



東北大學秦皇岛分校
Northeastern University at Qinhuangdao

毕业论文

基于 LSM-Tree 结构和 Raft 算法的分布式存储 系统

院别	计算机与通信工程学院
专业名称	计算机科学与技术
班级学号	1901-20197897
学生姓名	华令楠
指导教师	吕艳霞

2023 年 5 月 20 日

郑重声明

本人呈交的学位论文，是在导师的指导下，独立进行研究工作所取得的成果，所有数据、图片资料真实可靠。尽我所知，除文中已经注明引用的内容外，本学位论文的研究成果不包含他人享有著作权的内容。对本论文所涉及的研究工作做出贡献的其他个人和集体，均已在文中以明确的方式标明。本学位论文的知识产权归属于培养单位。

本人签名： 日期：

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。我们利用 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。我们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。通过不断精读原文和阅读参考并 demo 出 Google 开源 C++ 版本的 leveldb 实现的 LSM-Tree 以便尽最大限度复现 LSM-Tree 和 Raft 论文中提及的所有关键点，我们的评估表明，我们的系统在保持强一致性保证的同时实现了高性能和可扩展性，同时支持跨多平台的服务端部署，跨多个平台的客户端调用和多种语言的 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. We use 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. By continuously intensively reading the original text and reading references, and demoing the LSM-Tree implemented by Google's open source C++ version of leveldb, in order to reproduce all the key points mentioned in the LSM-Tree and Raft

papers as much as possible, Our evaluation shows that our system achieves high performance and scalability while maintaining strong consistency guarantees, while supporting server-side deployment across multiple platforms, client calls 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 相关背景知识介绍	1
1.1 开发工具和环境	1
1.2 LSM-Tree 存储结构	2
1.3 Raft 共识性算法	3
1.4 涉及的开源库	3
1.5 本章小结	4
2 Radds 存储系统需求分析	5
2.1 存储系统需求概述	5
2.2 存储系统功能需求分析	5
2.3 存储系统性能需求分析	6
2.4 存储系统可行性分析	7
2.5 本章小结	7
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 内存可变数据结构 memtable 总体设计	11
3.2.2 内存不可变数据结构 immutable memtable 总体设计	11
3.2.3 日志文件数据结构 journal 总体设计	11
3.2.4 磁盘持久化数据结构 sstable 总体设计	12
3.2.5 文件元数据 manifest 总体设计	12
3.2.6 版本号 current 总体设计	13
3.3 共识层功能总体设计	15
3.4 客户端功能总体设计	15
3.4.1 API 客户端服务平台总体设计	15
3.4.2 gRPC API 客户端总体设计	15
3.4.3 RESTful API 客户端总体设计	15

3.4.4	CLI 客户端总体设计	15
3.5	本章小结	15
4	Radds 存储系统详细设计与实现	16
4.1	基础层详细设计与实现	16
4.1.1	错误处理的实现	16
4.1.2	日志系统的实现	16
4.1.3	其他工具库的实现	20
4.2	存储层详细设计与实现	20
4.2.1	写数据的实现	20
4.2.2	读数据的实现	25
4.2.3	跳表数据结构的实现	27
4.2.4	内存数据库的实现	32
4.2.5	持久化数据存储的实现	34
4.2.6	缓存系统的实现	51
4.2.7	布隆过滤器的实现	56
4.2.8	数据压缩系统的实现	61
4.2.9	版本控制的实现	61
4.3	共识层详细设计与实现	62
4.4	客户端层详细设计与实现	62
4.4.1	API 客户端服务平台的实现	62
4.4.2	gRPC API 客户端的实现	62
4.4.3	RESTful API 客户端的实现	62
4.4.4	CLI 命令行客户端的实现	62
4.5	本章小结	62
5	Radds 存储系统部署、日志分析与客户端测试	63
5.1	分布式存储系统部署	63
5.1.1	在 X86-64 GNU/Linux Ubuntu22.04 操作系统部署	63
5.1.2	在 X86-64 Windows11 操作系统部署	63

5.1.3 在 Arm64 Darwin MacOS 操作系统部署	63
5.2 数据存储系统日志分析	63
5.3 共识性系统日志分析	63
5.4 客户端服务平台日志分析	63
5.5 客户端测试	63
5.5.1 gRPC API 客户端测试	63
5.5.2 RESTful API 客户端测试	63
5.5.3 CLI 客户端测试	63
5.6 本章小结	63
结论	64
致 谢	65
附 录	66
附录 A	66
附录 B	66
参考文献	66

1 相关背景知识介绍

1.1 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他实用工具。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 或更新的处理器的设备。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。类似其他类 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。

1.2 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 树和其他搜索树一样，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）。

1.3 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。

1.4 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他压缩库兼容为目标；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，可以与标准库做很好的搭配。

1.5 本章小结

2 Radds 存储系统需求分析

2.1 存储系统需求概述

图 2.1 Radds 存储系统需求概述

2.2 存储系统功能需求分析

1、功能需求 1

(2) 登录。省略一段文字省略一段文字省略一段文字省略一段文字省略一段文字
省略一段文字省略一段文字省略一段文字省略一段文字省略一段文字省略一段文字省
略一段文字省略一段文字省略一段文字省略一段文字省略一段文字省略一段文字省
略一段文字省略一段文字省略一段文字省略一段文字省略一段文字省略一段文字

2、功能需求 2

3、功能需求 3

(1) 省略一段文字省略一段文字省略一段文字省略一段文字省略一段文字省略一段文字省略一段文字省略一段文字省略一段文字省略一段文字省略一段文字省略一段文字省

(2) 省略一段文字省略一段文字省略一段文字省略一段文字省略一段文字省略一段文字省略一段文字省略一段文字省略一段文字省略一段文字省

4、功能需求 4

(1) 省略一段文字省略一段文字省略一段文字省略一段文字省略一段文字省略一段文字省略一段文字省略一段文字省略一段文字省略一段文字省略一段文字省

(2) 省略一段文字省略一段文字省略一段文字省略一段文字省略一段文字省略一段文字省略一段文字省略一段文字省略一段文字省

5、功能需求 5

(1) 省略一段文字省略一段文字省略一段文字省略一段文字省略一段文字省略一段文字省略一段文字省略一段文字省略一段文字省略一段文字省

(2) 省略一段文字省略一段文字省略一段文字省略一段文字省略一段文字省略一段文字省略一段文字省略一段文字省略一段文字省

2.3 存储系统性能需求分析

1、性能需求 1

省略一段文字省略一段文字省略一段文字省略一段文字省略一段文字省略一段文字省略一段文字省略一段文字省略一段文字省略一段文字省略一段文字省

2、性能需求 2

省略一段文字省略一段文字省略一段文字省略一段文字省略一段文字省略一段文字省略一段文字省略一段文字省略一段文字省略一段文字省

3、性能需求 3

省略一段文字省略一段文字省略一段文字省略一段文字省略一段文字省略一段文字省略一段文字省略一段文字省略一段文字省略一段文字省略一段文字省略一段文字省略一段文字省略一段文字省略一段文字省

4、性能需求 4

省略一段文字省略一段文字省略一段文字省略一段文字省略一段文字省略一段文字省略一段文字省略一段文字省略一段文字省略一段文字省略一段文字省略一段文字省略一段文字省略一段文字省

5、性能需求 5

省略一段文字省略一段文字省略一段文字省略一段文字省略一段文字省略一段文字省略一段文字省略一段文字省略一段文字省略一段文字省略一段文字省略一段文字省略一段文字省略一段文字省

2.4 存储系统可行性分析

2.5 本章小结

3 Radds 存储系统总体设计

Radds 存储系统总体架构组成如图 3.1 所示

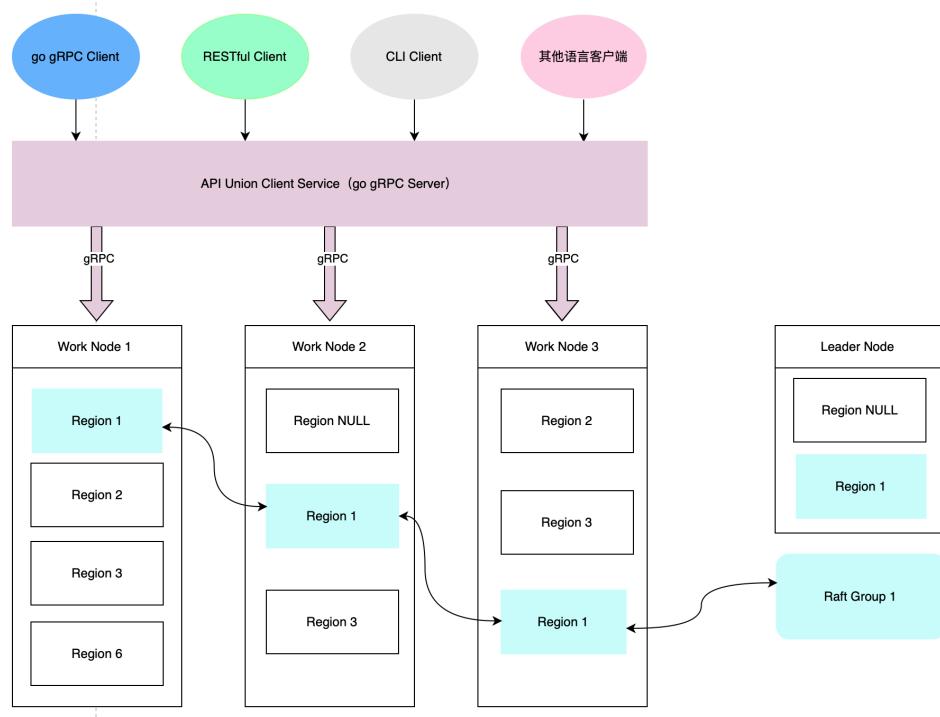


图 3.1 Radds 存储系统总体架构图

3.1 存储系统架构设计

系统的设计架构是 C/S 的方式，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 的其他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）。

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。

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，

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，还可以存储其他 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，他们分别属于不同的 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，可能的情况有：

- (1) Leader 宕机，一个 Group 的 Leader 宕机后，其他节点在超时时间内未收到心跳信息则会感知到 Group Leader 宕机，接下来根据 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，以便于其他 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，同时他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，这时他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，我根据 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 的存储结构我则是做

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，也是我的创新，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，可以添加新的功能，如鉴权，多租户系统等，这个灵感来源于阅读 SnowFlake 数据仓库和一些报告会论文。

