分类号_____ 学号 M201672669

学校代码 10487 密级_____

华中科技大学

硕士学位论文

一种基于混合存储的矩阵结构键值存储系统的研究与实现

学位申请人 朱承浩

学 科 专 业： 计算机系统结构

指 导 教 师： 万继光 教授

答 辩 日 期： 2019.5.22

A Thesis Submitted in Partial Fulfillment of the Requirements

for the Degree of Master of Engineering

Research and Implementation of An Efficient KV store with Matrix Tables based on A heterogeneous System

Candidate : Zhu Chenghao

Major : Computer Architecture

Supervisor : Prof. Wan Jiguang

Huazhong University of Science and Technology

Wuhan, Hubei 430074, P.R.China

独创性声明

本人声明所呈交的学位论文是我个人在导师指导下进行的研究工作及取得的研究成果。尽我所知，除文中已经标明引用的内容外，本论文不包含任何其他个人或集体已经发表或撰写过的研究成果。对本文的研究做出贡献的个人和集体，均已在文中以明确方式标明。本人完全意识到本声明的法律结果由本人承担。

学位论文作者签名：

日期：　　年　　月　　日

学位论文版权使用授权书

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

本论文属于　保密□，在_____年解密后适用本授权书。

　　不保密□。

（请在以上方框内打“√”）

学位论文作者签名：　　指导教师签名：

日期：　　年　　月　　日　　日期：　　年　　月　　日

摘 要

键值存储系统由于其优秀的性能及可无限扩展的特性被广泛地应用到现代数据中心。然而随着大数据时代的来临，业界对于存储系统的性能需求度越来越大，人们对 KV 键值系统的性能提出了更高的需求。非易失性存储（Non-Volatile Memory，NVM）的出现，使得这样的需求成为了现实。NVM 是一种新型存储设备，具有磁盘的非易失特性及 DRAM 的高速特性，能够给传统存储系统的性能带来一个质的提升。

针对于当前时代背景，以及传统日志结构合并树（Log-Structured Merged Tree，LSM 树）键值存储系统的性能波动与写放大等问题，本文提出了一种基于 LSM 树结构的混合存储系统 MatrixKV。MatrixKV 系统修改了传统 LSM 树结构，在 NVM 中设置了一种大容量的矩阵形式数据结构 Matrix-Table，代替 LSM 树的 L0 层，Matrix-Table 内部采用 table 堆叠成行的方式形成多层结构，并在每层 table 之间建立相应的索引关系，加速 Matrix-Table 内部的数据查找；同时针对于 Matrix-Table 设计了一种以列为单位的细粒度 Compaction 策略，减少单次 Compaction 所涉及的数据量，避免给系统带来较大的性能波动，并提升系统的性能；利用 NVM 设备的特性及索引指针，在不影响读性能的前提下，扩大了 L0 层的空间，减少 LSM 树的层次，降低了写放大。

课题基于开源 RocksDB 实现了 MatrixKV 系统，并在一种真实的 Intel 3D Xpoint 非易失性存储设备——Apache Pass 上进行了性能测试。测试结果表明 MatrixKV 可以有效缓解过多数据的 Compaction 操作给系统性能带来的波动影响，并较好地减少了 LSM 树结构的数据层次，降低了写放大。同时，MatrixKV 的随机写性能是 RocksDB 的 3.91~4.99 倍，是 NoveLSM 的 1.66~2.27 倍；随机读性能是 RocksDB 的 2.70~8.65 倍，是 NoveLSM 的 2.14~3.99 倍。

关键词：键值存储，非易失存储器，日志结构合并树，矩阵结构，列合并

Abstract

With the advent of the era of big data, the demand of high-performance storage device and system is growing significantly. Key-value stores have been widely used in data centers due to their excellent performance and unlimited horizontal scalability in theory. However, enormously-growing data requires better performance on key-value stores. The emerging of NVM(Non-Volatile Memory) make it a reality. NVM has both the non-volatile feature of HDD and the high-performance feature of DRAM. It can bring favourable performance improvement in conventional storage system.

In this paper, we propose MatrixKV, an LSM-tree based persistent KV store on the heterogeneous storage architecture, to mitigate system stalls and write amplification in LSM-tree. We propose a matrix table in NVM to manage L0's data, which adopts a table-stacking structure and index tables in different levels to accelerate query in matrix table. We design a fine granularity column compaction between L0 and L1 based on matrix table to reduce involved data in each compaction, resulting in less performance fluctuation and higher performance. We increase the size of L0 in matrix table to reduce LSM-tree's levels and thus mitigate write amplification without penalties.

We implement MartixKV based on RocksDB and evaluate it on a real 3D Xpoint NVM device—Apache Pass. Evaluation results show that MatrixKV gives more stable random write performance without system stalls and reduces both LSM-tree levels and write amplification. For random write, MatrixKV can improve throughput 3.91x to 4.99x compared to RocksDB and 1.66x to 2.27x compared to NoveLSM. For random read, MatrixKV can improve throughput 2.7x to 8.65x over RocksDB and 2.14x to 3.99x over NoveLSM.

Key words: Key-Value Store, Non-Volatile Memory, LSM-Tree, Matrix Structure, Column Compaction

目 录

1 绪论

1.1 研究背景与意义

自第三次工业革命以来，伴随着科技的进步，信息技术已经存在于人类生活的每一个角落。尤其是近几年人工智能技术的逐渐成熟，计算机可以通过不断地学习来模拟人类的某些思维过程与智能行为，代替人脑来承担和完成一些较为复杂的工作。信息科技的进步也伴随着数据爆炸的问题。人工智能的学习数据集、大数据分析处理等新型技术对数据量的需求越来越大。麦肯锡咨询公司曾就数据爆炸问题提出："数据，已经渗透到当今每一个行业和业务职能领域，成为重要的生产因素。人们对于海量数据的挖掘和运用，预示着新一波生产率增长和消费者盈余浪潮的到来[1]。"人类已然进入了大数据的时代。大数据时代背景之下，全球的数据总量以指数级速率增长，截至 2012 年，全球人类历史所有印刷书籍数据总量约为 0.2EB，而到 2018 年时，全球的数据总量达到了 33ZB，预计到 2025 年，这个数字将变为 175ZB[2]。因此如何存储和管理海量数据，对新时代背景下的计算机技术提出了新的挑战。

数据量的爆炸式扩大，根本原因在于现今计算机系统性能的飞速发展。1965 年 Gordon Moore 提出了著名的摩尔定律——当价格不变时，集成电路上可容纳的元器件的数目，约每隔 18-24 个月便会增加一倍，性能也将提升一倍[3]。伴随着时间的发展，计算机的处理计算能力在指数级增长。然而飞速发展的技术并没有给计算机系统的整体性能带来了质的变化。计算机存储的性能瓶颈正在严重限制着计算机技术的进步。传统的存储设备如机械硬盘与固态硬盘上的技术研究主要在于单位密度的存储容量上，而在主体的读写速度上并没有得到突破性进展。H Garnell 等人的研究[4]表明，自 20 世纪 80 年代起，之后的约 20 年里，CPU 的计算速度提升了约 1000 倍，而存储设备的速度仅仅提升了约 10 倍。

传统技术下的高速存储设备动态随机存储器（Dynamic Random Access Memory，DRAM）虽然拥有的较高的读写速度，但其由于易失的特性，以及相对

较小的存储密度，很难作为数据存储和持久化的设备，现今计算机系统中一般常应用 DRAM 作为内存设备。而常见的固态硬盘、机械硬盘等，其 I/O 速度只能达到 100M/s~500M/s，这一速度无法和内存以及更高速 CPU 保持平衡。低速的存储外设与高速的内存及 CPU 之间的速度差异是限制现今计算机系统发展的重要原因之一。传统的存储技术无法跟上 CPU 技术的发展，也很难应对海量数据场景下的大数据存储，因此，寻求新型的既有高速 I/O 性能，又拥有非易失可持久化特性的存储技术已然是计算机存储领域一大重要研究热点。

经过科学家的不断研究与探索，非易失存储技术[5]（Non-Volatile Memory，NVM）逐渐进入了人们的视野。NVM 是一种新型存储设备，不仅拥有 DRAM 的高速可随机读取的特性，也拥有着固态硬盘、机械硬盘所拥有的非易失特性，且随着其相关工艺成熟以来，NVM 设备的特性得到了飞速提升。

NVM 虽然拥有着高速、非易失等特性，但其在现今环境下仍然拥有着较为昂贵的单位存储价格，且其相对于传统的存储设备如机械硬盘等的寿命要短很多，因此完全使用 NVM 来进行数据存储在当前环境背景下仍然不够现实，而利用其一些优异的特性来设置一个与传统存储设备进行混合异构的存储架构，却是一个新的研究方向。

针对于当前的大数据背景，非关系型数据库（NoSQL）技术得到了迅猛发展。NoSQL 数据库的产生即是为了解决大规模数据集合多重数据种类带来的挑战，其与传统的关系型数据库之间最大的区别在于其扩展方式为横向扩展，传统的关系型数据库的主要瓶颈在于多表的笛卡儿积计算，这需要更强性能的计算机才可以突破上限，而 NoSQL 天生优势在于其横向扩展的特性，可以通过增加服务器来提升负载能力，理论上其性能没有上限。因此在大数据时代，NoSQL 数据库系统由于其高可扩展及高效的大数据处理能力，已经逐渐成为了主流的系统存储架构之一。

传统的 NoSQL 数据库如 Redis[6]、Memcache[7]等基于内存和哈希结构的存储系统拥有着极高的数据检索能力，但由于其内存数据库的特性，无法支持现今的海量数据的存储与持久化。而基于 KV 键值存储的 NoSQL 数据库在近年里得到

了迅速发展，如 LevelDB[8][9]、RocksDB[10]等。主流的键值存储系统均基于 LSM 树结构设计，采用追加写的方式将所有的随机写操作转变为顺序写操作保证系统的高性能，同时采用日志形式保障系统的数据可靠性，有着不错的研究前景。

在现今的大数据时代的背景下，NVM 存储设备的高 I/O 性能和键值存储系统的大规模与高可扩展性是主要的研究方向。本课题将二者进行结合，将 NVM 存储设备应用在基于 LSM 树结构的键值存储系统中，考虑到 NVM 大小与价格的限制，采用固态硬盘与 NVM 做混合存储，设计了一种基于 NVM 与固态硬盘的混合介质的键值存储系统，并针对现今采用 LSM 树结构的 KV 键值系统的性能波动较大的问题进行了改良，提高系统性能的平稳与适用性。

1.2 国内外研究现状

1.2.1 NVM 设备技术研究现状

NVM 设备即非易失性存储设备，其按照主要的制造材料和不同的工艺手段，NVM 设备有着很多不同的种类。现今的 NVM 设备主要工艺手段包括有磁变随机存取存储器[11]（Magnetic Random Access Memory，MRAM）、相变随机存取存储器[5]（Phase-Change Memory，PCM）、阻变随机存取存储器[12]（Resistive Random Access Memory，RRAM）以及铁电随机存取存储器[13]（Ferroelectric Random Access Memory，FRAM）等。NVM 主要的特性有：

① 远高于普通机械磁盘的读写速度；

② 设备内部的数据内容不需要外界提供能量来维持，即非易失性；

③ 相对于普通机械磁盘的寿命较短，不同的工艺手段拥有着不同的寿命周期；

而针对其的技术特性，现今世界上主流的研究目标有以下几种：

① 利用其高速特性，直接用 NVM 来代替 DRAM 作为计算机的主存，提升计算机系统的开机加载初始化的效率，同时减少计算机系统的能耗；

② 考虑到大部分材质的 NVM 的速度仍然与 DRAM 相差了一个数量级，因此将 DRAM 和 NVM 进行异构作为计算机的主存，使用高速的 DRAM 作为计算

空间，使用稳定且较大容量的 NVM 作为缓存空间；

③ 利用 NVM 高速与非易失性，利用 NVM 作为外存，大力提升系统 I/O 的速度。

针对 NVM 这一新型存储技术，由于其优秀的性能使得其在业界炙手可热。无数的业内大公司与研究学者都对其进行了技术研究。

企业产品方面上，Intel 公司与镁光公司于 2012 年合作成立公司，携手研发 3D XPoint[14]技术，Intel 公司并于 2017 年推出了第一款 3D XPoint 产品——Optane 固态硬盘与存储加速器，3D XPoint 技术基于 PCM 材料。PCM 以非晶相和结晶相存储信息，它可以通过外部电压进行数据位的取反。Optane 固态硬盘采用两层堆叠架构，使用 20nm 工艺可实现 128 千兆位的密度，其读取延迟约为 125ns，可擦写次数为 20 万次。

在学术研究中国内外学者对 NVM 在实际系统中的应用有着不同的方向。

Ranganathan 和 Chang 提出未来将由存储级内存来实现统一内外存系统[15]。由于计算机应用环境的变化，大数据时代的来临，存储设备的发展已经逐渐跟不上数据产生的速度，在这一背景环境下，在未来传统的内存与存储体系结构将发生变化，各类组件的区别将逐渐变小，利用 NVM 技术将可以使得内外存有机地统一，这一模式将是未来的研究发展方向。

Yang 等人[16]针对 NVM 高速 I/O 的特点，提出将传统的操作系统通过中断来处理所有异步 I/O 的方式改变为同步 I/O 的模式。使用轮询的方式来与 NVM 设备进行数据通信，而不是使用中断的方式，避免过多中断给系统带来的性能影响。Yang 等人针对这一想法进行了实验证明同步模式下的系统可以正常且快速运行。这为今后的计算机系统的架构设计指出了一条可行的方向。

Ramos 等人[17]提出了一种 NVM 存储中的页面管理策略。方案设计提出了一种 RaPP（Rank-based Page Placement）的基于等级的页面放置机制。机制主要内容为：DRAM 存储热度相对较高的数据，而热度相对较低的数据保存在 NVM 中，同时根据对数据页面的热度的频率变化对页面进行动态排序，将热度靠前的数据从 NVM 中移入 DRAM 中。方案通过对冷热数据进行划分，利用 DRAM 的高速

和 NVM 的非易失特性，充分发挥出了混合存储介质的性能优势。

1.2.2 键值存储系统研究现状

KV 键值存储是非关系型数据库的一种典型的存储方式，键值存储系统中数据按照 key-value 键值对的形式进行组织存储，其由于横向扩展的性能扩展方式，可以较好地应对现今大数据时代的需要，已经成为了现今存储行业的主流方式之一。如今市面上已经有数十种完善的键值存储系统。本节以下内容将对一些较为典型的系统进行相应介绍。

Memcache[7]是一个如今应用较为广泛且典型的内存分布式存储系统，由 Brad Fitzpatrick 于 2003 年开发，随后被广泛用于 Youtube、Twitter、Facebook 等各大互联网公司。Memcache 基于哈希结构实现 KV 键值存储，所有数据均存储于内存空间中，由于哈希表的特性，其无论数据的多少，均提供 O(1)的查询效率，但由于其全内存存储的结构，一旦发生断电等意外故障，系统将会导致数据完全丢失。因此，Memcache 的设计初衷是为了作为缓存设备的存储系统，不考虑持久化的问题。Memcache 由于定位问题，且年代相对较早，因此提供的基本类型较少，只支持较少的 KV 数据类型。后续基于此结构的系统如 Redis 等均对此进行了针对性优化。

Redis[6]是一个开源的基于内存的日志结构 KV 键值数据库，由 Salvatore Sanfilippo 于 2009 年开发完成并对外界开源发布，现今 Redis 的开发由 Pivotal 公司赞助。Redis 是一个完全基于内存的 KV 键值数据库，因此拥有着极高的 I/O 速度，其支持多种不同的数据结构类型，且对外提供了如 java、C/C++、Python、Erlang 等多种 API 结构，拥有着极高的可扩展性。Redis 与一般的内存数据库如 Memcached 不同，其拥有着较为完善的持久化机制，系统会周期性地将更新的数据备份写入到磁盘中，并将操作过程加入日志文件，并且对服务器实现了相应的主从同步机制，保证系统性能的高扩展性和系统安全性。

MongoDB[18]是一种应用范围广，可支持内容多的开源数据库。MongoDB 是一个适用于敏捷开发的数据库，其数据模式可以随着应用程序的发展而灵活地更新，拥有着高性能、可扩展性与可用性。MongoDB 支持全索引，拥有着高效的

查询速度，同时期面向文档存储，存储格式简单且实用。

LevelDB[8][9]是由 Google 公司开发并开源的键值存储系统，与 Redis 和 Memcached 不同，LevelDB 的主要存储介质为硬盘，其底层主要数据引擎采用了 LSM 树架构，通过这一数据结构，系统可以提供极高的随机写的性能，因此 LevelDB 常应用于写请求占比高的应用场景。LevelDB 与其他的数据库不同，其并不是一个完整的系统，而是一个 C/C++语言的库，接口操作简单，易于应用开发，但由于 LevelDB 为单机环境下的存储系统，因此对分布式事务支持较差。

RocksDB[10]由 Facebook 研发，旨在开发一个与高速存储设备尤其是固态硬盘的数据存储性能相当的数据库系统，以此来提供高性能的服务。其主要目标在于保证存储速率的高速和高负载服务器的性能，可以提供 PB 级数据存储管理访问的服务。RocksDB 系统底层存储引擎同样采用了 LSM 树结构，并相对于 LevelDB 多加入了多线程优化与列族等概念，扩大了接口支持，提供多 key 查询及范围查找等接口功能。同时针对于大数据环境下的应用场景，RocksDB 对 HDFS 也提供了相应的接口支持。如今 RocksDB 已成为最流行的 KV 键值存储数据库系统之一。

1.2.3 基于 NVM 设备的 KV 键值存储技术研究现状

考虑大数据时代背景下的数据存储需求，将高速的 NVM 设备与高可扩展的键值存储系统相结合，针对 NVM 的特性，将键值存储系统架构作相应的修改优化，可以达到强强联手的结果，来更好地解决现今的海量数据存储问题。NVM 在键值存储系统上的应用如今已是一个学术界较为火热的研究方向，本节以下内容将对一些较为典型的论文成果进行相应介绍。

Xia 等人提出的 HiKV 系统[19]针对极具潜力的 DRAM 与 NVM 混合结构的存储系统而设计，现有的混合内存键值存储系统的索引等设计未能较好地利用两种介质的特性，DRAM 的高速读写性能与 NVM 的相对较低速的读写性能等。Xia 等人提出了一种持久化的键值存储方案 HiKV，其核心的思想是在两种不同的存储介质之间设计了一种有效的混合存储索引。HiKV 结合了哈希索引与 B+树索引二者的优势，在 NVM 中建立哈希索引结构，通过高速的哈希索引在保持

NVM 所固有的快速索引搜索的特性，同时在 DRAM 中构建 B+树索引，借助 B+树的范围查找的特性使得系统得以实现范围搜索的功能。

Kannan 等人针对 NVM 与 DRAM 的混合异构存储介质提出了一种方案 NoveLSM[20]。NoveLSM 系统核心的目的在于利用 NVM 的高速与非易失的特性来解决 LSM 树写放大问题以及其导致的系统操作延时问题。方案针对 NVM 中的 memtable 中的数据结构针对 NVM 的固有特性实现了特有的可持久化的跳表，避免 memtable 到 NVM 的序列化和反序列化。同时为了减少 LSM 树的 Compaction 操作给系统带来的性能开销而采用 NVM 作为系统的持久化缓存，以一个大容量的空间来替代原有的小容量的 DRAM，增大系统的吞吐率。系统还利用了 NVM 的非易失且相对于 SSD 高速 I/O 的特性，采用 NVM 来进行系统日志的存储，减少写日志对系统性能的影响。

