，当内存损坏时，可以进行快速的恢复。同时，本文提出的系统还支持数据的增量备份和增量恢复，可以在数据发生变化时，进行快速的恢复。系统性能在系统性能方面，本文提出的系统具有以下几个优点：高可用性：本文提出的系统支持高可用性，当主节点故障时，可以自动切换到备用节点，保证系统的正常运行。同时，本文还提供了数据的增量备份和增量恢复功能，可以在数据发生变化时，进行快速的恢复。高性能：本文提出的系统采用 LSM 树数据结构来实现分布式存储，可以在多个节点之间实现数据的高效复制和同步。同时，本文还提供了数据的备份和恢复功能，可以在数据发生变化时，进行快速的恢复。安全性：本文提出的系统采用分布式存储和 Raft 协议来管理数据的持久性和一致性，可以保证数据的安全性和可靠性。同时，本文还提供了数据的增量备份和增量恢复功能，可以在数据发生变化时，进行快速的恢复。

1.4 论文组织结构

本文主要围绕相关技术选型，需求分析，系统整体设计、详细设计，部署与测试等方面来进行论述，共分为 6 章，各章内容如下：

第 1 章讲述分布式存储系统的发展和课题研究方法

第 2 章描述开发工具环境，LSM-Tree 结构和 Raft 算法，以及设计的开源库

第 3 章叙述需求分析，研究的内容和目标用户

第 4 章描述存储系统总体的架构设计和客户端总体设计

第 5 章详细描述存储系统和客户端具体实现细节

第 6 章对系统进行整体部署和客户端测试

2 相关背景知识介绍

2.1 开发工具和环境

1、开发工具：VS Code

Visual Studio Code（简称 VS Code）是一款由微软开发且跨平台的免费集成开发环境。该软件支持语法高亮、代码自动补全（又称 IntelliSense）、代码重构功能，并且内置了命令行工具和 Git 版本控制系统。用户可以更改主题和键盘快捷方式实现个性化设置，也可以通过内置的扩展程序商店安装扩展以拓展软件功能。VS Code 使用 Monaco Editor 作为其底层的代码编辑器。Visual Studio Code 的源代码以 MIT 许可证在 GitHub 上释出，而可执行文件使用了专门的许可证。

2、开发环境

（1）X86-64 GNU/Linux-Ubuntu22.04

Linux 是一种自由和开放源码的类 UNIX 操作系统。该操作系统的内核由林纳斯·托瓦兹在 1991 年 10 月 5 日首次发布，再加上用户空间的应用程序之后，就成为了 Linux 操作系统。Linux 严格来说是单指操作系统的内核，因操作系统中包含了许多用户图形接口和其它实用工具。如今 Linux 常用来指基于 Linux 的完整操作系统，内核则改以 Linux 内核称之。由于这些支持用户空间的系统工具和库主要由理查德·斯托曼于 1983 年发起的 GNU 计划提供，自由软件基金会提议将其组合系统命名为 GNU/Linux。

Ubuntu 是基于 Debian，以桌面应用为主的 Linux 发行版。Ubuntu 有三个正式版本，包括桌面版、服务器版及用于物联网设备和机器人的 Core 版。前述三个版本既能安装于实体电脑，也能安装于虚拟环境。

（2）Golang1.20

Go（又称 Golang）是 Google 开发的一种静态强类型、编译型、并发型，并具有垃圾回收功能的编程语言。罗伯特·格瑞史莫、罗勃·派克及肯·汤普逊于 2007 年 9 月开始设计 Go，稍后伊恩·兰斯·泰勒（Ian Lance Taylor）、拉斯·考克斯（Russ Cox）加入项目。Go 是基于 Inferno 操作系统所开发的。Go 于 2009 年 11 月正式宣布推出，成为开放源代码项目，支持 Linux、macOS、Windows 等操作系统。

3、测试环境

（1）X86-64 Windows11

Windows 11 是微软于 2021 年推出的 Windows NT 系列操作系统，为 Windows 10 的后继者。出于安全考虑，Windows 11 的系统需求比 Windows 10 有所提高。微软仅支持使用英特尔酷睿第 8 代或更新的处理器、AMD Zen+ 或更新的处理器及高通骁龙 850 或更新的处理器设备。Windows 11 不再支持 32 位 x86 架构或使用 BIOS 固件的设备。

（2）X86-64 GNU/Linux-Ubuntu22.04

前文已经提及

（3）Arm64 Darwin MacOS Ventura13.3

Darwin 是由苹果公司于 2000 年所发布的一个开放源代码操作系统。Darwin 是 macOS 和 iOS 操作环境的操作系统部分。苹果公司于 2000 年把 Darwin 发布给开放源代码社群。Darwin 是一种类 Unix 操作系统，包含开放源代码的 XNU 内核，其以微核心为基础的核心架构来实现 Mach，而操作系统的服务和用户空间工具则以 BSD 为基础。类似其它类 Unix 操作系统，Darwin 也有对称多处理器的优点，高性能的网络设施和支持多种集成的文件系统。Darwin 的内核是 XNU，它是一种混合内核，它采用了来自 OSF 的 OSFMK 7.3（Open Software Foundation Mach Kernel）和 FreeBSD 的各种要素（包括过程模型，网络堆栈和虚拟文件系统），还有一个称为 I/O Kit 的面向对象的设备驱动程序 API。混合内核设计使其具备了微内核的灵活性和宏内核的性能。

2.2 LSM-Tree 存储结构

在计算机科学中，日志结构合并树（也称为 LSM 树或 LSMT）是一种具有一定性能特征的数据结构，可以为具有高插入量的文件（例如事务日志）提供索引访问数据。LSM 树和其它搜索树一样，维护键值对。LSM 树将数据保存在两个或多个独立的结构中，每个结构都针对其各自的底层存储介质进行了优化；数据在两个结构之间有效地、批量地同步。

LSM 树的一个简单版本是两级 LSM 树。两级 LSM 树包含两个树状结构，称为 C0 和 C1。C0 较小，完全驻留在内存中，而 C1 驻留在磁盘上。新记录被插入到内存驻留的 C0 组件中。如果插入导致 C0 组件超过某个大小阈值，则从 C0 中删除一个连续的条目段，并合并到磁盘上的 C1 中。LSM 树的性能特征源于这样一个事实，即每个组件都根据其底层存储介质的特性进行调整，并且使用一种让人联想到归并排序的算法，数据可以滚动批次高效地跨介质迁移。

实践中使用的大多数 LSM 树都采用多个级别。0 级保存在主内存中，可以用树表

示。磁盘上的数据被组织成排序的数据运行。每次运行都包含按索引键排序的数据。一次运行可以在磁盘上表示为单个文件，或者表示为具有非重叠键范围的文件集合。要对特定键执行查询以获取其关联值，必须在 Level 0 树中进行搜索，并且每次都运行。LSM 树的 Stepped-Merge 版本是 LSM 树的变体，它支持多层次，每一层次都有多个树结构。一个特定的键可能会出现在多次运行中，这对查询意味着什么取决于应用程序。一些应用程序只需要具有给定键的最新键值对。某些应用程序必须以某种方式组合这些值以获得要返回的正确聚合值。例如，在 Apache Cassandra 中，每个值代表数据库中的一行，不同版本的行可能有不同的列集。为了降低查询成本，系统必须避免运行次数过多的情况。随着越来越多的读写工作负载在 LSM-tree 存储结构下共存，由于 LSM-tree 压缩操作经常使缓冲区缓存中的缓存数据失效，读取数据访问可能会遇到高延迟和低吞吐量。为了重新启用有效的缓冲区缓存以实现快速数据访问，提出并实现了一种日志结构缓冲合并树（LSbM-tree）。

2.3 Raft 共识性算法

Raft 是一种用于替代 Paxos 的共识算法。相比于 Paxos，Raft 的目标是提供更清晰的逻辑分工使得算法本身能被更好地理解，同时它安全性更高，并能提供一些额外的特性。Raft 能为在计算机集群之间部署有限状态机提供一种通用方法，并确保集群内的任意节点在某种状态转换上保持一致。Raft 算法的开源实现众多，在 Go、C++、Java 以及 Scala 中都有完整的代码实现。

2.4 涉及的开源库

1、Golang 跨平台文件系统通知库 fsnotify

项目地址：<https://github.com/fsnotify/fsnotify>

fsnotify 是一个 Go 库，用于在 Windows、Linux、macOS、BSD 和 illumos 上提供跨平台文件系统通知。

2、Golang 文件压缩库 snappy

项目地址：<https://github.com/golang/snappy>

Snappy 是一个压缩/解压库。它不以最大压缩或与任何其它压缩库兼容为目标；相反，它以非常高的速度和合理的压缩为目标。例如，与 zlib 的最快模式相比，Snappy 对大多数输入来说要快一个数量级，但由此产生的压缩文件要大 20% 到 100%。这个库的

优点有：快速：压缩速度为 250 MB/秒及以上，没有汇编代码。稳定：在过去几年里，Snappy 在谷歌的生产环境中压缩和解压缩了 PB 的数据。Snappy bitstream 格式是稳定的，不会在版本之间更改。稳健：Snappy 解压器旨在在遇到损坏或恶意输入时不会崩溃。golang/snappy 是 google/snappy (C++) 的官方实现。

3、Golang 测试库 Ginkgo | Gomega

项目地址：<https://github.com/onsi/ginkgo>

Ginkgo 是 Go 的一个测试框架，旨在帮助开发者编写富有表现力的测试。它与 Gomega 匹配器库搭配使用。结合使用时，Ginkgo 和 Gomega 为编写测试提供了丰富且富有表现力的 DSL（领域特定语言）。Ginkgo 有时被描述为“行为驱动开发”(BDD) 框架。实际上，Ginkgo 是一个通用测试框架，在各种测试环境中得到积极使用：单元测试、集成测试、验收测试、性能测试等。

4、Golang 性能度量库 go-metrics

项目地址：github.com/armon/go-metrics

go-metrics 是一个 Go 应用性能度量指标的库，go-metrics 提供的 meter、histogram 可以覆盖 Go 应用基本性能指标需求，如：吞吐性能、延迟数据分布等。此外，go-metrics 是模仿 JVM Metrics 开发的一个库，可以与 Golang 的 runtime 无缝集成。

5、Golang 断言库 testify

项目地址：<https://github.com/stretchr/testify>

testify 是一个具有常见断言和模拟的工具包，可以与 Golang 标准库做很好的搭配。

3 Radds 存储系统总体设计

3.1 存储系统架构设计

Radds 存储系统总体架构组成如下所示

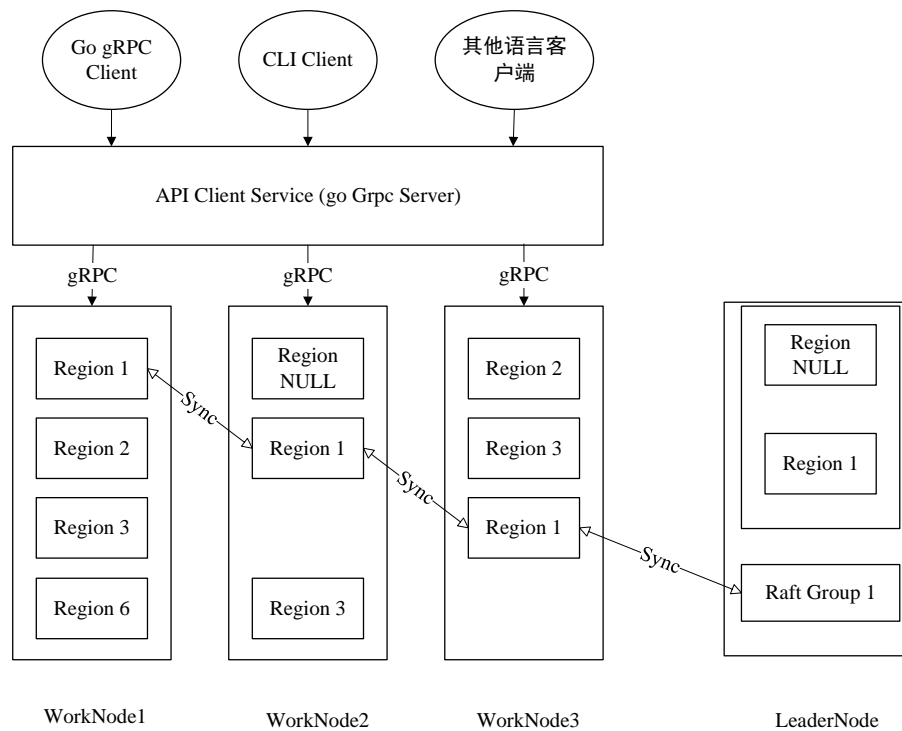


图 3.1 Radds 存储系统总体架构图

系统的设计架构是 C/S 的方式，并提供最大能力的可插拔特性。服务端支持单机模式和集群模式；客户端提供了 API Client 中台，默认使用 gRPC 通信，客户端支持多语言：默认支持 golang 语言的 gRPC 客户端，RESTful 形式的 HTTP 通信，还有命令行 CLI 方式，用户可以使用支持 gRPC 的其它语言来开发自定义的客户端（gRPC 支持绝大多数主流开发语言）。

1、服务端 API Client 中台

API Client 中台提供诸多功能，如：API 客户端调用日志，API 限流，数据查询日志，存储消息重定向等。

2、服务端单机模式

单机模式单机模式提供键值形式的数据存储，实现了 LSM-Tree 结构的存储引擎，单机模式可以看做是集群模式的一个存储节点，

3、服务端集群模式

集群模式以 Raft 算法为设计核心，集群间的节点通信使用 gRPC 协议的 message，这种方式通信的开销更小，因为 gRPC 仅仅建立在 TCP 协议之上。当然一个 WorkNode 不是只存储一个 Group 的数据，还可以存储其它 Group 的数据。集群的 Leader 通过数据规模和 API 调用次数分配工作节点数量，这个 Leader 仅仅是一个 Group 的 Leader，整个集群可以有多个 Leader，它们分别属于不同的 Group。集群中的一个节点如果作为一个 Group 的 Leader，那么它存储这个 Group 的所有数据：用户数据，元数据，缓存数据。集群中的一个节点如果作为一个 Group 的 WorkNode，那么它只存储这个 Group 的用户数据。集群中的一个节点会存在不同的状态，可能的状态有：

(1) Leader 宕机，一个 Group 的 Leader 宕机后，其它节点在超时时间内未收到心跳信息则会感知到 Group Leader 宕机，根据 Raft 算法选择 Group 的新一任 Leader，当新一任的 Leader 选举后会向客户端中台发送消息并从客户端中台取得必要的元数据。

(2) WorkNode 宕机，一个 Group 的 WorkNode 宕机后，Group 的 Leader 会在长时间未收到心跳后将此节点移除 Group 并重新通知客户端中台机器数量，以便于其它 Group 能分配到状态良好的机器。

(3) 节点存储空间异常，当一个节点的活跃 Group Region 的存储空间到达 95%，由于要保证分布式系统数据的强一致，所以无论是 Leader 还是 WorkNode 都会停止存储新的数据，这些节点的 Region 只能作为 WorkNode 存储数据提供数据查询服务，对于写操作不在响应，同时它们的数据版本永久定格在此刻，当新的写操作到达时，Leader 会通知客户端中台，客户端中台会重新选择集群节点，根据 Raft 算法选择新任 Leader。在数据归并的过程中，如果这些节点的存储空间由于数据的归并而释放到足够存储一个 Region 的大小（默认一个机器磁盘容量的 1/4），那么它可以成为一个 Group 的 Region，根据 Raft 算法强一致的特性，这些数据完全被丢弃成为 Region NULL，这时它就可以成为被集群选中的 WorkNode。

(4) 节点 I/O 异常，一个 Group 的不同节点只有一个节点是 Leader Node，Leader 对客户端进行写响应，在客户端平台看来 Leader 和 WorkNode 没有不同，当一个请求到达时，客户端中台分发请求到一个机器。由于所有请求都要经过客户端中台，所以当节点 I/O 异常，可以考虑是机器的故障，并上报中台是不可用节点。

由于 Raft 算法形式上比较清晰，本文根据 Raft 算法在机器层面做了改动，让一个

节点同时可以最多容纳 4 个 Region，这样保证 Leader 节点不会出现太大程度的浪费，同时充分利用 WorkNode 的存储空间存储用户数据。对于 LSM-Tree 的存储结构本文则是做全部的复现，存储引擎和数据复制，数据压缩是存储系统的工作核心。对于客户端中台的搭建，这是一个面向用户的组成部分，也是本文的创新，这是让一个存储引擎可用和好用的部分。同时客户端中台设计成了一个插件化的中心，后续也可以添加新的功能，如鉴权，资源隔离，甚至可以实现多租户方案等，这个灵感来源于阅读 Snowflake 数据仓库和一些数据库报告的相关论文。

3.2 存储层功能总体设计

3.2.1 存储引擎设计概述

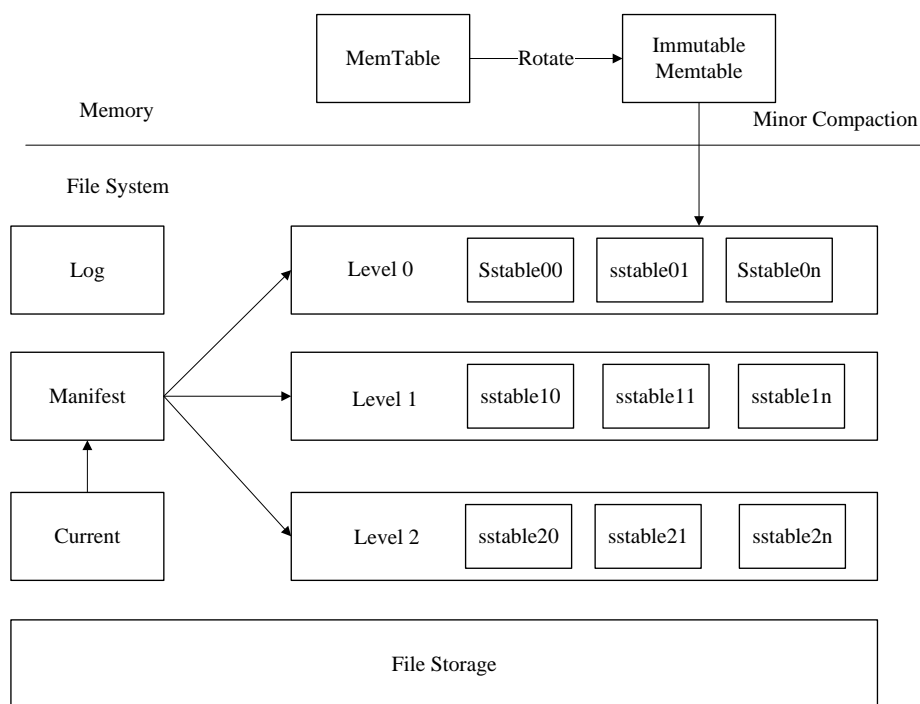


图 3.2 Radd's 存储层架构设计

上图是存储引擎的架构设计图，本文要实现具有写性能的存储引擎，则必须实现一个 LSM 树 (Log Structured-Merge Tree)。LSM 树的核心思想就是放弃部分读的性能，换取最大的写入能力。LSM 树写性能极高的原理，就是尽量减少随机写的次数。对于每次写入操作，并不是直接将最新的数据驻留在磁盘中，而是将其拆分成一次日志文件的顺序写和一次内存中的数据插入存储架构正是实践了这种思想，将数据首先更新在内存

中，当内存中的数据达到一定的阈值，将这部分数据真正刷新到磁盘文件中，因而获得了极高的写性能（顺序写 60MB/s, 随机写 45MB/s）。

3.2.2 内存可变结构 memtable 总体设计

之前提到，存储引擎的一次写入操作并不是直接将数据刷新到磁盘文件，而是首先写入到内存中作为代替，**memtable** 就是一个在内存中进行数据组织与维护的结构。**memtable** 中，所有的数据按用户定义的排序方法排序之后按序存储，等到其存储内容的容量达到阈值时（默认为 4MB），便将其转换成一个不可修改的 **memtable**，与此同时创建一个新的 **memtable**，供用户继续进行读写操作。**memtable** 底层使用了一种跳表数据结构，这种数据结构效率可以比拟二叉查找树，绝大多数操作的时间复杂度为 $O(\log n)$ 。

3.2.3 内存不可变结构 immutable memtable 总体设计

memtable 的容量到达阈值时，便会转换成一个不可修改的 **memtable**，也称为 **immutable memtable**。这两者的结构定义完全一样，区别只是 **immutable memtable** 是只读的。当一个 **immutable memtable** 被创建时，存储系统的后台压缩进程便会将利用其中的内容，创建一个 **sstable**，持久化到磁盘文件中。

3.2.4 日志文件结构 journal 总体设计

存储系统的写操作并不是直接写入磁盘的，而是首先写入到内存。假设写入到内存的数据还未来得及持久化，存储系统进程发生了异常，抑或是宿主机发生了宕机，会造成用户的写入发生丢失。因此存储系统在写内存之前会首先将所有的写操作写到日志文件中，也就是 **log** 文件。当以下五种异常情况发生时，均可以通过日志文件进行恢复。

- 1、写 **log** 期间进程异常。
- 2、写 **log** 完成，写内存未完成。
- 3、**write** 动作完成（即 **log**、内存写入都完成）后，进程异常。
- 4、**immutable memtable** 持久化过程中进程异常。
- 5、其它压缩异常（较为复杂，首先不在此处介绍）。

异常发生时，处理的情况分两种。当第一类情况发生时，数据库重启读取 **log** 时，发现异常日志数据，抛弃该条日志数据，即视作这次用户写入失败，保障了数据库的一致性。当第二类，第三类，第四类情况发生了，均可以通过 **redo** 日志文件中记录的写入操作完成数据库的恢复。每次日志的写操作都是一次顺序写，因此写效率高，整体写入

性能较好。此外，存储系统的用户写操作的原子性同样通过日志来实现。

3.2.5 磁盘持久化结构 sstable 总体设计

虽然存储系统采用了先写内存的方式来提高写入效率，但是内存中数据不可能无限增长，且日志中记录的写入操作过多，会导致异常发生时，恢复时间过长。因此内存中的数据达到一定容量，就需要将数据持久化到磁盘中。除了某些元数据文件，存储系统的数据主要都是通过 sstable 来进行存储。

虽然在内存中，所有的数据都是按序排列的，但是当多个 memetable 数据持久化到磁盘后，对应的不同的 sstable 之间是存在交集的，在读操作时，需要对所有的 sstable 文件进行遍历，严重影响了读取效率。因此存储系统后台会“定期”整合这些 sstable 文件，该过程也称为 compaction。随着 compaction 的进行，sstable 文件在逻辑上被分成若干层，由内存数据直接 dump 出来的文件称为 level 0 层文件，后期整合而成的文件为 level i 层文件，这也是以 leveldb 为原型的存储系统的这个名字的由来。所有的 sstable 文件本身的内容是不可修改的，这种设计带来了许多优势，简化了很多设计。

3.2.6 文件元数据 manifest 总体设计

存储系统中有个版本的概念，一个版本中主要记录了每一层中所有文件的元数据，元数据包括（1）文件大小（2）最大 key 值（3）最小 key 值。该版本信息十分关键，除了在查找数据时，利用维护的每个文件的最大、小 key 值来加快查找，还在其中维护了一些进行 compaction 的统计值，来控制 compaction 的进行。一个文件的元数据主要包括了最大最小 key，文件大小等信息。