3.2 存储层功能总体设计

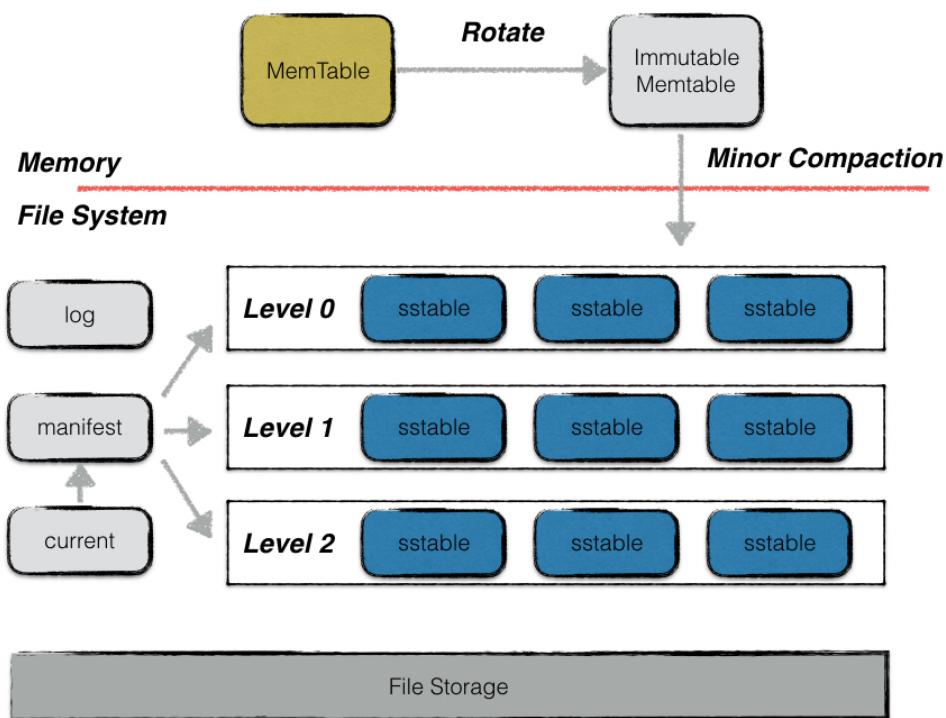


图 3.2 Radds 存储层架构设计

1、存储引擎设计概述

我们要实现一个写性能优秀的存储引擎，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，简单地来说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

- (1) 一次日志文件的顺序写
- (2) 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）。

3.2.1 内存可变数据结构 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 $O(\log n)$ 。

3.2.2 内存不可变数据结构 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。

3.2.3 日志文件数据结构 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，抑或是宿主机器发生了宕机，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、其他压缩异常（较为复杂，首先不在这里介绍）；

异常发生时，处理的情况分两种：

- 1、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；

2、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。

3.2.4 磁盘持久化数据结构 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 memtable 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。

注意，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。

3.2.5 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；代码清单 4.12

代码清单 3.1 tFile

```
type tFile struct {
    fd           storage.FileDesc
    seekLeft     int32
```

```
    size      int64  
    imin, imax internalKey  
}
```

3.2.6 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。

代码清单 3.2 tFile

```
type version struct {  
    s *session // session - version  
    levels []tFiles // file meta  
    cLevel int // next level  
    cScore float64 // current score  
    cSeek unsafe.Pointer  
    closing bool  
    ref     int  
    released bool  
}
```

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。

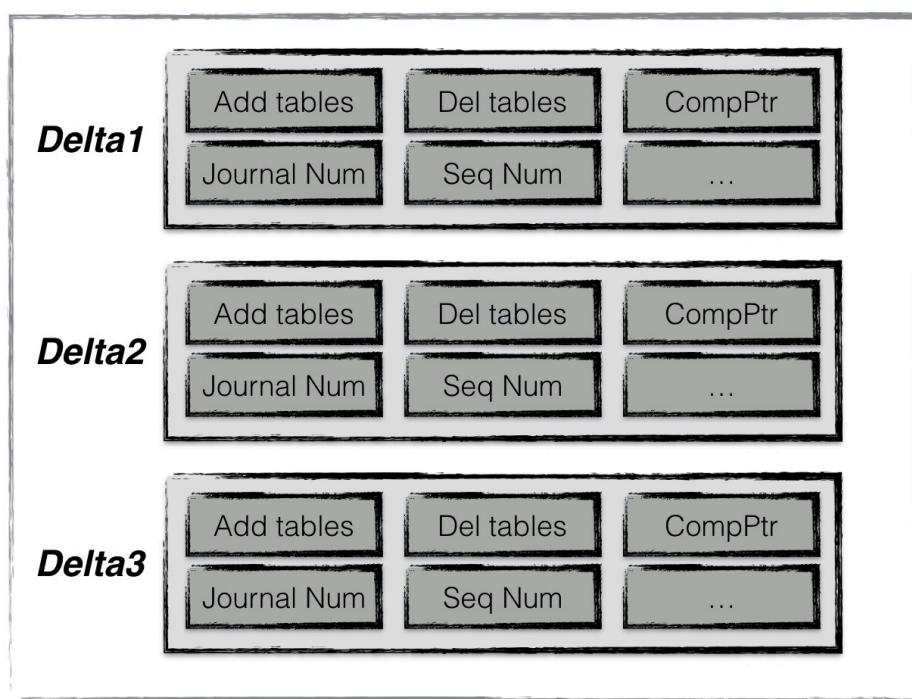


图 3.3 版本信息记录数据结构

3.3 共识层功能总体设计

3.4 客户端功能总体设计

3.4.1 API 客户端服务平台总体设计

3.4.2 gRPC API 客户端总体设计

3.4.3 RESTful API 客户端总体设计

3.4.4 CLI 客户端总体设计

3.5 本章小结

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 基础层详细设计与实现

4.1.1 错误处理的实现

针对 go 语言本身的特性，错误处理成为整个系统程序开发的首要项目，我们以轻量化，插件化的形式进行错误处理。

代码清单 4.1 Errors

4.1.2 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。

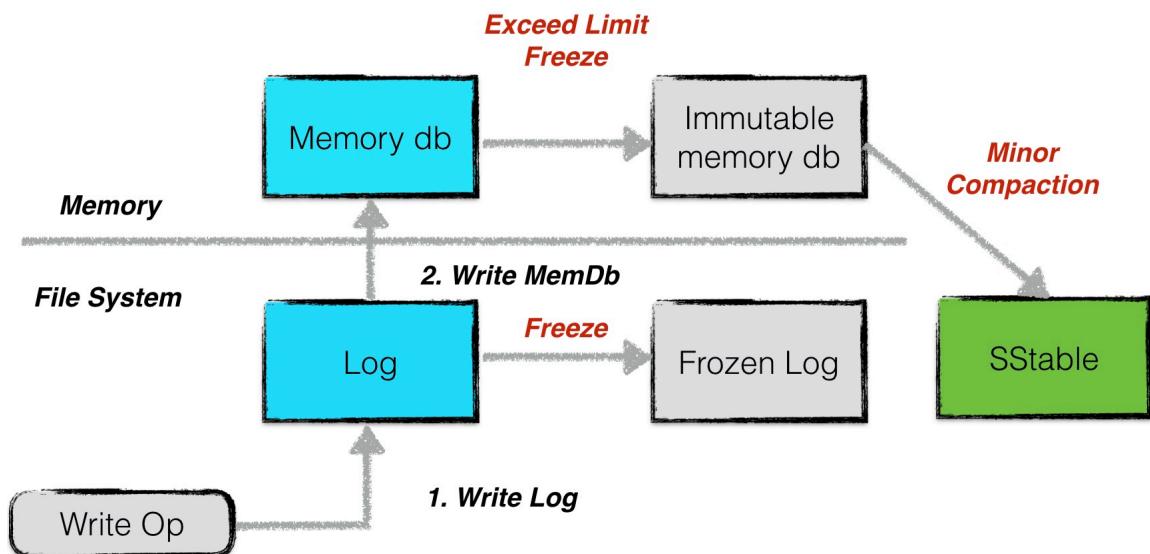


图 4.1 日志系统架构图

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。

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：

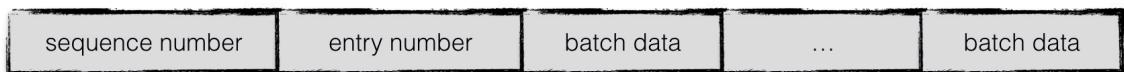


图 4.3 日志文件格式图

一条日志记录的内容包含：Header 和 Data 其中 Header 中有（1）当前 db 的 sequence number （2）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