Lu 等人提出的 WiscKey[21]同样也是针对于 LSM 树结构的严重写放大问题进行优化。WiscKey 在 LSM 树上采用 key 和 value 分离的方式进行存储，在 LSM 树结构中只对存储 key 以及 value 的位置进行操作管理，这一措施大大地减少了 LSM 树中每一层中的数据量，在每次 Compaction 过程中不需要对 value 进行重写，极大程度的减少了写放大。WiscKey 技术并不是针对于 NVM 设备提出，但其创新性的键值分离存储的方式在 NVM 与 SSD 混合存储环境下可以起到显著作用，可以将 key 和较热的 value 存入相对较小的 NVM，将较冷的 value 保存在 SSD 中，这可以较大程度地利用 NVM 的高速性，来提升系统的性能。

1.3 研究内容与论文结构组织

本文针对于当今大数据时代的背景，为了应对与日俱增的数据存储需求，将 NVM 存储设备与键值存储系统这两项如今较为热门的软硬件技术相结合，利用 NVM 设备的高 I/O 与非易失的特性，提升存储系统的读写速度，同时利用键值存储系统在大数据场景应用下的普适性，将二者有机地结合，设计实现了一种新型 KV 键值数据库存储系统，并在保证系统性能与可靠性提升的同时，利用 NVM 设备的特性，设计相应的数据结构，针对传统 LSM 树结构的 Compaction 操作带

来的系统性能极速波动的问题进行了改良与优化，保证了系统性能的平稳性。

本文的主要内容包括以下六个章节：

第一章，主要介绍了本课题的研究背景与研究意义，针对现今主流的 NVM 设备与键值存储系统方案的研究进行了介绍，分析了如今这两种技术的研究前景与方向，并阐述了本文的主要研究目的，以及本文的内容结构。

第二章，主要将于本文相关的关键性技术进行了分析说明，针对性描述了 NVM 设备的特性与使用以及对 LSM 树结构的键值存储系统的结构与特性进行了分析。

第三章，首先对常见的采用 LSM 树结构的键值存储系统的问题进行了分析，再针对该问题提出了本文的设计方案内容，对方案中主体的 NVM 中数据组织结构及 Compaction 过程进行了详细阐述。

第四章，详细描述了系统方案基于 RocksDB 框架的实现过程，通过读写以及合并等流程详细描述了系统的实现过程。

第五章，将所设计的系统与传统的 RocksDB、以及 NoveLSM 进行了详细的对比测试，分析测试结果，得出结论。

第六章，对本文的工作进行概括性总结，并分析了研究中的不足之处，并对未来的研究方向进行了展望。

2 相关技术分析

2.1 非易失存储设备分析

非易失存储设备（Non-Volatile Memory，NVM）是一种新型存储介质，拥有着非易失性、静态功耗低、存储容量相对较大等优点，这些特征使得 NVM 能够在未来的计算机系统应用中拥有较大的优势。前一章节已经介绍，NVM 如今主要有相变存储器(Phase Change Memory，PCM)、磁变随机存取存储器（Magnetic Random Access Memory，MRAM）等几种，不同材质的 NVM 拥有不同的存储特性，几种 NVM 和常见的存储介质的相关属性如表 2-1 所示。

表 2-1 几种不同的 NVM 设备基础性能对比[5]

芯片类型	DRAM	PCM	RRAM	MRAM	FeRAM	NAND
产品容量	约 16GB	约 8GB	约 1TB	约 64MB	约 64MB	约 1TB
理论工艺制式	约 20nm	约 5nm	约 11nm	约 32nm	约 65nm	约 16nm
读操作时间	小于 10ns	10~100ns	10~50ns	2~20ns	20~80ns	10~50μs
写操作时间	小于 10ns	10~120ns	10~50ns	5~35ns	10~5ns	0.1~1ms
非易失性质	易失	非易失	非易失	非易失	非易失	非易失
主要技术瓶颈	易失性	材料特性要求较高	原理不明确	容量小	容量小	寿命短，速度慢

从表中可以看出，几种常见材质的 NVM 的性能都远高于传统的固态硬盘（NAND SSD），不同介质的 NVM 有着不同的性能特征。如阻变随机存储器 RRAM 如今的存储机理仍不够明确，无法做到较好的量产与性能稳定，而磁变随机存储器 MRAM 与铁电随机存储器 FeRAM 由于工艺的问题导致了无法制造出较大的容量，相变存储器 PCM 虽然速率上并不能达到 MRAM 那么高速，但是由于其较为稳定的特性，以及相对较为合适的空间大小，已经逐渐成为各大公司的主流研究材料。如今炙手可热的由 Intel 与镁光联合开发的 3D Xpoint 技术虽然官方对外宣称并非采用 PCM 工艺且对外保密工艺类型，但是仍然有很多专家认为其实质仍然为 PCM 技术。

相变存储器实质是一种硫族化合物其利用材料可逆转的物理状态变化来模

拟数字信号，以此来进行数据信息存储。这种硫族化合物在通电升温的条件下，将会由非晶体状态转变为晶体状态，不同的状态拥有不同的电阻特性，对应着数字信号的 0 和 1，实现数据存储。由于相变存储器的状态信号是由化合物介质的状态来决定，且两种状态均属于稳定状态，不需要外界能量去维持，因此相变存储器就拥有了非易失性与低功耗性，同时由于材料的非晶体态与晶体态可直接转换，即 0 可以变为 1，0 可以直接变为 1，因此相变存储器无需类似于固态硬盘的擦除操作，可以直接覆盖写入数据，相对于固态硬盘少了垃圾回收的过程。同时由于是晶体状态来决定数据内容，相变存储器类似于 DRAM，以字节进行寻址，读写速率接近 DRAM。但是相变存储器仍然还有着较多的缺点，如材料价格昂贵、产品存储密度低，容量只能达到 GB 级别等问题。如今各类相变存储器芯片仍然处于测试等阶段，还不能满足真实环境下的应用需求。

2.2 PMDK 开发工具分析

由于 NVM 设备的数据访问机制以及数据管理等，计算机系统无法直接将其视为 DRAM 进行数据访问，且考虑到 NVM 的高速特性，通过访问外设的方式来访问 NVM 会一定程度上限制 NVM 设备的速度。因此，Intel 公司为了应对现今世界上对 NVM 存储技术的需求前景开发了相应持久化存储开发工具包（Persistent Memory Development Kit，PMDK）[22]，借助这一开发工具库包，系统可以绕过系统内核 I/O 调用接口，使用直接访问（Direct Access，DAX）的方式访问持久化存储设备，减少应用请求的获取文件数据的响应时间。PMDK 开发库将 NVM 中的文件通过内存映射的方式映射到内存中，借助此种方式来允许应用 load/store 数据内容到文件中。目前，PMDK 已支持 C/C++、JAVA 等各类热门编程语言，为在持久化内存上进行存储的研究提供了开发基础与保证。

2.3 LSM 树结构分析

日志结构合并树[23]（Log Structured Merged Tree,LSM-Tree）是一种通过牺牲系统的读性能来提升系统随机写性能的数据结构。LSM 树由日志结构的文件系

统启发转变而来，借鉴了日志的追加写的特性及 B 树的数据组织形式，采用了延迟更新与批量顺序写入磁盘的方案，减少磁盘磁头移动，提升写入效率。日志结构合并树，顾名思义，主题结构分为两个部分：日志结构与合并。

① 日志结构

日志结构即追加写形式。LSM 树的核心理念在于对磁盘的 I/O 采用批处理的方式，当数据写入到内存时，先在内存驻留，当内存中数据量达到一定阈值时，对这些数据进行排序合并后批量追加写入到磁盘尾部。

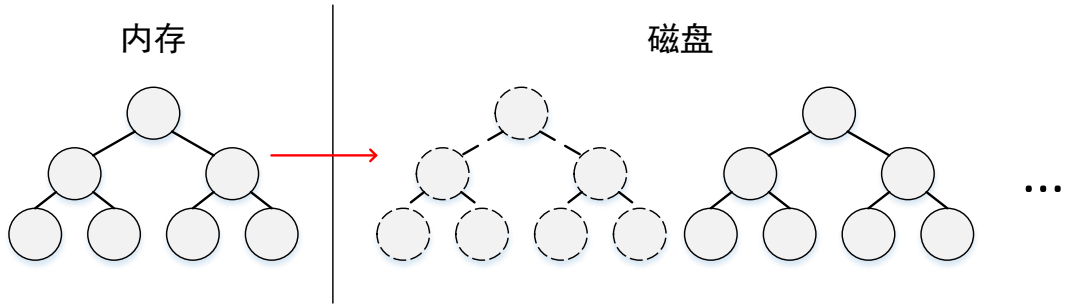

图 2-1 LSM 树结构

如图 2-1 所示，数据在内存中进行排序组合，生成一个有序树后，当有序树的容量达到一定的阈值时，将这个有序树追加写入到磁盘的尾部。这一操作将上层应用的所有随机写的操作均转变为写内存，内存中的有序树又以追加写的形式写入磁盘，避免了对磁盘的随机写，由于磁盘的读写不对称的特性，这一操作极大程度地提高了系统的写性能。

② 合并

由于上图中的操作会导致磁盘中会出现较多的有序树，这对系统的读性能造成很大的影响，每当上层应用发起对某个数据的访问时，需要依次遍历内存中的有序树、磁盘上的所有有序树，并对每一个有序树进行相应的查找才能得到目标结果，磁盘中过多的有序树数量会对这个查找过程带来极大的开销。因此，LSM 树中提出了合并的概念。

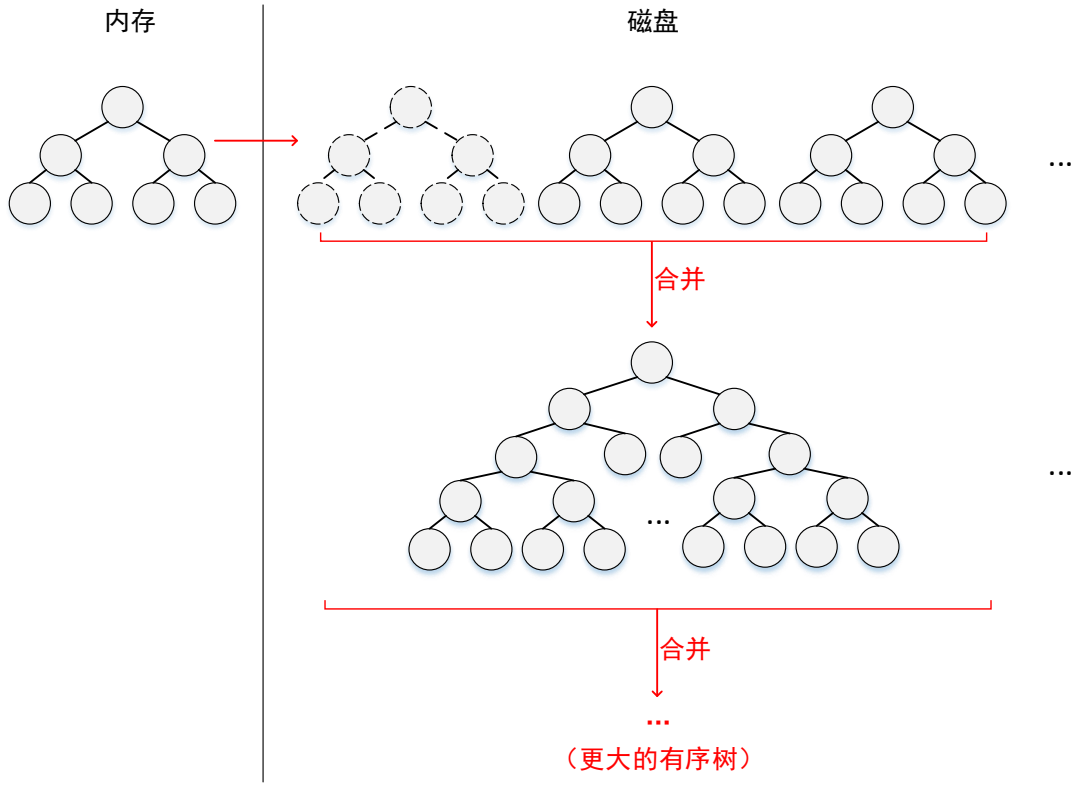

图 2-2 LSM 合并过程

如图 2-2 所示，当磁盘中的有序树数量达到一定上限时，几个小的有序树将会合并合并生成一个大的有序树，当数据量再增大时，大的有序树也会进行相互合并，生成更大的有序树。通过如此的合并操作，系统中的有序树数量将会保持稳定较少的数量，在每次对所有的有序树进行访问时，更多时间将应用在树内部进行 O(logN)效率的查找，减少了读操作的耗时。

2.4 典型 LSM 树系统 RocksDB 分析

如今 LSM 树结构已成为了 KV 键值存储的一个主流结构，典型的基于 LSM 树结构的存储系统 RocksDB[10]由 Facebook 开发，以应对大数据环境下的高负载问题。

RocksDB 系统分为内存与磁盘两个部分，内存中拥有 memtable 与 immutable Memtable 等部件，磁盘中为完整的 LSM 树结构。同时为了应对应用层的不同的数据访问形式，RocksDB 还提出了列族 ColumnFamily 的概念，每个 ColumnFamily

都拥有以上描述的所有组件，不同的 ColumnFamily 之间组件与数据相互隔离，RocksDB 的结构图如图 2-3 所示。

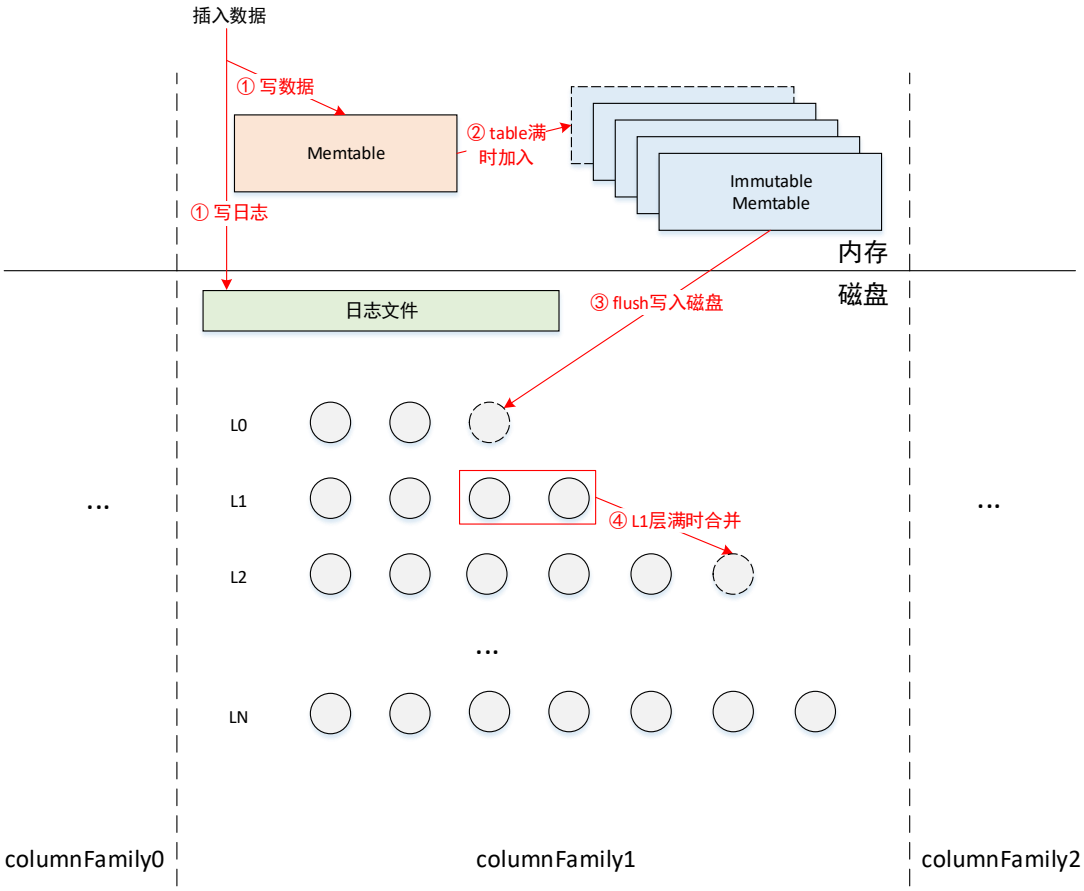

图 2-3 RocksDB 结构示意图

① 当应用层向系统插入数据时，首先系统写日志文件，保障系统的可靠性，再将写入的数据加入到 memtable 中。RocksDB 中的 memtable 与 immutable memtable 实质均为一个有序的跳表。数据加入到 memtable 的跳表中进行排序。

② 当 memtable 空间达到一定的阈值时，memtable 将发生转变，成为 immutable memtable，并加入到 immutable memtable 链表中，等待 Flush 线程将 table 写入到磁盘。

③ Flush 线程即将 immutable memtable 写入磁盘的线程，其主要工作为将 immutable memtable 转变为文件形式的 SSTable，并将 SSTable 加入到 LSM 树的最高层 L0 层。当 L0 层的文件数量达到系统预设的上限时，系统将启动

Compaction 合并线程。

④ Compaction 首先从 L0 层选择 SSTable，并选择以及 L0 层与 L1 层中与该 SSTable 的键值数据 key 范围有重叠的其他 SSTable，一起读入内存，进行新的排列组合，并去除无效数据后，再分为固定大小的 SSTable，写入 L1 层。当 LSM 树的其他层 SSTable 数量达到上限时也进行同样的操作并写入更下一层。

当用户层读取系统中的数据时，将自顶向下依次访问 memtable、immutable memtable、LSM 树 L0 层到最底层，直到命中该数据为止。由于 LSM 树结构的特性，为了保证系统的高速写的性能，牺牲了系统的读，以达到系统的性能的平衡。为了加速读操作，避免访问过多不包括目标数据的 table，系统使用了布隆过滤器(Bloom Filter)算法，并在 table 内部进行数据查找时采用了二分查找的方式，加速数据查找的速度。

布隆过滤器算法是一种使用位图及多个哈希函数来进行数据检索的算法。算法利用有限的位图，将数据通过多个哈希函数分别映射到位图的不同位置，如图 2-4 所示。同一个 key 通过不同的哈希函数映射到了 1、2、5 三个位置，当需要进行数据检索时，同时检索这三个位置的值是否为 1，就可以快速排查该数据是否一定不在集合中。若三个位置其中一个不为 1，则数据必定不存在于集合中，而若三个位置均为 1，考虑每个位置对应多个 key 值，因此可能这些位置是被其他的 key 置位，所以该 key 不一定存在于集合中。因此，布隆过滤器只可以筛除必定不存在的集合，不能准确判断目标一定在集合中，拥有一定的误差。在 RocksDB 中针对每一个 table 均设置了一个布隆过滤器，以快速筛查判断目标数据是否不存在于该 table 中。

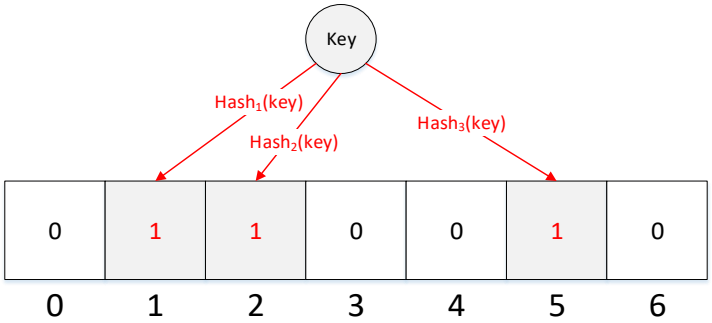

图 2-4 布隆过滤器原理示意图

2.5 本章小结

本章节首先对 NVM 技术进行了介绍与分析，并针对主流的相变存储器的原理进行了简要分析，然后对 NVM 等持久化存储的开发工具 PMDK 进行了简要介绍；再分析了 LSM 树结构的基本原理，并对 LSM 树的主要应用场景 RocksDB 进行了介绍，分析了 RocksDB 的框架结构，解读了其读写过程的原理，并简要说明了 RocksDB 中加速读操作的布隆过滤器技术。对这些要点内容的分析与理解，为本课题后续的方案设计与实现提供了理论基础。