3.2.7 版本号 current 总体设计

一个版本信息主要维护了每一层所有文件的元数据。当每次 compaction 完成（或者换一种更容易理解的说法，当每次 sstable 文件有新增或者减少），leveldb 都会创建一个新的 version，创建的规则是： $versionNew = versionOld + versionEdit$ versionEdit 指代的是基于旧版本的基础上，变化的内容（例如新增或删除了某些 sstable 文件）。manifest 文件就是用来记录这些 versionEdit 信息的。一个 versionEdit 数据，会被编码成一条记录，写入 manifest 文件中。如图 3.3 便是一个 manifest 文件的示意图，其中包含了 3 条 versionEdit 记录，每条记录包括（1）新增哪些 sst 文件（2）删除哪些 sst 文件（3）当前 compaction 的下标（4）日志文件编号（5）操作 seqNumber 等信息。通过这些信息，存

储系统便可以在启动时，基于一个空的 version，不断 apply 这些记录，最终得到一个上次运行结束时的版本信息。

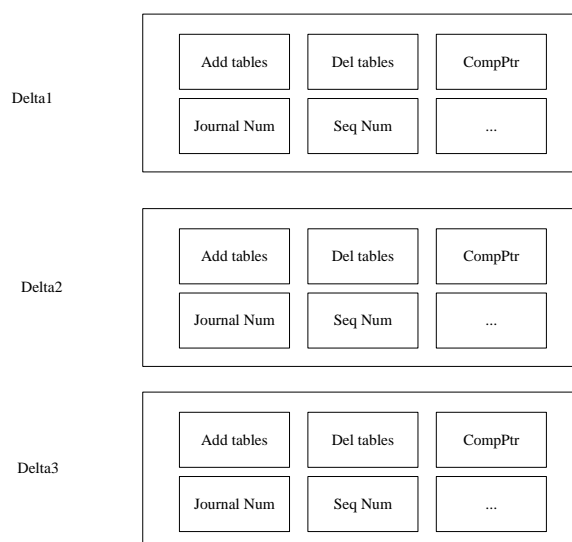


图 3.3 版本信息记录数据结构

3.3 共识层功能总体设计

在一个分布式系统中，寻找可靠的共识性算法是分布式系统保证一致性（Consistency），可用性（Availability），分区容错性（Partition Tolerance）的关键性问题。对于共识性算法，本文选取容易理解和实现在真实系统之中的 Raft 算法，Reliable, Replicated, Redundant, Fault-Tolerant 是 raft 算法的四个重要特性，也是算法的名字来源。

Raft 节点总是处于三种状态之一：follower、candidate 或 leader。所有节点最初都是作为跟随者开始的。在这种状态下，节点可以接受来自领导者的日志条目并进行投票。如果一段时间内没有收到任何条目，节点将自本文提升到候选状态。在候选状态中，节点向其对等节点请求投票。如果候选人获得法定人数的选票，则将其提升为领导者。领导者必须接受新的日志条目并复制给所有其它追随者。此外，如果过时的读取是不可接受的，则所有查询也必须在领导者上执行。

一旦集群有了领导者，它就能够接受新的日志条目。客户端可以请求领导者附加一个新的日志条目，这是一个不透明的二进制 blob 到 Raft。领导者然后将条目写入持久存储并尝试复制到追随者的法定人数。一旦日志条目被认为已提交，它就可以应用于有限状态机。有限状态机是特定于应用程序的，并使用接口实现。

一个明显的问题与复制日志的无限性质有关。**Raft** 提供了一种机制，可以对当前状态进行快照，并压缩日志。由于 **FSM** 抽象，恢复 **FSM** 的状态必须导致与重播旧日志相同的状态。这允许 **Raft** 捕获某个时间点的 **FSM** 状态，然后删除所有用于达到该状态的日志。这是自动执行的，无需用户干预，可防止无限制的磁盘使用，并最大限度地减少重放日志所花费的时间。

最后，还有在新服务器加入或现有服务器离开时更新对等集的问题。只要有法定数量的节点可用，这就不是问题，因为 **Raft** 提供了动态更新对等集合的机制。如果节点法定人数不可用，那么这将成为一个非常具有挑战性的问题。例如，假设只有 2 个对等点，**A** 和 **B**。仲裁大小也是 2，这意味着两个节点都必须同意提交日志条目。如果 **A** 或 **B** 失败，则现在不可能达到法定人数。这意味着集群无法添加或删除节点，或提交任何其它日志条目。这导致不可用。此时，需要手动干预以删除 **A** 或 **B**，并以引导模式重新启动其余节点。

3 个节点的 **Raft** 集群可以容忍单个节点故障，而 5 个节点的集群可以容忍 2 个节点故障。推荐的配置是运行 3 个或 5 个 **raft** 服务器。这样可以在不显著牺牲性能的情况下最大限度地提高可用性。

在性能方面，**Raft** 与 **Paxos** 不相上下。假设领导稳定，提交日志条目需要单次往返集群的一半。因此，性能受磁盘 I/O 和网络延迟的限制。

4 Radds 存储系统详细设计与实现

本章对存储系统的各层进行详细设计与实现，针对各层内部的子系统，各层之间的接口进行详细定义。

4.1 存储层详细设计与实现

4.1.1 日志系统的实现

为了防止写入内存的数据库因为进程异常、操作系统掉电等情况发生丢失，存储系统在写内存之前会将本次写操作的内容写入日志文件中。

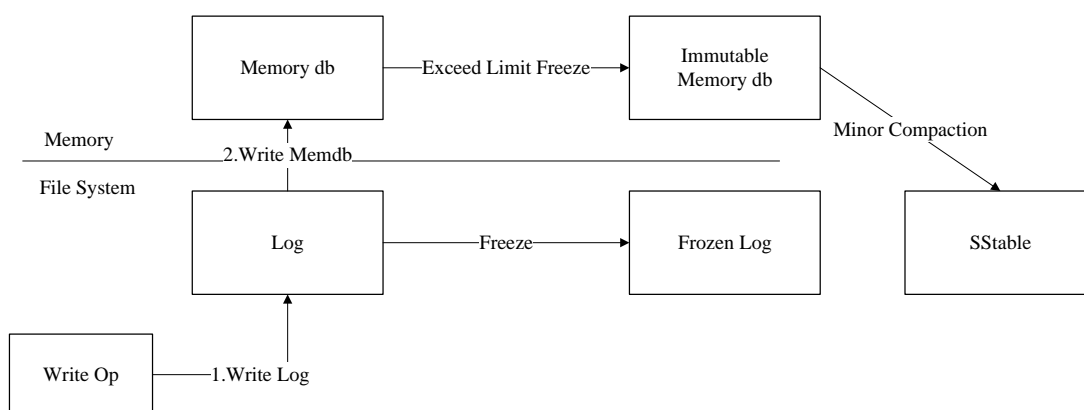


图 4.1 日志系统架构图

存储系统中，有两个 memory db，以及对应的两份日志文件。其中一个 memory db 是可读写的，当这个 db 的数据量超过预定的上限时，便会转换成一个不可写的 memory db，与此同时，与之对应的日志文件也变成一份 frozen log。

而新生成的 immutable memory db 则会由后台的 minor compaction 进程将其转换成一个 sstable 文件进行持久化，持久化完成，与之对应的 frozen log 被删除。如果 frozen log 被存档，但是另一个进程正在写入新数据，那么新数据会被当作缓存数据写入缓存，这部分的缓存是内存数据库，写入内存数据库后会通过日志定期刷新到所定位的日志文件，从而成为新的 frozen log，这样避免了 frozen log 被丢弃的风险，增加了容错性和可靠性，这是一种模仿 mysql 日志同步数据的解决方案，并不是独创性的，这部分的逻辑就是利用更多层次的缓存减少错误，增加可靠性，这也是日志复制的核心内容。关于新的日志复制如何使用全量和增量会在下一章节详细介绍。

1、日志结构

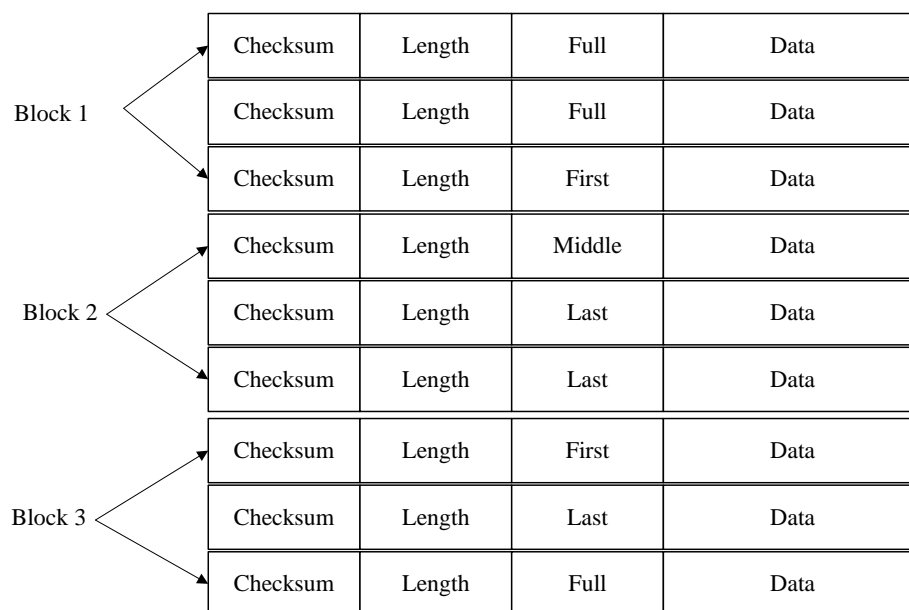


图 4.2 日志文件存储结构图

为了增加读取效率，日志文件中按照 **block** 进行划分，每个 **block** 的大小为 32KiB。每个 **block** 中包含了若干个完整的 **chunk**。一条日志记录包含一个或多个 **chunk**。每个 **chunk** 包含了一个 7 字节大小的 **header**，前 4 字节是该 **chunk** 的校验码，紧接的 2 字节是该 **chunk** 数据的长度，以及最后一个字节是该 **chunk** 的类型。其中 **checksum** 校验的范围包括 **chunk** 的类型以及随后的 **data** 数据。

chunk 共有四种类型：**full**，**first**，**middle**，**last**。一条日志记录若只包含一个 **chunk**，则该 **chunk** 的类型为 **full**。若一条日志记录包含多个 **chunk**，则这些 **chunk** 的第一个类型为 **first**，最后一个类型为 **last**，中间包含大于等于 0 个 **middle** 类型的 **chunk**。由于一个 **block** 的大小为 32KiB，因此当一条日志文件过大时，会将第一部分数据写在第一个 **block** 中，且类型为 **first**，若剩余的数据仍然超过一个 **block** 的大小，则第二部分数据写在第二个 **block** 中，类型为 **middle**，最后剩余的数据写在最后一个 **block** 中，类型为 **last**。

2、日志内容

日志的内容为写入的 **batch** 编码后的信息，具体的格式为：

Sequence number	Entry number	Batch data	...	Batch data
-----------------	--------------	------------	-----	------------

图 4.3 日志文件格式图

一条日志记录的内容包含：Header 和 Data 其中 Header 中有当前 db 的 sequence number 和本次日志记录中所包含的 put/del 操作的个数，紧接着写入所有 batch 编码后的内容。

3、日志文件写

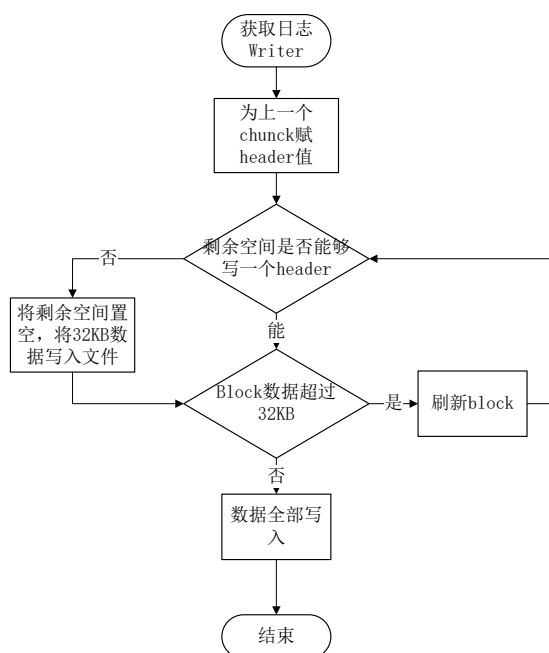


图 4.4 日志文件写流程图

日志写入流程较为简单，在存储系统内部，实现了一个 journal 的 writer。首先调用 Next 函数获取一个 singleWriter，这个 singleWriter 的作用就是写入一条 journal 记录。

singleWriter 开始写入时，标志着第一个 chunk 开始写入。在写入的过程中，不断判断 writer 中 buffer 的大小，若超过 32KiB，将 chunk 开始到现在做为一个完整的 chunk，为其计算 header 之后将整个 chunk 写入文件。与此同时 reset buffer，开始新的 chunk 的写入。若一条 journal 记录较大，则可能会分成几个 chunk 存储在若干个 block 中。

4、日志文件读

同样，日志读取也较为简单。为了避免频繁的 IO 读取，每次从文件中读取数据时，按 block（32KiB）进行块读取。每次读取一条日志记录，reader 调用 Next 函数返回一个 singleReader。singleReader 每次调用 Read 函数就返回一个 chunk 的数据。每次读取一个 chunk，都会检查这批数据的校验码、数据类型、数据长度等信息是否正确，若不正确，且用户要求严格的正确性，则返回错误，否则丢弃整个 chunk 的数据。循环调用

singleReader 的 read 函数，直至读取到一个类型为 Last 的 chunk，表示整条日志记录都读取完毕，返回。

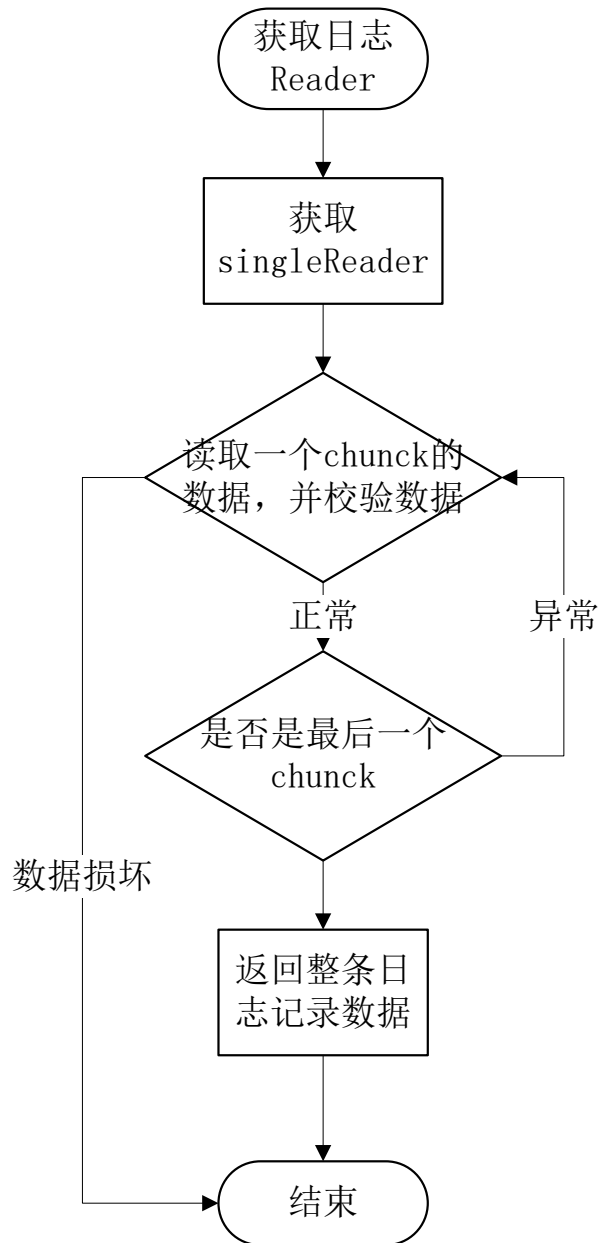


图 4.5 日志文件读流程图

4.1.2 写数据的实现

1、写入数据的整体流程

先来分析一下存储系统整个写入的流程，底层数据结构的支持以及为何能够优化本文的写入性能。

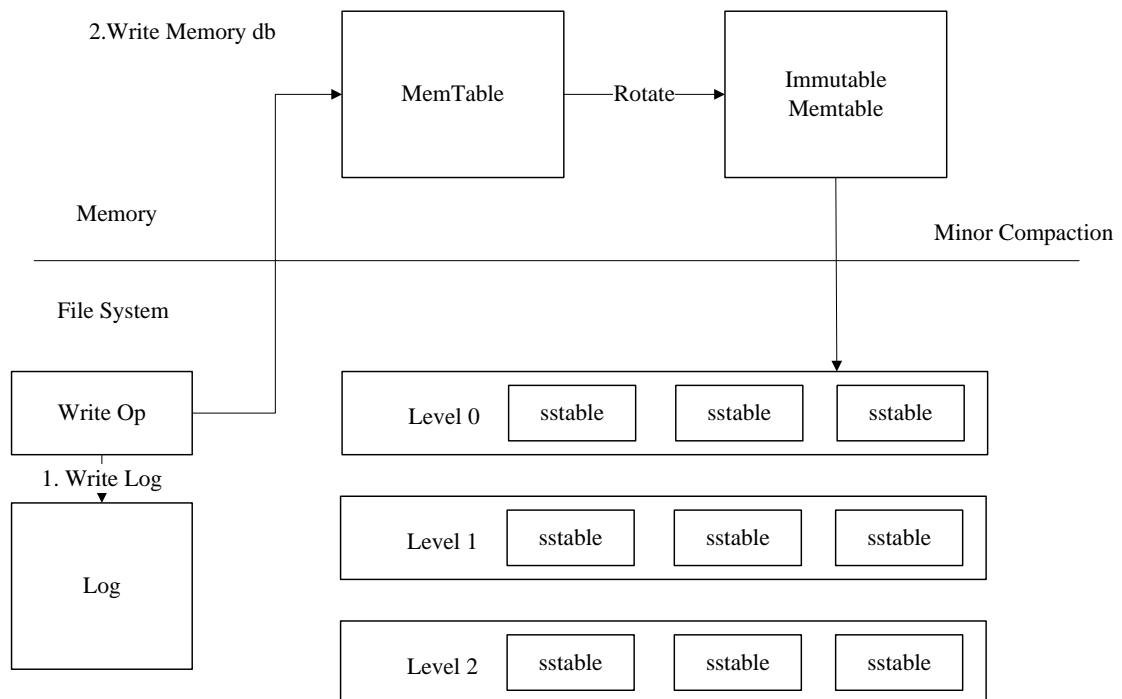


图 4.6 存储系统写数据流程

数据的一次写入分为两部分：将写操作写入日志；将写操作应用到内存数据库中；之前已经阐述过为何这样的操作可以优化写入性能，以及通过先写日志的方法能够保障用户的写入不丢失。

其实仍然存在写入丢失的隐患。在写设置为非同步的情况下，在写完日志文件以后，操作系统并不是直接将数据真正落到磁盘中，而是暂时留在操作系统缓存中，因此当用户写入操作完成，操作系统还未来得及落盘的情况下，发生系统宕机，就会造成写丢失；但是若只是进程异常退出，则不存在该问题。

2、写类型

由于是键值型非关系型数据存储，存储系统对外提供的写入接口有：**Put** 和 **Delete** 两种。这两种本质对应同一种操作，**Delete** 操作同样会被转换成一个 **value** 为空的 **Put** 操作。除此以外，本文还提供了一个批量处理的工具 **Batch**，用户可以依据 **Batch** 来完成批量更新操作，且这些操作是原子性的。

3、batch 结构

无论是 **Put/Del** 操作，还是批量操作，底层都会为这些操作创建一个 **batch** 实例作为一个数据库操作的最小执行单元。因此首先介绍一下 **batch** 的组织结构。

type	Key length	key	Value length	value
------	------------	-----	--------------	-------

图 4.7 batch 的数据结构

在 batch 中，每一条数据项都按照上图格式进行编码。每条数据项编码后的第一位是这条数据项的类型（更新还是删除），之后是数据项 key 的长度，数据项 key 的内容；若该数据项不是删除操作，则再加上 value 的长度，value 的内容。

batch 中会维护一个 size 值，用于表示其中包含的数据量的大小。该 size 值为所有数据项 key 与 value 长度的累加，以及每条数据项额外的 8 个字节。这 8 个字节用于存储一条数据项额外的一些信息。

4、key 值编码

当数据项从 batch 中写入到内存数据库中时，需要将一个 key 值的转换，即在存储系统内部，所有数据项的 key 是经过特殊编码的，这种格式称为 internalKey。

uKey	Sequence number(7byte)	Type(1 byte)
------	------------------------	--------------

图 4.8 internalkey 的数据结构

internalkey 在用户 key 的基础上，尾部追加了 8 个字节，用于存储该操作对应的 sequence number 和该操作的类型。每一个操作都会被赋予一个 sequence number。该计时器是在存储系统内部维护，每进行一次操作就做一个累加。由于在存储系统中，一次更新或者一次删除，采用的是 append 的方式，并非直接更新原数据。因此对应同样一个 key，会有多个版本的数据记录，而最大的 sequence number 对应的数据记录就是最新的。此外，存储系统的快照（snapshot）也是基于这个 sequence number 实现的，即每一个 sequence number 代表着数据库的一个版本。

5、数据合并写入

存储系统中，在面对并发写入时，做了一个处理的优化。在同一个时刻，只允许一个写入操作将内容写入到日志文件以及内存数据库中。为了在写入进程较多的情况下，减少日志文件的小写入，增加整体的写入性能，存储系统将一些“小写入”合并成一个“大

写入”。

当前写操作

(1) 第一个写入操作获取到写入锁。

(2) 在当前写操作的数据量未超过合并上限，且有其它写操作 pending 的情况下，将其它写操作的内容合并到自身。

(3) 若本次写操作的数据量超过上限，或者无其它 pending 的写操作了，将所有内容统一写入日志文件，并写入到内存数据库中。

(4) 通知每一个被合并的写操作最终的写入结果，释放或移交写锁。

其它写操作

(1) 等待获取写锁或者被合并。

(2) 若被合并，判断是否合并成功，若成功，则等待最终写入结果。

(3) 反之，则表明获取锁的写操作已经 **oversize** 了，此时，该操作直接从上个占有锁的写操作中接过写锁进行写入。

(4) 若未被合并，则继续等待写锁或者等待被合并。

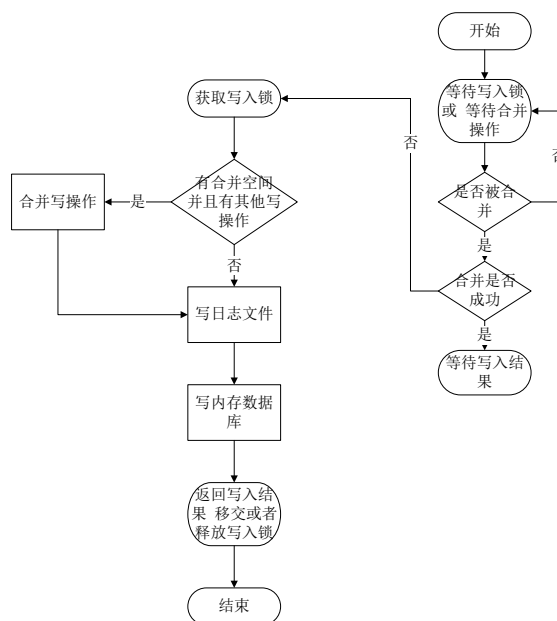


图 4.9 写合并的流程

6、原子性

存储系统的任意一个写操作（无论包含了多少次写），其原子性都是由日志文件实现的。一个写操作中所有的内容会以一个日志中的一条记录，作为最小单位写入。

考虑以下两种异常情况：写日志未开始，或写日志完成一半，进程异常退出；写日志完成，进程异常退出。前者中可能存储一个写操作的部分写已经被记载到日志文件中，仍然有部分写未被记录，这种情况下，当数据库重新启动恢复时，读到这条日志记录时，发现数据异常，直接丢弃或退出，实现了写入的原子性保障。后者，写日志已经完成，写入日志的数据未真正持久化，存储系统启动恢复时通过 redo 日志实现数据写入，仍然保障了原子性。