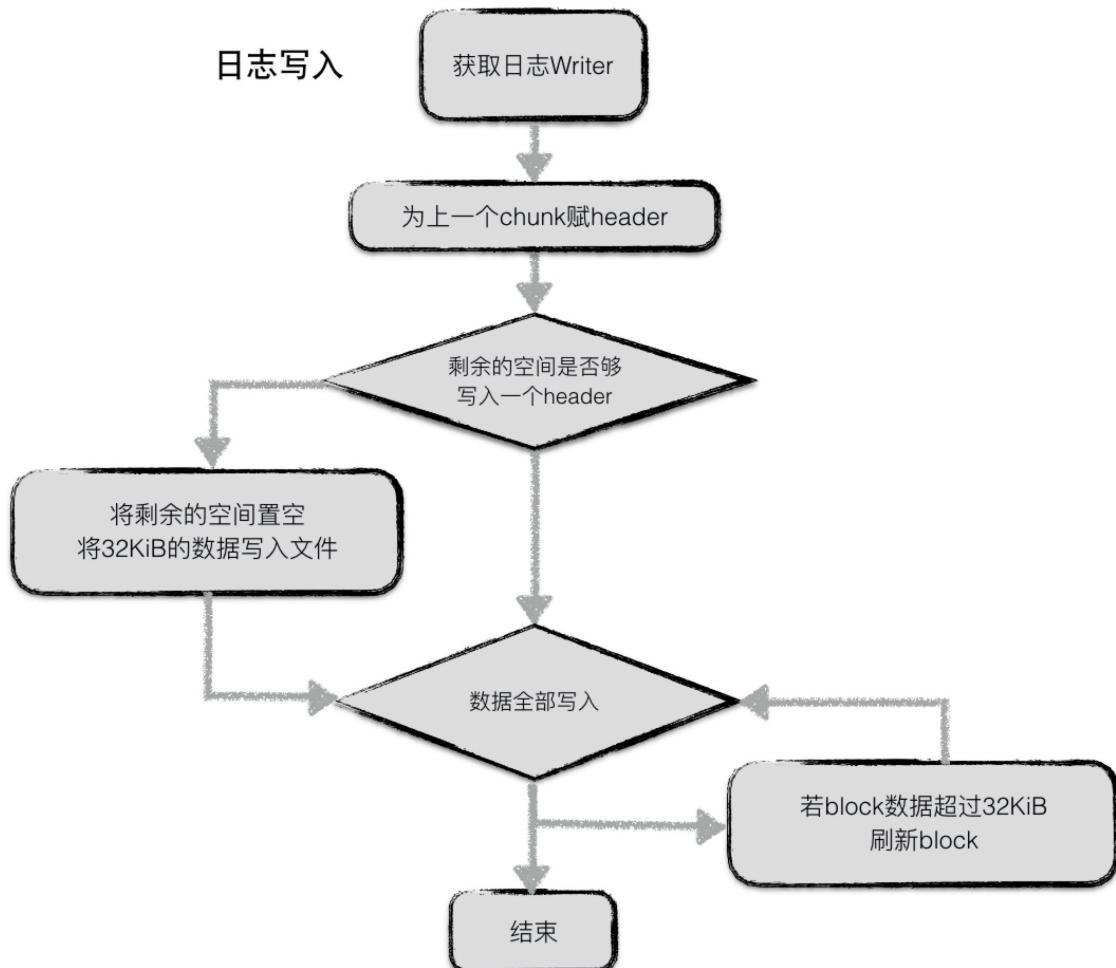


图 4.4 日志文件写流程图

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。

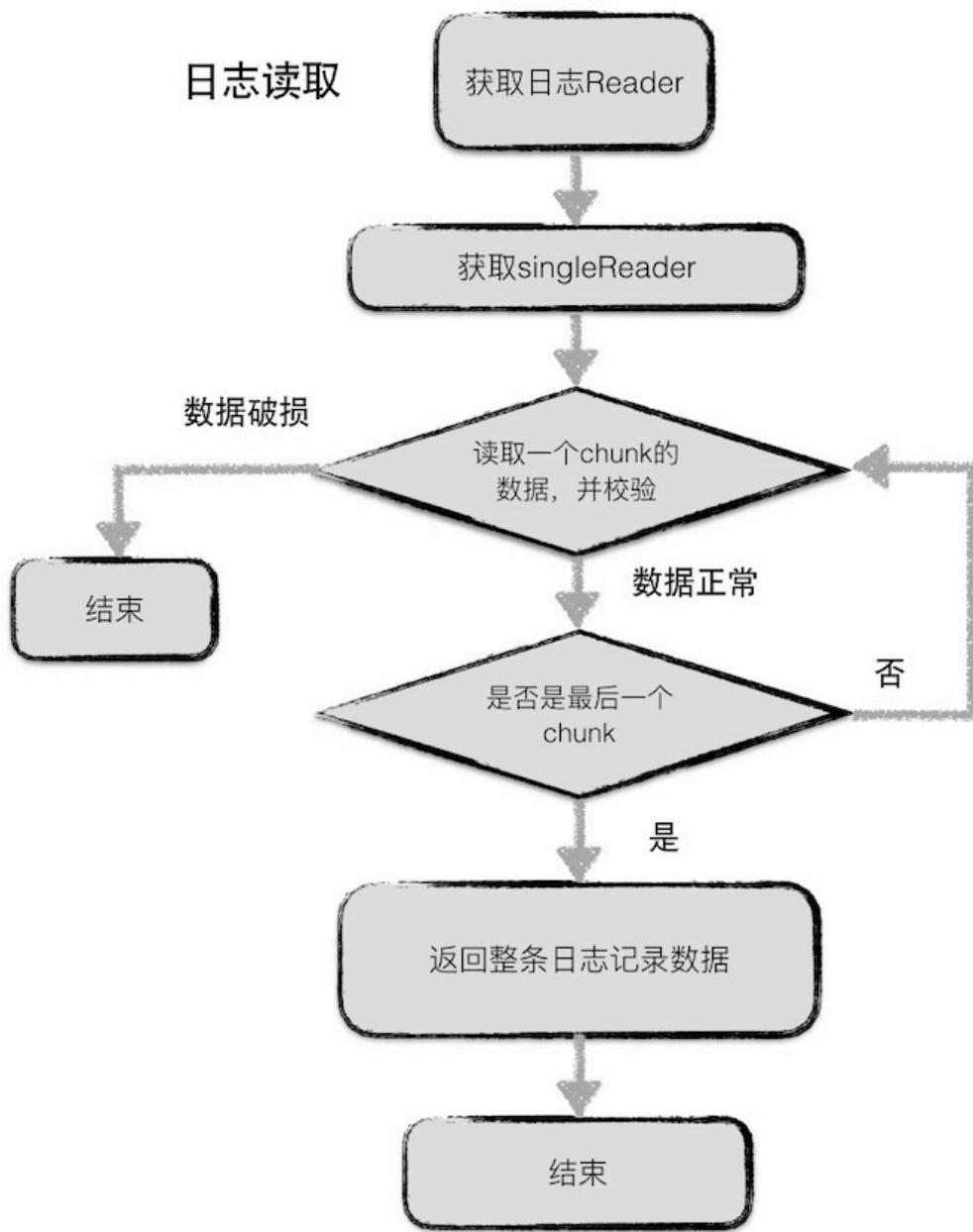


图 4.5 日志文件读流程图

4.1.3 其他工具库的实现

4.2 存储层详细设计与实现

4.2.1 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我们的写入性能。

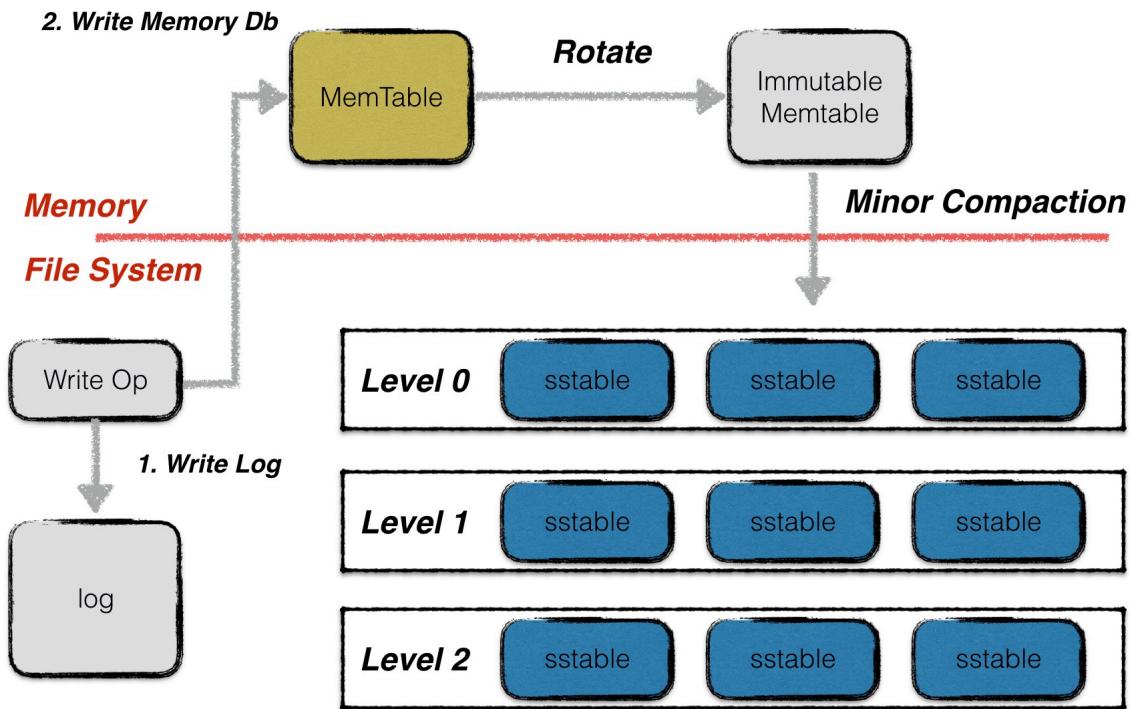


图 4.6 存储系统写数据流程

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这些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：(1) Put (2) 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，我们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。

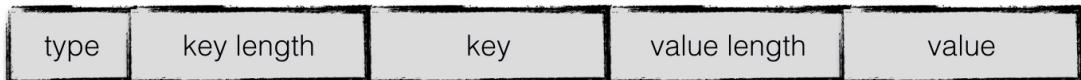


图 4.7 batch 的数据结构

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。

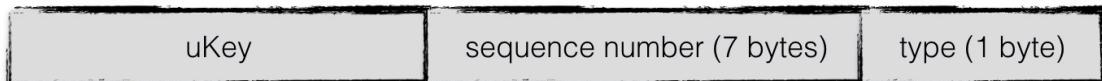


图 4.8 internalkey 的数据结构

internalkey 在用户 key 的基础上，尾部追加了 8 个字节，用于存储（1）该操作对应的 sequence number （2）该操作的类型。

其中，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，且有其他写操作 pending 的情况下，将其他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，或者无其他 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；