3 MatrixKV 系统设计

在本节中，首先针对现有基于 LSM 树结构的 KV 系统所存在的问题进行了分析，然后基于分析得到的问题设计提出了一种在 L0 层采用矩阵结构的 KV 系统 MatrixKV，本节后续内容对该系统进行了详细介绍。

3.1 问题分析

现今主流的键值存储系统如 RocksDB、LevelDB 等均基于 LSM 树结构而设计，然而，由于 LSM 树结构需要频繁进行 Compaction 的特性，采用 LSM 树结构的 KV 系统在传统 DRAM-SSD 存储架构中会有着较为严重的性能波动问题。为了较好地分析这些问题，本文基于 RocksDB 对系统性能波动的问题进行了相关测试，具体测试配置见测试章节。测试内容对系统进行 80GB 大小的数据写入，每 10 秒对系统的负载速度进行统计，同时统计在 LSM 树进行 Compaction 操作时 L0 与 L1 层涉及该次 Compaction 的数据大小。测试结果如图 3-1 所示。

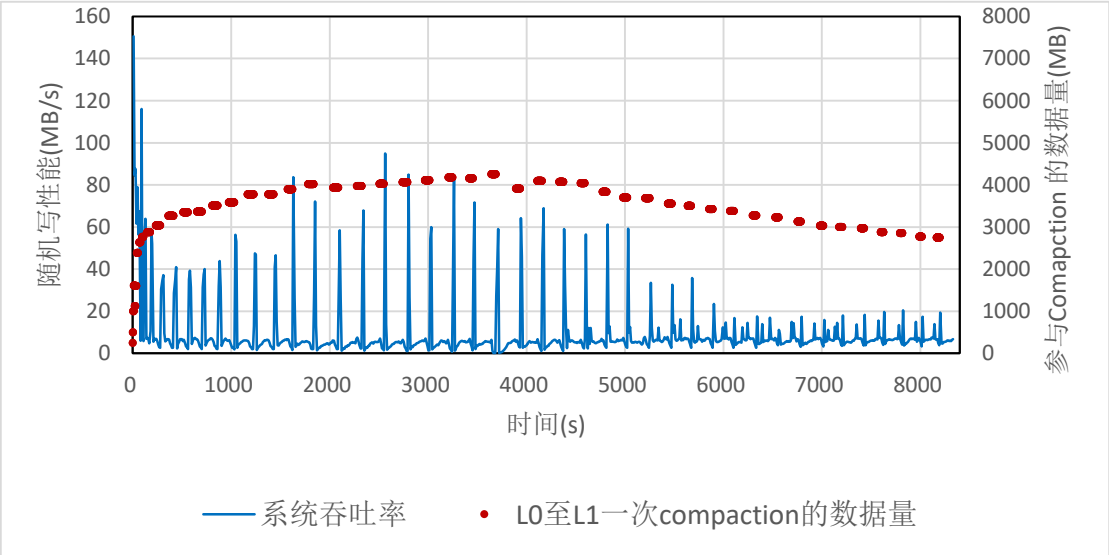

图 3-1 RocksDB 性能与 Compaction 数据量测试

图 3-1 中蓝线代表着系统的性能曲线，由图可以看出，系统性能会周期性地

变差，且峰值和谷值性能相差较大。过于频繁的波动会给用户带来系统性能不稳定以及系统性能的不可预见性的体验。红线表示系统在 L0 层至 L1 层 Compaction 过程所涉及的数据量，每次 compaction 数据量约为 3GB~4GB 左右，这对系统的性能造成了很大影响，每次 compaction 出现的时间与系统性能停顿延时的时间基本吻合。

上述的系统性能延时在一个基于 LSM 树的键值存储系统中主要由以下三个部分导致：

① Insert stall：当 memtable 满时，memtable 转变为 immutable memtable，而如果此时 immutable memtable 被挂起正在进行相应的 Flush 操作时，memtable 将需要等待整个 Flush 完成后才可以完成转变，此时，整个系统需要进行相应的等待；

② Flush stall：当 immutable memtable 向 L0 层进行 Flush 的过程中，L0 层的空间达到了上限，需要将数据 Compaction 到下一层中，此时 Flush 操作需要进行相应的等待；

③ Compaction stall：当 L0 层与 L1 层均达到空间上限时，L0 层向 L1 层进行 Compaction 的操作将需要等待更底层的 Compaction。

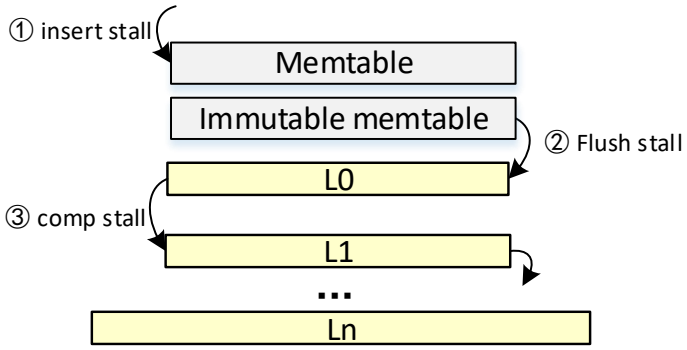

图 3-2 停顿延时示意图

以上三种停顿前后相关，Compaction stall 影响 Flush stall，Flush stall 影响 Insert stall，三者造成的停顿延时逐渐堆积，最后对整体系统性能造成了极大的影响。由以上分析可以看出，三种停顿延时的根本原因在于最底层的 Compaction stall。

对于现今常见的 LSM 树结构的键值存储系统，如 RocksDB、LevelDB 等，在 L0 层采用了 SSTable 范围可相互覆盖的原则。上层的 immutable memtable 直接 Flush 转变为 L0 层 SSTable，由于现实场景下的读写随机性，这导致了 L0 层的每个 SSTable 的 key 范围随机分布，大多数情况下有范围相互重叠的现象。在每次对一个 SSTable 进行 Compaction 的过程中，系统首先会在 L0 中选择出所有与其有范围重叠的 SSTable，再在 L1 层以同样的方式找出重叠的 SSTable，这导致了 L0 层向 L1 层的一次 Compaction 操作，会几乎涵括了所有 L0 与 L1 层的数据，如图 3-3 所示，若选择 key 范围为 1~158 的 SSTable 进行 Compaction，几乎需要对整个 L0 与 L1 层进行合并操作，这种大数据量的 Compaction 操作给系统带来极大的性能损耗，并加剧了其他两种停顿延时，带来了系统周期性的性能波动。

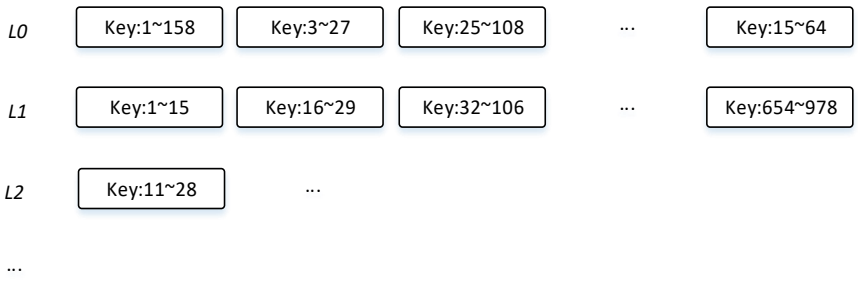

图 3-3 LSM 树数据范围结构

同时，由于 LSM 树的多层结构， LSM 树的层次会随着系统中数据量逐渐增大，而 SSTable 需要通过不断地进行 Compaction 才能够向下层移动，这意味着当 LSM 层次逐渐增多时，Compaction 的次数也会变多，写放大问题将越来越严重。Lanyue Lu 在 WiscKey[21]一文中对此进行研究得出写放大的倍数与 LSM 树的层数呈线性递归关系。由于 SSD 的带宽限制，层次越来越多的 LSM 树结构会极大地影响系统的性能。

为了解决以上 LSM 树结构键值存储系统所存在的停顿延时与写放大的问题，本文设计提出了一种基于混合存储的矩阵结构键值存储系统 MatrixKV。

3.2 MatrixKV 整体结构设计

MatrixKV 引入特殊的结构 Matrix-Table 代替传统的 L0 层，在 Matrix-Table 与 L1 层之间采用较细粒度的 Compaction 操作来降低系统 Compaction stall 带来的性能影响，同时采用 NVM 来作为 Matrix-Table 数据存储设备，以更快的存储设备来接受来自上层的数据 I/O，以此来解决设备速度不对等而带来的性能瓶颈。由于 LSM 树各层次的空间按比例增大，MatrixKV 通过扩大 Matrix-Table（即 L0 层）的空间来相应扩大后续层次空间，降低 LSM 的层次结构，减少多层次树结构给系统带来的写放大影响。

方案基于 DRAM、NVM 与 SSD 三层混合存储架构，设计总体结构图如图 3-4 所示。方案设计保留 LSM 树的结构不改变，总体结构分为三个部分：

① DRAM：系统在 DRAM 中结构与 RocksDB 相同，memtable 与 immutable memtable 来处理上层的写请求，当 memtable 空间满时，转变为 immutable memtable，然后由 immutable memtable 完成对 L0 层（Matrix-Table）的 Flush 操作；

② NVM：在 NVM 中，系统设计了一种矩阵形式的数据结构 Matrix-Table 来代替传统的 L0 层结构，Matrix-Table 对数据 table 进行堆叠存储，并在相邻层之间建立索引关系，加快 Matrix-Table 内数据查找速度，同时对数据按照 key 的范围进行细粒度切分成列，以列为单位进行 Compaction，减少这些数据在合并到 L1 层时的系统带宽占用；

③ SSD：由于 NVM 的空间限制，系统中 LSM 树的 L1 层及之后的层次依旧存储在 SSD 中，其结构与 RocksDB 相同，在保持 RocksDB 层间容量放大系数不变的情况下，由于 Matrix-Table（L0 层）空间增大，下层每层空间的容量均相应扩大，因此在同等负载数据量的情况下，MatrixKV 的数据层次要低于 RocksDB，以此来减少写放大性能开销。

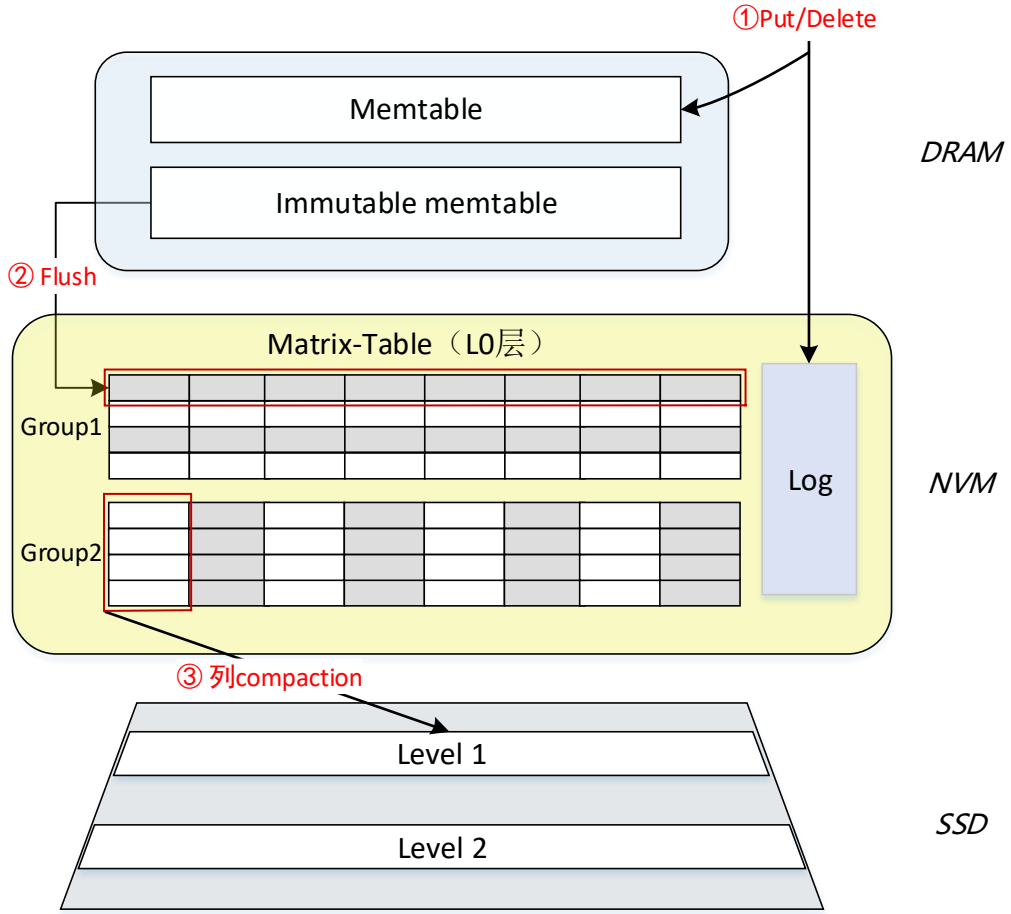

图 3-4 MatrixKV 系统整体结构图

MatrixKV 系统整体结构如图 3-4 所示，当系统处于 Compaction 过程时（步骤③），为了避免 Compaction 操作占用 Matrix-Table，导致了 Flush 过程的等待，系统将 Matrix-Table 划分为两个 group，group1 接受上层的 Flush 操作，group2 进行向下层的 Compaction 操作。同时采用数据可持久化的 NVM 来替代 SSD 作为日志的数据存储介质，加快写日志速度。

MatrixKV 针对前一小节中提出的性能问题而设计，相对于传统的采用 LSM 树结构的键值存储系统有以下几点改进：

① **高速的存储设备**：MatrixKV 采用相对于 SSD 更为高速的 NVM 作为 LSM 树结构中 L0 层的存储设备，提升了 immutable memtable 向 L0 层进行 Flush 的速度，使得 Flush 操作能够更为快速的完成，避免插入过程发生停顿延时；

② **快速索引结构：**MatrixKV 设计了一种可以快速进行数据索引查找的数据结构 Matrix-Table 代替原有的 L0 层，利用矩阵结构索引与 NVM 优秀的随机读写性能加速对 Matrix-Table 内部数据的访问；

③ **扩大 LSM 树各层的空间：**MatrixKV 采用较大空间的 Matrix-Table，LSM 树中其他层次的空间也按比例相应扩大，在总空间固定的情况下，每层空间增大，使得总层数减少，Compaction 次数相应减少，同时，MatrixKV 利用 Matrix-Table 的高速索引查找特性，避免了传统结构扩大 L0 层空间会而带来的读开销问题；

④ **按列分割 Compaction 策略：**针对扩大的 Matrix-Table，MatrixKV 采用按列来对其进行切分形成列块（Column Chunk），在 Compaction 过程中，以列块为单位进行 Compaction，将原先的一个大数据量大 key 值范围 SSTable 的 Compaction，转变为多个中等数据量小 key 值范围列块的 Compaction，将原来一次严重的性能波动转变为多次轻微的性能波动，以此来提升系统的平均性能与用户的使用体验。

3.3 Matrix-Table 结构设计

Immutable memtable 在 Flush 到 NVM 后，以 table 为单位堆叠形成 Matrix-Table，完成数据组织与存储。对于 KV 键值存储来说，在大多数应用场景中，value 大小一般为 1K 以上甚至更大，而 key 只有几十字节甚至更少，因此一个 KV 单元内 value 占据绝大部分数据空间。由于 value 大小的不定性，为了方便统一管理，且减少对 KV 单元的访问和拷贝数据量，MatrixKV 对整个 Matrix-Table 中的 KV 数据均建立对应的元数据(metadata)单元，Matrix-Table 对元数据进行操作，完成对数据的管理，减少数据操作过程中带来的 I/O 数据量。

3.3.1 多层索引数组 Matrix-Table

由于需要对 Matrix-Table 中的多层元数据 table 进行数据存管理，且 table 内部又包含多个元数据单元，因此需要采用二维元数据数组来对这些 table 进行存储。如若简单采用二维数组策略会出现以下两种问题：

① 在对某一个 key 进行查找时，需要从上至下对每个 table 都进行所有范围

的查找，直到查询到对应的 key，涉及数据量较大；

② 在进行范围查找时，需要对每个 table 都进行所有范围的查找，涉及数据量较大。

为了解决范围查找时需对所有 table 均进行全范围查找而带来的查找效率问题，本文提出一种在相邻 table 层次之间构建链接关系，将上下层 table 中的 key 按照数值大小或字典序的形式进行排序索引，借助索引来加速范围查找。

基本结构如图 3-5 所示。每个元数据单元的索引指向其下层相邻 table 中字典序大于等于该元数据的所有单元中的最小单元。即对于每一个元数据单元，该元数据索引对应的位置之后的所有元数据的 key 均大于等于该元数据，位置之前的所有元数据的 key 均小于该元数据。

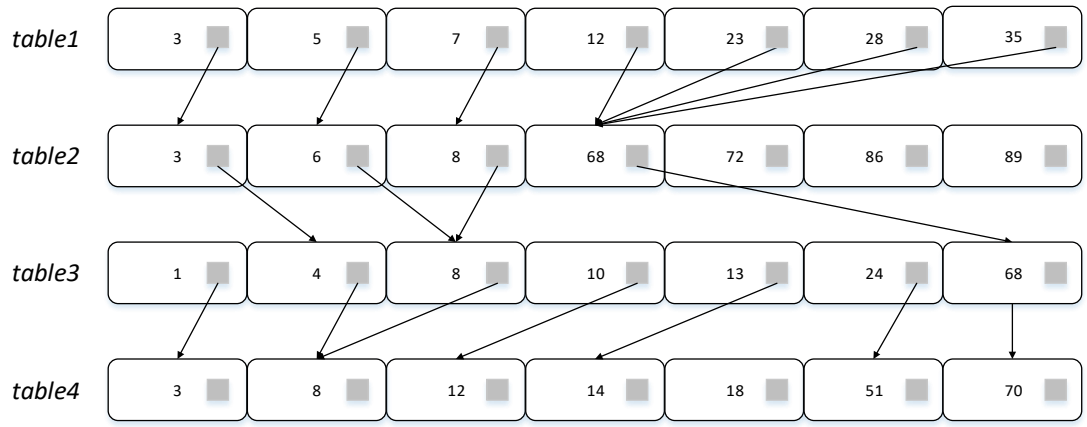

图 3-5 索引数组结构

3.3.2 多层索引数组加速原理

在一个基于 LSM 树结构的键值存储系统中，一般情况下，写操作的方式采用的是追加写形式，即在系统中同时可能会存储着多个相同 key 值的 KV 数据，这些相同 key 的数据按照写入的时间先后顺序（时间戳）来判断数据是否为最新值，在系统后续涉及数据合并的过程中，将旧的无效数据删除。

由于系统追加写这一特性，Matrix-Table 中数据查找的过程需要由上至下进行查找操作，即优先查找新写入的 table，再查找旧的 table，以保证查找过程中第一个命中的结果就是最新值。对于不采用索引的多层数组，由上至下的查询过

程中，需要对每一层的数组均进行从头至尾的二分查找，查找过程如图 3-6 所示。

图 3-6 二分查找过程

针对图中的数据例子，对 4 进行访问查找时，共需要进行 9 轮比较，最终得出结果。一般情况下，Matrix-Table 中每个 table 中 key 的数量较多（RocksDB 中采用的 table 默认值为 64M，KV 单元的平均大小假设为 64k，则每个 table 中 key 的数量约 1000 个），若 L0 中有较多的 table，过多的查找带来的开销也会给系统带来较大的性能问题。而本节问题分析小节中又提到，较小的 L0 层大小又会导致 LSM 树会产生更多的层次，带来较大的写放大开销。MatrixKV 设计的索引结构针对这一频繁的查找而设计，旨在于缩小二分查找的范围，减少查找过程中比较的次数，提升系统的性能。