4.1.3 读数据的实现

存储系统提供给用户两种进行读取数据的接口：直接通过 Get 接口读取数据；首先创建一个 snapshot，基于该 snapshot 调用 Get 接口读取数据；两者的本质是一样的，只不过第一种调用方式默认地以当前数据库的状态创建了一个 snapshot，并基于此 snapshot 进行读取。snapshot（快照）就是数据库在某一个时刻的状态。基于一个快照进行数据的读取，读到的内容不会因为后续数据的更改而改变。由于两种方式本质都是基于快照进行读取的，因此在介绍读操作之前，首先介绍快照。

1、snapshot（快照）

快照代表着数据库某一个时刻的状态，在存储系统中，巧妙地用一个整型数来代表一个数据库状态。在存储系统中，用户对同一个 key 的若干次修改（包括删除）是以维护多条数据项的方式进行存储的（直至进行 compaction 时才会合并成同一条记录），每条数据项都会被赋予一个序列号，代表这条数据项的新旧状态。一条数据项的序列号越大，表示其中代表的内容为最新值。

因此，每一个序列号，代表着存储系统的一个状态。当用户主动或者被动地创建一个快照时，存储系统会以当前最新的序列号对其赋值。例如图中用户在序列号为 98 的时刻创建了一个快照，并且基于该快照读取 key 为“name”的数据时，即便此刻用户将“name”的值修改为“dog”，再删除，用户读取到的内容仍然是“cat”。

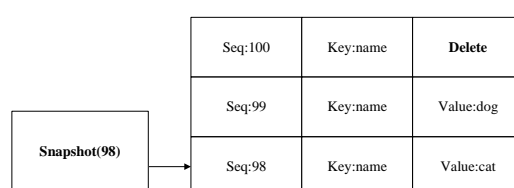


图 4.10 快照数据示例图

所以，利用快照能够保证数据库进行并发的读写操作。在获取到一个快照之后，存储系统会为本次查询的 **key** 构建一个 **internalKey**（格式如上文所述），其中 **internalKey** 的 **seq** 字段使用的便是快照对应的 **seq**。通过这种方式可以过滤掉所有 **seq** 大于快照号的数据项。

2、读数据流程

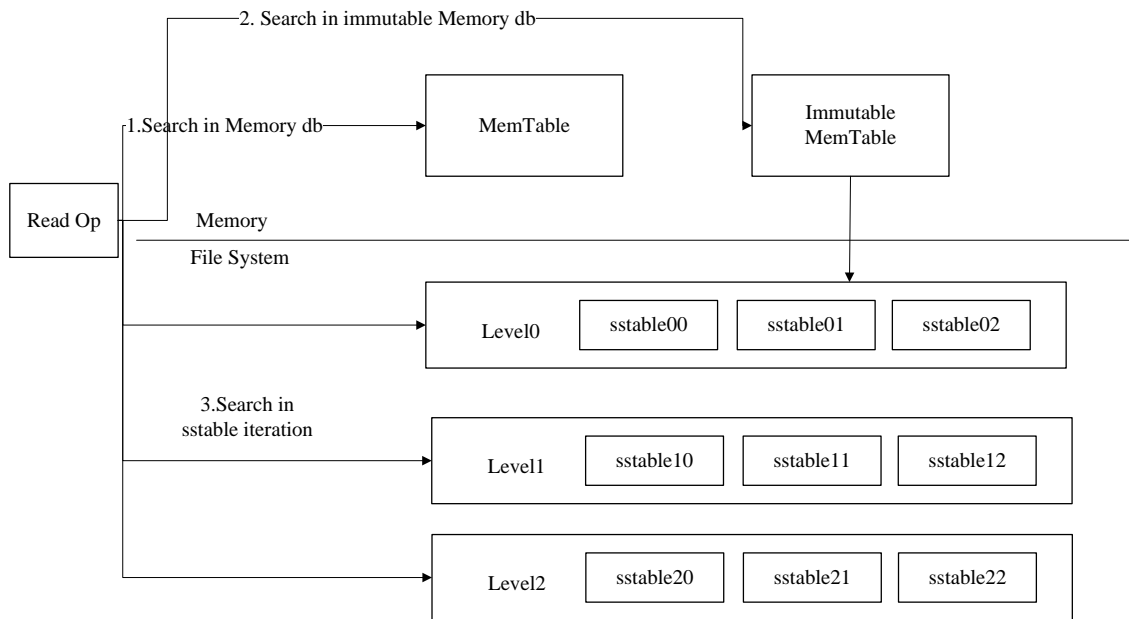


图 4.11 读数据流程

存储系统读取分为三步：

- (1) 在 **memory db** 中查找指定的 **key**，若搜索到符合条件的数据项，结束查找；
- (2) 在冻结的 **memory db** 中查找指定的 **key**，若搜索到符合条件的数据项，结束查找；
- (3) 按低层至高层的顺序在 **level i** 层的 **sstable** 文件中查找指定的 **key**，若搜索到符合条件的数据项，结束查找，否则返回 **Not Found** 错误，表示数据库中不存在指定的数据；

存储系统在每一层 **sstable** 中查找数据时，都是按序依次查找 **sstable** 的。0 层的文件比较特殊。由于 0 层的文件中可能存在 **key** 重合的情况，因此在 0 层中，文件编号大的 **sstable** 优先查找。理由是文件编号较大的 **sstable** 中存储的总是最新的数据。非 0 层文件，一层中所有文件之间的 **key** 不重合，因此存储系统可以借助 **sstable** 的元数据（一个文件

中最小与最大的 key 值) 进行快速定位, 每一层只需要查找一个 sstable 文件的内容。在 memory db 或者 sstable 的查找过程中, 需要根据指定的序列号拼接一个 internalKey, 查找用户 key 一致, 且 seq 号不大于指定 seq 的数据,

4.1.4 跳表数据结构的实现

内存数据库用来维护有序的 key-value 对, 其底层是利用跳表实现, 绝大多数操作 (读/写) 的时间复杂度均为 $O(\log n)$, 有着与平衡树相媲美的操作效率, 但是从实现的角度来说简单许多,

1、跳表的实现

跳表是利用概率均衡技术, 加快简化插入、删除操作, 且保证绝大多操作均拥有 $O(\log n)$ 的良好效率。

平衡树 (以红黑树为代表) 是一种非常复杂的数据结构, 为了维持树结构的平衡, 获取稳定的查询效率, 平衡树每次插入可能会涉及到较为复杂的节点旋转等操作。作者设计跳表的目的是借助概率平衡, 来构建一个快速且简单的数据结构, 取代平衡树。

2、跳表的结构

跳跃列表是按层建造的。底层是一个普通的有序链表。每个更高层都充当下面链表的”快速通道”, 这里在层 i 中的元素按某个固定的概率 p (通常为 0.5 或 0.25) 出现在层 $i+1$ 中。平均起来, 每个元素都在 $1/(1-p)$ 个列表中出现, 而最高层的元素 (通常是在跳跃列表前端的一个特殊的头元素) 在 $O(\log_{1/p} n)$ 个列表中出现。

3、跳表的查找

在介绍插入和删除操作之前, 本文首先介绍查找操作, 该操作是上述两个操作的基础。需要查找一个 key 的链表节点, 查找的过程为: 首先根据跳表的高度选取最高层的头节点; 若跳表中的节点内容小于查找节点的内容, 则取该层的下一个节点继续比较; 若跳表中的节点内容等于查找节点的内容, 则直接返回; 若跳表中的节点内容大于查找节点的内容, 且层高不为 0, 则降低层高, 且从前一个节点开始, 重新查找低一层中的节点信息; 若层高为 0, 则返回当前节点, 该节点的 key 大于所查找节点的 key。综合来说, 就是利用稀疏的高层节点, 快速定位到所需要查找节点的大致位置, 再利用密集的底层节点, 具体比较节点的内容。

4、跳表的插入

跳表的插入以查找为基础实现。在查找的过程中, 不断记录每一层的前任节点, 如

图中红色圆圈所表示的；为新插入的节点随机产生层高（随机产生层高的算法较为简单，依赖最高层数和概率值 p ，可见下文中的代码实现）；在合适的位置插入新节点，并依据查找时记录的前任节点信息，在每一层中，以链表插入的方式，将该节点插入到每一层的链接中。链表插入指：将当前节点的 `Next` 值置为前任节点的 `Next` 值，将前任节点的 `Next` 值替换为当前节点。

代码清单 4.1 skiplistRandHeight

```
func (p *DB) randHeight() (h int) {
    const branching = 4
    h = 1
    for h < tMaxHeight && p.rnd.Int()%branching == 0 {
        h++
    }
    return
}
```

5、跳表的删除

跳表的删除操作较为简单，依赖查找过程找到该节点在整个跳表中的位置后，以链表删除的方式，在每一层中，删除该节点的信息。链表删除指：将前任节点的 `Next` 值替换为当前节点的 `Next` 值，并将当前节点所占的资源释放。

6、跳表的迭代

（1）向后遍历

若迭代器刚被创建，则根据用户指定的查找范围 `[Start, Limit)` 找到一个符合条件的跳表节点。若迭代器处于中部，则取出上一次访问的跳表节点的后继节点，作为本次访问的跳表节点（后继节点为最底层的后继节点）。利用跳表节点信息（`keyvalue` 数据偏移量，`key`，`value` 值长度等），获取 `keyvalue` 数据。

（2）向前遍历

若迭代器刚被创建，则根据用户指定的查找范围 `[Start, Limit)` 在跳表中找到最后一个符合条件的跳表节点。若迭代器处于中部，则利用上一次访问的节点的 `key` 值，查找比该 `key` 值更小的跳表节点。利用跳表节点信息（`keyvalue` 数据偏移量，`key`，`value` 值长度等），获取 `keyvalue` 数据。

4.1.5 内存数据库的实现

在介绍完跳表这种数据结构的组织原理以后，本文介绍存储系统如何利用跳表来构建一个高效的内存数据库。

1、键值编码

在介绍内存数据库之前，首先介绍一下内存数据库的键值编码规则。由于内存数据库本质是一个 kv 集合，且所有的数据项都是依据 key 值排序的，因此键值的编码规则尤为关键。

内存数据库中，key 称为 internalKey，其由三部分组成：用户定义的 key：这个 key 值也就是原生的 key 值。序列号：存储系统中，每一次写操作都有一个 sequence number，标志着写入操作的先后顺序。由于在存储系统中可能会有多条相同 key 的数据项同时存储在数据库中，因此需要有一个序列号来标识这些数据项的新旧情况。序列号最大的数据项为最新值。类型：标志本条数据项的类型，为更新还是删除。

uKey	Sequence number(7byte)	Type(1 byte)
------	------------------------	--------------

图 4.12 内存数据库内部键图

2、键值比较

内存数据库中所有的数据项都是按照键值比较规则进行排序的。这个比较规则可以由用户自己定制，也可以使用系统默认的。默认的比较规则：首先按照字典序比较用户定义的 key (ukey)，若用户定义 key 值大，整个 internalKey 就大。若用户定义的 key 相同，则序列号大的 internalKey 值就小。通过这样的比较规则，则所有的数据项首先按照用户 key 进行升序排列。当用户 key 一致时，按照序列号进行降序排列，这样可以保证首先读到序列号大的数据项。

3、数据组织

```

type DB struct {
    cmp comparer.BasicComparer
    rnd *rand.Rand
    mu    sync.RWMutex
    kvData []byte
    
```

```

nodeData  []int
prevNode  [tMaxHeight]int
maxHeight int
n          int
kvSize     int
}

```

其中 kvData 用来存储每一条数据项的 key-value 数据，nodeData 用来存储每个跳表节点的链接信息。nodeData 中，每个跳表节点占用一段连续的存储空间，每一个字节分别用来存储特定的跳表节点信息。第一个字节用来存储本节点 key-value 数据在 kvData 中对应的偏移量。第二个字节用来存储本节点 key 值长度。第三个字节用来存储本节点 value 值长度。第四个字节用来存储本节点的层高。第五个字节开始，用来存储每一层对应的下一个节点的索引值。

4、基础操作

Put、Get、Delete、Iterator 等操作均依赖于底层的跳表的基本操作实现，不再赘述。

4.1.6 持久化数据存储的实现

1、sstable 概述

如本文之前提到的，系统是 LSM 树 (Log Structured-Merge Tree) 实现，即一次写入过程并不是直接将数据持久化到磁盘文件中，而是将写操作首先写入日志文件中，其次将写操作应用在 memtable 上。当其达到 checkpoint 点 (memtable 中的数据量超过了预设的阈值)，会将当前 memtable 冻结成一个不可更改的内存数据库 (immutable memory db)，并且创建一个新的 memtable 供系统继续使用。immutable memory db 会在后台进行一次 minor compaction，即将内存数据库中的数据持久化到磁盘文件中。LSM 树设计 Minor Compaction 的目的是为了：有效地降低内存的使用率。避免日志文件过大，系统恢复时间过长。当 memory db 的数据被持久化到文件中时，存储系统将以一定规则进行文件组织，这种文件格式成为 sstable。在本文中将详细地介绍 sstable 的文件格式以及相关读写操作。

2、sstable 文件格式

(1) 物理结构

为了提高整体的读写效率，一个 sstable 文件按照固定大小进行块划分，默认每个

块的大小为 4KiB。每个 Block 中，除了存储数据以外，还会存储两个额外的辅助字段：压缩类型和 CRC 校验。压缩类型说明了 Block 中存储的数据是否进行了数据压缩，若是，采用了哪种算法进行压缩。存储系统中默认采用 Snappy 算法进行压缩。CRC 校验码是循环冗余校验校验码，校验范围包括数据以及压缩类型。

Data	Compression Type	CRC
Data	Compression Type	CRC
Data	Compression Type	CRC
Data	Compression Type	CRC

图 4.13 sstable 物理结构图

（2）逻辑结构

在逻辑上，根据功能不同，存储系统在逻辑上又将 sstable 分为：data block: 用来存储 key value 数据对；filter block: 用来存储一些过滤器相关的数据（布隆过滤器），但是若用户不指定存储系统使用过滤器，存储系统在该 block 中不会存储任何内容；meta Index block: 用来存储 filter block 的索引信息（索引信息指在该 sstable 文件中的偏移量以及数据长度）；index block: index block 中用来存储每个 data block 的索引信息；footer: 用来存储 meta index block 及 index block 的索引信息。

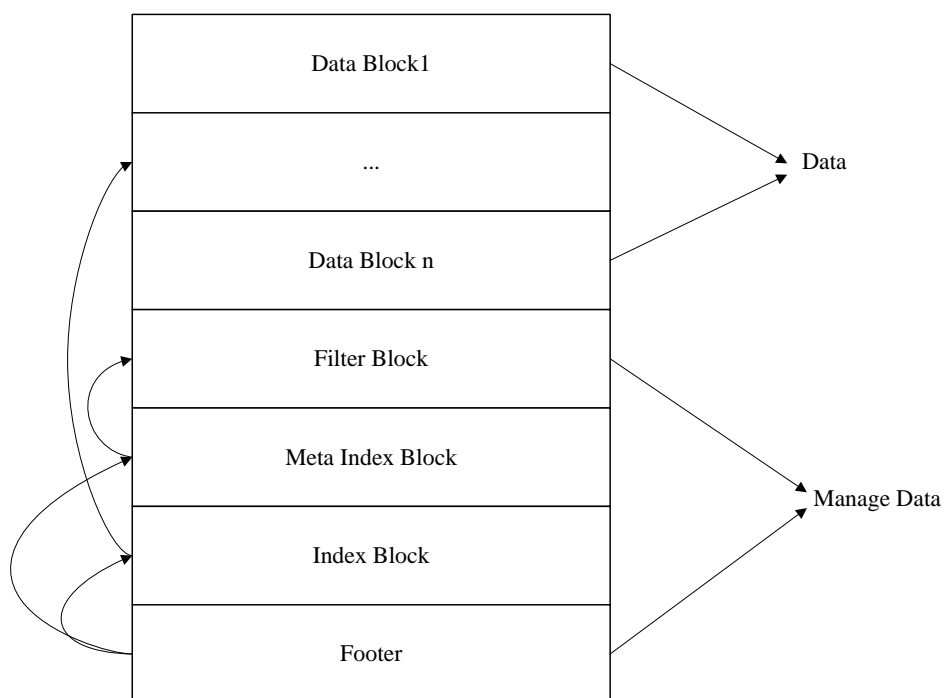


图 4.14 sstable 逻辑结构图

每个区块都会有自己的压缩信息以及 CRC 校验码信息。

（3）datablock 结构

data block 中存储的数据是存储系统中的 keyvalue 键值对。其中一个 data block 中的数据部分（不包括压缩类型、CRC 校验码）按逻辑又以下图进行划分：

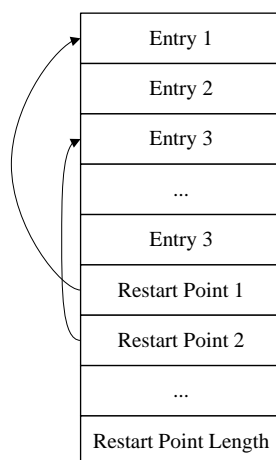


图 4.15 sstable 数据块分布图

第一部分用来存储 keyvalue 数据。由于 sstable 中所有的 keyvalue 对都是严格按序存储的，为了节省存储空间，存储系统并不会为每一对 keyvalue 对都存储完整的 key 值，而是存储与上一个 key 非共享的部分，避免了 key 重复内容的存储。每间隔若干个 keyvalue 对，将为该条记录重新存储一个完整的 key。重复该过程（默认间隔值为 16），每个重新存储完整 key 的点称之为 Restart point。存储系统设计 Restart point 的目的是在读取 sstable 内容时，加速查找的过程。由于每个 Restart point 存储的都是完整的 key 值，因此在 sstable 中进行数据查找时，可以首先利用 restart point 点的数据进行键值比较，以便于快速定位目标数据所在的区域；当确定目标数据所在区域时，再依次对区间内所有数据项逐项比较 key 值，进行细粒度地查找；该思想有点类似于跳表中利用高层数据迅速定位，底层数据详细查找的理念，降低查找的复杂度。

每个数据项格式如下图所示：

Shared key length	Unshared key length	Value length	Unshared key content	Value
-------------------	---------------------	--------------	----------------------	-------

图 4.16 sstable 条目格式图

一个 entry 分为 5 部分内容：与前一条记录 key 共享部分的长度；与前一条记录 key 不共享部分的长度；value 长度；与前一条记录 key 非共享的内容；value 内容。

三组 entry 按上图的格式进行存储。值得注意的是 restart_interval 为 2，因此每隔两个 entry 都会有一条数据作为 restart point 点的数据项，存储完整 key 值。因此 entry3 存储了完整的 key。此外，第一个 restart point 为 0（偏移量），第二个 restart point 为 16，restart point 共有两个，因此一个 datablock 数据段的末尾添加了下图所示的数据：尾部数据记录了每一个 restart point 的值，以及所有 restart point 的个数。

（4）filter block 结构

为了加快 sstable 中数据查询的效率，在直接查询 datablock 中的内容之前，存储系统首先根据 filter block 中的过滤数据判断指定的 datablock 中是否有需要查询的数据，若判断不存在，则无需对这个 datablock 进行数据查找。filter block 存储的是 data block 数据的一些过滤信息。这些过滤数据一般指代布隆过滤器的数据，用于加快查询的速度。

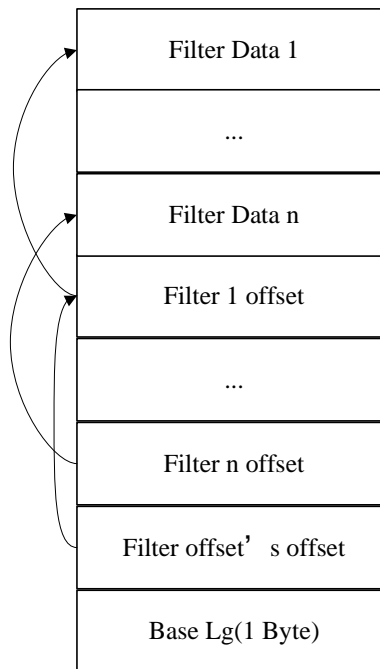


图 4.17 sstable 过滤块格式图

filter block 存储的数据主要可以分为两部分：（1）过滤数据（2）索引数据。

其中索引数据中，filter i offset 表示第 i 个 filter data 在整个 filter block 中的起始偏移量，filter offset's offset 表示 filter block 的索引数据在 filter block 中的偏移量。在读取 filter block 中的内容时，可以首先读出 filter offset's offset 的值，然后依次读取 filter i offset，根据这些 offset 分别读出 filter data。Base Lg 默认值为 11，表示每 2KB 的数据，创建一个新的过滤器来存放过滤数据。

一个 sstable 只有一个 filter block，其内存存储了所有 block 的 filter 数据。具体来说，filter_data_k 包含了所有起始位置处于 $[base*k, base*(k+1)]$ 范围内的 block 的 key 的集合的 filter 数据，按数据大小而非 block 切分主要是为了尽量均匀，以应对存在一些 block 的 key 很多，另一些 block 的 key 很少的情况。索引和 BloomFilter 等元数据可随文件一起创建和销毁，即直接存在文件里，不用加载时动态计算，不用维护更新

（5）meta index block 结构

meta index block 用来存储 filter block 在整个 sstable 中的索引信息。meta index block 只存储一条记录：该记录的 key 为：“filter.” 与过滤器名字组成的常量字符串该记录的 value 为：filter block 在 sstable 中的索引信息序列化后的内容，索引信息包括：（1）在

sstable 中的偏移量（2）数据长度。

（6）index block 结构

与 meta index block 类似，index block 用来存储所有 data block 的相关索引信息。indexblock 包含若干条记录，每一条记录代表一个 data block 的索引信息。一条索引包括以下内容：data block i 中最大的 key 值；该 data block 起始地址在 sstable 中的偏移量；该 data block 的大小。

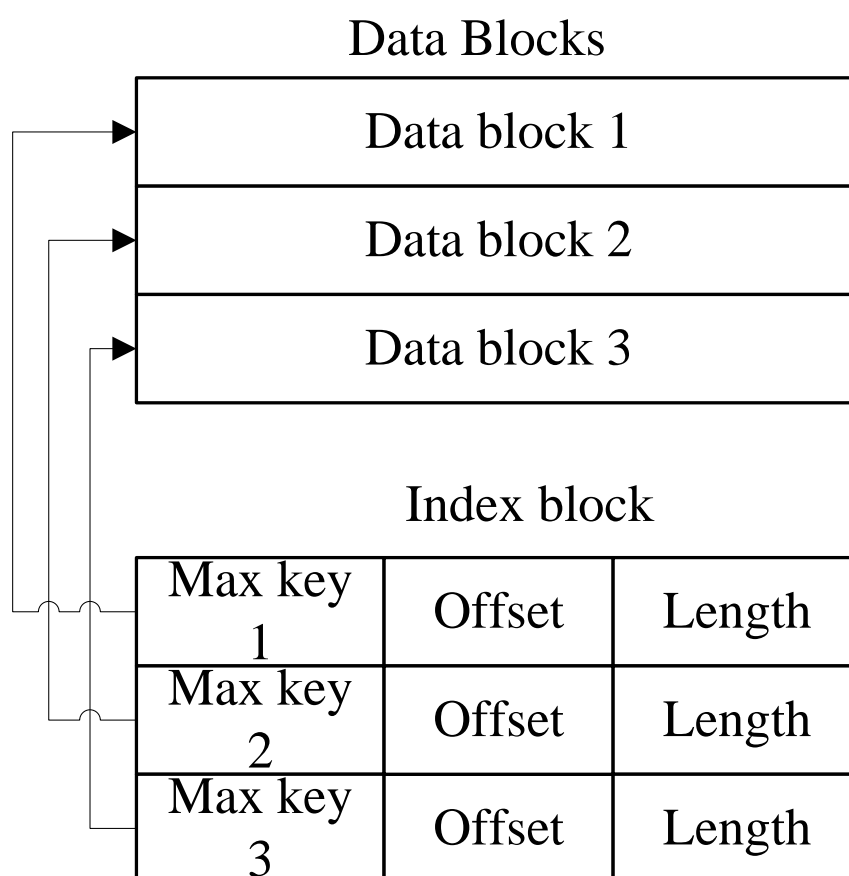


图 4.18 sstable 序号块格式图

其中，data block i 最大的 key 值还是 index block 中该条记录的 key 值。依次比较 index block 中记录信息的 key 值即可实现快速定位目标数据在哪个 data block 中。