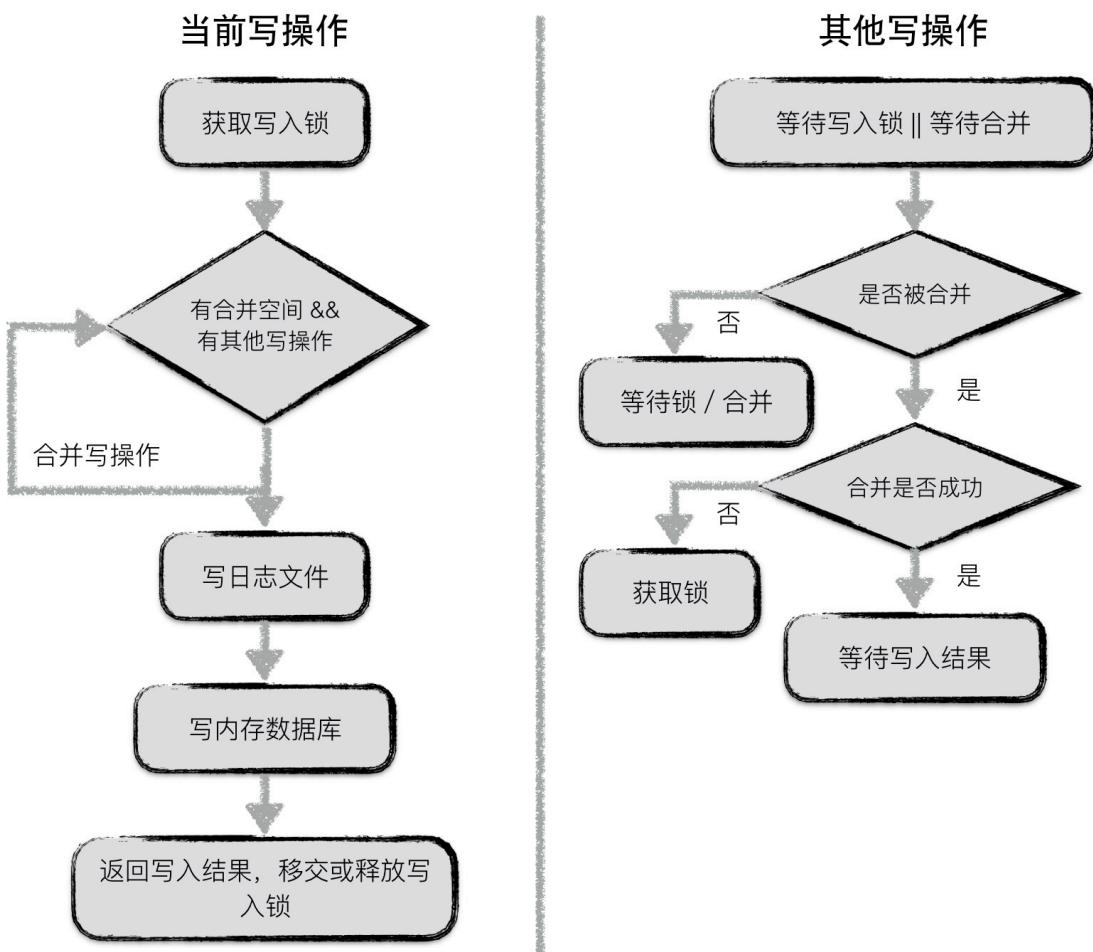


图 4.9 写合并的流程

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：

- (1) 写日志未开始，或写日志完成一半，进程异常退出；(2) 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。

4.2.2 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。

读者可能不了解 snapshot（快照）到底是什么？简单地来说，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，其实就代表着存储系统的一个状态。换句话说，每一个序列号都可以作为一个状态快照。

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”。

Seq: 100	Key: "name"	Delete
Seq: 99	Key: "name"	Value: "dog"
Seq: 98	Key: "name"	Value: "cat"

Snapshot(98) →

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

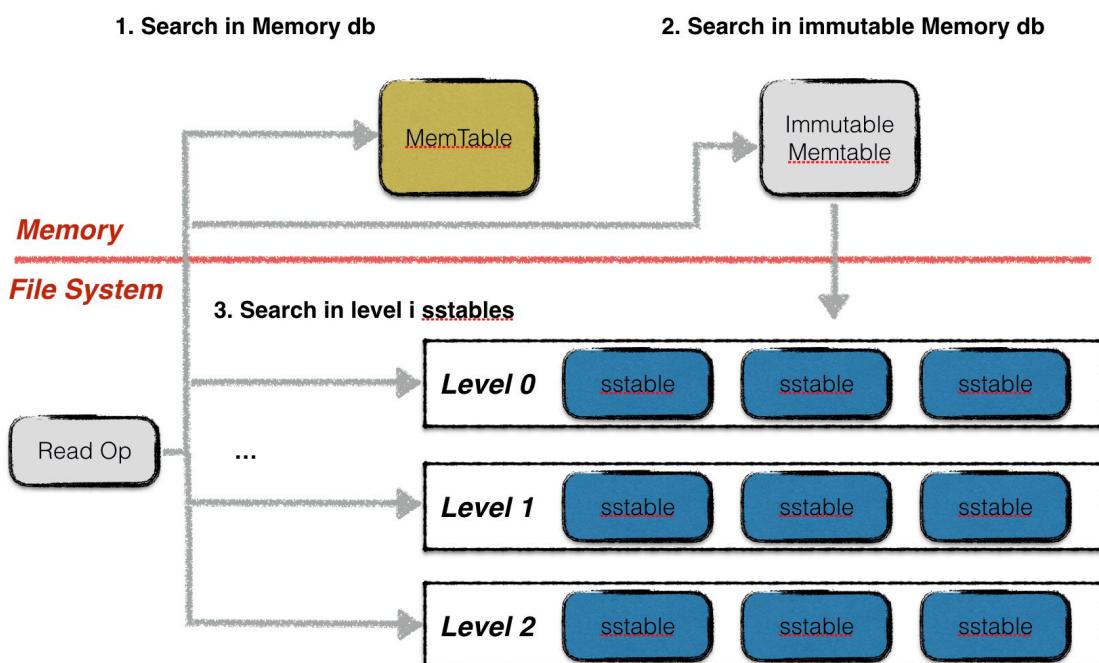


图 4.11 读数据流程

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；

注意存储系统在每一层 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，

4.2.3 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，接下来将介绍一下内存数据库的实现细节。

1、跳表的实现

跳表（SkipList）是由 William Pugh 提出的。他在论文《Skip lists: a probabilistic alternative to balanced trees》中详细地介绍了有关跳表结构、插入删除操作的细节。

这种数据结构是利用概率均衡技术，加快简化插入、删除操作，且保证绝大多数操作均拥有 $O(\log n)$ 的良好效率。

原文的一段话道出了跳表在数据结构中运用离散数学知识的精髓：数据结构是离散的，计算机的本质是离散的。

Skip lists are a data structure that can be used in place of balanced trees. Skip lists use probabilistic balancing rather than strictly enforced balancing and as a result the algorithms for insertion and deletion in skip lists are much simpler and significantly faster than equivalent algorithms for balanced trees.

图 4.12 跳表的影响

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就是借助概率平衡，来构建一个快速且简单的数据结构，取代平衡树。

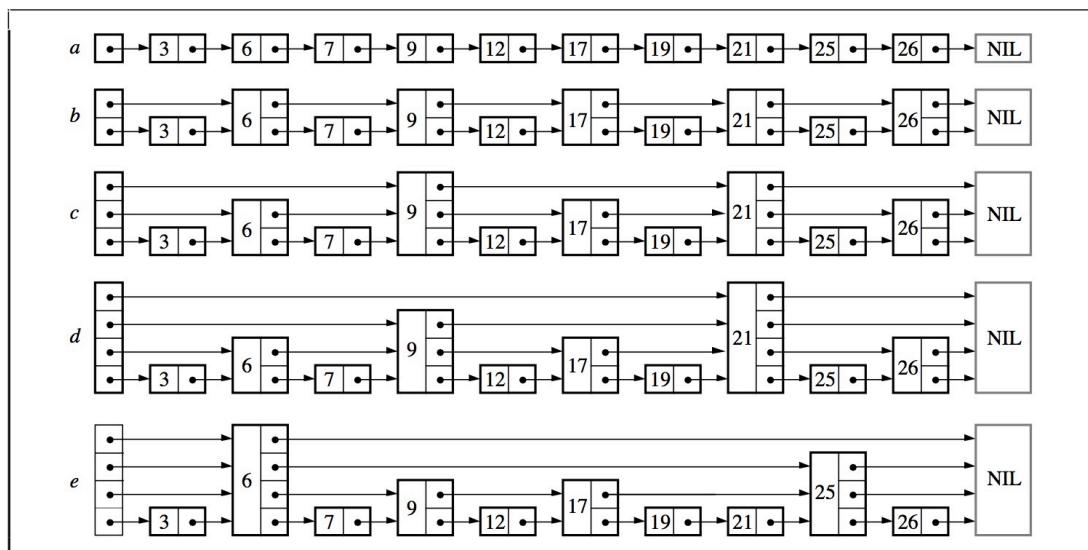


FIGURE 1 - Linked lists with additional pointers

图 4.13 一个跳表的图

作者从链表讲起，一步步引出了跳表这种结构的由来。

图 a 中，所有元素按序排列，被存储在一个链表中，则一次查询之多需要比较 N 个链表节点；

图 b 中，每隔 2 个链表节点，新增一个额外的指针，该指针指向间距为 2 的下一个节点，如此以来，借助这些额外的指针，一次查询至多只需要 $\lceil n/2 \rceil + 1$ 次比较；

图 c 中，在图 b 的基础上，每隔 4 个链表节点，新增一个额外的指针，指向间距为 4 的下一个节点，一次查询至多需要 $\lceil n/4 \rceil + 2$ 次比较；

作者推论，若每隔 2 个节点，新增一个辅助指针，最终一次节点的查询效率为 $O(\log n)$ 。但是这样不断地新增指针，使得一次插入、删除操作将会变得非常复杂。

一个拥有 k 个指针的结点称为一个 k 层结点 (level k node)。按照上面的逻辑，50% 的结点为 1 层节点，25% 的结点为 2 层节点，12.5%。若保证每层节点的分布如上述概率所示，则仍然能够相同的查询效率。图 e 便是一个示例。

维护这些辅助指针将会带来较大的复杂度，因此作者将每一层中，每个节点的辅助指针指向该层中下一个节点。故在插入删除操作时，只需跟操作链表一样，修改相关的前后两个节点的内容即可完成，作者将这种数据结构称为跳表。