上节已经提出，每个元数据中的索引指向下层中最小的大于等于自身的元数据，借助这一索引，我们可以通过上一层最后一次二分查找的边界来缩小下当前层二分查找的起始范围。假设上一层最后的二分查找的范围是[a,b]，a 索引指向 c，b 索引指向 d，显然目标值 target 所处于的范围是(a,b)。按照索引的定义，本

层中在 c 之前的所有元数据值均小于 a，d 之后的所有元数据值均大于等于 b，设 c 前一位元数据值为 c_{-1}，则几者之间的大小关系如下所示：

$$\begin{cases} a < target < b & (1) \\ c_{-1} < a <= c & (2) \\ b <= d & (3) \end{cases}$$

显然，target 在(c_{-1},d)范围内，因此，若本层中存在目标值，那么该目标值必然存在于 c_{-1} 与 d 两个元数据单元之间，因为本轮的二分查找的起始范围可以选定为(c_{-1},d)。特别的，若 c_{-1} 不存在，即 c 为本层第一个单元，那么二分查找的起点为本层的起点；同样若 d 不存在，即 b 大于本层中任何一个单元，那么二分查找的终点为本层的最后一个单元。最坏情况下为全范围查找，这与普通的二分查找开销相同。因此，采用索引这一策略很大程度地降低了查找带来的开销，图 3-6 中的案例以范围索引查找的过程如图 3-7 所示，比较的次数减少了 1/3。随着 L0 中层次逐渐增多的情况，层间索引带来的性能提升会越来越大。

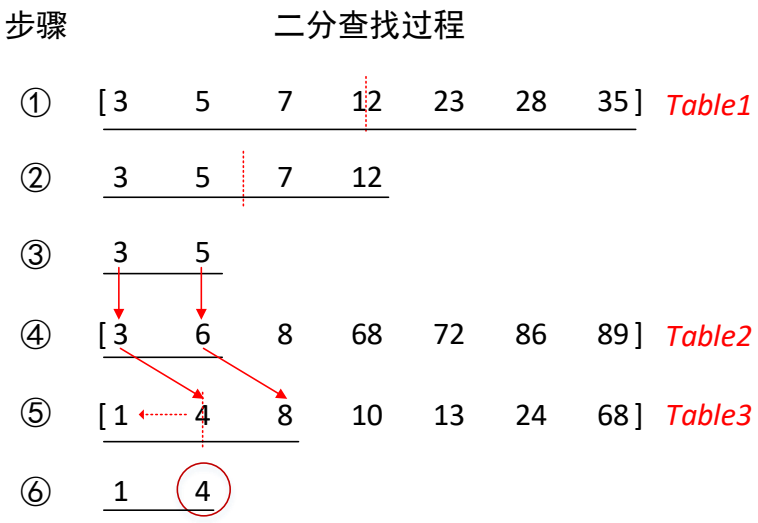

图 3-7 层间索引查找过程

3.3.3 多层索引数组的构建

当 immutable memtable 中的数据以 table 为单位向 NVM 中进行 Flush 时，需要将新的 table 与原有的 Matrix-Table 建立索引。由前文可知，在 Matrix-Table 中

的操作单位为各项 kv 数据的元数据单元。immutable memtable 在 Flush 进入 NVM 时，首先构建元数据单元结构数组，再将数组与原 Matrix-Table 最上层的 table 数组构建相应的索引关系，然后加入到 Matrix-Table 中。由于 immutable memtable 内部 key 的有序性，生成的元数据单元数组也是严格按照 key 值大小排序的数组，同样，由于 immutable memtable 加入到 Matrix-Table 中时并未对其内部的顺序结构进行修改破坏，因此 Matrix-Table 的最上层 table 数组也是一个严格有序的数组，新数组的插入其实是两个有序数组的归并排序过程。插入链接过程的示意图如图 3-8 所示。

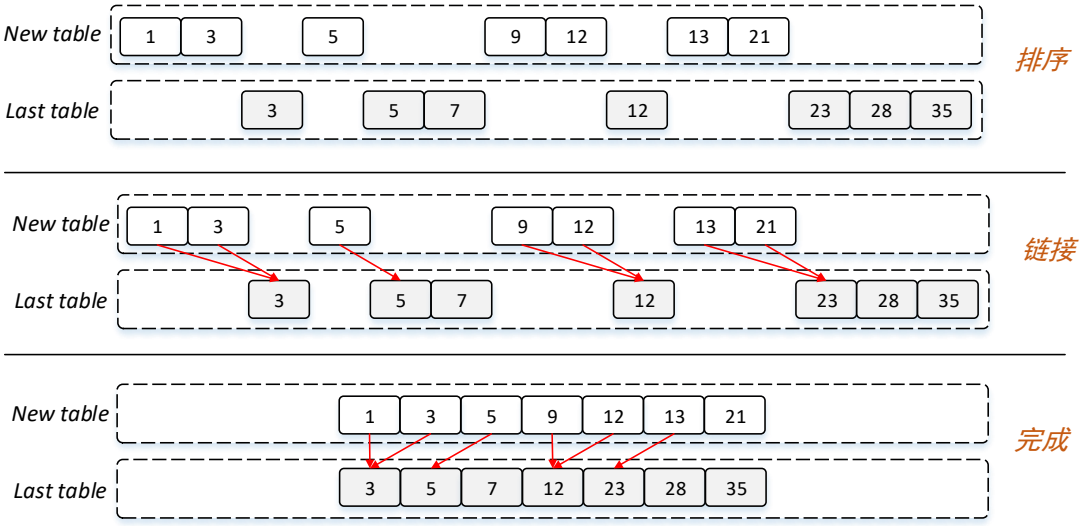

图 3-8 矩阵索引数组的创建

特殊情况下，当出现连续多个元数据单元都指向下层同一个单元，如图 3-9 所示，table1 中 12 及之后的所有单元的下层指针同时指向 key 为 68 的位置。在这种情况下，索引指针并不能较好地缩小二分查找的范围，如图 3-9 中查找 24 的过程。

由于上层多个元数据单元指向了下层的同一个元数据单元，这表明这段范围在下层不存在，如图 3-9 中的 table1 范围[12,35]，在 table2 中不存在任何数据属于该范围。针对这种情况，MatrixKV 系统提出了一种加速策略——跨层指针索引。

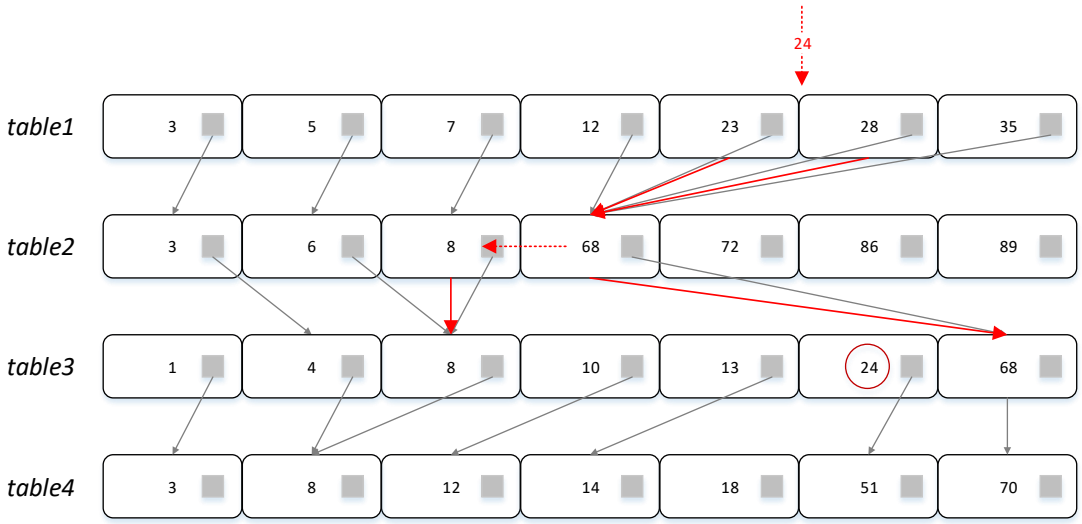

图 3-9 矩阵索引数组查找过程

3.3.4 跨层指针索引

由于在 table2 中不存在[12,35]范围之间的任何数据，当上层对此范围内的数据进行访问查询时，在 table2 必然不会命中目标，因此，table1 中此范围内的数据的索引可以直接跳过 table2，直接指向 table3，加速索引的效率。

针对于上述出现的 table2 中不存在 table1 某个数据段中任何数据的情况，假设该数据段范围为[c1,f1]，如图 3-10 所示，系统读取目标 target 有以下几种可能：

① target 不在[b1,g1]范围内——此情况下 target 与[c1,f1]范围没有关系，即 target 可能存在于 table2 中，可以直接按照前一小节中的索引方式查找 table2 层；

② target 在[b1,c1]范围内——此情况中在 table2 层可能会存在目标数据（如查找 d2），因此需要在 table2 层展开查找，所以 c1 必须要有指向 table2 层的索引指针；

③ target 在[c1,f1]范围内——由前文可知，此情况下 table2 层不存在目标的数据，因此可以直接跳过 table2 层，即此范围内的单元可以指向 table3 层；

④ target 在[f1,g1]范围内——此情况与情况②相同，f1 必须要有指向 table2 层的索引指针。

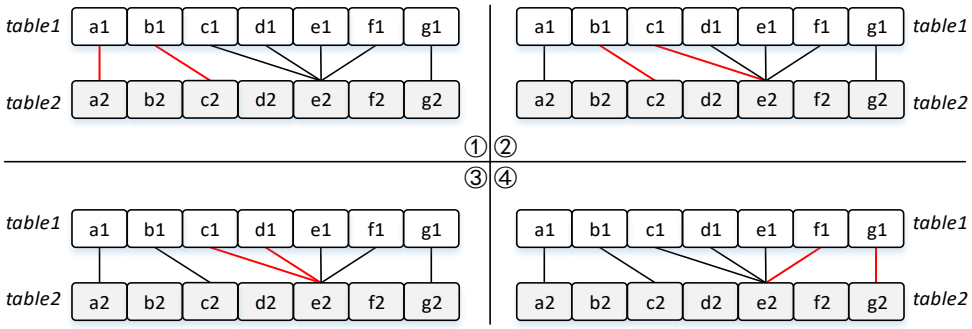

图 3-10 查询范围的四种情况

综合以上四种情况可以了解，当出现多个数据指向下层同一个元数据单元的情况时，由于②④的限制，第一个和最后一个单元需要保留指向 table2 中索引指针，而中间的元数据单元可以直接跳过 table2 层直接指向 table3 层最小的大于等于该数据的单元，如图 3-11 所示。

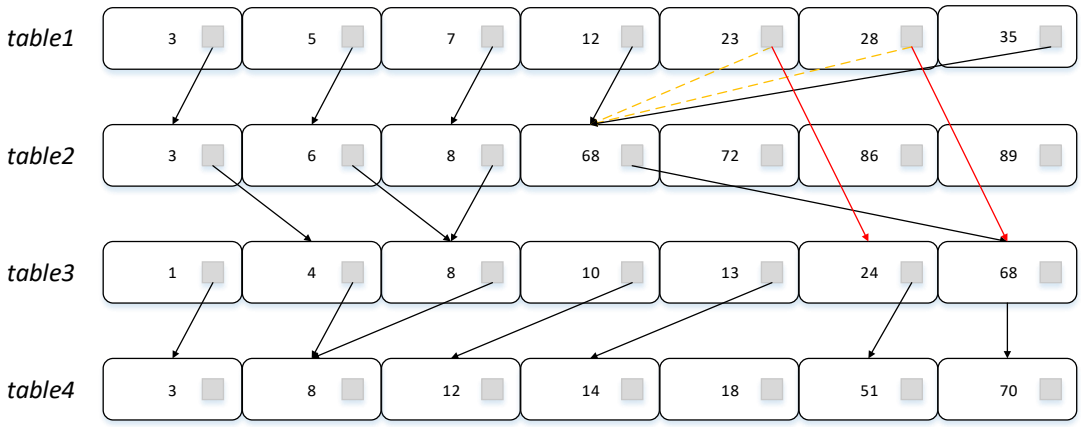

图 3-11 跨层索引示意图

在此种优化方案中，第①②④三种情况，因为没有涉及到任何跨层索引，所以与原来的查询方式相同；针对第③种情况，可能会出现两个索引一个指向 table2，另一个指向了 table3 的情况，由于 table2 中必然不存在目标数据，因此直接将 table2 层的指针进行判断是否是左边界（如若是左边界则前移一位，如若是右边界则不进行操作），然后通过索引进入 table3 层。如图 3-12 中对 13 的访问。

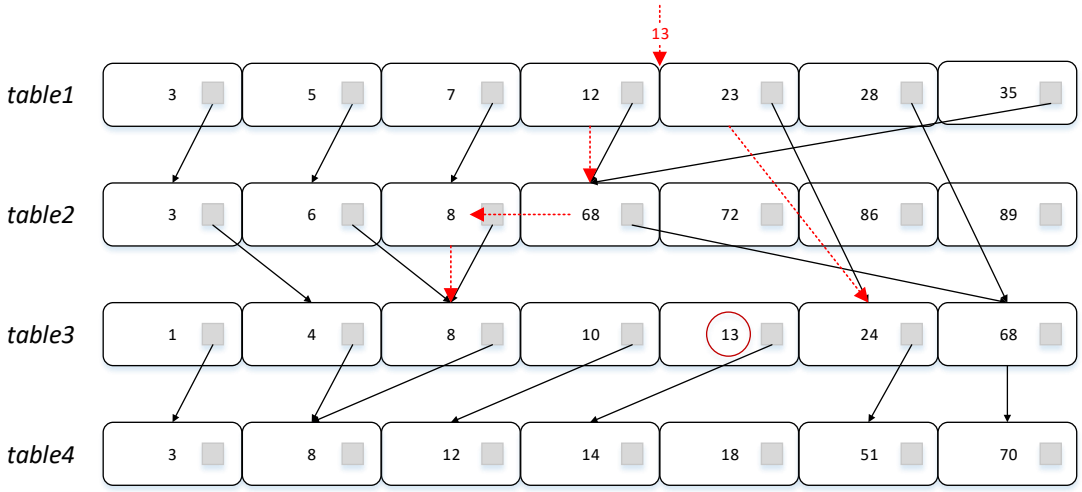

图 3-12 跨层索引下的数据查找

3.3.5 读性能优化

伴随着系统运行时间的增长，Matrix-Table 中数据逐渐变多，而考虑到 Matrix-Table 与传统基于 LSM 树结构的键值存储系统较小的 L0 不同， Matrix-Table 的空间相对较大（1G~128G），而 memtable 的大小一般为 64M，这意味着 Matrix-Table 的层次最高可能会达到几百甚至上千层，虽然系统采用了索引加速的方式，但对如此多层的 table 进行遍历访问仍然会有较大的访问开销。

针对 Matrix-Table 中数据层次过多而导致的系统读性能问题，本文设计了以下两种读优化的方案。

3.3.5.1 布隆过滤器

由于 Matrix-Table 中数据 table 层次过多，系统对每一个 table 均进行二分查找会是一种极大的开销，因此 MatrixKV 需要设计一种可行的方案，来避免一部分对不包含目标数据 table 的查找。在现有采用 LSM 树结构的键值存储系统中，如 LevelDB、RocksDB 等均采用布隆过滤器来进行判断目标数据是否在某个 SSTable 中。在 MatrixKV 中，也同样利用了布隆过滤器能够快速筛选数据的特性，对 Matrix-Table 中的 table 进行判断与过滤。

布隆过滤器利用一个多位二进制数组和对应的一系列映射散列函数来检索

元素是否在集合中，而由于本方案中 Matrix-Table 的数据 table 相对较多，如若对每一层 table 均设置布隆过滤器来进行判断，这不仅会在资源上有着较多浪费，且由于 table 数量的过多，对每个 table 都进行判断，相对来说仍然需要较多的开销。MatrixKV 对 Matrix-Table 中的 table 进行分段（segment），连续一定数量的 table 组成一个段，以段为单位进行过滤判断，每一段 table 的数量由系统配置是设置，默认设置为 50。当某一个段被布隆过滤器过滤掉时，系统将对下一个命中的段进行判断查找，由于新的段的第一个 table 没有来自于上层的索引，在此 table 将从两端开始二分查找。查找示意图如图 3-13 所示。

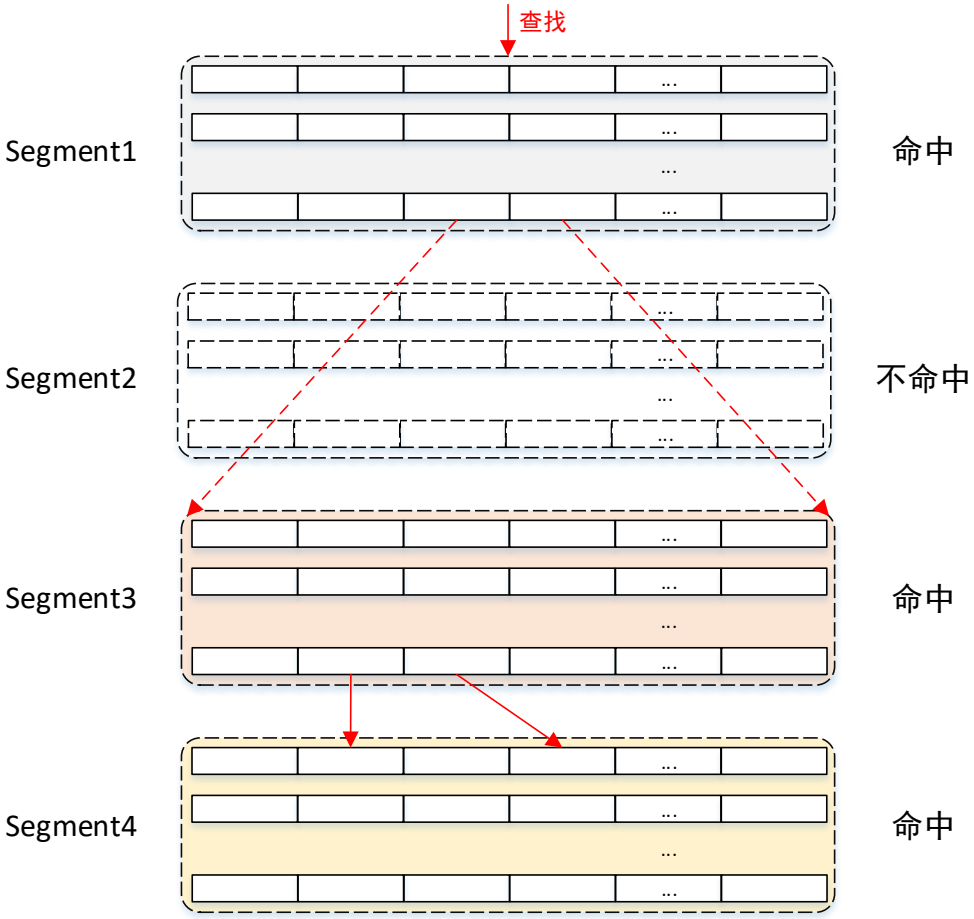

图 3-13 Matrix-Table 的分段结构

分段结构的应用，很大程度地扩大了布隆过滤器过滤的粒度，一次性筛选过滤掉整个段，提升了过滤操作的作用效果。但是随着系统中数据层次越来越多，

段的数量也会逐渐增加，布隆过滤器给系统带来的性能提升也逐渐变小，而由于NVM 的 I/O 速度远大于 SSD，在 NVM 空间足够时，直接对数据进行 Compaction 不是一个明智的选择，因此，除向 LSM 树下层进行 Compaction 操作外，方案设计了另外一种可以减少 Matrix-Table 中 table 数量的策略——table 合并策略。