（7）footer 结构 footer 大小固定，为 48 字节，用来存储 meta index block 与 index block 在 sstable 中的索引信息，另外尾部还会存储一个 magic word。

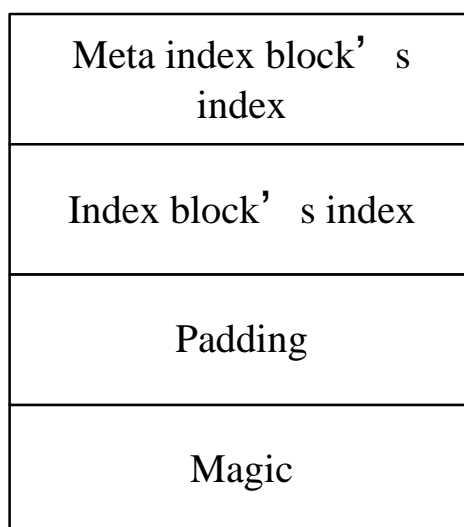


图 4.19 sstable 底部格式图

3、sstable 读写操作

在介绍完 sstable 文件具体的组织方式之后，本文再来介绍一下相关的读写操作。将首先介绍写操作。

（1）写操作

sstable 的写操作通常发生在：memory db 将内容持久化到磁盘文件中时，会创建一个 sstable 进行写入；存储系统后台进行文件 compaction 时，会将若干个 sstable 文件的内容重新组织，输出到若干个新的 sstable 文件中；对 sstable 进行写操作的数据结构为 tWriter，具体定义如下：

代码清单 4.2 tWriter

```
type tWriter struct {
    t *tOps

    fd storage.FileDesc // 文件描述符
    w  storage.Writer      // 文件系统 writer
    tw *table.Writer

    first, last []byte
}
```

主要包括了一个 sstable 的文件描述符，底层文件系统的 writer，该 sstable 中所有数据项最大最小的 key 值以及一个内嵌的 tableWriter。

一次 sstable 的写入为一次不断利用迭代器读取需要写入的数据，并不断调用 tableWriter 的 Append 函数，直至所有有效数据读取完毕，为该 sstable 文件附上元数据的过程。

该迭代器可以是一个内存数据库的迭代器，写入情景对应着上述的第一种情况；该迭代器也可以是一个 sstable 文件的迭代器，写入情景对应着上述的第二种情况；sstable 的元数据包括：文件编码，大小，最大 key 值，最小 key 值。

i.tableWriter

代码清单 4.3 tableWriter

```
// Writer is a table writer.
type Writer struct {
    writer io.Writer
    // Options
    blockSize    int // 默认是4KiB

    dataBlock    blockWriter // data块Writer
    indexBlock   blockWriter // indexBlock块Writer
    filterBlock  filterWriter // filter块Writer
    pendingBH    blockHandle
    offset       uint64
    nEntries     int // key-value 键值对个数
}
```

其中 blockWriter 与 filterWriter 表示底层的两种不同的 writer，blockWriter 负责写入 data 数据的写入，而 filterWriter 负责写入过滤数据。pendingBH 记录了上一个 dataBlock 的索引信息，当下一个 dataBlock 的数据开始写入时，将该索引信息写入 indexBlock 中。

ii.Append

一次 append 函数的主要逻辑如下：若本次写入为新 dataBlock 的第一次写入，则将上一个 dataBlock 的索引信息写入；将 keyvalue 数据写入 datablock；将过滤信息写

入 `filterBlock`；若 `datablock` 中的数据超过预定上限，则标志着本次 `datablock` 写入结束，将内容刷新到磁盘文件中。

代码清单 4.4 Append

```
func (w *Writer) Append(key, value []byte) error {
    w.flushPendingBH(key)
    // Append key/value pair to the data block.
    w.dataBlock.append(key, value)
    // Add key to the filter block.
    w.filterBlock.add(key)

    // Finish the data block if block size target reached.
    if w.dataBlock.bytesLen() >= w.blockSize {
        if err := w.finishBlock(); err != nil {
            w.err = err
            return w.err
        }
    }
    w.nEntries++
    return nil
}
```

iii.dataBlock.append

该函数将编码后的 `kv` 数据写入到 `dataBlock` 对应的 `buffer` 中，编码的格式如上文提到的数据项的格式。此外，在写入的过程中，若该数据项为 `restart` 点，则会添加相应的 `restart point` 信息。

iv.filterBlock.append

该函数将 `kv` 数据项的 `key` 值加入到过滤信息中，具体可见《布隆过滤器》。

v.finishBlock

若一个 `datablock` 中的数据超过了固定上限，则需要将相关数据写入到磁盘文件中。在写入时，需要做以下工作：封装 `dataBlock`，记录 `restart point` 的个数；若 `dataBlock` 的数据需要进行压缩（例如 `snappy` 压缩算法），则对 `dataBlock` 中的数据进行压缩；

计算 checksum; 封装 dataBlock 索引信息 (offset, length); 将 datablock 的 buffer 中的数据写入磁盘文件; 利用这段时间里维护的过滤信息生成过滤数据, 放入 filterBlock 对用的 buffer 中。

vi.Close

当迭代器取出所有数据并完成写入后, 调用 tableWriter 的 Close 函数完成最后的收尾工作:

若 buffer 中仍有未写入的数据, 封装成一个 datablock 写入; 将 filterBlock 的内容写入磁盘文件; 将 filterBlock 的索引信息写入 metaIndexBlock 中, 写入到磁盘文件; 写入 indexBlock 的数据; 写入 footer 数据; 至此为止, 所有的数据已经被写入到一个 sstable 中了, 由于一个 sstable 是作为一个 memory db 或者 Compaction 的结果原子性落地的, 因此在 sstable 写入完成之后, 将进行更为复杂的存储系统的版本更新, 将在如下的文章中继续介绍。

(2) 读操作

读操作作为写操作的逆过程, 写操作和读操作是相互转化的。下图为在一个 sstable 中查找某个数据项的流程图:

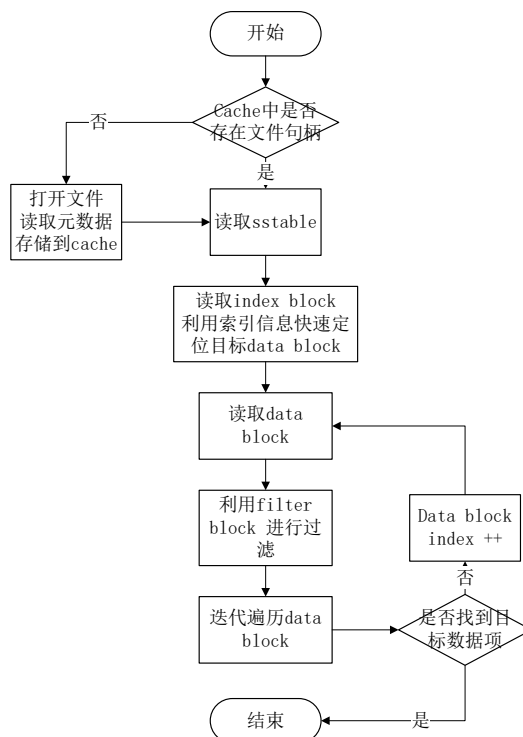


图 4.20 sstable 读过程过程图

大致流程为：

首先判断“文件句柄”cache 中是否有指定 sstable 文件的文件句柄，若存在，则直接使用 cache 中的句柄；否则打开该 sstable 文件，按规则读取该文件的元数据，将新打开的句柄存储至 cache 中；利用 sstable 中的 index block 进行快速的数据项位置定位，得到该数据项有可能存在的两个 data block；利用 index block 中的索引信息，首先打开第一个可能的 data block；利用 filter block 中的过滤信息，判断指定的数据项是否存在于该 data block 中，若存在，则创建一个迭代器对 data block 中的数据进行迭代遍历，寻找数据项；若不存在，则结束该 data block 的查找；若在第一个 data block 中找到了目标数据，则返回结果；若未查找成功，则打开第二个 data block，重复步骤 4；若在第二个 data block 中找到了目标数据，则返回结果；若未查找成功，则返回 Not Found 错误信息。

i.缓存

在存储系统中，使用 cache 来缓存两类数据：sstable 文件句柄及其元数据；data block 中的数据；因此在打开文件之前，首先判断能够在 cache 中命中 sstable 的文件句柄，避免重复读取的开销。

ii.元数据读取

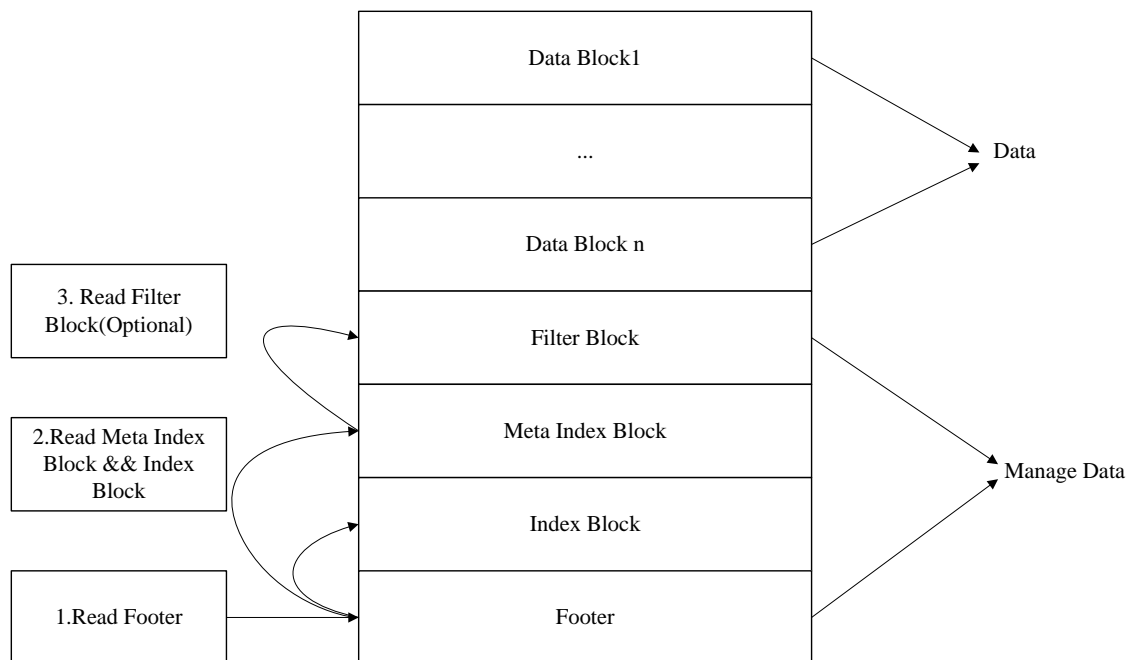


图 4.21 sstable 元数据结构图

由于 sstable 复杂的文件组织格式，因此在打开文件后，需要读取必要的元数据，才能访问 sstable 中的数据。

元数据读取的过程可以分为以下几个步骤：

读取文件的最后 48 字节的利用，即 Footer 数据；读取 Footer 数据中维护的 (1) Meta Index Block(2) Index Block 两个部分的索引信息并记录，以提高整体的查询效率；利用 meta index block 的索引信息读取该部分的内容；遍历 meta index block，查看是否存在“有用”的 filter block 的索引信息，若有，则记录该索引信息；若没有，则表示当前 sstable 中不存在任何过滤信息来提高查询效率；

iii.数据项的快速定位

sstable 中存在多个 data block，倘若依次进行“遍历”显然是不可取的。但是由于一个 sstable 中所有的数据项都是按序排列的，因此可以利用有序性已经 index block 中维护的索引信息快速定位目标数据项可能存在的 data block。

一个 index block 的文件结构示意图如下：

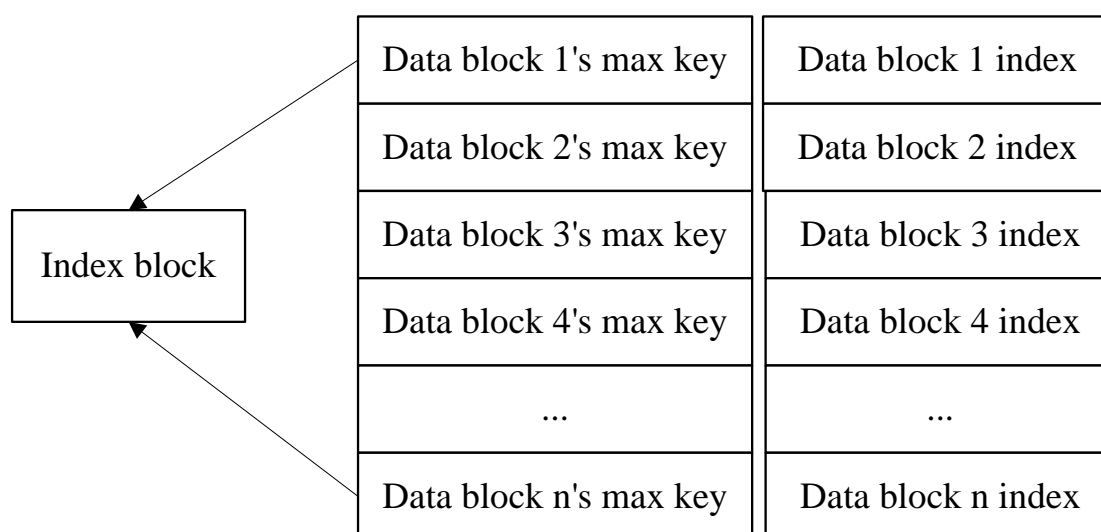


图 4.22 sstable 序号块示意图

index block 是由一系列的键值对组成，每一个键值对表示一个 data block 的索引信息。键值对的 key 为该 data block 中数据项 key 的最大值，value 为该 data block 的索引信息（offset, length）。因此若需要查找目标数据项，仅仅需要依次比较 index block 中的这些索引信息，倘若目标数据项的 key 大于某个 data block 中最大的 key

值，则该 data block 中必然不存在目标数据项。故通过这个步骤的优化，可以直接确定目标数据项落在哪个 data block 的范围区间内。值得注意的是，与 data block 一样，index block 中的索引信息同样也进行了 key 值截取，即第二个索引信息的 key 并不是存储完整的 key，而是存储与前一个索引信息的 key 不共享的部分，区别在于 data block 中这种范围的划分粒度为 16，而 index block 中为 2。也就是说，index block 连续两条索引信息会被作为一个最小的“比较单元”，在查找的过程中，若第一个索引信息的 key 小于目标数据项的 key，则紧接着会比较第三条索引信息的 key。这就导致最终目标数据项的范围区间为某两个“data block”。

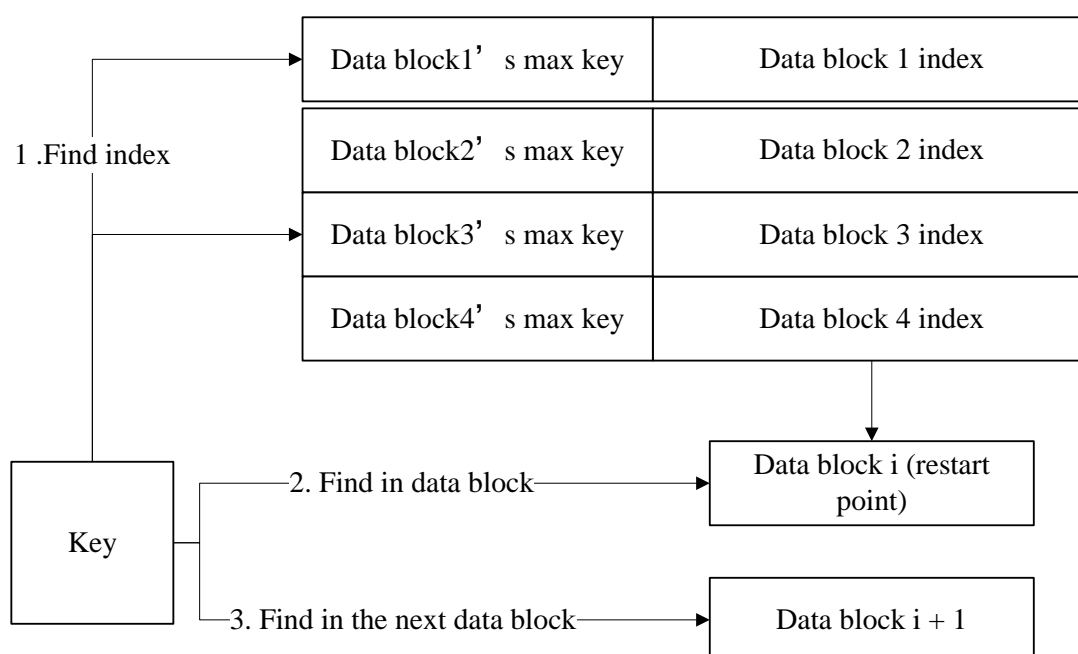


图 4.23 sstable 序号块查找图

iv.过滤 data

若 sstable 存有每一个 data block 的过滤数据，则可以利用这些过滤数据对 data block 中的内容进行判断，“确定”目标数据是否存在于 data block 中。

过滤的原理为：

若过滤数据显示目标数据不存在于 data block 中，则目标数据一定不存在于 data block 中；若过滤数据显示目标数据存在于 data block 中，则目标数据可能存在于 data block 中；因此利用过滤数据可以过滤掉部分 data block，避免发生无谓的查找。

v.查找 datablock

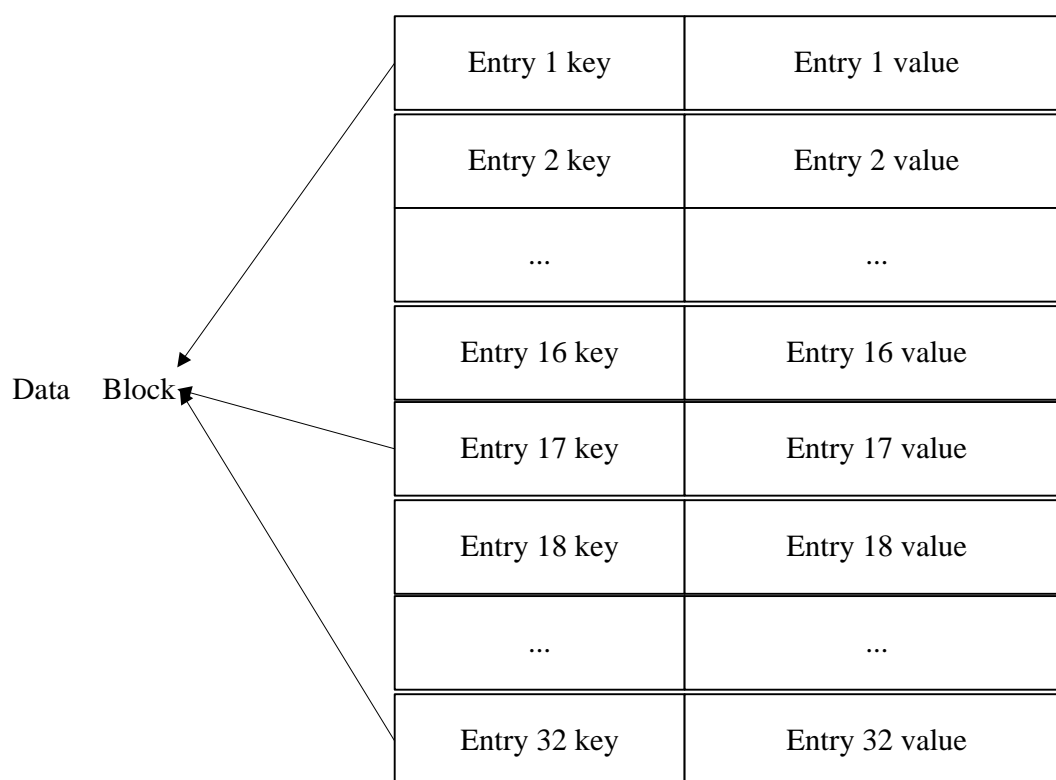


图 4.24 sstable 数据块结构格式图

在 **data block** 中查找目标数据项是一个简单的迭代遍历过程。虽然 **data block** 中所有数据项都是按序排序的，但是作者并没有采用“二分查找”来提高查找的效率，而是使用了更大的查找单元进行快速定位。与 **index block** 的查找类似，**data block** 中，以 16 条记录为一个查找单元，若 **entry 1** 的 **key** 小于目标数据项的 **key**，则下一条比较的是 **entry 17**。因此查找的过程中，利用更大的查找单元快速定位目标数据项可能存在于哪个区间内，之后依次比较判断其是否存在与 **data block** 中。可以看到，**sstable** 很多文件格式设计（例如 **restart point**，**index block**，**filter block**，**max key**）在查找的过程中，都极大地提升了整体的查找效率。

4、sstable 文件特点

（1）只读性

sstable 文件为 **compaction** 的结果原子性的产生，在其余时间是只读的。

（2）完整性

一个 **sstable** 文件，其辅助数据：

索引数据和过滤数据都直接存储于同一个文件中。当读取是需要使用这些辅助数据时，无须额外的磁盘读取；当 **sstable** 文件需要删除时，无须额外的数据删除。简要地说，辅助数据随着文件一起创建和销毁。

（3）并发访问友好性

由于 **sstable** 文件具有只读性，因此不存在同一个文件的读写冲突。存储系统采用引用计数维护每个文件的引用情况，当一个文件的计数值大于 0 时，对此文件的删除动作会等到该文件被释放时才进行，因此实现了无锁情况下的并发访问。

（4）Cache 一致性

sstable 文件为只读的，因此 **cache** 中的数据永远于 **sstable** 文件中的数据保持一致。

4.1.7 缓存系统的实现

缓存对于一个数据库读性能的影响十分巨大，倘若存储系统的每一次读取都会发生一次磁盘的 IO，那么其整体效率将会非常低下。存储系统中使用了一种基于 **LRUCache** 的缓存机制，用于缓存：已打开的 **sstable** 文件对象和相关元数据；**sstable** 中的 **dataBlock** 的内容；使得在发生读取热数据时，尽量在 **cache** 中命中，避免 IO 读取。在介绍如何缓存、利用这些数据之前，本文首先介绍一下存储系统使用的 **cache** 是如何实现的。

1、Cache 结构

存储系统中使用的 **cache** 是一种 **LRUcache**，其结构由两部分内容组成 **Hash table**：用来存储数据；**LRU**：用来维护数据项的新旧信息。