2、跳表的结构

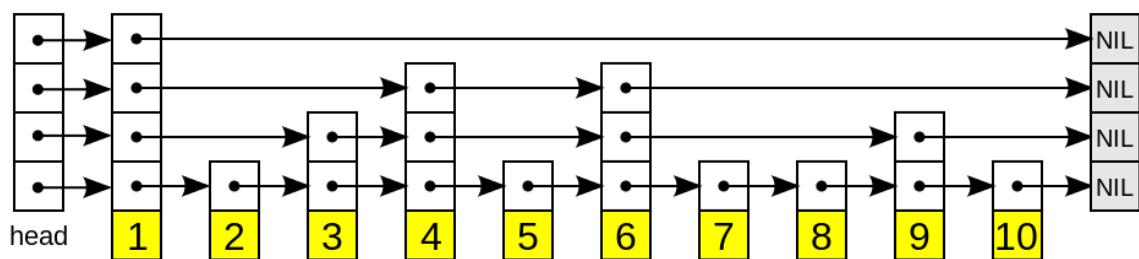


图 4.14 跳表的结构

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

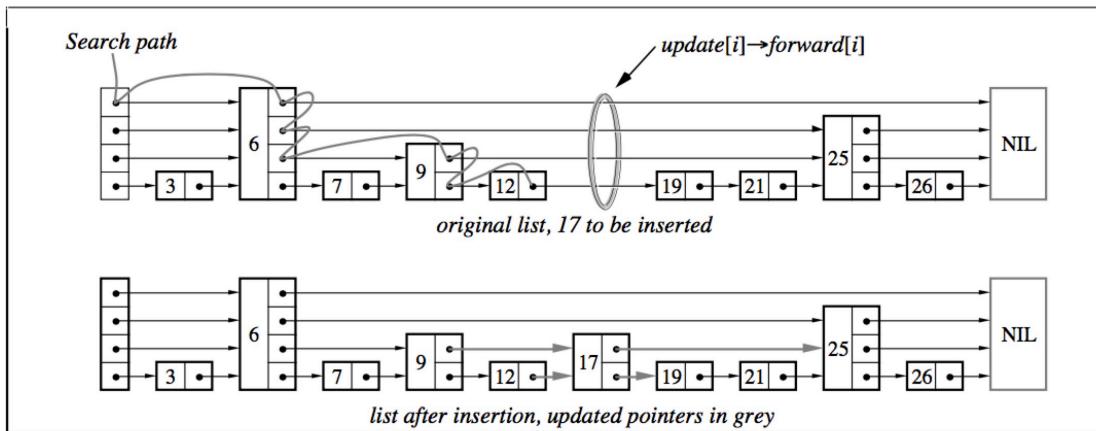


FIGURE 3 - Pictorial description of steps involved in performing an insertion

图 4.15 跳表的查找图

在介绍插入和删除操作之前，我们首先介绍查找操作，该操作是上述两个操作的基础。

例如图中，需要查找一个值为 17 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

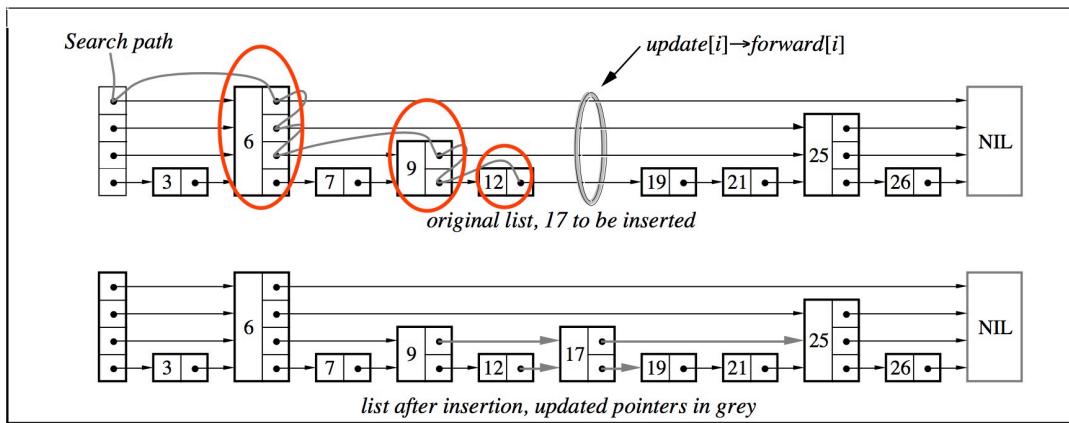


FIGURE 3 - Pictorial description of steps involved in performing an insertion

图 4.16 跳表的插入图

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（例如图中节点 12 与节点 19 之间），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 4.2 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；

4.2.4 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，我们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；

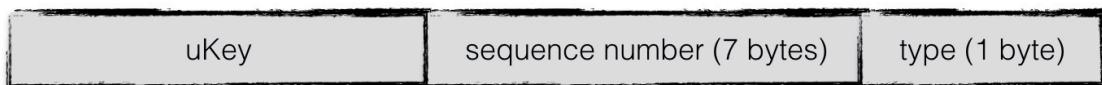


图 4.17 内存数据库内部键

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。在这里介绍一下系统默认的比较规则。

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
    // Node data:
    // [0]          : KV offset
    // [1]          : Key length
    // [2]          : Value length
    // [3]          : Height
    // [3..height] : Next nodes
    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。