3.3.5.2 Table 合并策略

由于基于 LSM 树结构的键值存储系统采用的写数据方式为追加写，这导致了整个系统中会保存着很多同一个 key 不同时期的值，而这些值除了最新一次的结果外，其余的值均为无效值。即整个系统中的空间利用率相对较低。当 Matrix-Table 数据 table 层数逐渐变多时，实际有效数据只占据了一小部分空间，而这些无效数据在 compaction 进入 SSD 后，又会被不断地读取加载进入内存参与之后的 compaction 操作，造成了较大的性能浪费。考虑到 Matrix-Table 存储介质 NVM 的特性，如若在高速的 NVM 内部直接消除这些无效数据可以一定程度地减少性能损耗。因此，本文设计了一种对 Matrix-Table 中进行内部 table 合并的策略，有效地回收一部分无效数据占据的空间，并降低 Matrix-Table 的层次数量。

当系统中 Matrix-Table 的段数量达到一定的阈值时，系统将启动 Table 合并策略。根据时间局部性原理，新加入到 Matrix-Table 的 table 由于将会被较多机率被访问，而对此部分 table 的合并将会对系统性能造成较大程度的阻塞延迟。因此合并过程针对于底层在 Matrix-Table 存在较久的数据段，以段为基本单位，从 Matrix-Table 中的最底层开始向上开始进行合并。

如图 3-14 所示，合并过程从 Matrix-Table 的底部开始，每若干数量（默认值为 5）个段组成一个小组，每个小组内进行相应的数据合并。合并过程中，将小组内所有的数据进行合并处理，去除无效的数据，得到最终的结果后，按照原 table 大小的 150%倍容量进行切分，行成新的较大容量 table，再组合成新的段，加入到 Matrix-Table 中。这一操作删除了 Matrix-Table 中大部分无效数据，并扩大了 table 的容量，减少了 Matrix-Table 的层次。

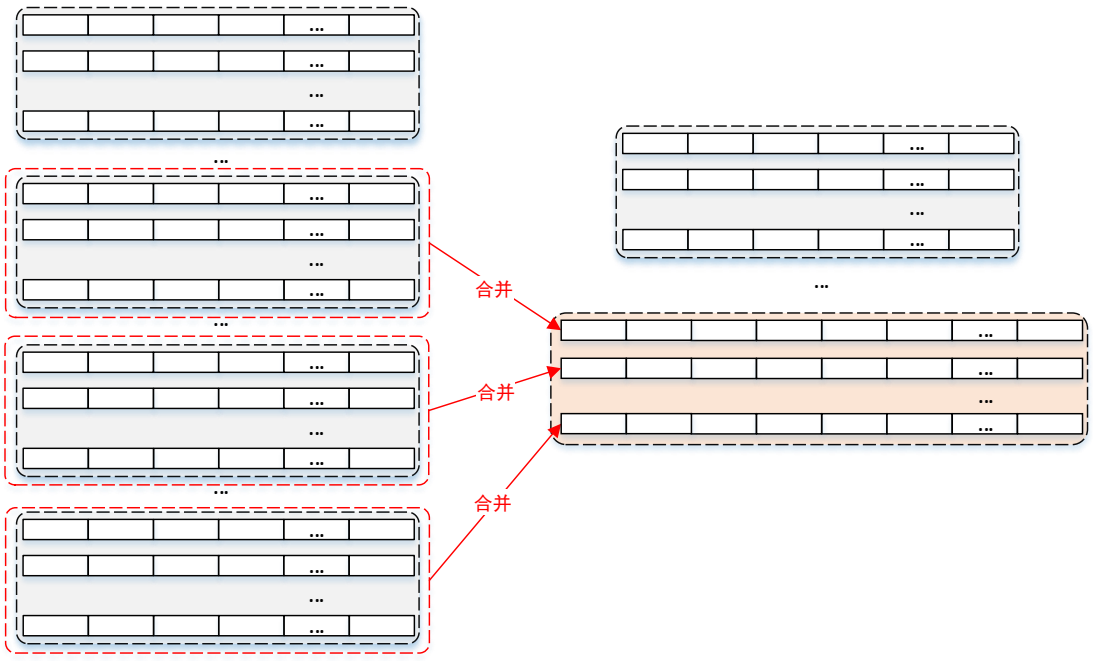

图 3-14 Matrix-Table 的合并

3.3.6 Matrix-Table 大小性能分析设计

3.1 节问题分析中已经提出，由于 LSM 树结构的特性，不同层的空间大小以固定倍数扩大，因此，扩大 L0 层的大小可以相应地扩大 LSM 树每一层的空间。这使得同等数据量的数据存储需求，将会使用较少层次的数据空间，减少了系统性能的写放大。

因此，在 MatrixKV 系统中，理论上 Matrix-Table 的空间大小越大，系统的写放大越少，系统的写性能越强；然而随着 Matrix-Table 空间的增大，Matrix-Table 中多层索引数组的层次也逐渐变多，伴随而来的是段的数据量增多，table 的合并次数也相应增多，合并开销变大，同时，在对 Matrix-Table 中的数据进行读访问时，虽然拥有着高速的索引指针加速索引，但由于层次过多，将会使得索引指针的加速效果变得微乎其微，极大程度地造成了读放大。因此，Matrix-Table 的空间大小与系统的写性能成正比，而与系统的读性能成反比，在不同的应用需求下，可以相应地设置不同的 Matrix-Table 的大小，以期得到最好的性能。在本文的设计过程中，为了与 NoveLSM 进行较好的对比，采用了 NoveLSM 所默认使用的

8GB 的 NVM 空间来进行 Matrix-Table 的存储。

3.4 细粒度 Compaction 流程设计

通过问题分析章节可以了解，Compaction 的速度是系统性能最大的制约因素。Compaction 造成的停顿延时间接性地影响了其他两种停顿延时，而绝大多数的 Compaction 操作均来至于 L0 到 L1 的 Compaction。传统 LSM 树结构下的键值存储系统由于 L0 中各个 SSTable 之间的乱序性，导致每次 L0 到 L1 的 Compaction 会涉及大量 L0 层中的 SSTable，甚至会出现所有 L0 层的文件数据均参与 Compaction，这种过多过大的数据 I/O 必然会给系统性能会带来较为严重的性能阻塞，频繁的 Compaction 导致了系统性能的频繁波动，也影响了系统的整体性能。

前文已经指出，为了减少 Compaction 对系统性能的影响，MatrixKV 采用了相对较大的 Matrix-Table 空间，并使用高性能的存储设备 NVM 来存储数据，以此减少 Compaction 的次数并提升 I/O 的速度。但是对 LSM 树结构的键值存储系统来说，Compaction 永远是不可避免的过程，一旦发生 Compaction 必然会导致系统性能大幅度下降，影响系统性能，造成较差的系统性能体验。

针对上述性能大幅度波动的问题，MatrixKV 提出了一种以列为单位细粒度可控的 Compaction。系统通过将大容量的 Matrix-Table 进行切分成多个小范围段的数据列块(Column Chunk)，再对这些小数据列块块进行 Compaction，将一个长时间的大系统延时转变为多个短时间的小系统延时，这种方案虽然会降低系统峰值速度，但是会大大提升系统的最差性能，优化平均速度。

3.4.1 Matrix-Table 的空间组成

为了避免 Compaction 过程占用整个 Matrix-Table，而导致上层的 Flush 请求造成等待，再实际进行 Compaction 过程时，系统将 Matrix-Table 的整体空间划分为两个部分，如图 3-15 所示。

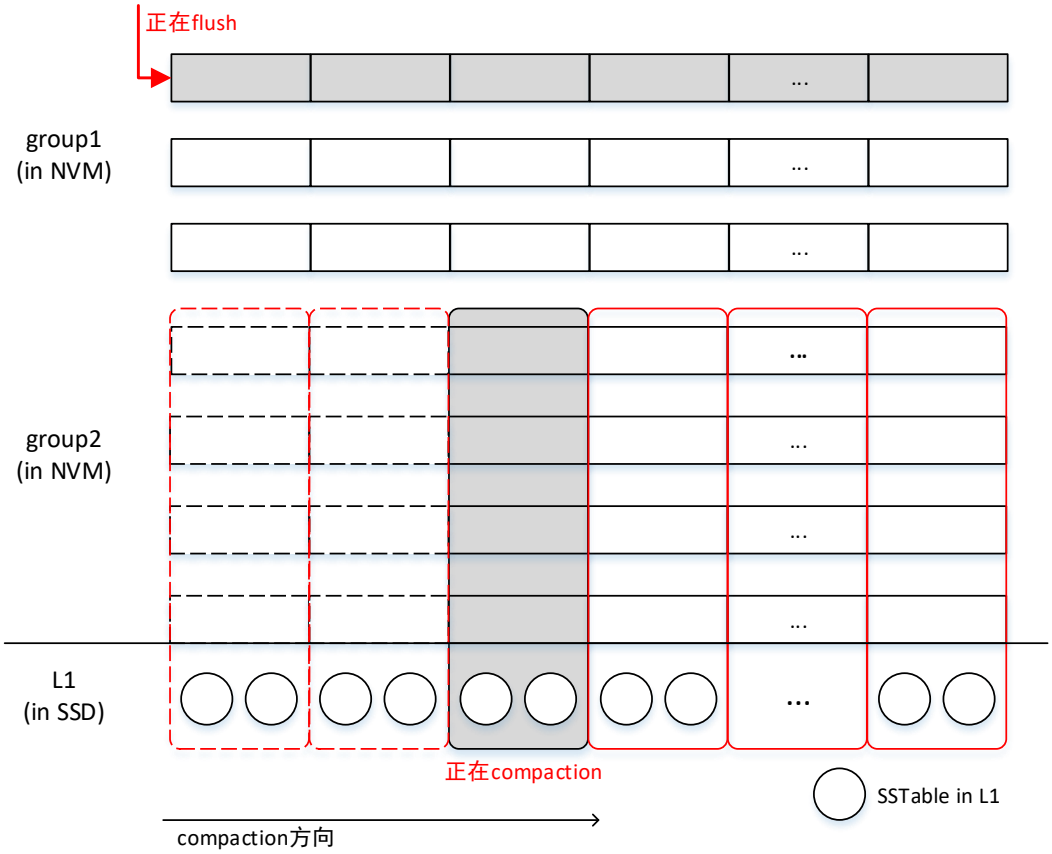

图 3-15 Compaction 示意图

NVM 分为两个部分，group1 保持 table 的结构，继续接受来自于 immutable memtable 的 table 插入过程，维持基本的 Matrix-Table 结构不改变；而 group2 部分按照一定的规则纵向切分成若干个大小相似的列块，以这些列块为基本单位依次向下层的 SSD 进行 Compaction。

由于 NVM 空间有限，group1 与 group2 的总空间容量一定，当 NVM 中数据量达到阈值时（如总容量的 2/3），系统将启动 Compaction 过程，将 Matrix-Table 现有的所有 table 均加入 group2，开始 Compaction 操作，完成 Compaction 的列块中的所有数据将会被回收，回收的空间用于存储 Flush 到 Matrix-Table 的新数据。而在系统开始 Compaction 过程中，所有 Flush 进入 NVM 的 table 将均加入到 group1 中，组成一个新的 Matrix-Table 结构。当 NVM 不足写入一个完整的 table 时，系统将产生 Flush 停顿延时，等待 Compaction 过程释放出相应空间后再进行 Flush 操作。

3.4.2 列块的切分算法

由本章节问题分析可以了解，L0 向 L1 中 Compaction 操作的性能开销除了受到 L0 中参与 Compaction 的数据量大小的影响外，还受到 L1 中参与 Compaction 的数据量大小的影响。Compaction 操作将以 L0 层所有数据 key 值所在的范围在 L1 层中选取与该范围有重叠的 SSTable，一起加载进入内存，进行合并，再写回到 SSD 中的 L1 层。因此，为了减少 Compaction 过程的次数，降低性能开销，且为了避免和 L1 层较多的 SSTable 进行合并而增大 I/O 带来的性能开销，Compaction 过程中 chunk 数据块的选择需要遵循以下原则——每次 Compaction 过程的数据范围尽可能小，以保证 Compaction 过程中不会涉及到过多的 L1 层文件。

在分块进行 Compaction 的设计中，可能会产生一个严重影响系统性能的问题——在一轮 Compaction 中，多个分块的 Compaction 操作可能会涉及 L1 层的同一个 SSTable，这就导致了这个 SSTable 数据不断地读入内存、写回 SSD，极大地增加了系统的性能开销。考虑到范围问题，以及避免多次读取同一个 SSTable 进入内存进行 Compaction 的问题，方案设计以 L1 层中 SSTable 的范围来进行列块的切分。

为了避免系统产生大幅的性能波动，列块的大小需要尽可能保证大小相近，由于方案设计以 L1 层中 SSTable 的范围来进行切分，现实环境下，这种切分方式将无法实现最理想的每个列块均相同大小的情况。由于 NVM 的引入，Matrix-Table 的容量极大程度地扩大，由当前主流系统默认的 64M 或 128M 扩大到了 GB 甚至几十 GB 的级别，这也使得了 L0 层中 SSTable 数量增多，采用累积合凑的方式可以近似达到每个分列类似相等的情况。

设 NVM 中 Matrix-Table 的总容量为 S，设定的分片数量为 N，预期每个分片的大小为 S/N。从小到大对所有的 SSTable 的范围进行排序切分，如图 3-16 所示。进行以下步骤：

① 有 L0 层 SSTable 中得到若干个范围边界点 t_0 至 t_n；

② 设置起点 start = t_0;

③ 对 group2 中的所有数组同时从左向右开始遍历；

④ 达到一个边界点 t_i 时，计算当前的总大小 s，若 s 大于 S/N，则将[start,t_i]为一个列块范围，使 start = t_i 作为起点，s 重新置为 0，继续执行③、④操作，若 s 小于 S/N，直接执行③、④操作；

⑤ 若 NVM 中 group2 数据遍历完毕，[start,∞)为最后一个列块范围。

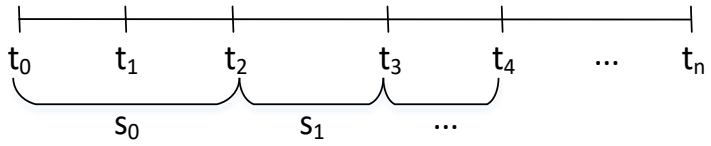

图 3-16 范围切分示意图

通过以上五步操作，所有切分出的列块大小基本接近于 S/N，不会出现极大或极小的块，而导致系统性能波动较大。

由于 L1 层中所有 SSTable 的范围两两之间均不重叠，以上方法每次切分的界限为 SSTable 的界限，即一个列块对应多个 SSTable，而一个 SSTable 只对应一个列块，如图 3-17 所示。每一个列块的 Compaction 均只会涉及该列块范围内的 L0 层 SSTable，避免了多次对同一个 SSTable 进行读写，避免了写放大。

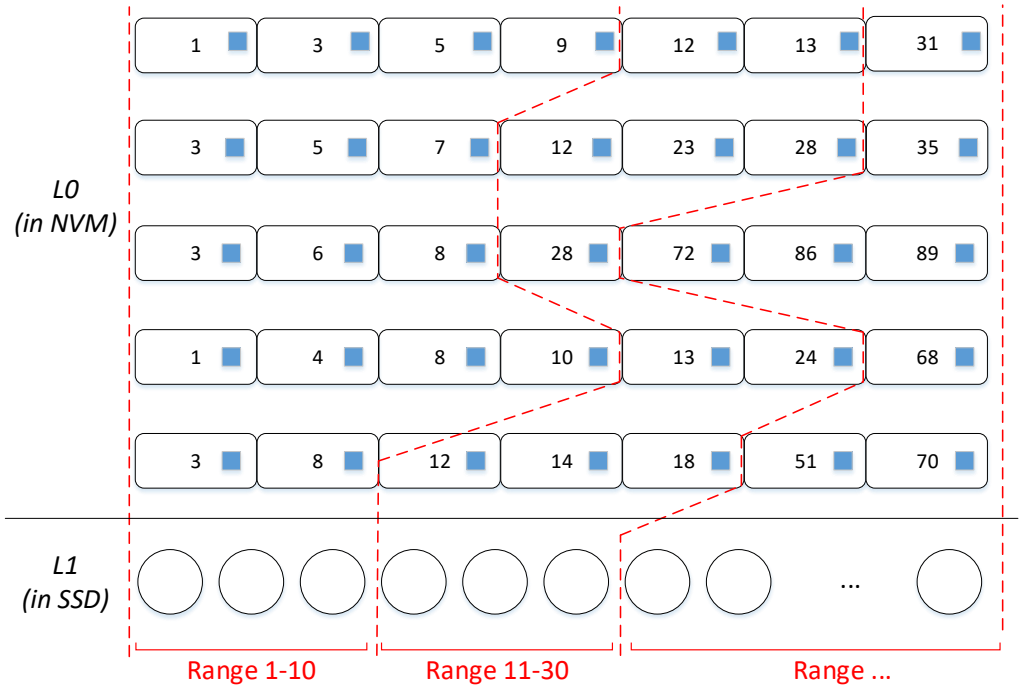

图 3-17 列块的切分

3.5 本章小结

本章首先分析了传统 LSM 树结构的键值存储系统所存在的问题——三种停顿延时以及 LSM 树结构的层次过多而带来的写放大问题，并针对问题提出了一种采用 NVM 与固态硬盘进行混合存储的持久化键值存储系统 MatrixKV。MatrixKV 主要包含代替 L0 层的 Matrix-Table 结构与细粒度列 Compaction 两个部分。方案采用 NVM 作为 Matrix-Table 的存储介质，通过扩大 Matrix-Table（L0 层）的空间来减少系统 LSM 树结构的层次，并在 Matrix-Table 中设计了一种矩阵结构的数据组织形式，提出了一种快速索引技术，保证系统在扩大 L0 层容量的情况下不对读性能造成影响；系统针对于停顿延时的问题，设计了一种细粒度以列为单位进行 Compaction 的方案，将大范围大容量的 Matrix-Table 进行切分，转变为对多个小范围小容量的列块进行 Compaction，避免了过多的数据 I/O 对系统性能造成严重的波动。

4 MatrixKV 系统实现

本文基于现今较为火热的键值存储系统——RocksDB 来完成对 MatrixKV 的实现，并采用 Intel 的 PMDK(Persistent Memory Development Kit) [22]来完成对 NVM 设备的驱动使用。

4.1 系统主要结构模块

由第三章节中整体结构图可知，MatrixKV 基本的结构与 RocksDB 类似，在 DRAM 中的基本结构与 RocksDB 相同，设置了 memtable、immutable memtable 结构。在底层数据持久化设备 SSD 中也与 RocksDB 类似，同样设置了 LSM 树存储结构，但 MatrixKV 方案将 LSM 树的每一层空间容量都进行了大幅度的扩大，以 Matrix-Table 代替 L0 层，并将 Matrix-Table 存储在 NVM 设备中。系统在 Matrix-Table 中采用 table 堆叠的形式，建立了一种索引矩阵结构的数据组织形式。并借助 Matrix-Table 的索引矩阵结构来对 L0 层（Matrix-Table）至 L1 层的 Compaction 进行了按列 Compaction 的细粒度切分优化。系统的主要结构模块关系图如图 4-1 所示。