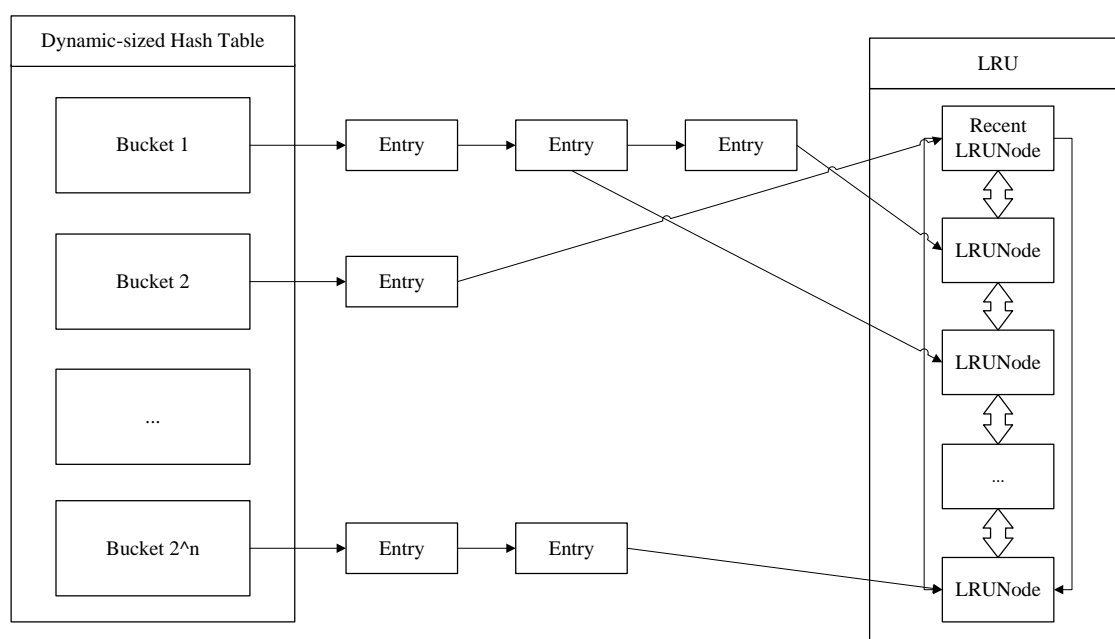


图 4.25 sstable 缓存架构图

其中 Hash table 是基于 Yujie Liu 等人的论文《Dynamic-Sized Nonblocking Hash Table》实现的，用来存储数据。由于 hash 表一般需要保证插入、删除、查找等操作的时间复杂度为 $O(1)$ 。当 hash 表的数据量增大时，为了保证这些操作仍然保有较为理想的操作效率，需要对 hash 表进行 **resize**，即改变 hash 表中 bucket 的个数，对所有的数据进行重散列。基于该文章实现的 hash table 可以实现 **resize** 的过程中不阻塞其它并发的读写请求。LRU 中则根据 **Least Recently Used** 原则进行数据新旧信息的维护，当整个 cache 中存储的数据容量达到上限时，便会根据 LRU 算法自动删除最旧的数据，使得整个 cache 的存储容量保持一个常量。

2、Dynamic-sized NonBlocking Hash table

在 hash 表进行 **resize** 的过程中，保持 **Lock-Free** 是一件非常困难的事。一个 hash 表通常由若干个 bucket 组成，每一个 bucket 中会存储若干条被散列至此的数据项。当 hash 表进行 **resize** 时，需要将“旧”桶中的数据读出，并且重新散列至另外一个“新”桶中。假设这个过程不是一个原子操作，那么会导致此刻其它的读、写请求的结果发生异常，甚至导致数据丢失的情况发生。因此，liu 等人提出了一个新颖的概念：一个 bucket 的数据是可以冻结的。这个特点极大地简化了 hash 表在 **resize** 过程中在不同 bucket 之间转移数据的复杂度。

（1）散列

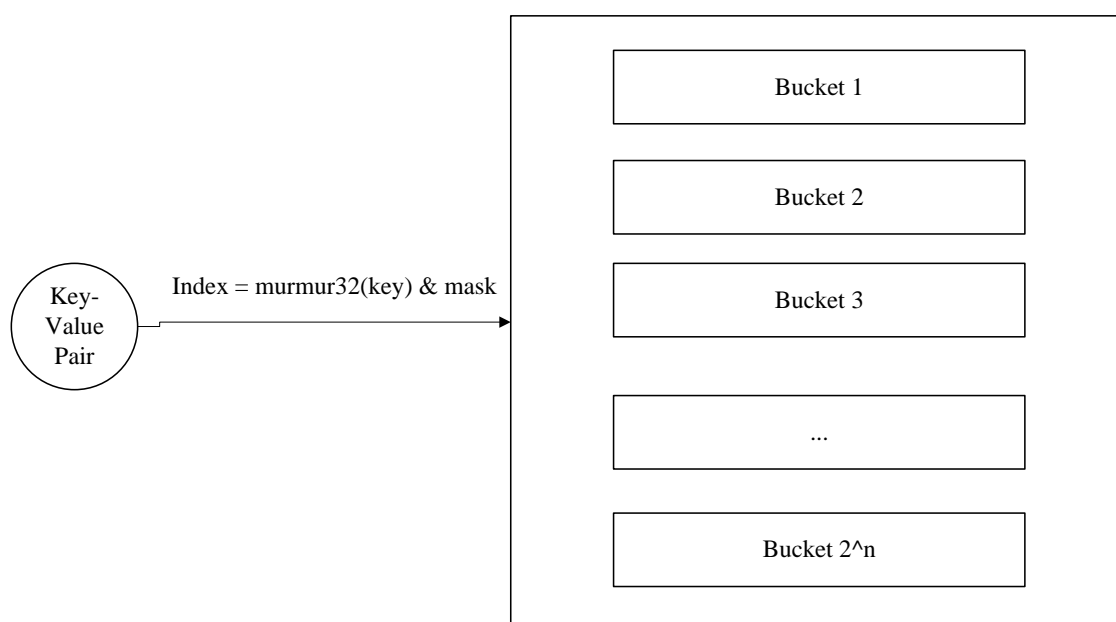


图 4.26 sstable 缓存查找图

该哈希表的散列与普通的哈希表一致，都是借助散列函数，将用户需要查找、更改的数据散列到某一个哈希桶中，并在哈希桶中进行操作。

由于一个哈希桶的容量是有限的（一般不大于 32 个数据），因此在哈希桶中进行插入、查找的时间复杂度可以视为是常量的。

（2）扩大

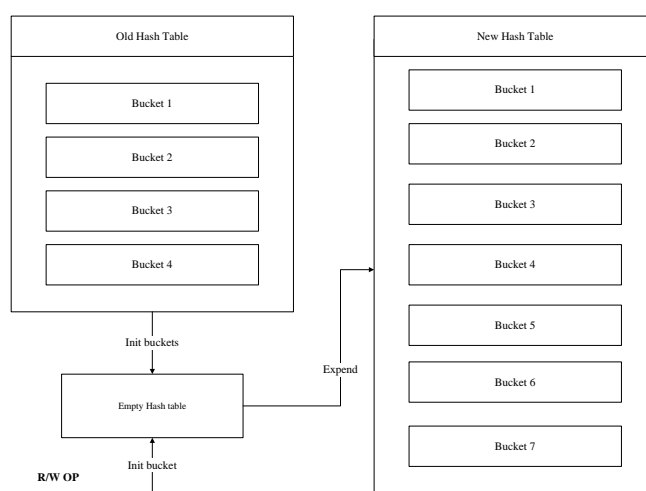


图 4.27 sstable 缓存扩展图

当 `cache` 中维护的数据量太大时，会发生哈希表扩张的情况。以下两种情况是为“`cache` 中维护的数据量过大”：

整个 `cache` 中，数据项（`node`）的个数超过预定的阈值（默认初始状态下哈希桶的个数为 16 个，每个桶中可存储 32 个数据项，即总量的阈值为哈希桶个数乘以每个桶的容量上限）；当 `cache` 中出现了数据不平衡的情况。当某些桶的数据量超过了 32 个数据，即被视作数据发生散列不平衡。当这种不平衡累积值超过预定的阈值（128）个时，就需要进行扩张；一次扩张的过程为：

计算新哈希表的哈希桶个数（扩大一倍）；创建一个空的哈希表，并将旧的哈希表（主要为所有哈希桶构成的数组）转换一个“过渡期”的哈希表，表中的每个哈希桶都被“冻结”；后台利用“过渡期”哈希表中的“被冻结”的哈希桶信息对新的哈希表进行内容构建；值得注意的是，在完成新的哈希表构建的整个过程中，哈希表并不是拒绝服务的，所有的读写操作仍然可以进行。哈希表扩张过程中，最小的封锁粒度为哈希桶级别。当有新的读写请求发生时，若被散列之后得到的哈希桶仍然未构建完成，则“主动”进行构建，并将构建后的哈希桶填入新的哈希表中。后台进程构建到该桶时，发现已经被构建了，则无需重复构建。

因此如上图所示，哈希表扩张结束，哈希桶的个数增加了一倍，于此同时仍然可以对外提供读写服务，仅仅需要哈希桶级别的封锁粒度就可以保证所有操作的一致性跟原子性。

构建哈希桶当哈希表扩张时，构建一个新的哈希桶其实就是将一个旧哈希桶中的数据拆分成两个新的哈希桶。拆分的规则很简单。由于一次散列的过程为：利用散列函数对数据项的 `key` 值进行计算；将第一步得到的结果取哈希桶个数的余，得到哈希桶的 `ID`；因此拆分时仅需要将数据项 `key` 的散列值对新的哈希桶个数取余即可。

（3）缩小

当哈希表中数据项的个数少于哈希桶的个数时，需要进行收缩。收缩时，哈希桶的个数变为原先的一半，2 个旧哈希桶的内容被合并成一个新的哈希桶，过程与扩张类似，在这里不展开详述。

3、LRU

除了利用哈希表来存储数据以外，存储系统还利用 LRU 来管理数据。存储系统中，LRU 利用一个双向循环链表来实现。每一个链表项称之为 `LRUNode`。

代码清单 4.5 lruNode

```

type lruNode struct {
    n    *Node // customized node
    h    *Handle
    ban  bool

    next, prev *lruNode
}
    
```

一个 **LRUNode** 除了维护一些链表中前后节点信息以外，还存储了一个哈希表中数据项的指针，通过该指针，当某个节点由于 **LRU** 策略被驱逐时，从哈希表中“安全的”删除数据内容。

LRU 提供了以下几个接口：

Promote 若一个 **hash** 表中的节点是第一次被创建，则为该节点创建一个 **LRUNode**，并将 **LRUNode** 置于链表的头部，表示为最新的数据；若一个 **hash** 表中的节点之前就有相关的 **LRUNode** 存在与链表中，将该 **LRUNode** 移至链表头部；若因为新增加一个 **LRU** 数据，导致超出了容量上限，就需要根据策略清除部分节点。

Ban 将 **hash** 表节点对应的 **LRUNode** 从链表中删除，并“尝试”从哈希表中删除数据。由于该哈希表节点的数据可能被其它线程正在使用，因此需要查看该数据的引用计数，只有当引用计数为 0 时，才可以真正地从哈希表中进行删除。

4、缓存数据

存储系统利用上述的 **cache** 结构来缓存数据。其中：

cache：来缓存已经被打开的 **sstable** 文件句柄以及元数据（默认上限为 500 个）；
bcache：来缓存被读过的 **sstable** 中 **dataBlock** 的数据（默认上限为 8MB）；当一个 **sstable** 文件需要被打开时，首先从 **cache** 中寻找是否已经存在相关的文件句柄，若存在则无需重复打开；若不存在，则从打开相关文件，并将 (1) **indexBlock** 数据，(2) **metaIndexBlock** 数据等相关元数据进行预读。

4.1.8 布隆过滤器的实现

Bloom Filter 是一种空间效率很高的随机数据结构，它利用位数组很简洁地表示一个集合，并能判断一个元素是否属于这个集合。**Bloom Filter** 的这种高效是有一定代价

的：在判断一个元素是否属于某个集合时，有可能会把不属于这个集合的元素误认为属于这个集合（false positive）。因此，Bloom Filter 不适合那些“零错误”的应用场合。而在能容忍低错误率的应用场合下，Bloom Filter 通过极少的错误换取了存储空间的极大节省。

存储系统中利用布隆过滤器判断指定的 key 值是否存在于 sstable 中，若过滤器表示不存在，则该 key 一定不存在，由此加快了查找的效率。

1、结构

bloom 过滤器底层是一个位数组，初始时每一位都是 0

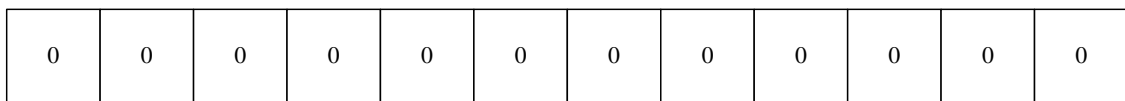


图 4.28 sstable 布隆过滤器示意图 1

当插入值 x 后，分别利用 k 个哈希函数（图中为 3）利用 x 的值进行散列，并将散列得到的值与 bloom 过滤器的容量进行取余，将取余结果所代表的那一位值置为 1。

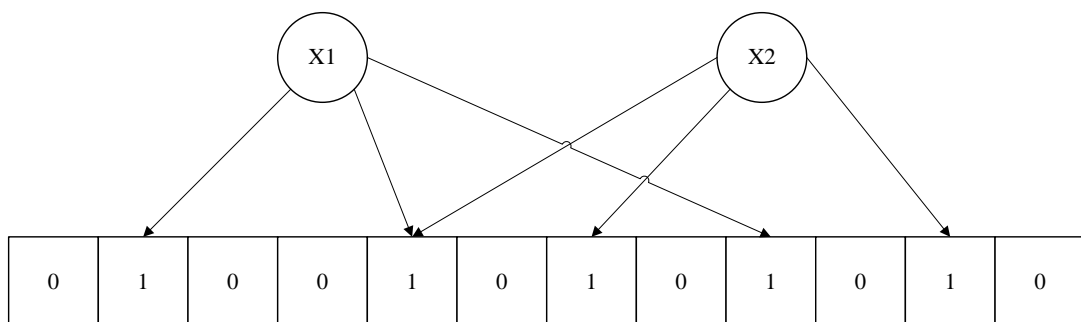


图 4.29 sstable 布隆过滤器示意图 2

一次查找过程与一次插入过程类似，同样利用 k 个哈希函数对所需要查找的值进行散列，只有散列得到的每一个位的值均为 1，才表示该值“有可能”真正存在；反之若有任意一位的值为 0，则表示该值一定不存在。例如 y1 一定不存在；而 y2 可能存在。

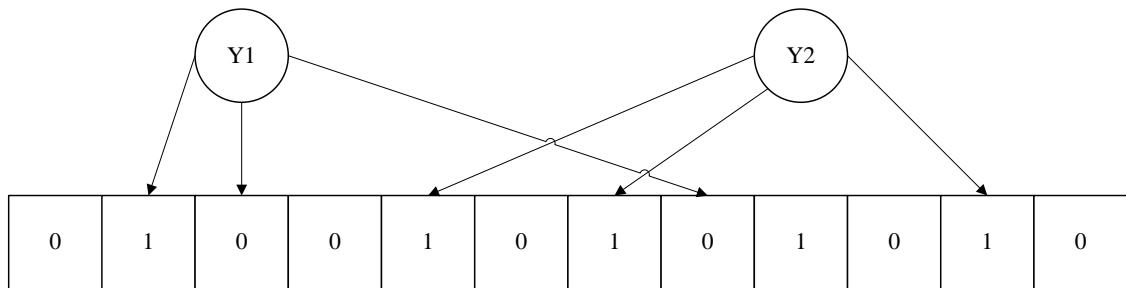


图 4.30 sstable 布隆过滤器示意图 3

2、数学结论

<http://blog.csdn.net/jiaomeng/article/details/1495500> 该文中从数学的角度阐述了布隆过滤器的原理，以及一系列的数学结论。

首先，与布隆过滤器准确率有关的参数有：

哈希函数的个数 k ；布隆过滤器位数组的容量 m ；布隆过滤器插入的数据数量 n ；主要的数学结论有：为了获得最优的准确率，当 $k = \ln 2 * (m/n)$ 时，布隆过滤器获得最优的准确性；在哈希函数的个数取到最优时，要让错误率不超过 ϵ ， m 至少需要取到最小值的 1.44 倍；

3、代码实现

存储系统中的布隆过滤器实现较为简单，以 go 存储系统为例，有关的代码在 `filter/bloom.go` 中。定义如下，bloom 过滤器只是一个 int 数字。

代码清单 4.6 tFile

```
type bloomFilter int
```

创建一个布隆过滤器时，只需要指定为每个 key 分配的位数即可，如结论 2 所示，只要该值 (m/n) 大于 1.44 即可，一般可以取 10。

代码清单 4.7 NewBloomFilter

```
func NewBloomFilter(bitsPerKey int) Filter {
    return bloomFilter(bitsPerKey)
}
```

创建一个 generator, 这一步中需要指定哈希函数的个数 k , 可以看到 $k = f * \ln 2$, 而 $f = m/n$, 即数学结论 1。返回的 generator 中可以添加新的 key 信息, 调用 generate 函数时, 将所有的 key 构建成一个位数组写在指定的位置。

代码清单 4.8 NewGenerator

```
func (f bloomFilter) NewGenerator() FilterGenerator {
    // Round down to reduce probing cost a little bit.
    k := uint8(f * 69 / 100) // 0.69 ≈ ln(2)
    if k < 1 {
        k = 1
    } else if k > 30 {
        k = 30
    }
    return &bloomFilterGenerator{
        n: int(f),
        k: k,
    }
}
```

generator 主要有两个函数:

Add

Generate

Add 函数中, 只是简单地将 key 的哈希散列值存储在一个整型数组中

代码清单 4.9 Add

```
func (g *bloomFilterGenerator) Add(key []byte) {
    // Use double-hashing to generate a sequence of hash values.
    // See analysis in [Kirsch,Mitzenmacher 2006].
    g.keyHashes = append(g.keyHashes, bloomHash(key))
}
```

Generate 函数中, 将之前一段时间内所有添加的 key 信息用来构建一个位数组, 该位数组中包含了所有 key 的存在信息。位数组的大小为用户指定的每个 key 所分配的位

数乘以 key 的个数。位数组的最末尾用来存储 k 的大小。

代码清单 4.10 Generate

```

func (g *bloomFilterGenerator) Generate(b Buffer) {
    // Compute bloom filter size (in both bits and bytes)
    // len(g.keyHashes) 可以理解为n, g.n可以理解为m/n
    // nBits可以理解为m
    nBits := uint32(len(g.keyHashes) * g.n)
    // For small n, we can see a very high false positive rate.
    // Fix it
    // by enforcing a minimum bloom filter length.
    if nBits < 64 {
        nBits = 64
    }
    nBytes := (nBits + 7) / 8
    nBits = nBytes * 8

    dest := b.Alloc(int(nBytes) + 1)
    dest[nBytes] = g.k

    for _, kh := range g.keyHashes {
        // Double Hashing
        delta := (kh >> 17) | (kh << 15) // Rotate right 17
        bits
        for j := uint8(0); j < g.k; j++ {
            bitpos := kh % nBits
            dest[bitpos/8] |= (1 << (bitpos % 8))
            kh += delta
        }
    }

    g.keyHashes = g.keyHashes[:0]
}

```

Contain 函数用来判断指定的 key 是否存在。

代码清单 4.11 tFile

```

func (f bloomFilter) Contains(filter, key []byte) bool {
    nBytes := len(filter) - 1
    if nBytes < 1 {
        return false
    }
    nBits := uint32(nBytes * 8)
    // Use the encoded k so that we can read filters generated
    by
    // bloom filters created using different parameters.
    k := filter[nBytes]
    if k > 30 {
        // Reserved for potentially new encodings for short
        bloom filters.
        // Consider it a match.
        return true
    }
    kh := bloomHash(key)
    delta := (kh >> 17) | (kh << 15) // Rotate right 17 bits
    for j := uint8(0); j < k; j++ {
        bitpos := kh % nBits
        if (uint32(filter[bitpos/8]) & (1 << (bitpos % 8))) == 0
        {
            return false
        }
        kh += delta
    }
    return true
}

```

4.1.9 数据压缩系统的实现

Compaction 是存储系统最为复杂的过程之一，同样也是存储系统的性能瓶颈之一。其本质是一种内部数据重合整合的机制，同样也是一种平衡读写速率的有效手段，因此在下文中，首先介绍下存储系统中设计 compaction 的原由，再来介绍下 compaction 的具体过程。

1、Compaction 作用

（1）数据持久化

存储系统是典型的 LSM 树实现，因此需要对内存中的数据进行持久化。一次内存数据的持久化过程，在存储系统中称为 Minor Compaction。一次 minor compaction 的产出是一个 0 层的 sstable 文件，其中包含了所有的内存数据。但是若干个 0 层文件中是可能存在数据 overlap 的。

（2）提高读写效率

正如前面的文章提到，存储系统是一个写效率十分高的存储引擎，存储的过程非常简单，只需要一次顺序的文件写和一个时间复杂度为 $O(\log n)$ 的内存操作即可。相比之下，存储系统的读操作就复杂不少。首先一到两次读操作需要进行一个复杂度为 $O(\log n)$ 的查询操作。若没有在内存中命中数据，则需要在按照数据的新旧程度在 0 层文件中依次进行查找遍历。由于 0 层文件中可能存在 overlap，因此在最差情况下，可能需要遍历所有的文件。假设存储系统中就是以这样的方式进行数据维护，那么随着运行时间的增长，0 层的文件个数会越来越多，在最差的情况下，查询一个数据需要遍历所有的数据文件，这显然是不可接受的。因此存储系统设计了一个 *Major Compaction* 的过程，将 0 层中的文件合并为若干个没有数据重叠的 1 层文件。对于没有数据重叠的文件，一次查找过程就可以进行优化，最多只需要一个文件的遍历即可完成。因此，存储系统设计 compaction 的目的之一就是为了提高读取的效率。

（3）平衡读写差异

有了 minor compaction 和 major compaction，所有的数据在后台都会被规定的次序进行整合。但是一次 major compaction 的过程其本质是一个多路归并的过程，既有大量的磁盘读开销，也有大量的磁盘写开销，显然这是一个严重的性能瓶颈。但是当用户写入的速度始终大于 major compaction 的速度时，就会导致 0 层的文件数量还是不断上升，用户的读取效率持续下降。所以存储系统中规定：当 0 层文件数量超过 SlowdownTrigger

时，写入的速度主要减慢；当 0 层文件数量超过 `PauseTrigger` 时，写入暂停，直至 `Major Compaction` 完成；故 `compaction` 也可以起到平衡读写差异的作用。

（4）整理数据

存储系统的每一条数据项都有一个版本信息，标识着这条数据的新旧程度。这也就意味着同样一个 `key`，在存储系统中可能存在着多条数据项，且每个数据项包含了不同版本的内容。为了尽量减少数据集所占用的磁盘空间大小，存储系统在 `major compaction` 的过程中，对不同版本的数据项进行合并。

2、Compaction 过程

由上述所示，`compaction` 分为两类：`minor compaction` 和 `major compaction`，这两类 `compaction` 负责在不同的场景下进行不同的数据整理。