4.2.5 持久化数据存储的实现

1、sstable 概述

如我们之前提到的，系统是的 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；CRC 校验码

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

图 4.18 sstable 物理结构

(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；

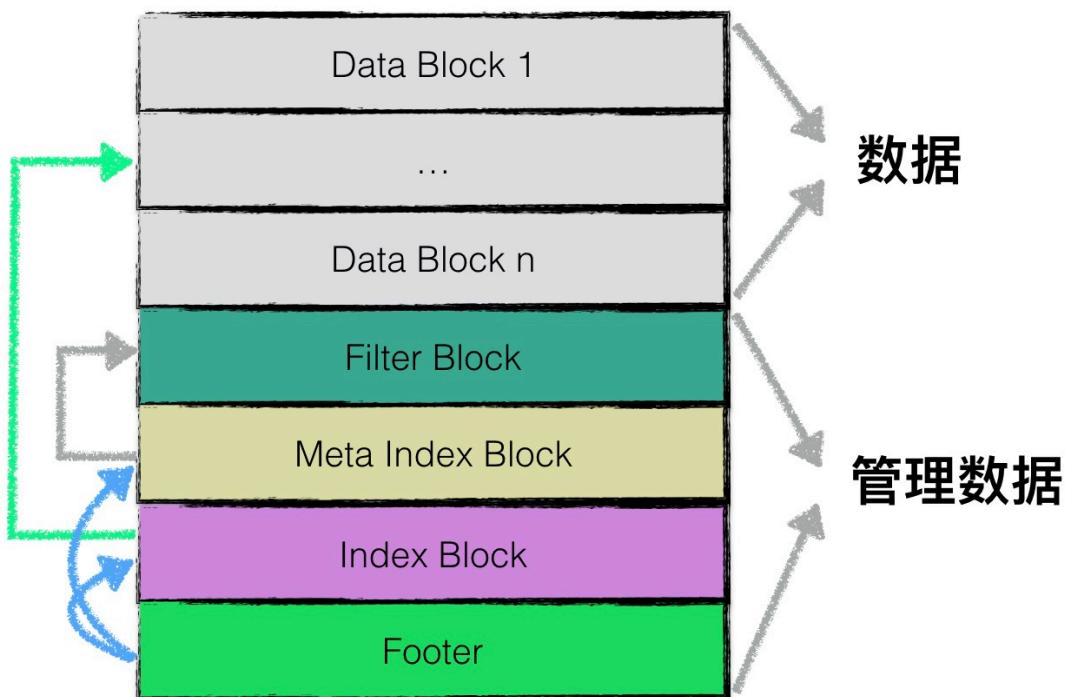


图 4.19 sstable 逻辑结构

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：

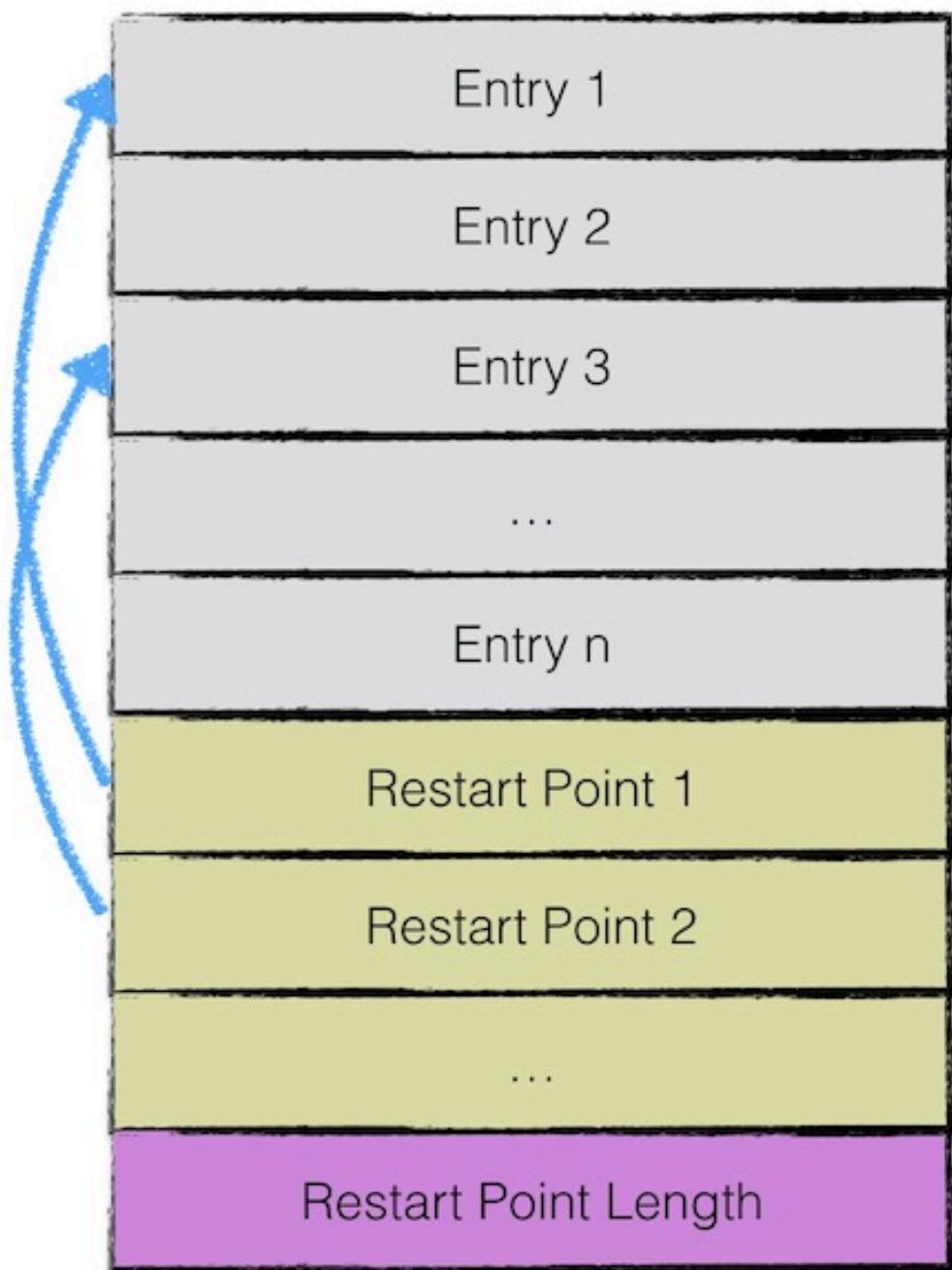


图 4.20 sstable data block

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 4.21 sstable entry format

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；



图 4.22 sstable data block 示例图 1

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：

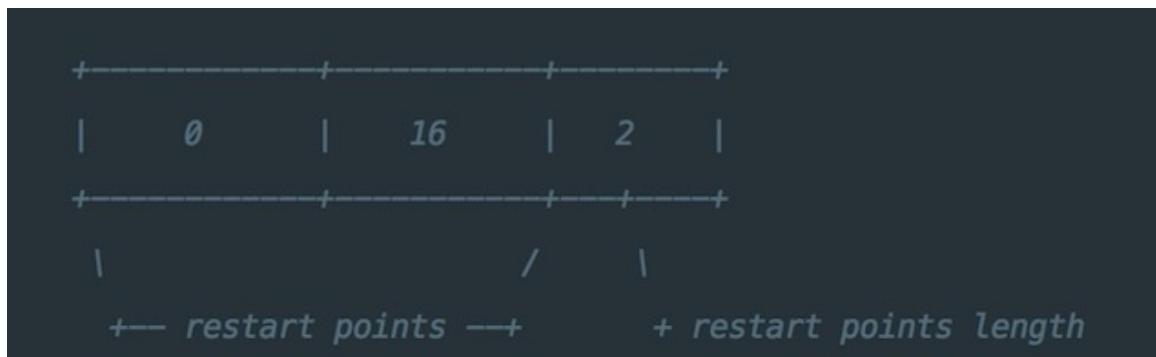


图 4.23 sstable data block 示例图 2

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，

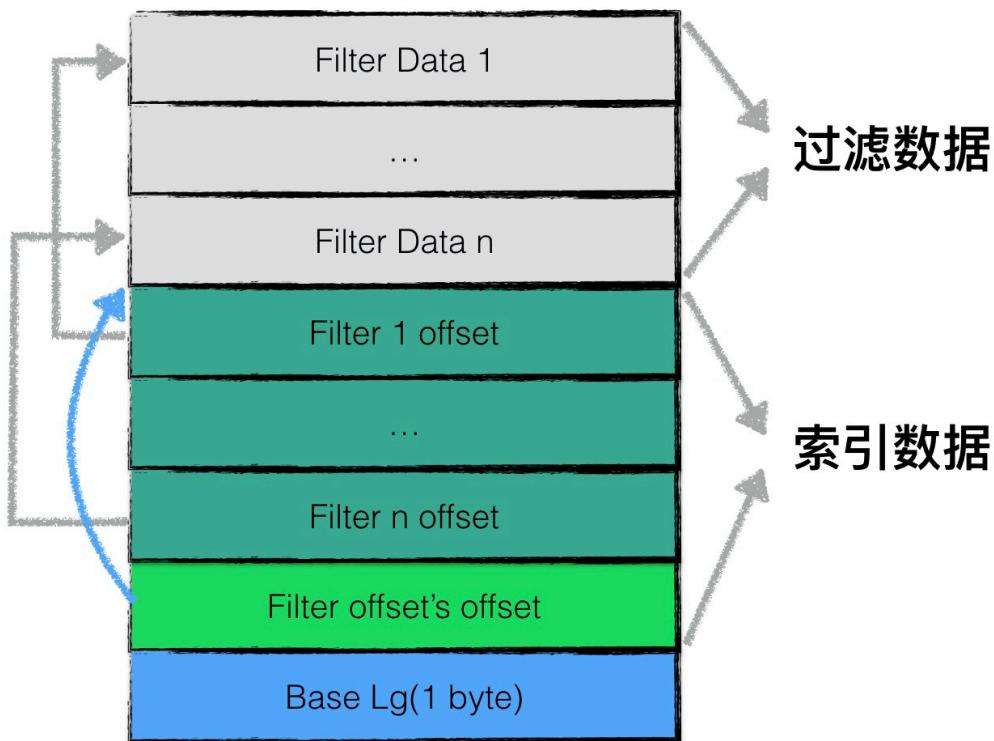


图 4.24 sstable filterblock format

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；

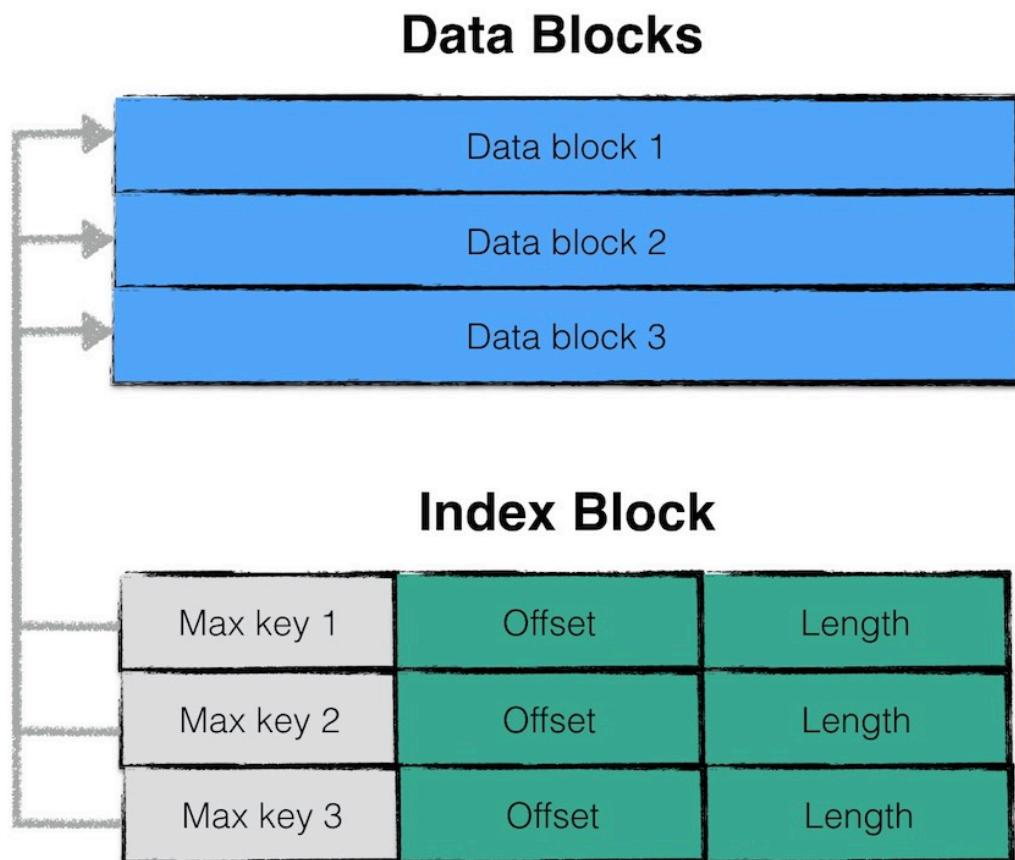


图 4.25 sstable indexblock format

其中，data block i 最大的 key 值还是 index block 中该条记录的 key 值。

如此设计的目的是，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，内容为：“<http://code.google.com/p/>”字符串 sha1 哈希的前 8 个字节。



图 4.26 sstable footer format

3、sstable 读写操作

在介绍完 sstable 文件具体的组织方式之后，我们再来介绍一下相关的读写操作。为了便于理解，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 4.3 tWriter

```
// tWriter wraps the table writer. It keep track of file descriptor
// and added key range.

type tWriter struct {

    t *t0ps

    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：(1) 文件编码 (2) 大小 (3) 最大 key 值 (4) 最小 key 值 故，理解 tableWriter 的 Append 函数是理解整个写入过程的关键。

i.tableWriter

在介绍 append 函数之前，首先介绍一下 tableWriter 这个数据结构。主要的定义如下：

代码清单 4.4 Writer

```
// 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 4.5 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，编码的格式如上文中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《存储系统源码解析 - 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，将在接下来的文章中继续介绍。