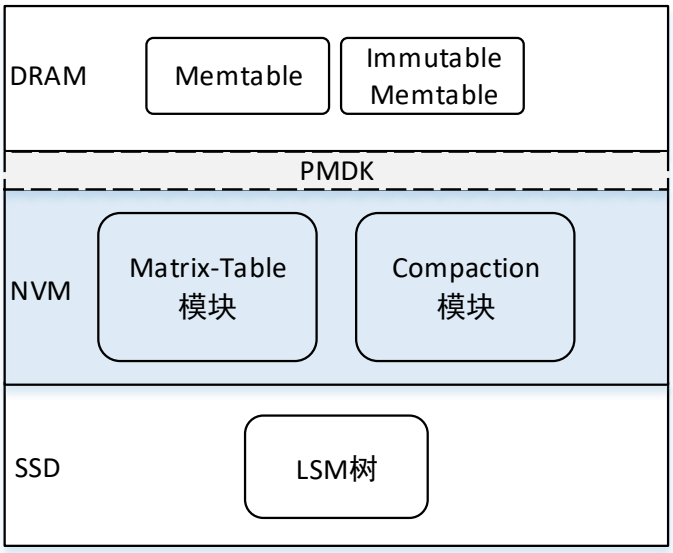

图 4-1 主要结构模块关系图

本章接下来内容将主要从 Matrix-Table 数据结构、Compaction 模块与读写流程三个部分介绍 MatrixKV 系统的实现过程。

4.2 Matrix-Table 模块实现

4.2.1 Matrix-Table 数据结构

Matrix-Table 包含两个部分，1）存储数据的数据区；2）存储元数据和数据之间关系的多维数组区。数据区的每一个 kv 数据单元，在元数据区都有与之对应元数据单元。

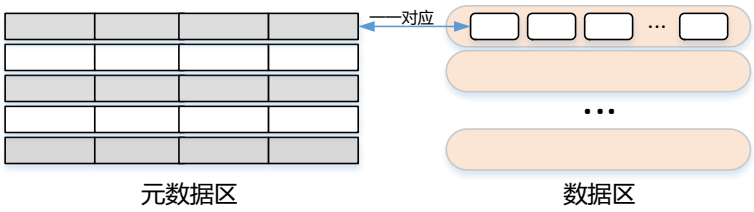

图 4-2 数据分区示意图

数据区中，每一个 immutable memtable 下发的 table 数据对应一个字符串数组，一个字符串数组成员对应一个 kv 数组。元数据区即矩阵结构的多维数组，其中每个单元为每一个 key 的相关信息，有着指向该 key 在数据区位置的指针，同时为了加速检索，每个元数据单元中还有对应的指针指向下层的单元。每个数据单元主要结构成员如下所示：

```
class matrix_union{
    p<int> ind;
    p<char[]> key;
    persistent_ptr<char> data;
}
```

结构基于 PMDK 工具开发包，p<template>表示存储于 NVM 上的持久化 template 结构对象，persistent_ptr<template>表示存储于 NVM 上的持久化 template 结构的对象指针。ind 表示单元指向的下层数组中相应单元的位置，key 表示本单元 key 的值，data 指针指向数据区对应的数据。

这些 matrix_union 结构单元数组，组成了一个 table 中所有数据的元数据数

组，同时多个元数据数组之间的 key 值关系指针又将这些一维数组构建成了一个二维矩阵数组。

4.2.2 Flush 过程实现

当 memtable 中数据大小达到上限后，将转变为 immutable memtable，同时启动系统 Flush 调度线程，将 immutable memtable 加入调度队列中，然后 Flush 线程启动进行 Flush 操作。Flush 操作的流程如图 4-3 所示。

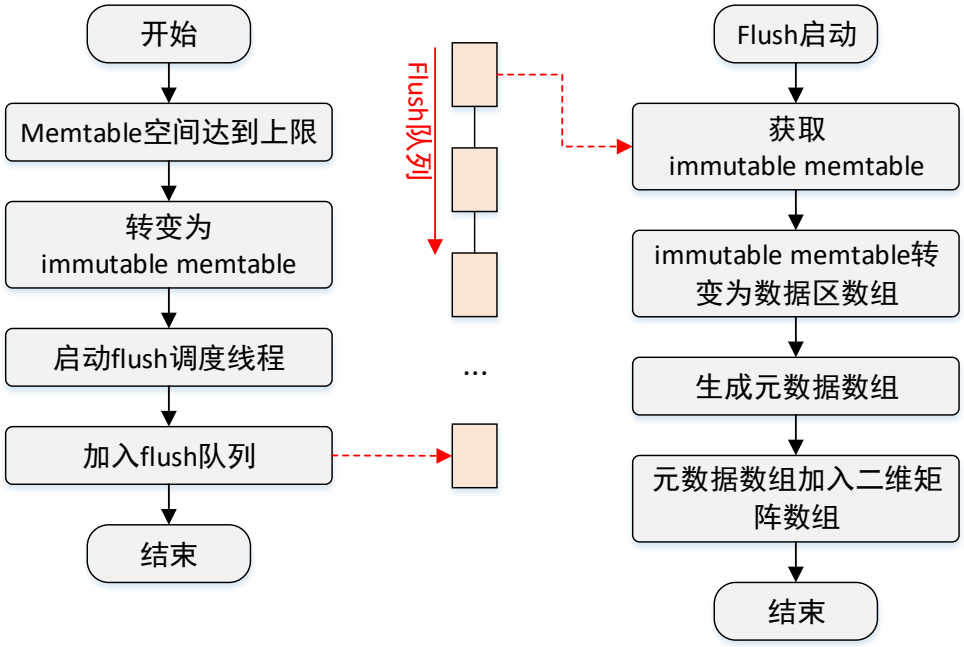

图 4-3 Flush 流程图

Flush 队列采用类似于拥塞队列的结构形式，当主线程中有 immutable memtable 加入到队列中时，主线程通知并启动 Flush 线程，进行相应的 Flush 操作。和 RocksDB 原有的 Flush 操作所区别是，除 L0 层的结构不同之外，系统的 Flush 操作的目标位置为 NVM，NVM 中采用类似于内存结构的管理形式，而非 RocksDB 使用的 SSD 的文件系统形式。

Immutable memtable 中数据按照原有的格式结构，直接转变为数据数组写入到数据区，同时提取相应的元数据信息，归并插入到二维矩阵数组中。二维矩阵数组在新增一层数据时，采用拉链式归并的策略，分别使用两个游标指针指向新数组 immutable memtable 和原二维矩阵数组最上层数组的最小值位置，比较两个

游标指针指向的值大小。若新数组指针指向的值较小，则将该数组单元的 pos 设为旧数组指针指向的单元，新数组游标向后移动一格继续比较；若旧数组指针的值较小，则指针后移一格，直到指针指向的单元 key 的值大于新数组指针指向的值。插入过程的伪代码实现如下所示。

伪代码 4-1 Table 的插入过程

```
输入：Immutable memtable,插入前的 Matrix-Table
输出：新生成的 Matrix-Table
1.    NewTable ← initArrayList(immutable memtable.size)
2.    newtable_index, lasttable_index ← 0
3.    LastTable ←Matrix-Table [Matrix-Table.size - 1]
4.    Flag ← true  //标记 LastTable 是否读完
5.    for InsertKey in immutable memtable do
6.    if Flag is not true then
7.          NewTable[newtable_index] ← package(InsertKey,-1)
8.          ++newtable_index
9.        else
10.         Key ← InsertKey – 1 //key 设定初始值，恒小于 InsertKey
11.         while Flag is true and InsertKey > Key then
12.           if lasttable_index >= LastTable.size then
13.               Flag ← false
14.               offset ← -1
15.           else
16.               Key ← LastTable[lasttable_index].key
17.               offset ← LastTable[lasttable_index].offset
18.               ++lasttable_index
19.           end
20.         end
21.         NewTable[newtable_index] ← package(InsertKey, offset)
22.         ++newtable_index
23.       end
24.   end
25.   Matrix-Table.append(NewTable)
```

当整个 immutable memtable 中的数据均插入到 Matrix-Table 中，本次 Flush 过程完成。

4.2.3 矩阵跳表的合并

当 Matrix-Table 中的 table 层次达到一定的阈值时，为了避免过多的层次给系统带来较大的读操作开销，系统需要对 Matrix-Table 内部的 table 进行合并。合并流程如图 4-4 所示。

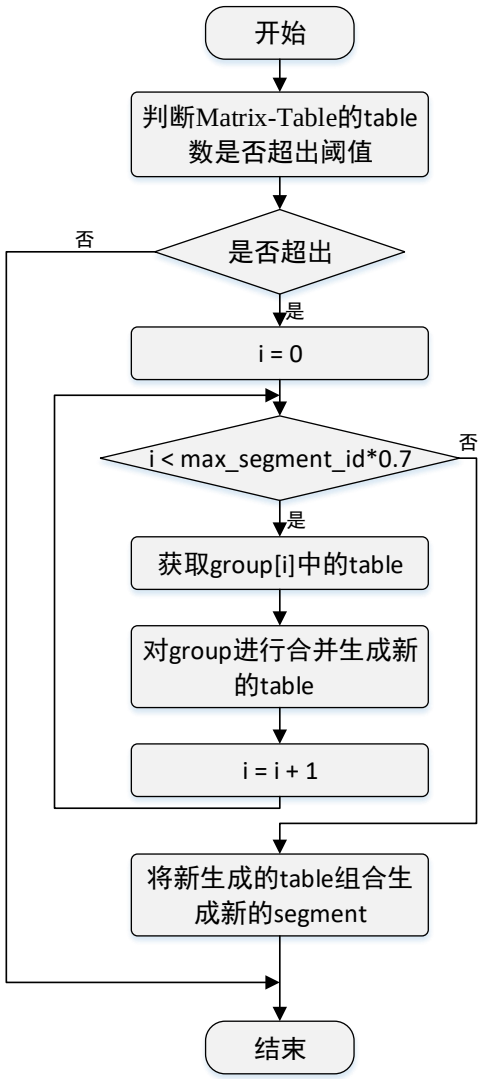

图 4-4 Matrix-Table 合并流程图

系统首先判断 Matrix-Table 中的 table 数量是否超过了阈值，若超过则执行

合并操作，否则不执行。合并操作以段(segment)为基本单位，从最下层的段开始，依次向上进行遍历，并对每个一个 segment 进行合并生成多个原 table 大小 150%倍的新 table，组成新的段加入到 Matrix-Table 中。由于考虑到最上层的 segment 可能会被频繁访问，因此合并过程只合并 Matrix-Table 自底向上的 70%的段。

4.3 细粒度 Compaction 流程实现

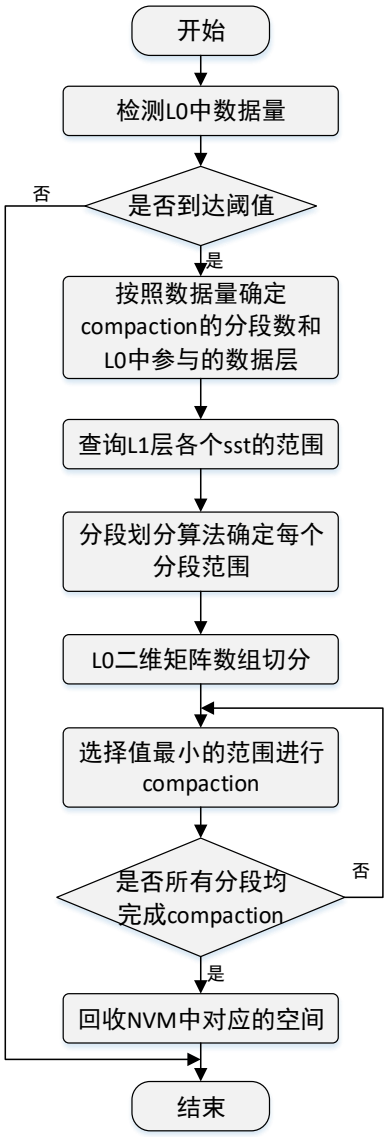

图 4-5 Compaction 整体流程图

当 Matrix-Table 数据达到一定的空间大小时，会对 Matrix-Table 中的数据进

行 Compaction，压缩写入到 LSM 树的更下层位置。由前一章节中的问题分析中可以了解到，系统的性能好坏极大程度地取决于 Compaction 过程的性能。如何较好地实现 Compaction 过程是本研究中最为关键的问题。为了保证系统性能的稳定性，并降低 Compaction stall 给系统带来的性能延时，MatrixKV 设计并实现了一种细粒度可控的 Compaction 方案。MatrixKV 中 Compaction 过程的流程图如图 4-5 所示。由流程图可以了解到，Compaction 过程可以分为三个阶段：1）判断是否需要进行 Compaction 过程；2）Compaction 分段选择和切分过程；3）向下层 SSD 中进行数据 Compaction 写入过程。下文将从这三个阶段来分别描述 Compaction 的实现过程。

4.3.1 Compaction 触发条件

当 NVM 中数据量达到一定的阈值时，需要对 Matrix-Table 进行 Compaction 操作。系统维护了一个关于 Matrix-Table 相关状态的数据结构。主要结构成员如表 4-1 所示：

表 4-1 二维矩阵相关状态数据结构成员

成员	类型	说明
Compact_begin_size	p<Unsigned long>	系统预设的开始 Compaction 的大小
Total_size	p<Unsigned long>	Matrix-Table 大小
Table_size	p<unsigned long[]>	Matrix-Table 中每一层 table 大小
Table_num	p<Unsigned int>	Matrix-Table 中 table 的数量
Compact_cnt	p<Unsigned long>	table 已完成 Compaction 位置
Data	persistent_ptr<Class data>	指向二维矩阵数组数据区的指针

每次向 Matrix-Table 中插入新的数据时，均要对以上结构体中的相关信息进行修改，插入 table 时需要对 NVM 的空间进行判定，即检测 Total_size 是否达到阈值，若达到则触发 Compaction 操作。

4.3.2 Compaction 前的准备

为了避免 Compaction 操作给系统带来过大的性能波动，需要对 Matrix-Table

中的数据进行切分转变为较小数据量的列块，再进行分段 Compaction。在准备过程中，首先对 L1 层所有 SSTable 的元数据信息(每个 SSTable 中 key 的范围)进行访问，再对 Matrix-Table 进行访问，按照一定的切分规则对其进行切分。整个切分过程需要遵循以下两个原则：

① 尽可能保证每个分块的大小相对一致；

② 尽可能按照 L1 层的 SSTable 范围进行切分，即同一个 range 范围只对应一个分片。

在此原则下，系统不会因为过大或过小的分块而导致了性能波动，同时 L1 每个 SSTable 只对应一个分片，避免 Compaction 过程中有过多的 SSTable 参与，导致较多的写放大问题。

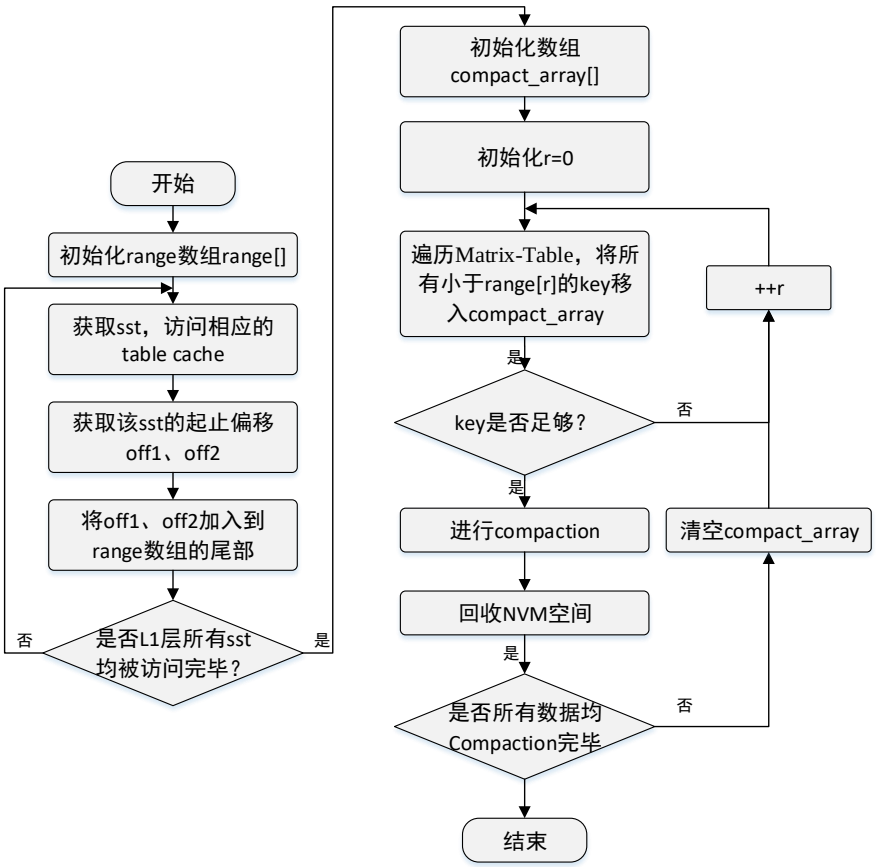

图 4-6 Compaction 准备阶段

如图 4-6 所示，整个准备过程分为两个阶段：

① range 确定阶段——访问 L1 层中所有 SSTable 的元数据，获取到每个 SSTable 中 key 对应的范围，以这些范围的起始和终止进行范围切分，得到相应的 range 数组。

② 数据切割阶段——按照预设的分块大小，对 Matrix-Table 中的数据进行访问，每当访问完一个 range 时，判断当前的数据量是否达到分块大小，若达到则在此处切割，分块数据量清零，并进入下一个 range 进行计数；若未达到分块大小，则直接进入下一个 range 进行计数。

当数据切割完成后得到 compact_array 数组，修改 Matrix-Table 的状态结构信息中的 Compaction_cnt，分别对应每个 table 中当前 Compaction 切块所到达的位置，以此记录来标记下一次 Compaction 时每个 table 中 key 的位置。同时将 compact_array 中 key 对应的数据单元进行访问读取，开始对相应的列块进行 Compaction 操作。

4.3.3 Compaction 过程

当compact_array选定完毕后，系统将启动后台线程BackgroundCompaction。该线程的主要工作内容为采用后台的方式，将 compact_array 中所有的 key 对应的 kv 数据信息与 SSD 中 L1 层 SSTable 进行合并，再写回 SSD 中，成为新的 L1 层 SSTable。Compaction 过程的主要流程如图 4-7 所示。

由图可知，Matrix-Table 中的数据和 L1 层的数据均会通过 builder 结构对元数据的访问得到具体内容后，再分别从 NVM 和 SSD 中读取加载入内存。在内存中，这些数据将合并，由于基于 LSM 树结构的键值存储系统的追加写特性，对于相同 key 值的数据，根据其自身的版本号 seqnumber 的大小，来判断最新写入到系统中的数据，判断完毕后，将旧的无效数据舍去，所有的新数据内容按照系统预设的固定 SSTable 大小进行切分，生成新的 SSTable 写回到 SSD 中。由于 Compaction 准备过程的存在，Matrix-Table 中的数据按照 L1 中 SSTable 的范围进行分割切分，即一个 Matrix-Table 分块对应多个 SSTable，一个 SSTable 对应一个 Matrix-Table 列块。这避免了在 PickFileToCompact 过程中，不同的列块的 Compaction 选择了同一个 SSTable，减少了文件冲突等待，也避免了更多的写放