（1）Minor Compaction

一次 `minor compaction` 非常简单，其本质就是将一个内存数据库中的所有数据持久化到一个磁盘文件中。

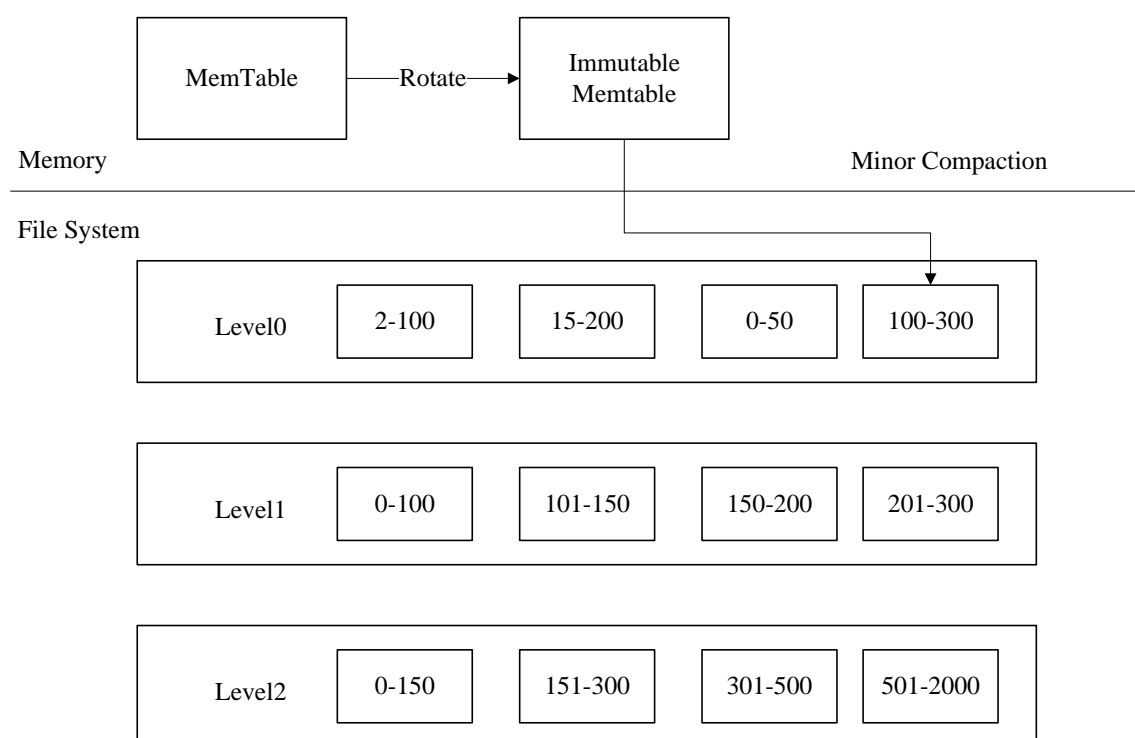


图 4.31 sstable 结构镜像归并

每次 `minor compaction` 结束后，都会生成一个新的 `sstable` 文件，也意味着存储系统

的版本状态发生了变化，会进行一个版本的更替。有关版本控制的内容，将在接下去一篇文章中详细展开。值得注意的是，minor compaction 是一个时效性要求非常高的过程，要求其在尽可能短的时间内完成，否则就会堵塞正常的写入操作，因此 minor compaction 的优先级高于 major compaction。当进行 minor compaction 的时候有 major compaction 正在进行，则会首先暂停 major compaction。

（2）Major Compaction

相比于 minor compaction，major compaction 就会复杂地多。首先看一下一次 major compaction 的示意图。

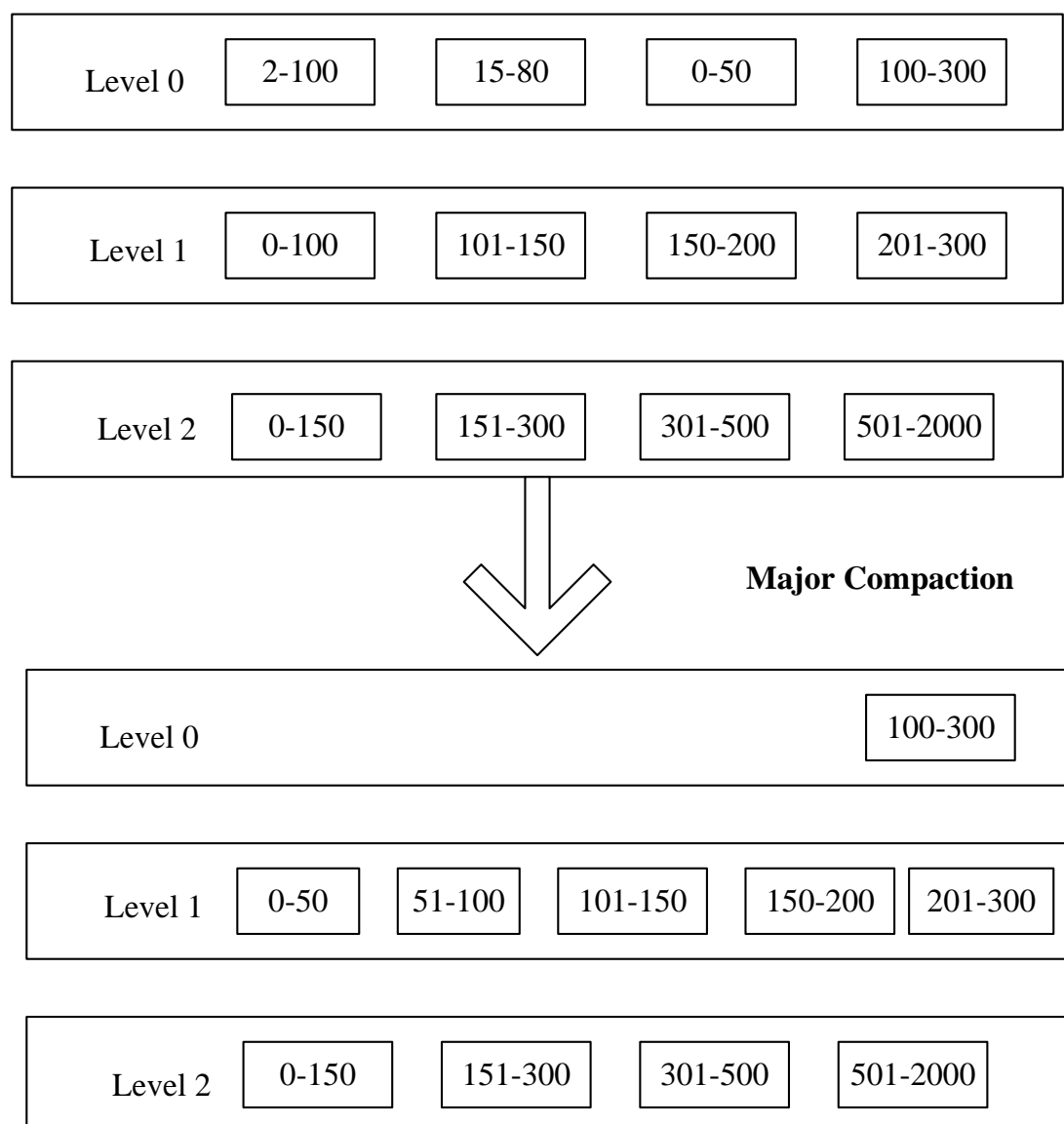


图 4.32 sstable 结构主归并

0 层中浅蓝色的三个 sstable 文件，加上 1 层中的绿色的 sstable 文件，四个文件进行了合并，输出成两个按序组织的新的 1 层 sstable 文件进行替换。

i. 条件

何时会触发存储系统进行 major compaction 呢。总结地来说为以下三个条件：当 0 层文件数超过预定的上限（默认为 4 个）；当 level i 层文件的总大小超过 $(10^i)\text{MB}$ ；当某个文件无效读取的次数过多；第 0 层文件数据大小限制：由于 compaction 的其中一个目的是为了提高读取的效率，因此存储系统不允许 0 层存在过多的文件数，一旦超过了上限值，即可进行 major compaction。非 0 层文件数据大小限制：对于 level i ($i \geq 1$) 的情况来说，一个读取最多只会访问一个 sstable 文件，因此，本身对于读取效率的影响不会太大。针对于这部分数据发生 compaction 的条件，从提升读取效率转变成了降低 compaction 的 IO 开销。假设存储系统的合并策略只有第一条，那么会导致 1 层文件的个数越来越多或者总的的数据量越来越大，而通常一次合并中，0 层文件 key 的取值范围是很大的，导致每一次 0 层文件与 1 层文件进行合并时，1 层文件输入文件的总数据量非常庞大。所以不仅需要控制 0 层文件的个数，同样，每一层文件的总大小同样需要进行控制，使得每次进行 compaction 时，IO 开销尽量保持常量。故存储系统规定，1 层文件总大小上限为 10MB，2 层为 100MB，依次类推，最高层（7 层）没有限制。以上两个机制能够保证随着合并的进行，数据是严格下沉的，但是仍然存在一个问题。假设 0 层文件完成合并之后，1 层文件同时达到了数据上限，同时需要进行合并。更加糟糕的是，在最差的情况下，0-n 层的文件同时达到了合并的条件，每一层都需要进行合并。

其中一种优化机制是：source 层的文件个数只有一个；source 层文件与 source+1 层文件没有重叠；source 层文件与 source+2 层的文件重叠部分不超过 10 个文件；当满足这几个条件时，可以将 source 层的该文件直接移至 source+1 层。

但是该条件非常苛刻，还是无法解决上述问题。为了避免可能存在这种“巨大”的合并开销，存储系统引入了第三个机制：“错峰合并”。那么（1）如何找寻这种适合错峰合并的文件（2）以及如果判断哪个时机是适合进行错峰合并的呢？对于问题（1），一个文件一次查询的开销为 10ms，若某个文件的查询次数过多，且查询在该文件中不命中，那么这种行为就可以视为无效的查询开销，这种文件就可以进行错峰合并。对于问题（2），对于一个 1MB 的文件，对其合并的开销为 25ms。因此当一个文件

1MB 的文件无效查询超过 25 次时，便可以对其进行合并。对于一个 1MB 的文件，其合并开销为（1）source 层 1MB 的文件读取，（2）source+1 层 10-12MB 的文件读取（3）source+1 层 10-12MB 的文件写入。总结 25MB 的文件 IO 开销，除以 100MB / s 的文件 IO 速度，估计开销为 25ms。

ii. 采样探测

在每个 sstable 文件的元数据中，还有一个额外的字段 seekLeft，默认为文件的大小除以 16KB。存储系统在正常的数据访问时，会顺带进行采样探测。正常的数据访问包括（1）用户直接调用 Get 接口（2）用户使用迭代器进行访问。采样的规则：记录本次访问的第一个 sstable 文件。若在该文件中访问命中，则不做任何处理；若在该文件中访问不命中，则对该文件的 seekLeft 标志做减一操作。知道某一个文件的 seekLeft 标志减少到 0 时，触发对该文件的错峰合并。故以上三种机制可以保障每次进行 compaction 的时候，总体开销不会呈现上升趋势。

iii. 过程

整个 compaction 可以简单地分为以下几步：寻找合适的输入文件；根据 key 重叠情况扩大输入文件集合；多路合并；积分计算。

A. 寻找输入文件：

不同情况下发起的合并动作，其初始的输入文件不同。对于 level 0 层文件数过多引发的合并场景或由于 level i 层文件总量过大的合并场景，采用轮转的方法选择起始输入文件，记录了上一次该层合并的文件的最大 key，下一次则选择在此 key 之后的首个文件。对于错峰合并，起始输入文件则为该查询次数过多的文件。

B. 扩大输入文件集合

该过程如下：

红星标注的为起始输入文件；在 level i 层中，查找与起始输入文件有 key 重叠的文件，如图中红线所标注，最终构成 level i 层的输入文件；利用 level i 层的输入文件，在 level i+1 层找寻有 key 重叠的文件，结果为绿线标注的文件，构成 level i, i+1 层的输入文件；最后利用两层的输入文件，在不扩大 level i+1 输入文件的前提下，查找 level i 层的有 key 重叠的文件，结果为蓝线标准的文件，构成最终的输入文件；

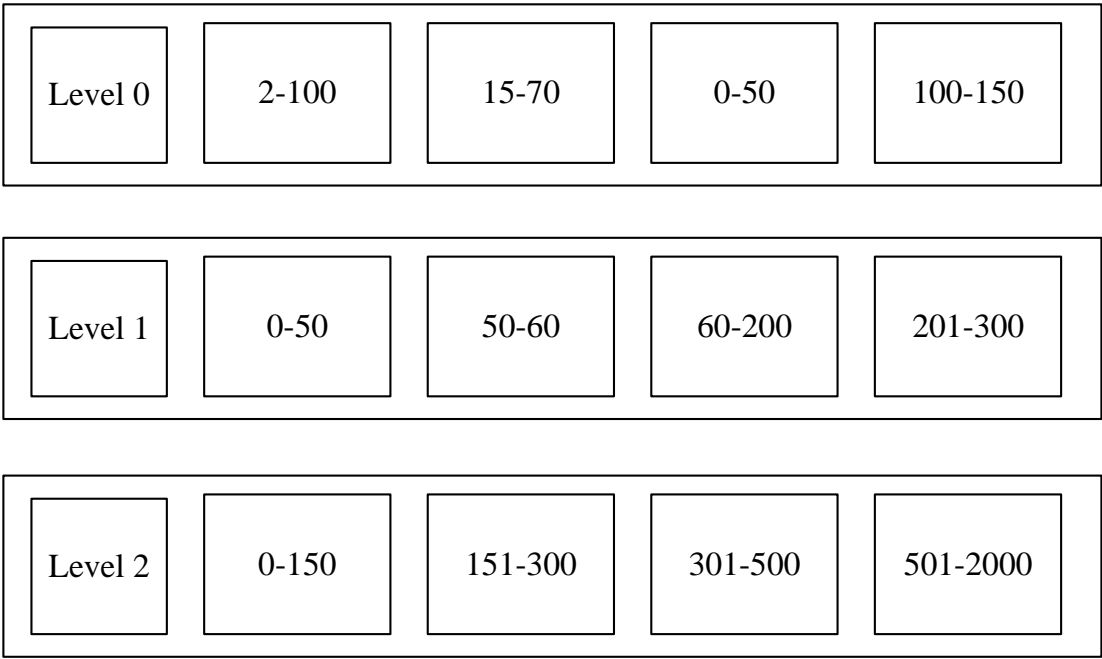


图 4.33 sstable 归并扩展图

C.多路合并：

多路合并的过程比较简单，即将 level i 层的文件，与 level i+1 层的文件中的数据项，按序整理之后，输出到 level i+1 层的若干个新文件中，即合并完成。

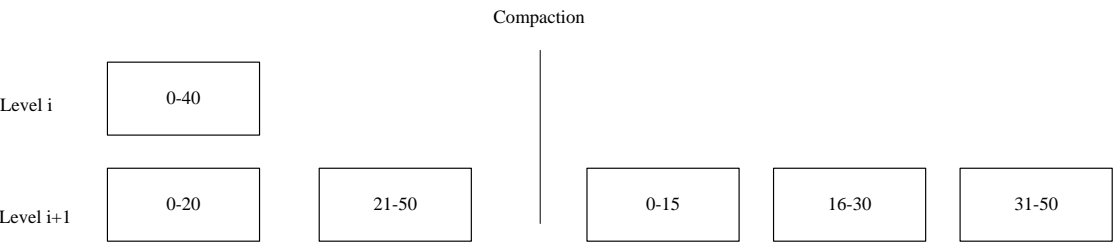


图 4.34 sstable 表合并图

注意在整理的过程中，需要将冗余的数据进行清理，即同一条数据的多个版本信息，只保留最新的那一份。但是要注意，某些仍然在使用的旧版本的数据，在此时不能立刻删除，而得等到用户使用结束，释放句柄后，根据引用计数来进行清除。

D.积分计算

每一次 compaction 都会消除若干 source 层的旧文件，新增 source+1 层的新文

件，因此触发进行合并的条件状态可能也发生了变化。故在存储系统中，使用了计分牌来维护每一层文件的文件个数及数据总量信息，来挑选出下一个需要进行合并的层数。计分的规则很简单：对于 0 层文件，该层的分数为文件总数 / 4；对于非 0 层文件，该层的分数为文件数据总量 / 数据总量上限；将得分最高的层数记录，若该得分超过 1，则为下一次进行合并的层数。

3、用户行为

由于存储系统内部进行 **compaction** 时有 **trivial move** 优化，且根据内部的文件格式组织，用户在使用存储系统时，可以尽量将大批量需要写入的数据进行预排序，利用空间局部性，尽量减少多路合并的 IO 开销。

4.1.10 版本控制的实现

存储系统每次新生成 **sstable** 文件，或者删除 **sstable** 文件，都会从一个版本升级成另外一个版本。换句话说，每次 **sstable** 文件的更替对于存储系统来说是一个最小的操作单元，具有原子性。版本控制对于存储系统来说至关重要，是保障数据正确性的重要机制。在本文中，将着重从版本数据的格式以及版本升级的过程进行展开。

1、Manifest

manifest 文件专用于记录版本信息。存储系统采用了增量式的存储方式，记录每一个版本相较于上一个版本的变化情况。一个 **Manifest** 文件中，包含了多条 **Session Record**。一个 **Session Record** 记录了从上一个版本至该版本的变化情况。

变化情况大致包括：（1）新增了哪些 **sstable** 文件（2）删除了哪些 **sstable** 文件（由于 **compaction** 导致）（3）最新的 **journal** 日志文件标号等借助这个 **Manifest** 文件，存储系统启动时，可以根据一个初始的版本状态，不断地应用这些版本改动，使得系统的版本信息恢复到最近一次使用的状态。

一个 **Manifest** 文件的格式示意图如下所示：

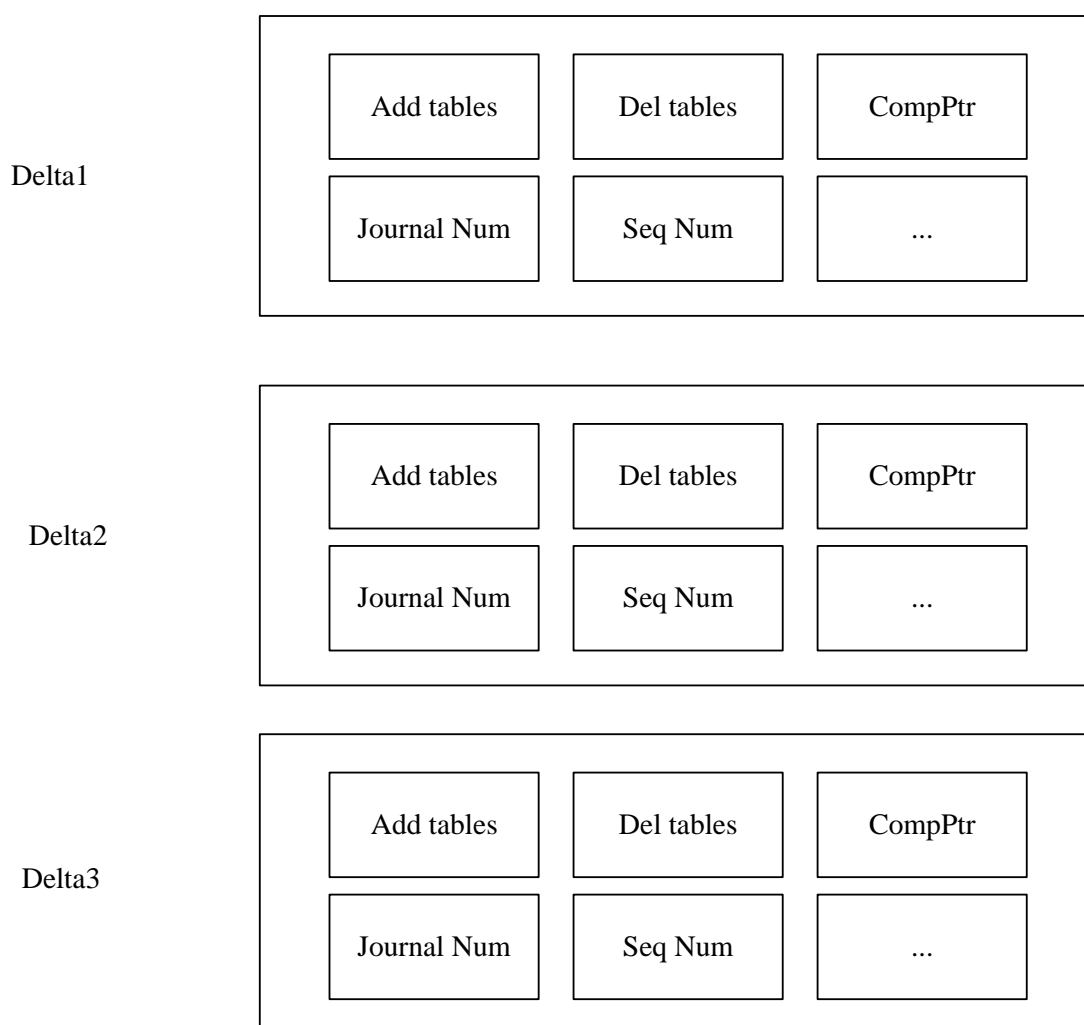


图 4.35 元数据图

一个 Manifest 内部包含若干条 Session Record，其中第一条 Session Record 记载了当时存储系统的全量版本信息，其余若干条 Session Record 仅记录每次更迭的变化情况。因此，每个 manifest 文件的第一条 Session Record 都是一个记录点，记载了全量的版本信息，可以作为一个初始的状态进行版本恢复。一个 Session Record 可能包含以下字段：

Comparer 的名称；最新的 journal 文件编号；下一个可以使用的文件编号；数据库已经持久化数据项中最大的 sequence number；新增的文件信息；删除的文件信息；compaction 记录信息。

2、Commit

每当（1）完成一次 major compaction 整理内部数据或者（2）通过 minor compaction

或者重启阶段的日志重放新生成一个 0 层文件，都会触发存储系统进行一个版本升级。、一次版本升级的过程如下：

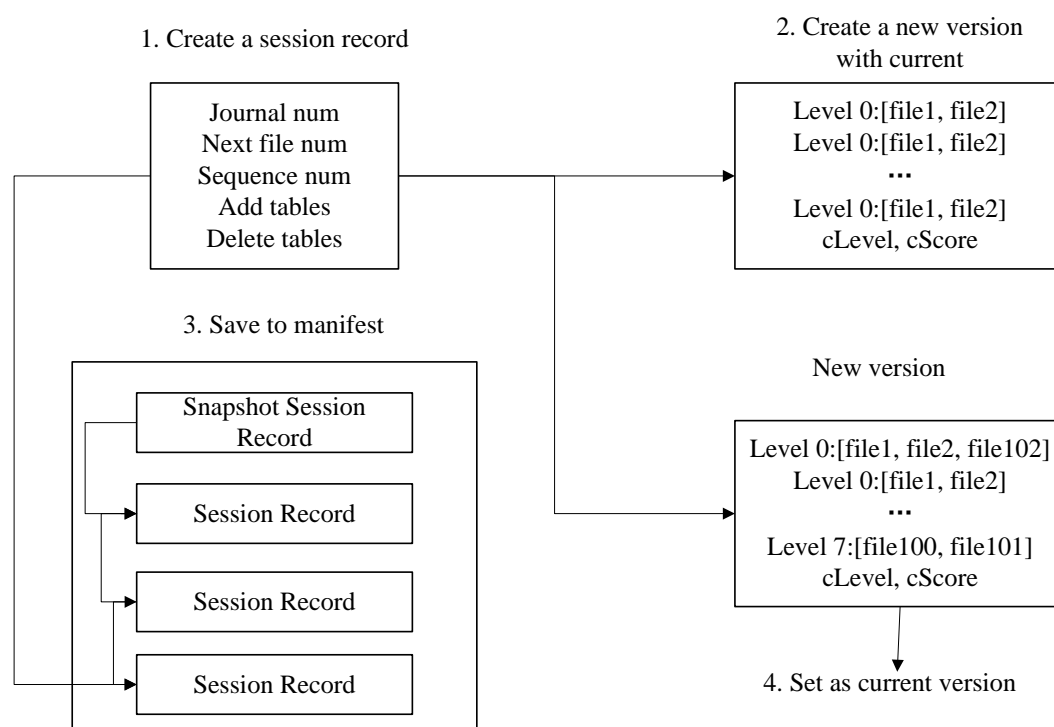


图 4.36 版本更新示意图

新建一个 session record，记录状态变更信息；若本次版本更新的原因是由于 minor compaction 或者日志 replay 导致新生成了一个 sstable 文件，则在 session record 中记录新增的文件信息、最新的 journal 编号、数据库 sequence number 以及下一个可用的文件编号；若本次版本更新的原因是由于 major compaction，则在 session record 中记录新增、删除的文件信息、下一个可用的文件编号即可；利用当前的版本信息，加上 session record 的信息，创建一个全新的版本信息。相较于旧的版本信息，新的版本信息更改的内容为：（1）每一层的文件信息；（2）每一层的计分信息；将 session record 持久化；若这是数据库启动后的第一条 session record，则新建一个 manifest 文件，并将完整的版本信息全部记录进 session record 作为该 manifest 的基础状态写入，同时更改 current 文件，将其指向新建的 manifest；若数据库中已经创建了 manifest 文件，则将该条 session record 进行序列化后直接作为一条记录写入即可；将当前的 version 设置为刚创建的 version。