（2）读操作

读操作作为写操作的逆过程，充分理解了写操作，将会帮助理解读操作。

下图为在一个 sstable 中查找某个数据项的流程图：

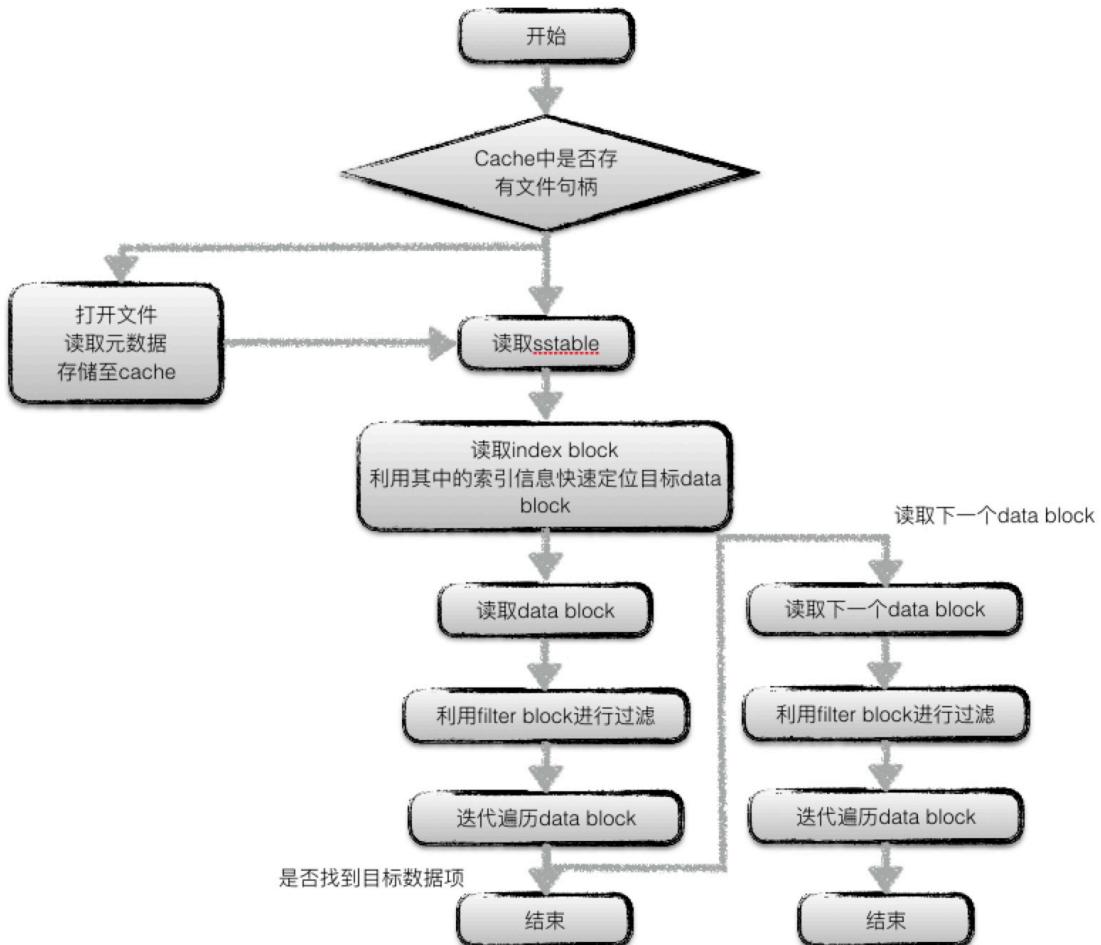


图 4.27 sstable read procedure

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

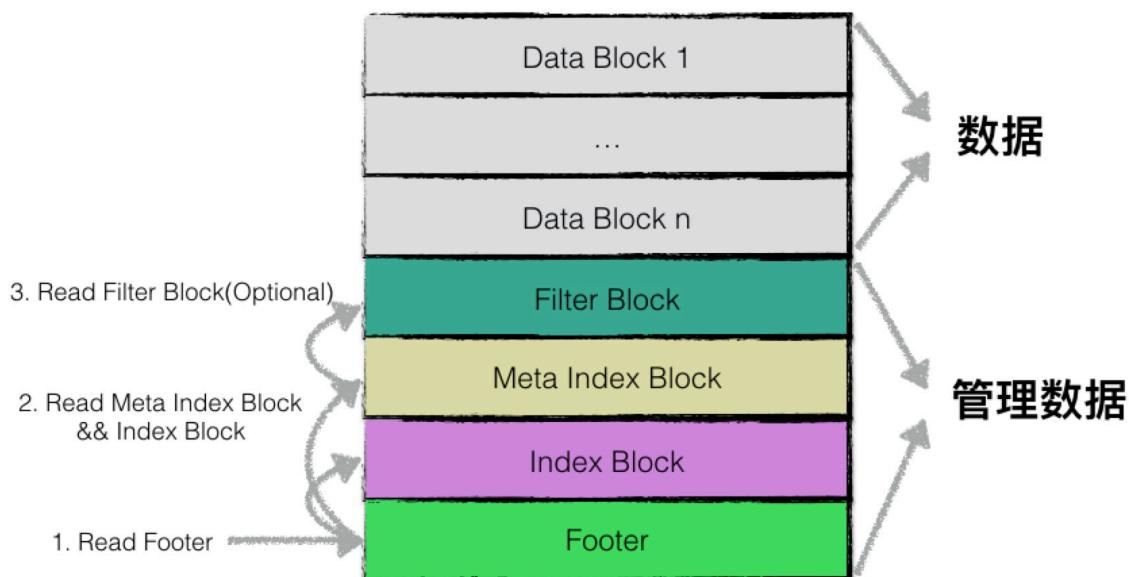


图 4.28 sstable metadata

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：

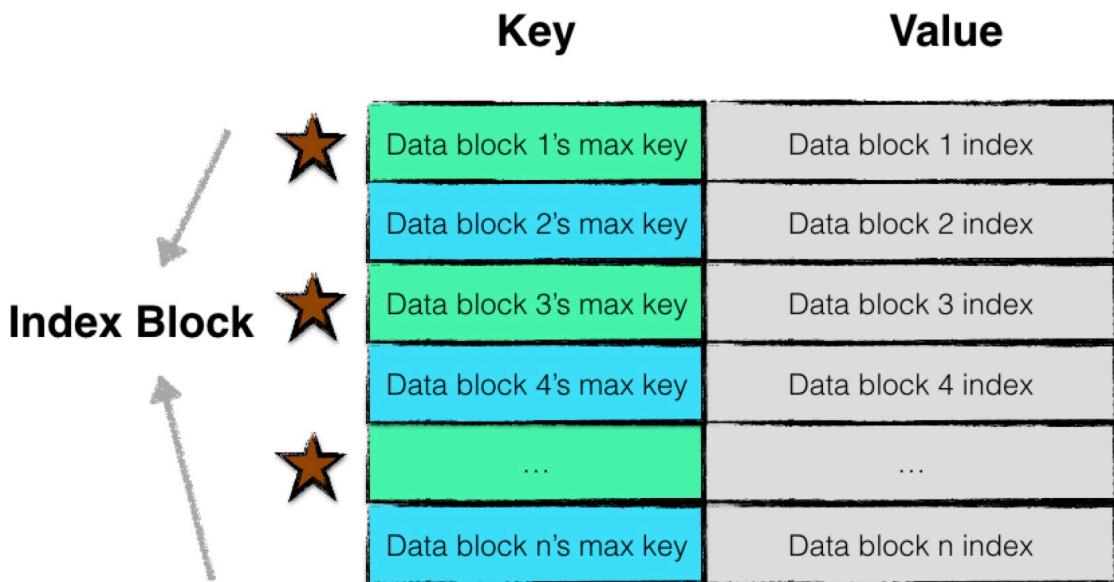


图 4.29 sstable index block

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”。

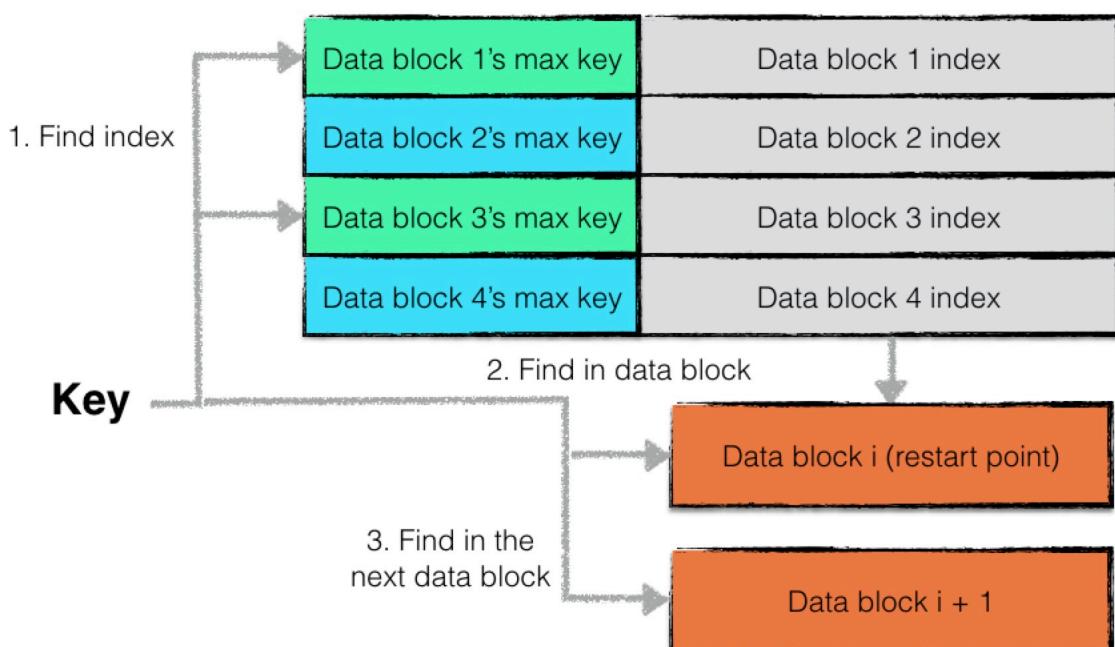


图 4.30 sstable index block find

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

data block 中；具体的原理可能参见《布隆过滤器》。

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

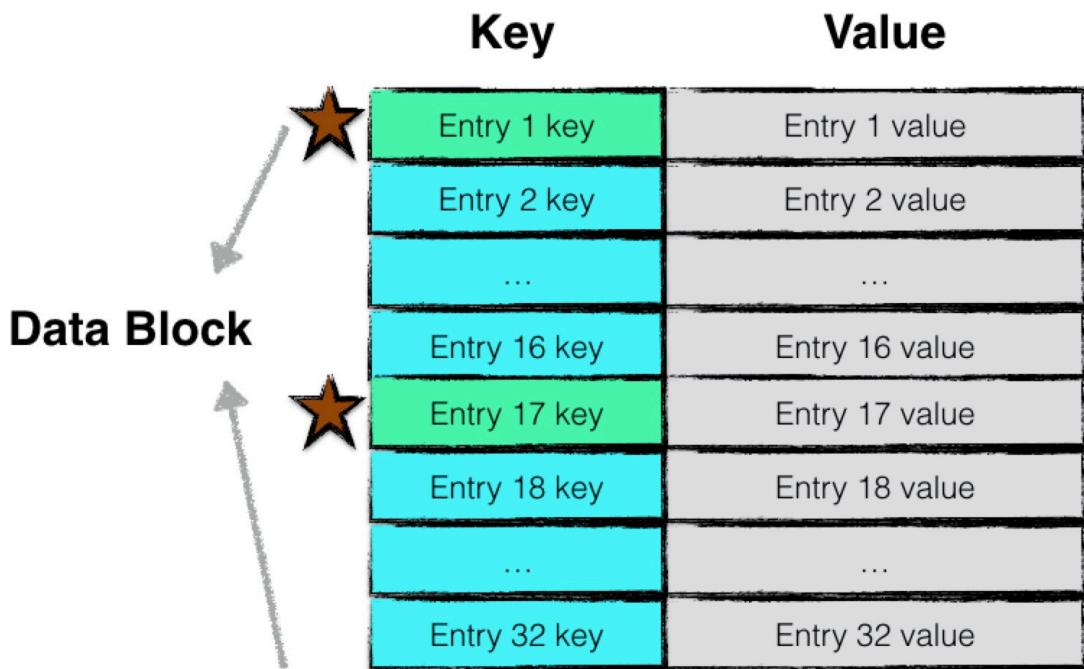


图 4.31 sstable datablock format

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。当读取时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。

4.2.6 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，我们首先介绍一下 **leveldb** 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；