大，提升系统的性能。

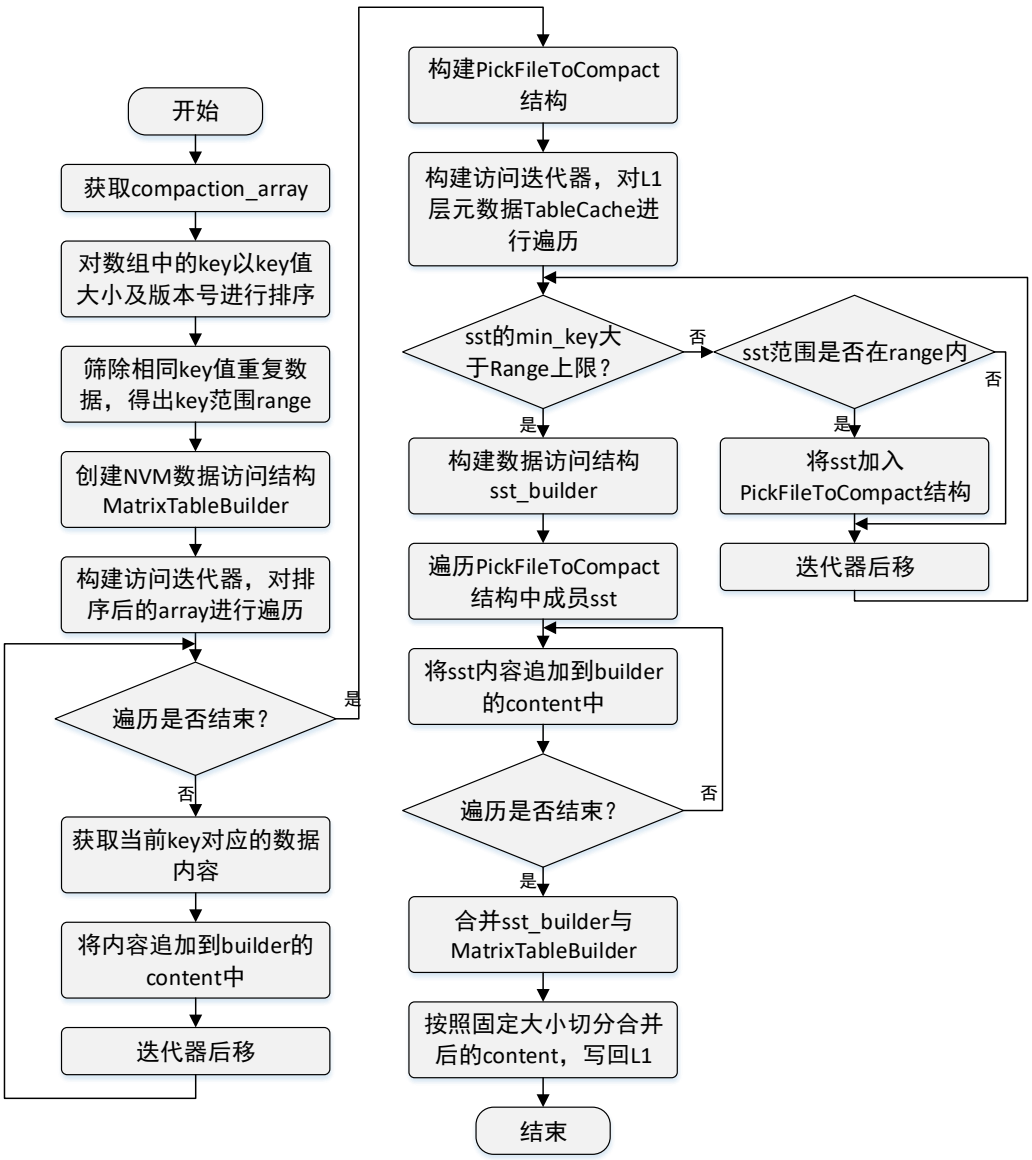

图 4-7 Compaction 过程

4.3.4 性能优化加速

现今的服务器系统一般采用了至少十六核的 cpu 核心，而本课题研究中所使用到的线程数在并行的状态下并不会让整个 cpu 满负载工作。因为基于这一场景，MatrixKV 系统在针对 Matrix-Table 进行切分时的循环遍历场景，使用 OpenMP 技术进行并行化加速。OpenMP 技术可将串行的非关联循环并行化，提升循环的

执行速度。OpenMP 在系统中的应用之一——遍历 Matrix-Table 主要部分的伪代码如下所示：

伪代码 4-2 OpenMP 加速示意

```
输入：Matrix-Table, compact_array
输出：目标结果
1.  table_num = Matrix-Table.table_num;
2.  i = 0;
3.  #pragma omp parallel for schedule(dynamic) num_threads(cpu_core_num)
    //开启 openmp
4.  do
5.      table_array = Matrix-Table.data[i];
6.      j = 0;
7.      do
8.          do_compare (L0_matrix); //在 Matrix-Table 内部进行比较操作
9.          ++j;
10.     while j < table_array;
11.     ++i
12. while i < table_num;
13. #pragma omp barrier;      //关闭 openmp
14. return compact_array;
```

由于 Matrix-Table 中的数据量相对较多，如若使用单线程对其进行循环遍历，会有比较大的性能开销，而使用 OpenMP 技术后，可以将开销缩短至原来的几分之一。

4.4 读写流程实现

4.4.1 读流程实现

采用 LSM 树结构的键值存储系统有着类似的读流程，从内存到底层磁盘中的 LSM 树中的最下层，本方案的读流程也与之类似，但在 Matrix-Table 中有着自己的读数据的方式。整个系统的读流程如图 4-8 所示。

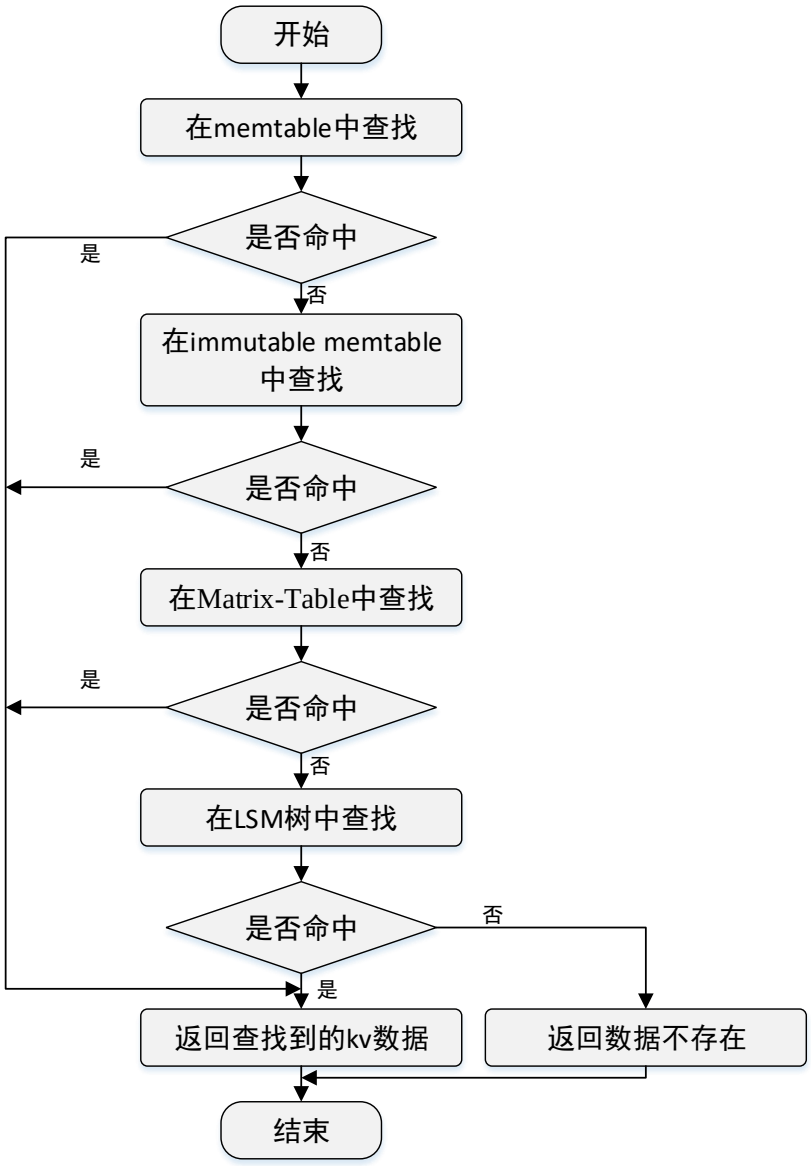

图 4-8 系统读流程图

由流程图可以看出，当目标 key 值在内存 memtable 与 immutable memtable 中未命中时，将会在 Matrix-Table 中进行查找。在 Matrix-Table 中查找的规则采用二分查找的形式，但是采用了矩阵跳表结构的形式来加速查找。查找过程的伪代码实现如下所示。

伪代码 4-3 查找过程

```
输入：目标 key，Matrix-Table
输出：目标 value
1.  N = Matrix-Table.table_num；
2.  min = 0;
3.  max = k;
4.  i = 1;
5.  do
6.      mid = (max − min)/2 ;
7.      do
8.          if Ri[mid] == key then
9.              return (Success,Ri[mid].meta);
10.         end
11.         if Ri[mid] > key then
12.             max = mid - 1;
13.         else
14.             min = mid - 1;
15.         end
16.     while min < max ;
        //向下一层 table 移动
17.     min = Ri[min].next – 1;
18.     max = Ri[max].next;
19.     i++;
20. while i ≤ N ;
21. return (Failure, null);
```

从代码可以看出从 Matrix-Table 的最上层开始，进行整个范围的二分查找，若在当前层命中，则直接返回；若当前层不命中，在确定到包含目标值的最小范围后，通过该范围索引到下层最小大于等于该 key 的元数据单元，快速确定在下一层中的查找范围，借此加速查找。

若在 Matrix-Table 仍未命中，则继续去 SSD 中查询剩余 LSM 树，查询方式与 RocksDB 相同。

4.4.2 写流程实现

与传统的基于 LSM 树结构的键值存储系统类似，MatrixKV 系统的写操作均由 memtable 接受与处理用户请求，并将数据直接插入到 memtable 的跳表中。写操作流程的实现流程图如图 4-9 所示。

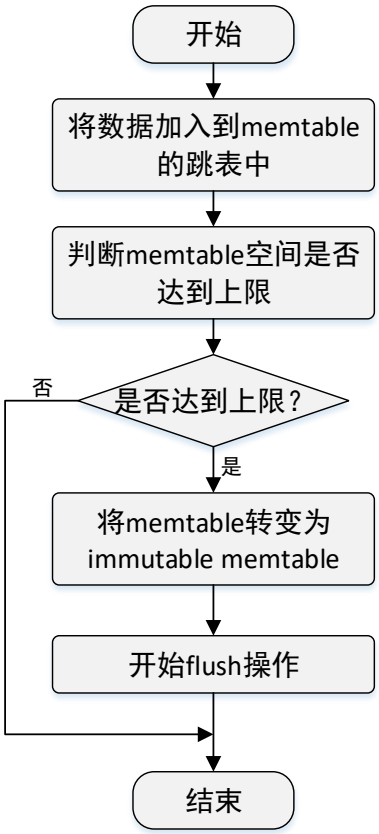

图 4-9 系统写流程图

插入完成后，判断系统 memtable 的空间是否已达到上限，若达到上限即触发 Flush 操作。Flush 操作在 4.2.2 小节中进行了介绍。

4.5 本章小结

本章节对 MatrixKV 系统的具体实现过程进行了详细介绍。第一小节介绍了 MatrixKV 的整体结构模块及各个模块的功能；第二小节对 Matrix-Table 结构的实现进行了介绍，详细说明了 Matrix-Table 的数据结构组成、Flush 操作的流程

以及 Matrix-Table 中 table 的合并优化；第三小节介绍了细粒度 Compaction 操作的实现细节及在 Compaction 实现过程中的代码优化策略；第四小节详细描述了系统读写流程的实现过程与细节。

5 MatrixKV 系统测试与结果分析

5.1 测试环境

实验测试所采用的服务器配置环境如表 5-1 所示。

表 5-1 服务器相关配置

操作系统	Federal 27
内核版本	Linux 4.13.9
处理器	Intel(R) Xeon(R) Gold 6140 CPU @ 2.30GHz *18 Core
主存	DDR3REG 32GB
SSD	Intel SSDSC2BB800G7 800GB
NVM	Apache Pass(Intel 3D Xpoint NVM 设备)[24]128G * 2

实验测试环境由 Intel 公司提供，测试所用 SSD 的基准读写速度如表 5-2 所示，由于 NVM 设备尚未对外公开发布，读写性能为 SSD 速度的 10 倍以上，具体详细性能数据不表。

表 5-2 SSD 设备基础性能

读性能		**写性能**	
随机读	顺序读	随机写	顺序写
290MB/s	356MB/s	245MB/s	273MB/s

测试主要针对课题的 MatrixKV 与 RocksDB、NoveLSM 两种系统的性能对比。考虑到传统 RocksDB 的存储介质为 SSD，为了更好地对比不同存储设备对系统性能的影响，测试设置了两种分别采用 SSD 作为存储介质与 NVM 作为存储介质的 RocksDB 对比方案。几种测试系统方案如下所示：

① MatrixKV，使用 8G NVM 作为 Matrix-Table 的数据存储空间，使用 800G 的 SSD 作为 LSM 树其余层次的存储设备；

② NoveLSM，使用 8G NVM 空间，其中 4G 空间作为 memtable，剩余 4G 空间作为 immutable memtable，使用 800G 的 SSD 作为 LSM 树的存储设备；

③ RocksDB-SSD，标准的 RocksDB 系统，使用 800G 的 SSD 作为 LSM 树的存储设备；

④ RocksDB-NVM，以 NVM 作为块设备代替传统 RocksDB 中的底层存储设备 SSD，使用 256G 的 NVM 作为 LSM 树的存储设备。

考虑 NoveLSM 系统基于 LevelDB 设计实现，不支持多线程优化等技术，为了保证测试的均衡与公平性，所有四种测试系统均采用单线程 Compaction 操与单线程 Flush 设置。Memtable、immutable memtable、及 SSTable 的大小设置为 RocksDB 默认的 64MB。

与传统存储系统不同，由于基于 LSM 树结构的存储系统在底层数据存储设备上均是顺序写 SSTable 的形式进行数据存储，本次实验中，顺序读写特指对连续递增的 key 值进行读写，随机读写指打乱 key 值顺序随机进行读写。

测试工具采用 RocksDB 自带的 db_bench 工具，使用 db_bench 生成对应的顺序随机读写数据集，分别对四种系统进行相应的数据测试。同时，还采用了 YCSB(Yahoo!Cloud Serving Benchmark)来对系统的性能进行了测试比较。

5.2 性能波动测试

第三章节已经指出，MatrixKV 方案设计针对于基于 LSM 树结构的键值存储系统由于过多且频繁的 Compaction 操作，形成大量的 Compaction 停顿延时导致系统性能严重波动的问题，提出了相应的解决方案。因此课题首先针对于系统的性能波动现象进行了针对性测试。

测试对比方案为 5.1 节中介绍的②③④方案，特别地，针对于同样使用了异构存储设备的 NoveLSM 进行了着重对比。为了较好地模拟实际应用场景下的性能负载，测试采用随机写的方式进行性能测试。测试内容为分别对四种系统随机写入 80GB 的 KV 键值数据，其中 key 键大小为 16B，value 值大小为 4KB，修改 db_bench 中的源码，使程序每隔固定一段时间输出这段时间内的系统写数据的平均速度。

5.2.1 系统性能波动对比测试

为了较为直观地观察四种系统方案当前时间的实时写入速率，且考虑到80GB 数据的写入所需时间相对较长，测试方案设置为每 30 秒对四种系统方案进行计算 30 秒内系统写入速率，绘制成速率与时间的变化曲线，如图 5-1 所示。

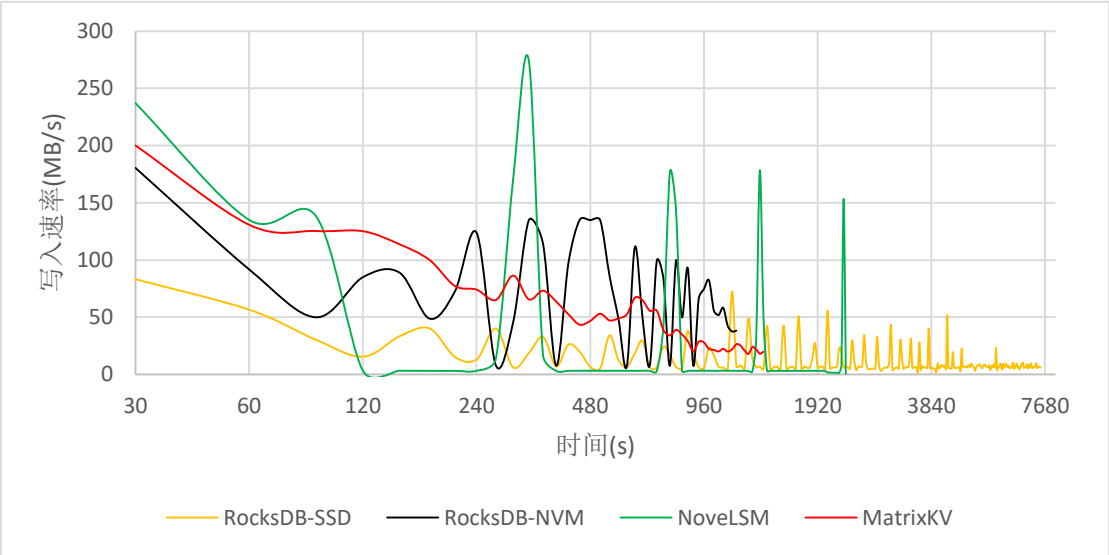

图 5-1 系统性能波动测试结果

由于采用 SSD 作为底层存储介质的 RocksDB 完成 80GB 数据写入所需时间相对较长，因此横坐标采用对数坐标显示。由图中可以得出以下结论：

① 根据时间的长短可以明显看出，MatrixKV 的性能要远高于 NoveLSM 与采用 SSD 作为存储介质的 RocksDB-SSD，速度接近于采用纯 NVM 作为存储介质的 RocksDB-NVM，这一定程度上显示了大容量的 memtable 数据集中 Flush 在性能上会对系统造成较大的影响，而采用分块的方式向 SSD 写入数据可以较好地缓解低速的写 SSD 给系统带来性能等待的影响，说明了本课题方案的细粒度 Compaction 的合理性。

② 由图中可以看出，RocksDB 与 NoveLSM 均存在着明显的系统停顿，系统的性能大幅度且周期性地降低至极低甚至为 0，然后再逐渐恢复。由于 RocksDB 的机制，在系统运行至一定时间时，L0 中的 SSTable 的数量到达触发系统的减慢前台请求的阈值（如 20 个 SSTable）时，系统将会开始进行 Compaction

操作，并降低前台处理写请求的速度。而当 L0 中的 SSTable 的数量到达触发系统的停止前台请求的阈值（如 20 个 SSTable）时，RocksDB 将停止前台的请求，将全部下层存储设备的 I/O 带宽用于 SSTable 的 Compaction。这一现象在第三章节问题分析中的图 3-1 中也得以体现，当 L0 到 L1 参与 Compaction 的数据量越多时，系统处理前台写请求的速度越慢。尽管采用 NVM 进行 SSTable 存储提供了极高的写 SSTable 速度，但仍然与传统的采用磁盘的系统拥有同样的停顿延时的问题。而针对于 NoveLSM，当系统内存中的 memtable 满时，NVM 中的 memtable 会替代内存进行写请求的缓冲处理，大容量的 NVM memtable 拥有较大的空间，可以一定程度地处理较多的写请求，然而，当 NVM memtable 转变为 NVM immutable memtable 并被写入到 L0 中，成为 SSTable 后，大容量 SSTable 在 L0 和 L1 之间的 Compaction 过程中，会导致大范围的 key 进行 Compaction，极大地增加了 Compaction 的数据量，引起了系统严重的性能停顿延时，如图中绿线中极长的约为零的时间段，显然，与其他三个系统对比，NoveLSM 系统的性能停顿延时最长。