注意，对于存储系统来说，增减某些 sstable 文件需要作为一个原子性操作，状态变更前后需要保持数据库的一致性。在整个过程中，原子性体现在：整个操作的完成标志

为 manifest 文件中完整的写入了一条 session record，在此之前，即便某些文件写入失败导致进程退出，数据库重启启动时，仍然能够恢复到崩溃之前正确的状态，而将这些无用的 sstable 文件删除，重新进行 compaction 动作。一致性体现在：存储系统状态变更的操作都是以 version 更新为标记，而 version 更新是整个流程的最后一步，因此数据库必然都是从一个一致性的状态变更到另外一个一致性的状态。

3、Recover

数据库每次启动时，都会有一个 recover 的过程，简要地来说，就是利用 Manifest 信息重新构建一个最新的 version。

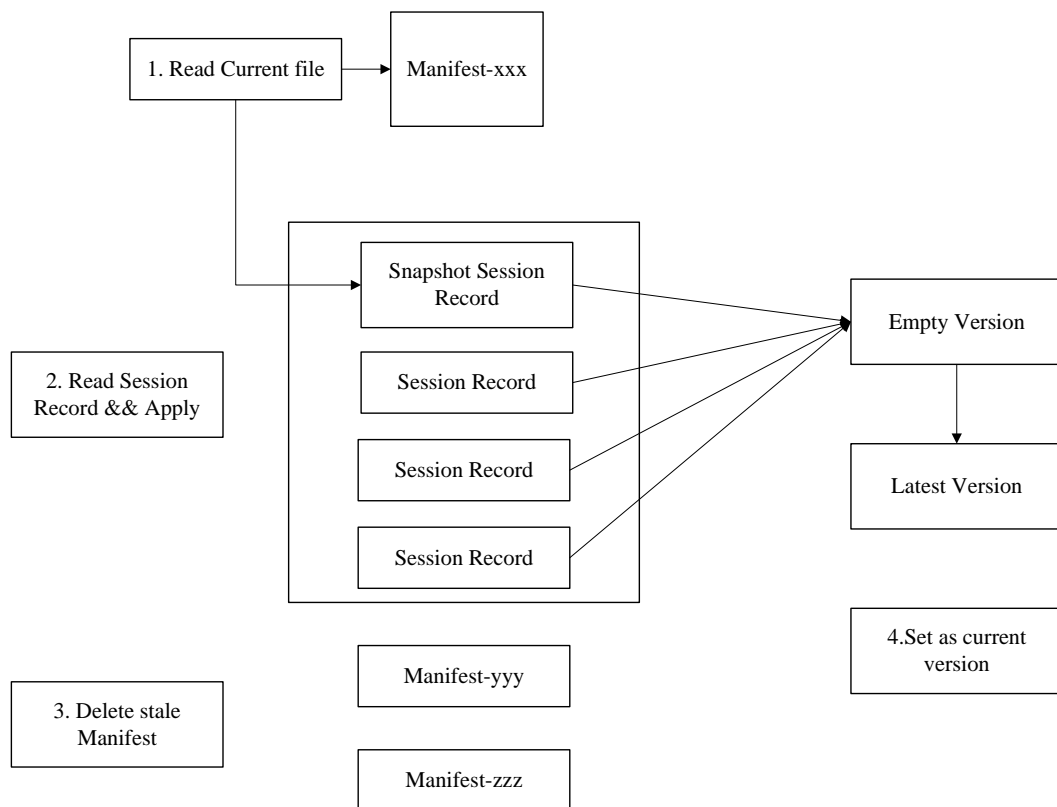


图 4.37 版本复原示意图

过程如下：

利用 Current 文件读取最近使用的 manifest 文件；创建一个空的 version，并利用 manifest 文件中的 session record 依次作 apply 操作，还原出一个最新的 version，注意 manifest 的第一条 session record 是一个 version 的快照，后续的 session record 记录的都是增量的变化；将非 current 文件指向的其它过期的 manifest 文件删除；将新建的 version

作为当前数据库的 **version**；注意，随着存储系统运行时间的增长，一个 **manifest** 中包含的 **session record** 会越来越多，故存储系统在每次启动时都会重新创建一个 **manifest** 文件，并将第一条 **session record** 中记录当前 **version** 的快照状态。其它过期的 **manifest** 文件会在下次启动的 **recover** 流程中进行删除。存储系统通过这种方式，来控制 **manifest** 文件的大小，但是数据库本身没有重启，**manifest** 还是会一直增长。

4、Current

由于每次启动，都会新建一个 **Manifest** 文件，因此存储系统当中可能会存在多个 **manifest** 文件。因此需要一个额外的 **current** 文件来指示当前系统使用的到底是哪个 **manifest** 文件。该文件中只有一个内容，即当前使用的 **manifest** 文件的文件名。

5、异常处理

倘若数据库中的 **manifest** 文件丢失，存储系统一定会进行自动修复。当存储系统的 **manifest** 文件丢失时，所有版本信息也就丢失了，但是本身的数据文件还在。因此存储系统提供了 **Repairer** 接口供用户进行版本信息恢复，具体恢复的过程如下：按照文件编号的顺序扫描所有的 **sstable** 文件，获取每个文件的元数据（最大最小 **key**），以及最终数据库的元数据（**sequence number** 等）；将所有 **sstable** 文件视为 0 层文件（由于 0 层文件允许出现 **key** 重叠的情况，因此不影响正确性）；创建一个新的 **manifest** 文件，将扫描得到的数据库元数据进行记录；但是该方法的效率十分低下，首先需要对整个数据库的文件进行扫描，其次 0 层的文件必然将远远大于 4 个，这将导致极多的 **compaction** 发生。

6、多版本并发控制

存储系统中采用了 **MVCC** 来避免读写冲突。试想一下，当某个迭代器正在迭代某个 **sstable** 文件的内容，而后台的 **major compaction** 进程完成了合并动作，试图删除该 **sstable** 文件。那么假设没有任何控制并发的机制，就会导致迭代器读到的内容发生了丢失。最简单的处理方式就是加锁，当发生读的时候，后台所有的写操作都进行阻塞，但是这就机制就会导致存储系统的效率极低。故存储系统采用了多版本并发控制的方法来解决读写冲突。具体体现在：**sstable** 文件是只读的，每次 **compaction** 都只是对若干个 **sstable** 文件进行多路合并后创建新的文件，故不会影响在某个 **sstable** 文件读操作的正确性；**sstable** 都是具有版本信息的，即每次 **compaction** 完成后，都会生成新版本的 **sstable**，因此可以保障读写操作都可以针对于相应的版本文件进行，解决了读写冲突；**compaction** 生成的文件只有等合并完成后才会写入数据库元数据，在此期间对读操作来

说是透明的，不会污染正常的读操作；采用引用计数来控制删除行为。当 **compaction** 完成后试图去删除某个 **sstable** 文件，会根据该文件的引用计数作适当的删除延迟，即引用计数不为 0 时，需要等待至该文件的计数为 0 才真正进行删除。

4.2 共识层详细设计与实现

4.2.1 共识层角色和状态的实现

1、Cluster

raft 配置中的一组对等节点，几个节点组成的一个集合。

2、Peer

参与共识协议的节点。**peer** 可能处于以下状态之一：**follower**、**candidate** 或 **leader**。

3、Log

完整的日志条目集。

4、Log Entry

日志中的条目。每个条目都有一个索引，用于相对于其它日志条目对其进行排序。

5、Committed

如果将日志条目应用于状态机是安全的，则该日志条目被视为已提交。一旦创建条目的领导者已将其复制到大多数对等点，就会提交日志条目。一旦条目被持久化，对等节点就成功地复制了条目。

6、Applied

应用于状态机 (FSM) 的日志条目。

7、Term

raft 将时间划分为任意长度的项。任期用连续的整数编号。每个任期都以选举开始，其中一名或多名候选人试图成为领导者。如果选举以分裂投票结束，任期将以无领导者结束。

8、FSM

有限状态机，存储集群状态

9、Client

客户端输入 **kv** 键值对，使用集群的客户端

4.2.2 共识层角色操作的实现

1、Leader Write 大多数写操作必须在领导者上执行。

(1) RequestConfigChange

更新 raft 对等节点的配置

(2) Apply

将日志条目应用于大多数对等点和 FSM 上的日志。有关更多详细信息，请参阅 raft 申请。

(3) Barrier

不修改 FSM 的特殊 Apply，用于等待之前的日志被应用

(4) LeadershipTransfer

停止接受客户端请求，并告诉不同的节点开始领导选举

(5) Restore (快照)

用快照的内容覆盖集群状态（不包括集群配置）

(6) VerifyLeader

向所有投票者发送心跳以确认对等点仍然是领导者

2、Follower Write

BootstrapCluster 在本地日志存储集群配置

3、Read 可以在任何状态的对等点上执行读取操作。

(1) AppliedIndex

获取应用于 FSM 的最后一个日志条目的索引

(2) GetConfiguration

返回最新的集群配置

(3) LastContact

获取此对等方最后一次与领导者联系的时间

(4) LastIndex

获取最新存储的日志条目的索引

(5) Leader

获取当前领导者的对等方的地址

(6) Snapshot

将 FSM 的当前状态快照到一个文件中

(7) State

返回对等体的状态

(8) Stats

返回一些关于对等点和集群的统计信息

4.2.3 共识层多线程的实现

Raft 使用以下线程来处理操作。线程的名称以粗体显示，后面是对线程处理的操作的简短描述。主线程负责处理许多操作。

1、run (main thread) 基于对等状态的不同行为

(1) follower

i.processRPC (from rpcCh)

A.AppendEntries

B.RequestVote

C.InstallSnapshot

D.TimeoutNow

ii.liveBootstrap (from bootstrapCh)

iii.periodic heartbeatTimer (HeartbeatTimeout)

(2) candidate 被调用时自己进行一轮选举

i.processRPC (from rpcCh) 与 follower 相同的操作

ii.acceptVote (from askPeerForVote)

(3) leader

首先开始复制到所有对等点，并应用 Noop 日志以确保新领导者已提交到提交索引

i.processRPC (from rpcCh)

与跟随者相同，但是本文实际上并不期望收到除 RequestVote 之外的任何 RPC

ii.leadershipTransfer (from leadershipTransferCh)

iii.commit (from commitCh)

iv.verifyLeader (from verifyCh)

v.user restore snapshot (from userRestoreCh)

vi.changeConfig (from configurationChangeCh)

vii.dispatchLogs (from applyCh)

通过将日志持久化到磁盘并通知复制 goroutines 复制新日志来处理客户端 Raft.Apply 请求

viii.checkLease (periodically LeaseTimeout)

2、runFSM

具有对 FSM 的独占访问权限，所有读写操作都必须向该线程发送消息。命令：从 fsmMutateCh、processLogs、leaderLoop (leader) 或 appendEntries RPC (follower/candidate) 向 FSM 应用日志从 fsmMutateCh、从 restoreUserSnapshot (leader) 或 installSnapshot RPC (follower/candidate) 恢复快照到 FSM 捕获快照，来自 fsmSnapshotCh，来自 takeSnapshot (runSnapshot 线程)

3、runSnapshot

处理拍摄快照的较慢部分。根据 FSM.Snapshot 操作捕获的指针，此线程通过调用 FSMSnapshot.Persist 来保留快照。还调用 compactLogs 来删除旧日志。定期 (SnapshotInterval) takeSnapshot 进行日志压缩用户快照，来自 userSnapshotCh，takeSnapshot 返回给用户

4、askPeerForVote (candidate only)

短暂的 goroutine，同步发送 RequestVote RPC 给所有投票点，并等待响应。每个投票节点一个 goroutine。

5、replicate (leader only)

将日志条目 AppendEntry RPC 同步发送到所有对等点的长时间运行的 goroutine。还启动心跳线程，可能还有 pipelineDecode 线程。AppendEntry 失败时运行 sendLatestSnapshot。

(1) heartbeat (leader only)

长时间运行的 goroutine，同步发送心跳 AppendEntry RPCs 给所有对等点。

(2) pipelineDecode (leader only)

5 Radds 存储系统部署、日志分析与测试

5.1 存储系统部署

1、选择合适的硬件机器

由于 Golang 语言跨平台的特性，本文选择的文件系统操作方案也具有跨平台的特性，所以 Radds 存储系统具有跨平台的特点。

2、下载相应平台上的 Golang 编译器

在 Golang 官网下载相应机器架构，相应操作系统的 Golang 编译器

3、编译源代码工程文件

对本文写好的源代码工程文件进行编译，和自动化测试

5.2 存储系统客户端测试

进入命令行客户端后，输入命令

1、命令行命令概览

下图是命令行客户端的命令总体概览

```

● chisato@ChisatodeMacBook-Pro radss % go install radss
● chisato@ChisatodeMacBook-Pro radss % radss
  😊 App Name is radss-cli , Author Name is , Author Favor is Code
● chisato@ChisatodeMacBook-Pro radss % radss help
  😊 Radss cli is a Command Line Interface that can transfer SQL/NoSQL instruction to the
    data storage system which are hosted on local, cloud, or distributed environment,
    and then receive the query results.

Usage:
  radss [flags]
  radss [command]

Available Commands:
  ask          Ask me whatever you want!
  completion   Generate the autocompletion script for the specified shell
  help         Help about any command
  query        🙋 Input your query instruction

Flags:
  -a, --author string   Author Name (default "Chisaot-X")
  -f, --favor string    Author Favor (default "Code")
  -h, --help            help for radss

Use "radss [command] --help" for more information about a command.
    
```

图 5.1 命令行客户端命令总体概览

2、命令行查询命令概览

下图是命令行客户端的查询命令概览

```

chisato@ChisatodeMacBook-Pro radss % radss query help
"👉" Query mode can be selected as SQL Mode OR NoSQL Mode, the Query Engine will transfer, compile,
optimize, execute and return the query result to the View of Line.

Usage:
  radss query [command]

Available Commands:
  get          Get command
  set          Set command

Flags:
  -h, --help            help for query
  -m, --mode string     Query Mode: NoSQL OR SQL (default "nosql")

Global Flags:
  -a, --author string   Author Name (default "Chisaot-X")
  -f, --favor string    Author Favor (default "Code")

Use "radss query [command] --help" for more information about a command.
    
```

图 5.2 命令行客户端查询命令概览

3、命令行操作数据

下图是如何使用命令行客户端操作数据的方法

```

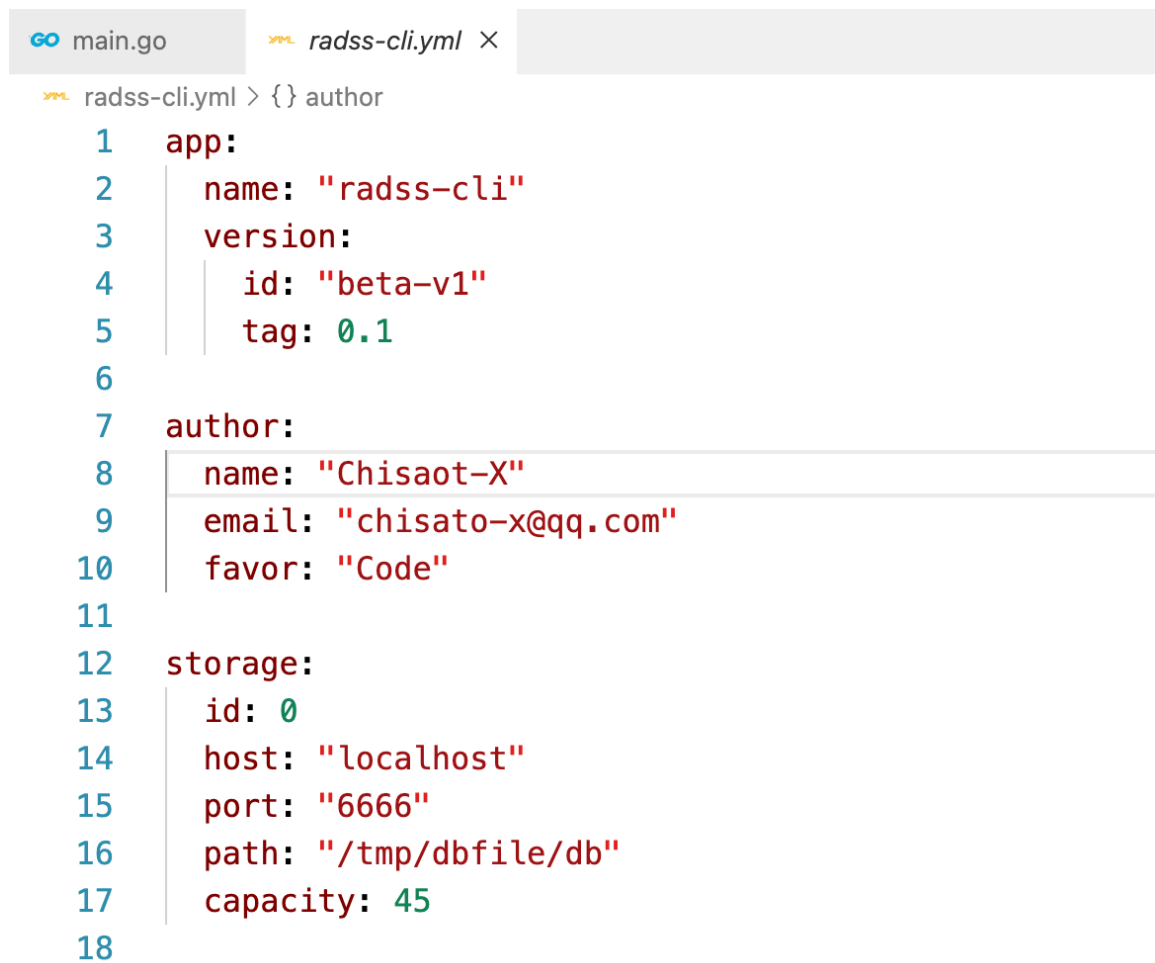
chisato@ChisatodeMacBook-Pro radss % radss query get chisato:name
Key : chisato:name has not been found
chisato@ChisatodeMacBook-Pro radss % radss query set chisato:name chisato-val
( chisato:name , chisato-val ) has setted
chisato@ChisatodeMacBook-Pro radss % radss query get chisato:name
chisato-val
chisato@ChisatodeMacBook-Pro radss % radss query set chisato:name chisato-val1
Value updated
chisato@ChisatodeMacBook-Pro radss % radss query set chisato:name chisato-val2
Value updated
chisato@ChisatodeMacBook-Pro radss % radss query set chisato:name chisato-val3
Value updated
chisato@ChisatodeMacBook-Pro radss % radss query get chisato:name
chisato-val3
    
```

图 5.3 命令行操作数据

5.3 存储系统日志分析

本文针对此存储系统的键值型存储方案，文件压缩方法，日志复制过程做日志分析。

如图是配置文件的一个示例图，可以针对不同操作系统做变形



```

GO main.go  radss-cli.yml ×
radss-cli.yml > {} author
1  app:
2    name: "radss-cli"
3    version:
4      id: "beta-v1"
5      tag: 0.1
6
7  author:
8    name: "Chisaot-X"
9    email: "chisato-x@qq.com"
10   favor: "Code"
11
12 storage:
13   id: 0
14   host: "localhost"
15   port: "6666"
16   path: "/tmp/dbfile/db"
17   capacity: 45
18

```

图 5.4 存储系统配置文件

本文进入操作系统的文件存储目录，打开日志文件：LOG

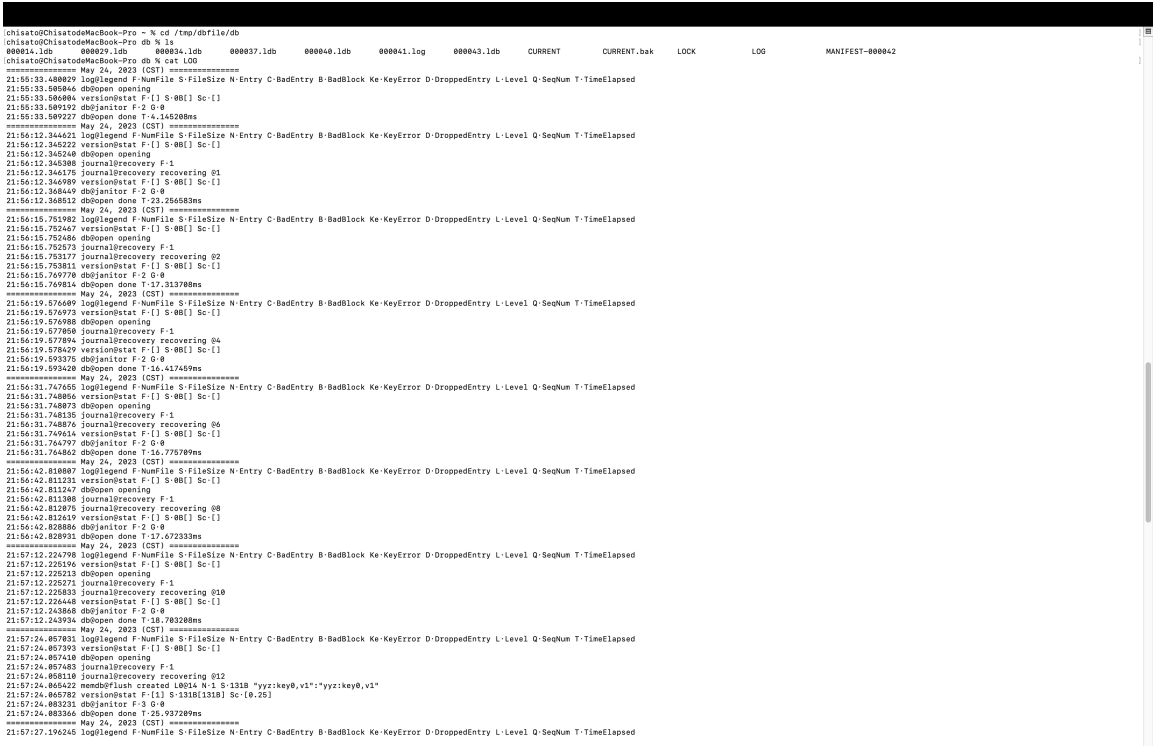


图 5.5 存储系统日志分析

将上述日志转换为表格：

操作类型	次数
open	10
recovery	10
flush	5
janitor	10
compaction	2

表 5.1 数据库操作的次数

结 致 论

综上所述，本文讨论了使用 Golang 编程语言中的 Raft 共识算法和 LSM-tree 数据结构实现分布式数据存储系统。Raft 共识算法提供了一种跨节点集群复制状态机的容错方式。本文在 Golang 中实现了 Raft 算法，以实现集群中节点之间的共识。

LSM-tree 数据结构针对写入密集型工作负载进行了优化，并使用分层方法来存储数据。本文在 Golang 中实现了 LSM-tree 数据结构，以分布式方式存储键值对。本文的实现包括内存缓冲区、磁盘上的一组排序文件以及用于合并排序文件和减少文件数量的压缩机制。

本文将 Raft 共识算法和 LSM-tree 数据结构相结合，用 Golang 构建分布式数据存储系统。本文的系统由多个节点组成，这些节点使用 Raft 算法相互通信。每个节点存储 LSM-tree 数据结构的副本。当客户端向系统写入数据时，首先将数据写入领导节点上 LSM-tree 的内存缓冲区。领导节点然后使用 Raft 算法将数据复制到集群中的所有其它节点。复制数据后，领导节点会将内存缓冲区刷新到磁盘并将数据添加到 LSM-tree。