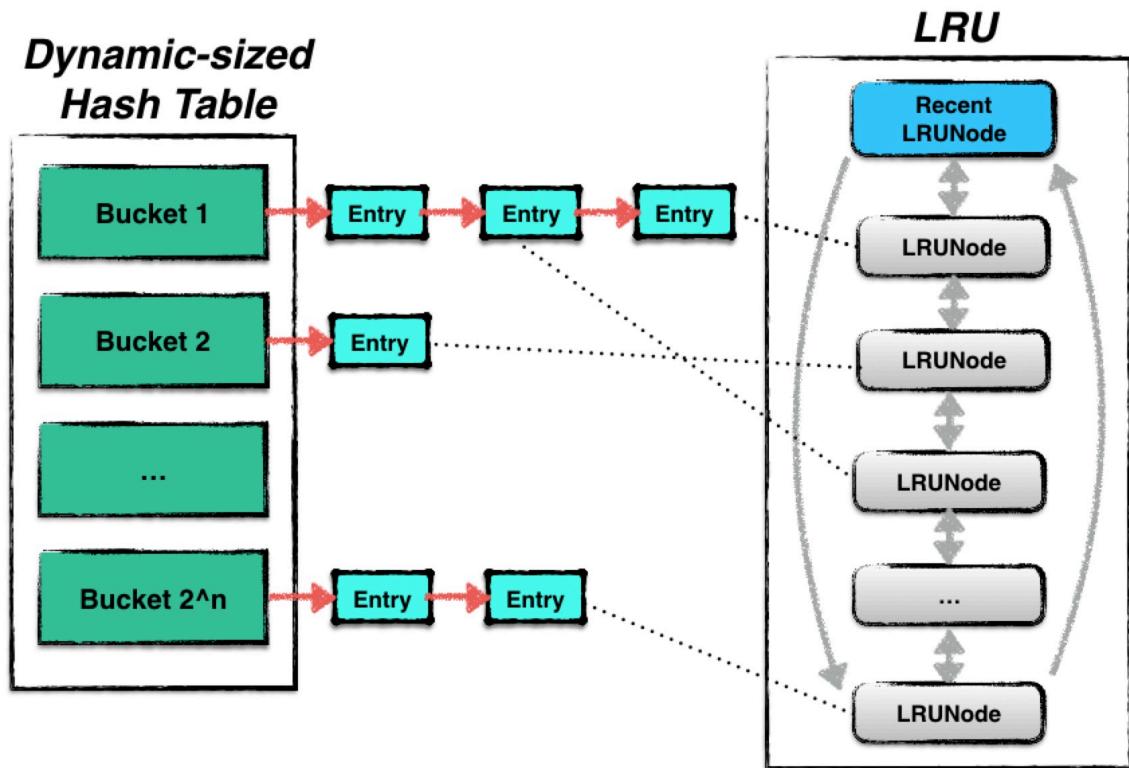


图 4.32 sstable cache arch

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 的过程中不阻塞其他并发的读写请求。

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，那么会导致此刻其他的读、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

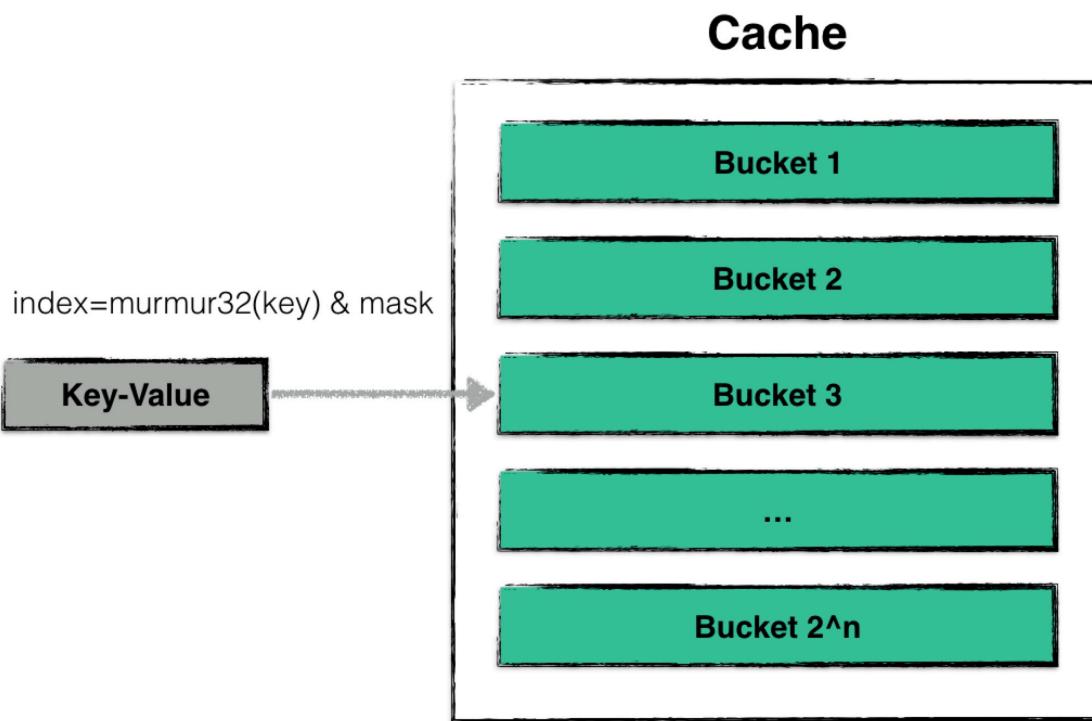


图 4.33 sstable cache select

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

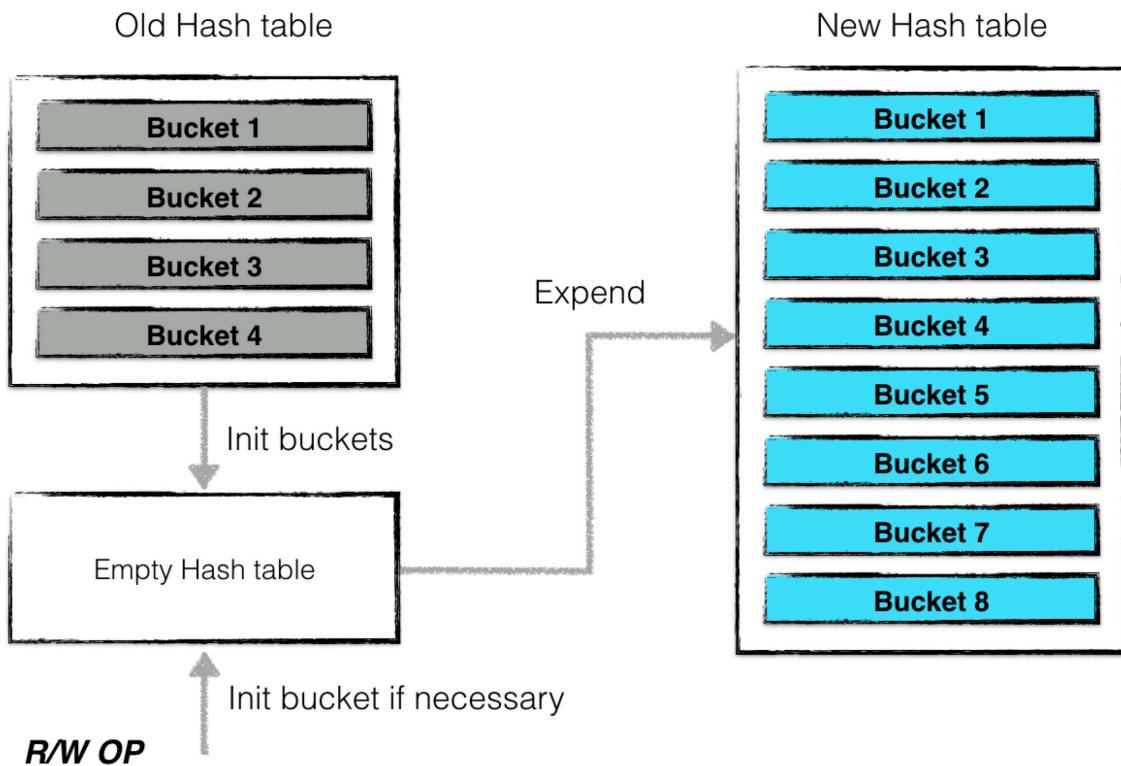


图 4.34 sstable cache expand

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 4.6 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他线程正在使用，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。

4.2.7 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。

leveldb 中利用布隆过滤器判断指定的 key 值是否存在与 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 4.35 sstable bloom1

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。

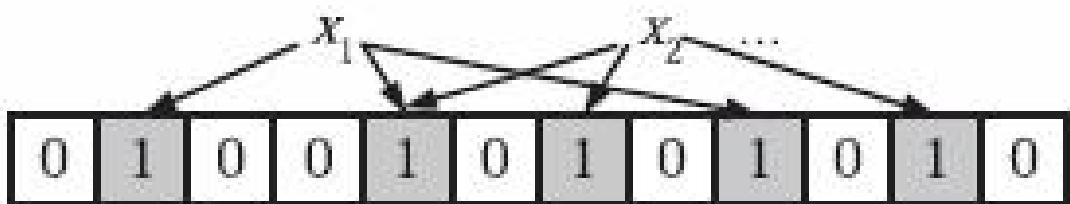


图 4.36 sstable bloom2

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 y_1 一定不存在；而 y_2 可能存在。

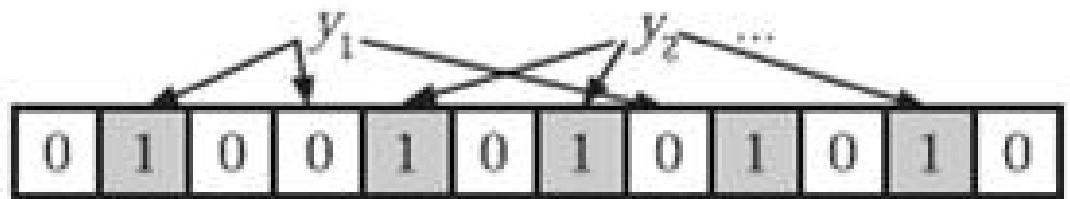


图 4.37 sstable bloom3

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

leveldb 中的布隆过滤器实现较为简单，以 goleveldb 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 4.7 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 4.8 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 4.9 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 4.10 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 4.11 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 4.12 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
}
```

4.2.8 数据压缩系统的实现

4.2.9 版本控制的实现

4.3 共识层详细设计与实现

4.4 客户端层详细设计与实现

4.4.1 API 客户端服务平台的实现

4.4.2 gRPC API 客户端的实现

4.4.3 RESTful API 客户端的实现

4.4.4 CLI 命令行客户端的实现

4.5 本章小结

5 Radds 存储系统部署、日志分析与客户端测试

5.1 分布式存储系统部署

5.1.1 在 X86-64 GNU/Linux Ubuntu22.04 操作系统部署

5.1.2 在 X86-64 Windows11 操作系统部署

5.1.3 在 Arm64 Darwin MacOS 操作系统部署

5.2 数据存储系统日志分析

5.3 共识性系统日志分析

5.4 客户端服务平台日志分析

5.5 客户端测试

5.5.1 gRPC API 客户端测试

5.5.2 RESTful API 客户端测试

5.5.3 CLI 客户端测试

5.6 本章小结

结论

- 1、调研了...
- 2、调研...
- 3、设计...
- 4、依据...
- 5、对整个系统...

致 谢

附录

附录 A

In Search of an Understandable Consensus Algorithm(Extended Version)

中文译文 A

Raft 寻找一种易于理解的共识性算法

附录 B

The Log-Structured Merge-Tree

中文译文 B

日志结构归并树