当客户端从系统读取数据时，读取请求被发送到领导节点。领导节点从 LSM-tree 中读取数据并将其发送回客户端。如果读请求被发送到一个 follower 节点，follower 节点将请求重定向到 leader 节点。

本文设计的客户端满足客户端设计的艺术法则（the-art-of-the-command-line），能够高效的使用并进行系统数据操作和集群操作。

总之，Raft 共识算法和 LSM-tree 数据结构的结合提供了一种以分布式方式存储和检索数据的可靠且高效的方式。Golang 编程语言为实现分布式数据存储系统提供了一个强大而高效的平台。本文的实施展示了结合这些技术构建分布式数据存储系统的有效性。

致 谢

在本科四年的学习生活中，我得到了许多人的帮助和支持，他们让我在学业和成长上都有了很大的收获。在此，我想向他们表达我的衷心感谢。

首先，我要感谢我的导师吕艳霞教授，她在我毕业论文的选题、撰写和修改过程中给予了我很多的指导和建议，她严谨的治学态度和深厚的学术造诣让我受益匪浅。她不仅教授了我专业知识和技能，还培养了我独立思考和创新的能力，对我的成长有着重要的影响。

其次，我要感谢我的同学和朋友们，他们在我学习和生活上都给予了我很多的帮助和鼓励。特别是我的室友们，我们一起度过了许多难忘的时光，分享了彼此的喜怒哀乐，他们是我最亲密的伙伴。还有我的团队成员们，我们一起完成了很多有意义的项目，互相学习和交流，共同进步。

再次，我要感谢 2017 级的两位学长金韬和覃辉，他们在我进入实验室后给予了我很多的指导和帮助，他们用自己的经验和见解让我开阔了视野，提高了思维水平。还有实验室的其他成员们，他们在实验室中营造了一个良好的学习氛围，与我分享了很多有价值的资料和信息。

最后，我要感谢我的家人，他们是我坚强的后盾，无论遇到什么困难和挫折，他们都给予了我无条件的爱和支持。他们为我的成长付出了很多的努力和牺牲，是最感激和敬佩的人。

在此，我再次向所有关心和帮助过我的人表示衷心的感谢！

参考文献

- [1] Alfred V. Aho, John E. Hopcroft, and Jeffrey D. Ullman, "The Design and Analysis of Computer Algorithms", Addison-Wesley.
- [2] Anon et al., "A Measure of Transaction Processing Power", Readings in Database Systems, edited by Michael Stonebraker, pp 300-312, Morgan Kaufmann, 1988.
- [3] R. Bayer and M Schkolnick, "Concurrency of Operations on B-Trees", Readings in Database Systems, edited by Michael Stonebraker, pp 129-139, Morgan Kaufmann 1988.
- [4] P. A. Bernstein, V. Hadzilacos, and N. Goodman, "Concurrency Control and Recovery in Database Systems", Addison-Wesley, 1987.
- [5] D. Comer, "The Ubiquitous B-tree", Comput. Surv. 11, (1979), pp 121-137.
- [6] George Copeland, Tom Keller, and Marc Smith, "Database Buffer and Disk Configuring and the Battle of the Bottlenecks", Proc. 4th International Workshop on High Performance Transaction Systems, September 1991.
- [7] P. Dadam, V. Lum, U. Pradel, G. Shlageter, "Selective Deferred Index Maintenance & Concurrency Control in Integrated Information Systems," Proceedings of the Eleventh International VLDB Conference, August 1985, pp. 142-150.
- [8] Dean S. Daniels, Alfred Z. Spector and Dean S. Thompson, "Distributed Logging for Transaction Processing", ACM SIGMOD Transactions, 1987, pp. 82-96.
- [9] R. Fagin, J. Nievergelt, N. Pippenger and H.R. Strong, Extendible Hashing — A Fast Access Method for Dynamic Files, ACM Trans. on Database Systems, V 4, N 3 (1979), pp 315-344
- [10] H. Garcia-Molina, D. Gawlick, J. Klein, K. Kleissner and K. Salem, "Coordinating Multi-Transactional Activities", Princeton University Report, CS-TR-247-90, February 1990.
- [11] Hector Garcia-Molina and Kenneth Salem, "Sagas", ACM SIGMOD Transactions, May 1987, pp. 249-259.
- [12] Jim Gray and Franco Putzolu, "The Five Minute Rule for Trading Memory for Disk Accesses and The 10 Byte Rule for Trading Memory for CPU Time", Proceedings of the 1987 ACM SIGMOD Conference, pp 395-398.
- [13] Jim Gray and Andreas Reuter, "Transaction Processing, Concepts and Techniques", Mor-

gan Kaufmann 1992.

- [14] Curtis P. Kolovson and Michael Stonebraker, "Indexing Techniques for Historical Databases", Proceedings of the 1989 IEEE Data Engineering Conference, pp 138-147.
- [15] BOLOSKY, W. J., BRADSHAW, D., HAAGENS, R. B., KUSTERS, N. P., AND LI, P. Paxos replicated state machines as the basis of a high-performance data store. In Proc. NSDI'11, USENIX Conference on Networked Systems Design and Implementation (2011), USENIX, pp. 141–154.
- [16] BURROWS, M. The Chubby lock service for loosely- coupled distributed systems. In Proc. OSDI'06, Symposium on Operating Systems Design and Implementation (2006), USENIX, pp. 335–350.
- [17] CAMARGOS, L. J., SCHMIDT, R. M., AND PEDONE, F. Multicoordinated Paxos. In Proc. PODC'07, ACM Symposium on Principles of Distributed Computing (2007), ACM, pp. 316–317.
- [18] CHANDRA, T. D., GRIESEMER, R., AND REDSTONE, J. Paxos made live: an engineering perspective. In Proc. PODC'07, ACM Symposium on Principles of Distributed Computing (2007), ACM, pp. 398–407.
- [19] CHANG, F., DEAN, J., GHEMAWAT, S., HSIEH, W. C., WALLACH, D. A., BURROWS, M., CHANDRA, T., FIKES, A., AND GRUBER, R. E. Bigtable: a distributed storage system for structured data. In Proc. OSDI'06, USENIX Symposium on Operating Systems Design and Implementation (2006), USENIX, pp. 205–218.
- [20] CORBETT, J. C., DEAN, J., EPSTEIN, M., FIKES, A., FROST, C., FURMAN, J. J., GHEMAWAT, S., GUBAREV, A., HEISER, C., HOCHSCHILD, P., HSIEH, W., KANTHAK, S., KOGAN, E., LI, H., LLOYD, A., MELNIK, S., MWAURA, D., NAGLE, D., QUINLAN, S., RAO, R., ROLIG, L., SAITO, Y., SZYMANKI, M., TAYLOR, C., WANG, R., AND WOODFORD, D. Spanner: Google's globally-distributed database. In Proc. OSDI'12, USENIX Conference on Operating Systems Design and Implementation (2012), USENIX, pp. 251–264.
- [21] COUSINEAU, D., DOLIGEZ, D., LAMPORT, L., MERZ, S., RICKETTS, D., AND VANZETTO, H. TLA+ proofs. In Proc. FM'12, Symposium on Formal Methods (2012),

- D. Giannakopoulou and D. M'ery, Eds., vol. 7436 of Lecture Notes in Computer Science, Springer, pp. 147–154.
- [22] GHEMAWAT, S., GOBIOFF, H., AND LEUNG, S.-T. The Google file system. In Proc. SOSP'03, ACM Symposium on Operating Systems Principles (2003), ACM, pp. 29–43.
 - [23] GRAY, C., AND CHERITON, D. Leases: An efficient fault-tolerant mechanism for distributed file cache consistency. In Proceedings of the 12th ACM Symposium on Operating Systems Principles (1989), pp. 202–210.
 - [24] HERLIHY, M. P., AND WING, J. M. Linearizability: a correctness condition for concurrent objects. ACM Transactions on Programming Languages and Systems 12 (July 1990), 463–492.
 - [25] HUNT, P., KONAR, M., JUNQUEIRA, F. P., AND REED, B. ZooKeeper: wait-free coordination for internet-scale systems. In Proc ATC'10, USENIX Annual Technical Conference (2010), USENIX, pp. 145–158.
 - [26] JUNQUEIRA, F. P., REED, B. C., AND SERAFINI, M. Zab: High-performance broadcast for primary-backup systems. In Proc. DSN'11, IEEE/IFIP Int'l Conf. on Dependable Systems & Networks (2011), IEEE Computer Society, pp. 245–256.
 - [27] ONGARO, D., AND OUSTERHOUT, J. In search of an understandable consensus algorithm. In Proc ATC'14, USENIX Annual Technical Conference (2014), USENIX.
 - [28] OUSTERHOUT, J., AGRAWAL, P., ERICKSON, D., KOZYRAKIS, C., LEVERICH, J., MAZI 'ERES, D., MITRA, S., NARAYANAN, A., ONGARO, D., PARULKAR, G., ROSENBLUM, M., RUMBLE, S. M., STRATMANN, E., AND STUTSMAN, R. The case for RAMCloud. Communications of the ACM 54 (July 2011), 121–130.
 - [29] Raft consensus algorithm website. <http://raftconsensus.github.io>.
 - [30] REED, B. Personal communications, May 17, 2013.
 - [31] ROSENBLUM, M., AND OUSTERHOUT, J. K. The design and implementation of a log-structured file system. ACM Trans. Comput. Syst. 10 (February 1992), 26–52.
 - [32] SCHNEIDER, F. B. Implementing fault-tolerant services using the state machine approach: a tutorial. ACM Computing Surveys 22, 4 (Dec. 1990), 299–319.

附 录

附录 A

Raft - In Search of an Understandable Consensus Algorithm

Raft is a consensus algorithm for managing a replicated log. It provides the same function and performance as the Paxos algorithm, but its algorithm structure is different from Paxos, making the Raft algorithm easier to understand and easier to build an actual system. To improve understandability, Raft decomposes the consensus algorithm into several key modules, such as leader election, log replication, and security. At the same time it reduces the number of states that need to be considered by enforcing a stronger consistency. The results of a user study show that the Raft algorithm is easier for students to learn than the Paxos algorithm. The Raft algorithm also includes a new mechanism to allow dynamic changes in cluster membership, which utilizes overlapping majorities for safety. Consensus algorithms allow a group of machines to work as a whole and continue to work even if some of them fail. Because of this, consensus algorithms play an important role in building trustworthy large-scale software systems. For the past 10 years, the Paxos algorithm has dominated the field of consensus algorithms: the vast majority of implementations are based on or influenced by Paxos. At the same time, Paxos has also become an example in the teaching field when explaining consistency issues.

But unfortunately, the Paxos algorithm is still very difficult to understand despite a lot of work trying to reduce its complexity. Moreover, the algorithm structure of Paxos itself needs to be greatly modified before it can be applied to the actual system. Therefore, both industry and academia are very troubled by the Paxos algorithm. After working hard on the Paxos algorithm, we set out to find a new consensus algorithm that could provide a better basis for building practical systems and teaching. Unlike Paxos, our primary goal is understandability: can we define a consensus algorithm in real systems that is easier to learn than the Paxos algorithm. Furthermore, we hope that the algorithm facilitates the development of the system builder's intuition. It's not just that the algorithm works, it's that you know exactly why it works. The Raft consensus algorithm is the result of this work. When designing the Raft algorithm, we use some special techniques to improve its understandability, including algorithm decomposition

(Raft is mainly divided into three modules of leader election, log replication and security) and reducing the state of the state machine (compared to Like Paxos, Raft reduces non-determinism and the way servers are inconsistent with each other). A study of 43 students at two universities showed that Raft is significantly easier to understand than Paxos. After these students learned both algorithms, 33 of them were able to answer questions about Raft compared to Paxos.

Consensus algorithms are proposed in the context of replicated state machines. In this approach, a state machine on a set of servers produces a copy of the same state and can continue to run even if some machines go down. Replicated state machines are used in distributed systems to solve many fault tolerance problems. For example, there is usually a cluster leader in large-scale systems, such as GFS, HDFS and RAMCloud, and a typical application is an independent replication state machine to manage leader election and store configuration information survived. Such as Chubby and ZooKeeper. A consensus algorithm manages a replicated log of instructions from clients. The state machine processes the same instructions in the same order from the log, so the results are the same.

Replicated state machines are usually implemented based on replicated logs. Each server stores a log containing a series of instructions and executes them in the order of the log. Every log contains the same commands in the same order, so every server executes the same sequence of commands. Because every state machine is deterministic, every execution of an operation produces the same state and the same sequence.

The task of the consensus algorithm is to ensure the consistency of the replicated log. The consistency module on the server receives the instructions sent by the client and adds them to its own log. It communicates with consistency modules on other servers to ensure that the logs on each server eventually contain the same requests in the same order, even if some servers fail. Once the commands are replicated correctly, each server's state machine processes them in log order, and the output is returned to the client. Thus, the cluster of servers appears to form a highly reliable state machine.

Consensus algorithms used in practical systems usually have the following characteristics: Safety guarantee (never return a wrong result): In the case of non-Byzantine errors, errors including network delays, partitions, packet loss, duplication, and disorder are guaranteed to be

correct. Availability: As long as most of the machines in the cluster are operational and able to communicate with each other and clients, they can be guaranteed to be available. Therefore, a typical cluster of 5 nodes can tolerate the failure of two nodes. A server is considered a failure if it is stopped. They may later recover from the reliably stored state and rejoin the cluster. Do not rely on timing for consistency: physical clock errors or extreme message delays cause availability problems only in the worst case. Typically, an instruction can complete as quickly as possible when most nodes in the cluster respond to a round of remote procedure calls. A small number of slower nodes will not affect the overall performance of the system.

中文译文 A

Raft-一种易于理解的共识性算法

Raft 是一种为了管理复制日志的一致性算法。它提供了和 **Paxos** 算法相同的功能和性能，但是它的算法结构和 **Paxos** 不同，使得 **Raft** 算法更加容易理解并且更容易构建实际的系统。为了提升可理解性，**Raft** 将一致性算法分解成了几个关键模块，例如领导人选举、日志复制和安全性。同时它通过实施一个更强的一致性来减少需要考虑的状态的数量。一项用户研究的结果表明，对于学生而言，**Raft** 算法比 **Paxos** 算法更加容易学习。**Raft** 算法还包括一个新的机制来允许集群成员的动态改变，它利用重叠的大多数来保证安全性。一致性算法允许一组机器像一个整体一样工作，即使其中一些机器出现故障也能够继续工作下去。正因为如此，一致性算法在构建可信赖的大规模软件系统中扮演着重要的角色。在过去的 10 年里，**Paxos** 算法统治着一致性算法这一领域：绝大多数的实现都是基于 **Paxos** 或者受其影响。同时 **Paxos** 也成为了教学领域里讲解一致性问题的示例。

但是不幸的是，尽管有很多工作都在尝试降低它的复杂性，但是 **Paxos** 算法依然十分难以理解。并且，**Paxos** 自身的算法结构需要进行大幅的修改才能够应用到实际的系统中。因此工业界和学术界都对 **Paxos** 算法感到十分头疼。努力研究过 **Paxos** 算法之后，本文开始寻找一种新的一致性算法，可以为构建实际的系统和教学提供更好的基础。与 **Paxos** 不同，本文的首要目标是可理解性：本文是否可以在实际系统中定义一个一致性算法，并且比 **Paxos** 算法更容易学习。此外，本文希望该算法方便系统构建者的直觉的发展。重要的不仅仅是算法能够工作，更重要的是能够很清楚地知道它为什么能工作。**Raft** 一致性算法就是这些工作的结果。在设计 **Raft** 算法的时候，本文使用一些特别的技

巧来提升它的可理解性，包括算法分解（Raft 主要被分成了领导人选举，日志复制和安全三个模块）和减少状态机的状态（相对于 Paxos，Raft 减少了非确定性和服务器互相处于非一致性的方式）。一份针对两所大学 43 个学生的研究表明 Raft 明显比 Paxos 算法更加容易理解。在这些学生同时学习了这两种算法之后，和 Paxos 比起来，其中 33 个学生能够回答有关于 Raft 的问题。

一致性算法是从复制状态机的背景下提出的。在这种方法中，一组服务器上的状态机产生相同状态的副本，并且在一些机器宕掉的情况下也可以继续运行。复制状态机在分布式系统中被用于解决很多容错的问题。例如，大规模的系统中通常都有一个集群领导人，像 GFS、HDFS 和 RAMCloud，典型应用就是一个独立的复制状态机去管理领导选举和存储配置信息并且在领导人宕机的情况下也要存活下来。比如 Chubby 和 ZooKeeper。一致性算法管理着来自客户端指令的复制日志。状态机从日志中处理相同顺序的相同指令，所以产生的结果也是相同的。

复制状态机通常都是基于复制日志实现的。每一个服务器存储一个包含一系列指令的日志，并且按照日志的顺序进行执行。每一个日志都按照相同的顺序包含相同的指令，所以每一个服务器都执行相同的指令序列。因为每个状态机都是确定的，每一次执行操作都产生相同的状态和同样的序列。

一致性算法的任务是保证复制日志的一致性。服务器上的一致性模块接收客户端发送的指令然后添加到自己的日志中。它和其它服务器上的一致性模块进行通信来保证每一个服务器上的日志最终都以相同的顺序包含相同的请求，即使有些服务器发生故障。一旦指令被正确的复制，每一个服务器的状态机按照日志顺序处理它们，然后输出结果被返回给客户端。因此，服务器集群看起来形成了一个高可靠的状态机。

实际系统中使用的一致性算法通常含有以下特性：安全性保证（绝对不会返回一个错误的结果）：在非拜占庭错误情况下，包括网络延迟、分区、丢包、重复和乱序等错误都可以保证正确。可用性：集群中只要有大多数的机器可运行并且能够相互通信、和客户端通信，就可以保证可用。因此，一个典型的包含 5 个节点的集群可以容忍两个节点的失败。服务器被停止就认为是失败。它们稍后可能会从可靠存储的状态中恢复并重新加入集群。不依赖时序来保证一致性：物理时钟错误或者极端的消息延迟只有在最坏情况下才会导致可用性问题。通常情况下，一条指令可以尽可能快的在集群中大多数节点响应一轮远程过程调用时完成。小部分比较慢的节点不会影响系统整体的性能。

附录 B

The Log-Structured Merge-Tree

High-performance transaction system applications typically insert rows in a History table to provide an activity trace; at the same time the transaction system generates log records for purposes of system recovery. Both types of generated information can benefit from efficient indexing. An example in a well-known setting is the TPC-A benchmark application, modified to support efficient queries on the History for account activity for specific accounts. This requires an index by account-id on the fast-growing History table. Unfortunately, standard disk-based index structures such as the B-tree will effectively double the I/O cost of the transaction to maintain an index such as this in real time, increasing the total system cost up to fifty percent. Clearly a method for maintaining a real-time index at low cost is desirable. The Log-Structured Merge-tree (LSM-tree) is a disk-based data structure designed to provide low-cost indexing for a file experiencing a high rate of record inserts (and deletes) over an extended period. The LSM-tree uses an algorithm that defers and batches index changes, cascading the changes from a memory-based component through one or more disk components in an efficient manner reminiscent of merge sort. During this process all index values are continuously accessible to retrievals (aside from very short locking periods), either through the memory component or one of the disk components. The algorithm has greatly reduced disk arm movements compared to a traditional access methods such as B-trees, and will improve cost-performance in domains where disk arm costs for inserts with traditional access methods overwhelm storage media costs. The LSM-tree approach also generalizes to operations other than insert and delete. However, indexed finds requiring immediate response will lose I/O efficiency in some cases, so the LSM-tree is most useful in applications where index inserts are more common than finds that retrieve the entries. This seems to be a common property for History tables and log files, for example. The conclusions of Section 6 compare the hybrid use of memory and disk components in the LSM-tree access method with the commonly understood advantage of the hybrid method to buffer disk pages in memory.

As long-lived transactions in activity flow management systems become commercially available there will be increased need to provide indexed access to transactional log records.

Traditionally, transactional logging has focused on aborts and recovery, and has required the system to refer back to a relatively short-term history in normal processing with occasional transaction rollback, while recovery was performed using batched sequential reads. However, as systems take on responsibility for more complex activities, the duration and number of events that make up a single long-lived activity will increase to a point where there is sometimes a need to review past transactional steps in real time to remind users of what has been accomplished. At the same time, the total number of active events known to a system will increase to the point where memory-resident data structures now used to keep track of active logs are no longer feasible, notwithstanding the continuing decrease in memory cost to be expected. The need to answer queries about a vast number of past activity logs implies that indexed log access will become more and more important.

Even with current transactional systems there is clear value in providing indexing to support queries on history tables with high insert volume. Networking, electronic mail, and other nearly-transactional systems produce huge logs often to the detriment of their host systems. To start from a concrete and well-known example, we explore a modified TPC-A benchmark in the following Examples 1.1 and 1.2. Note that examples presented in this paper deal with specific numeric parametric values for ease of presentation; it is a simple task to generalize these results. Note too that although both history tables and logs involve time-series data, the index entries of the LSM-Tree are not assumed to have identical temporal key order. The only assumption for improved efficiency is high update rates compared to retrieval rates.

中文译文 B

日志结构归并树

高性能事务系统应用程序通常在一个提供活动轨迹的历史表；同时交易系统产生日志用于系统恢复的记录。两种类型的生成信息都可以从中受益高效的索引。一个众所周知的例子是 TPC-A 基准应用程序，修改以支持对特定帐户的帐户活动历史记录进行有效查询。这需要在快速增长的 History 表上按 account-id 建立索引。不幸的是，基于硬磁盘的索引结构，如 B-tree 将有效地使 I/O 成本加倍事务来实时维护这样的索引，增加了总系统成本高达百分之五十。显然，需要一种以低成本维护实时索引的方法。这 Log-Structured Merge-tree (LSM-tree) 是一种基于磁盘的数据结构，旨在提供对经历高记

录插入（和删除）率的文件进行低成本索引延展期。**LSM-tree** 使用延迟和批处理索引更改的算法，**cas-** 将来自基于内存的组件的更改缓存到一个或多个磁盘组件中高效的方式让人联想到归并排序。在此过程中，所有指标值都在连续非常容易检索（除了非常短的锁定期），要么通过内存组件或磁盘组件之一。该算法大大减少了磁盘臂与传统的访问方法（如 **B** 树）相比，移动将提高成本在使用传统访问方法插入磁盘臂成本的域中的性能不堪重负的存储介质成本。**LSM-tree** 方法也推广到其它操作比插入和删除。然而，需要立即响应的索引查找将丢失 **I/O ef-** 在某些情况下效率很高，因此 **LSM-tree** 在需要插入索引的应用程序中最有用比检索条目的发现更常见。

随着活动流管理系统中的长期交易变得商业可用提供索引访问的需求将增加到事务日志记录。传统上，事务日志集中于中止和重新覆盖，并要求系统在正常情况下返回相对较短的历史记录处理偶尔的事务回滚，而恢复是使用批处理执行的顺序读取。然而，随着系统承担起更复杂活动的责任，构成单个长期活动的持续时间和事件数量将增加到一个点有时需要实时回顾过去的交易步骤以提醒用户已经完成的事情。同时，已知的活动事件总数系统将增加到内存驻留数据结构现在用于保存的程度尽管内存持续减少，但活动日志的跟踪不再可行预期的成本。需要回答有关大量过去活动日志的查询意味着索引日志访问将变得越来越重要。

即使对于当前的事务系统，提供索引以支持也具有明显的价值查询具有高插入量的历史表。网络、电子邮件和其它近事务性系统会产生大量日志，而这些日志通常会损害其主机系统。到从一个具体和众所周知的例子开始，本文探索了一个修改后的 **TPC-A** 基准。