③ 与 RocksDB 和 NoveLSM 相比，本课题提出的方案 MatrixKV 的性能更加稳定。方案采用细粒度的列块参与 Compaction，列块 Compaction 的策略将整个大数据量的 Matrix-Table 按照范围拆分成多个列，依次进行 Compaction 操作。对于每个列是小范围小数据量的 Compaction 过程，从而将大数据量的一次 Compaction，转变为小数据量的多次 Compaction，虽然在进行 Compaction 的过程中系统处理前台写操作的速率有所下降，但由于每次 Compaction 的数据量较小，一定程度地解决了系统性能的波动。

5.2.2 异构存储系统性能对比测试

为了更好地分析在大空间容量的 L0 层设计下细粒度可控的列块 Compaction 的合理与适用性，课题对 MatrixKV 和同样拥有较大 L0 层空间且采用异构存储结构的 NoveLSM 系统进行了细时间粒度的性能测试。测试环境和条件与上节相同，采用每 10 秒输出该 10 秒内系统的平均性能，绘制相应的数据曲线如图 5-2 所示。

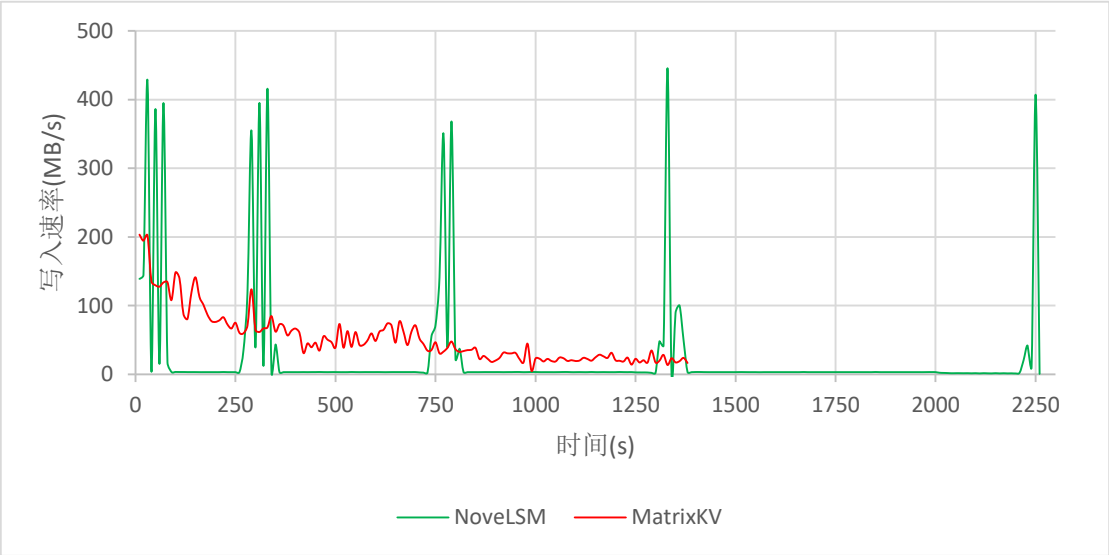

图 5-2 异构存储系统性能波动测试结果

由更细粒度的性能曲线图可以明显的看出 NoveLSM 在系统运行阶段中会有周期性的性能停顿延时，且这些性能停顿有着两种截然不同的状态：

① 如 0 到 100 秒之间，在短暂的时间内系统的性能从峰值直降到了接近零的水平，再迅速恢复峰值。这一阶段是由于 NoveLSM 系统 NVM 中的 memtable 进行数据 Flush 到 SSD 中所导致。由于 NVM memtable 的空间较大(对比测试方案设置为 4G)，较多的数据在 NVM memtable 满时转变为 immutable memtable 写入到 SSD 中的 L0 层，这对系统的性能消耗造成了一定的损耗，但由于数据量相对适中，因此系统的性能产生了一定的波动后，待数据写入完毕，性能又恢复峰值。

② 如 100 秒到 250 秒之间，系统面临了一个长时间的性能停顿，在这段时间里系统几乎完全停止了处理前台的数据请求。这个过程是因为由于 L0 层中的 SSTable 的数量达到了阈值，开始触发了系统的 Compaction 过程。由前一小节可以了解，由于 NoveLSM 结构的特殊性，L0 层 SSTable 较大，这很大程度地增加了 Compaction 过程的数据量，导致了较为严重的停顿延时，即第三章问题分析中提出的 Compaction stall。随着时间的增长，由于 L0 层中的 SSTable 的数据量越来越多，Compaction 过程涉及的数据量越发增大，这种停顿延时也越来越长，

由图中也可以明显看出，Compaction 带来的性能停顿时间由一开始的约 150 秒到最后变成了 800 多秒。

而本课题提出的 MatrixKV 由于 memtable 的大小为 64MB，所以较早时间开始了 Flush 过程，且采用 NVM 作为 L0 层的数据存储介质，且在 immutable memtable Flush 数据到 NVM 中时，需要进行组织管理建立矩阵索引链表，导致了在 Flush 过程中需要相对较多的开销，因此，系统的峰值性能相对于 NoveLSM 较低，且提前进入了性能下滑过程。当 L0 层数据达到阈值触发 Compaction 时，系统开始细粒度可控的 Compaction 过程，因此系统会有一定较长时间的周期性性能波动，如 100 秒到 250 秒之间的周期波动。随着 L0 层中数据数量逐渐增多，Compaction 过程变得越来越频繁，系统性能逐步下降。

5.3 读写性能测试

衡量一个存储系统性能的好坏，显然系统的读写速度是最为重要的指标。本课题针对于系统的顺序随机读写的性能进行了相应的测试，分析 MatrixKV 与三种对比方案的性能差异。

5.3.1 随机写性能测试

随机写的性能测试采用自带的 db_bench 程序，测试在固定 key 大小为 16 字节，不同 value 大小情况下，四种系统之间的性能差别。测试数据 80GB。测试结果如图 5-3 与图 5-4 所示。

由测试结果可以看出，随着 value 值大小的增大，系统的性能均有较小幅度的增大，原因之一在于由于 value 逐渐增大，单个 SSTable 中的 KV 键值对个数相对减少，Compaction 过程中覆盖的范围相对较小，开销相对较小，所以性能均有一定的提升。相对于采用 SSD 进行数据存储的 RocksDB 与 NoveLSM 来说，MatrixKV 在几种不同 value 大小的情况下，均有一定的性能优势。由于 80GB 数据相对较大，导致了 LSM 树的层次较多，过多的写放大给系统带来了较多的性能浪费，采用 SSD 作为底层存储的 RocksDB-SSD 性能相对较低，只有 7.52~11.14MB/s，远低于 SSD 的基本随机写性能，且由于 RocksDB-SSD 与

NoveLSM 和 MatrixKV 不同，没有使用新型的 NVM 介质，导致其性能远低于其他三种系统。而 NoveLSM 的性能只较 RocksDB-SSD 优，这与 5.2 章节中结果类似，大容量 memtable 给系统的 Flush 和 Compaction 带来了极大的性能影响，导致了虽然拥有着极高的峰值速率，但均值速度要低于同样使用 8GB 的 MatrixKV。MatrixKV 的性能要远高于 RocksDB-SSD 与 NoveLSM，性能接近于采用了全 NVM 进行 SSTable 存储的 RocksDB-NVM。由于过多的 Compaction 带来的写放大与性能停顿延时，使得所有数据均存储在高速存储设备的系统 RocksDB-NVM 性能在较小 Value 大小的状况下基本一致，较大 Value 大小状况下比使用了 8GB 的 NVM 空间的 MatrixKV 性能快约 40%。这样的测试结果表明了原生的 RocksDB 直接应用在 NVM 上在小 KV 场景下并不能完全利用 NVM 的性能优势，同时也说明了 MatrixKV 系统的性能拥有一定的优势。

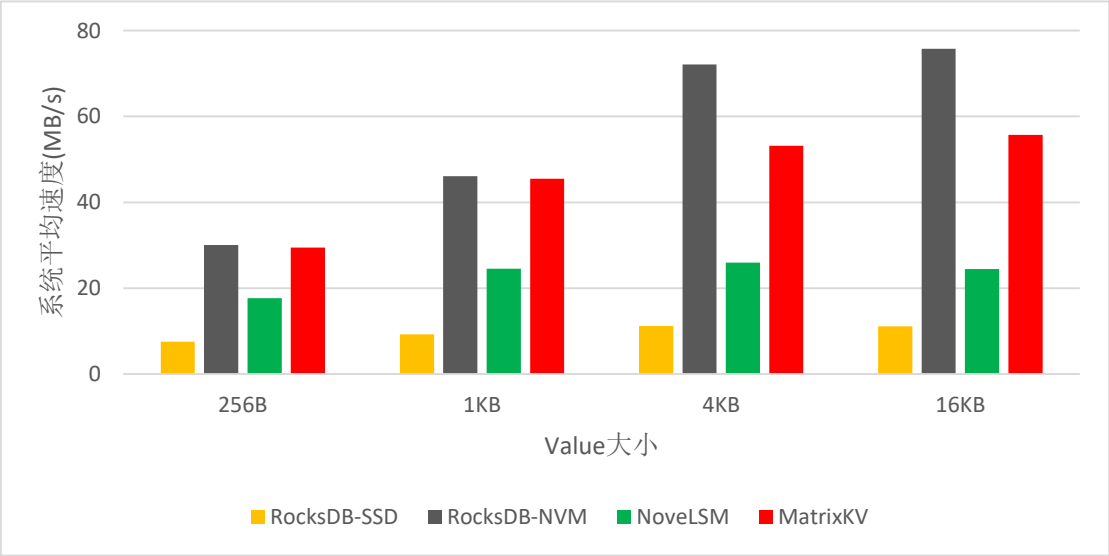

图 5-3 系统随机写测试速度

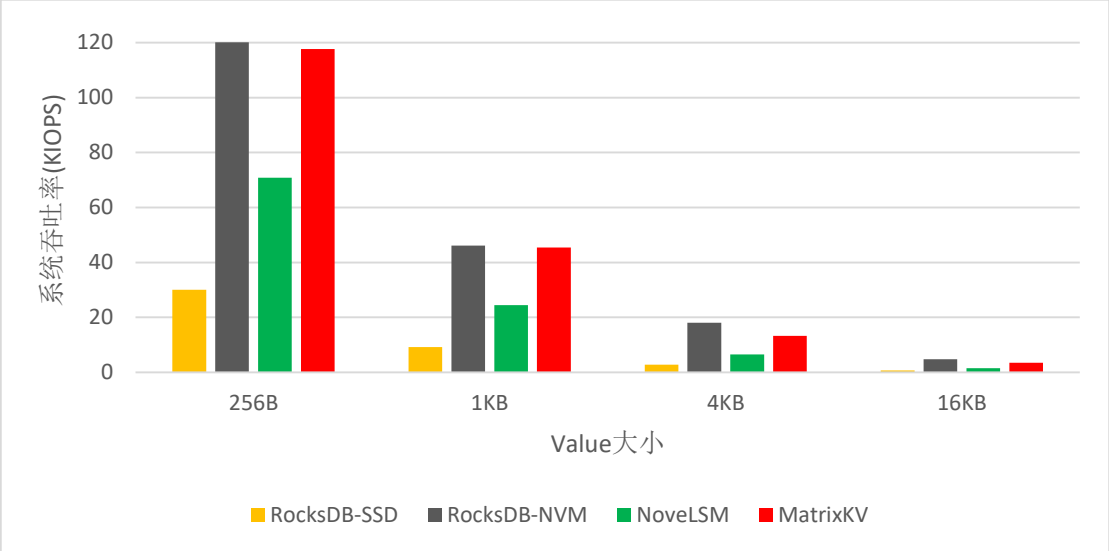

图 5-4 系统随机写测试 IOPS

5.3.2 随机读性能测试

四种系统在 db_bench 测试工具下的对随机读性能的性能进行了测试。测试内容为对前一小节测试中随机写入到数据库中的 80GB 数据进行随机读取其中的 100w 条数据内容，计算平均速度。测试结果如图 5-5 与图 5-6 所示。

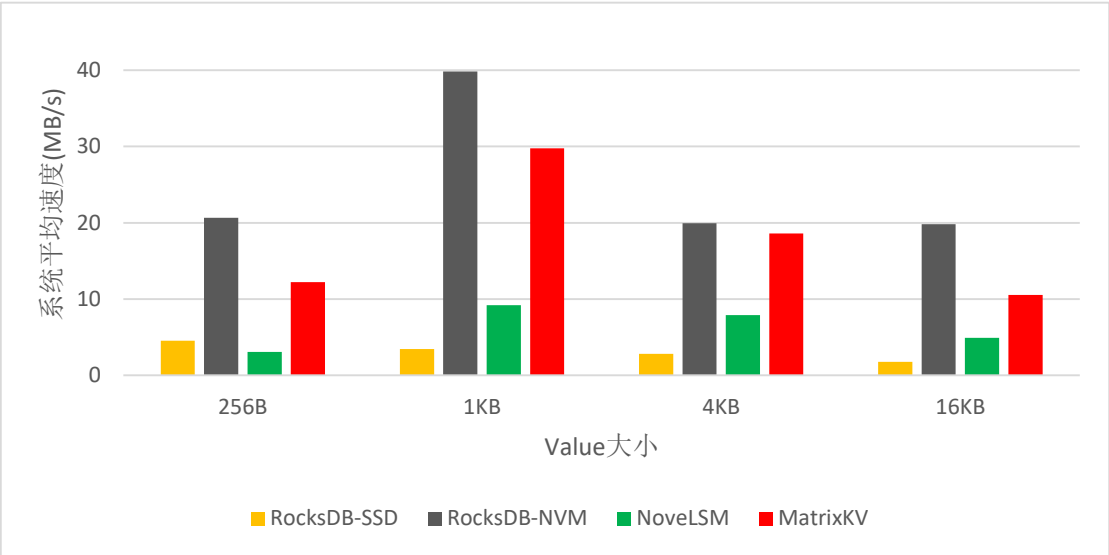

图 5-5 系统随机读测试速度

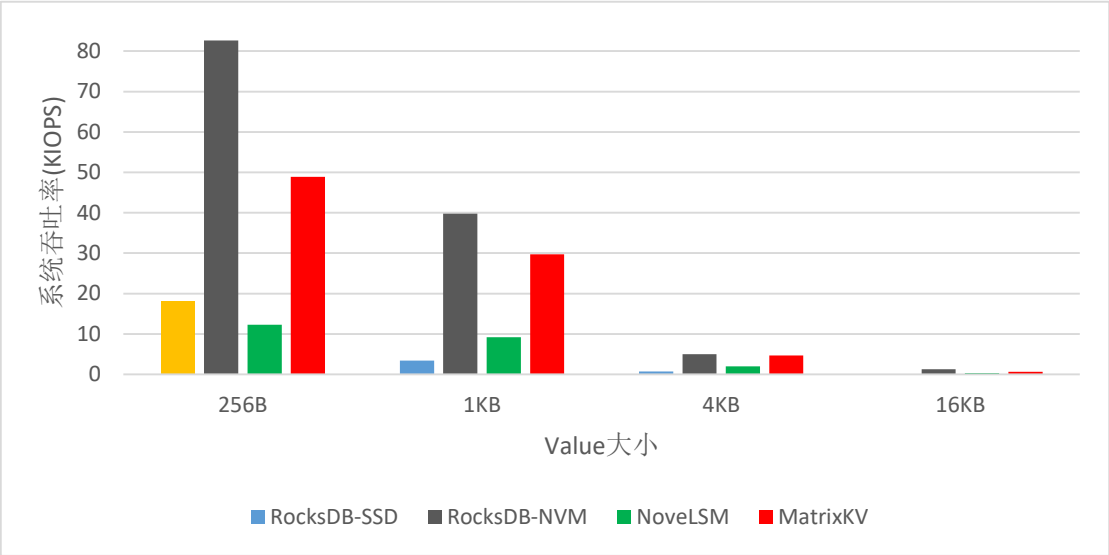

图 5-6 系统随机读测试 IOPS

由图可以看出，MatrixKV 的读性能在不同 value 大小下的随机读性能均高于 NoveLSM 与传统的 RocksDB-SSD，随机读的性能比采用 SSD 设备进行数据存储的 RocksDB-SSD 高出 2.7~8.6 倍，比同样使用了 NVM 设备的 NoveLSM 快了 2.1~3.9 倍。而完全采用了 NVM 作为存储设备的 RocksDB-NVM 仅比 MatrixKV 的随机读性能高出了 1.1 到 1.9 倍。这是由于 MatrixKV 对于 L0 层(Matrix-Table)类似于采用了一个大文件的形式，文件内部建立了多层索引指针的结构，且由于 NVM 高速特性的原因，系统对 Matrix-Table 中的数据访问拥有着极高的性能，而 RocksDB 与 NoveLSM 的 L0 层 SSTable 均处于 SSD 中，且相互之间未排序，这导致了每次的查找过程需要对所有的这些 L0 层的 SSTable 进行遍历，造成了较多开销。而采用 NVM 作为存储设备的 RocksDB-NVM 由于 NVM 的高速性，降低了这些开销的影响，因此性能较高。

5.3.3 顺序写性能测试

由于 LSM 结构的特殊性，顺序操作在 RocksDB 等系统中拥有着极快的性能优势。针对于顺序写过程，db_bench 程序将依次递增 key 值写入数据到系统中，这就导致了系统产生的每一个 immutable memtable 的 key 值范围均不相同，即 L0 层的 SSTable 的 key 值范围相互不重叠，这就给 Compaction 操作带来了极大

的便利，每次的 Compaction 数据量相对较小，性能损耗低。

课题对四种系统顺序写入 80GB 数据的性能进行了测试，性能结果如图 5-7 与图 5-8 所示。

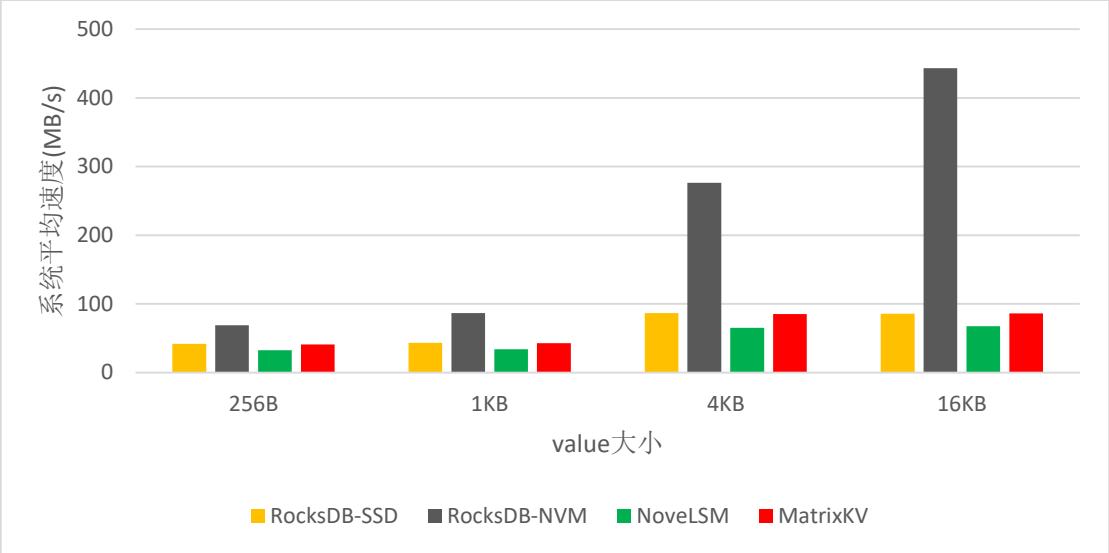

图 5-7 系统顺序写测试速度

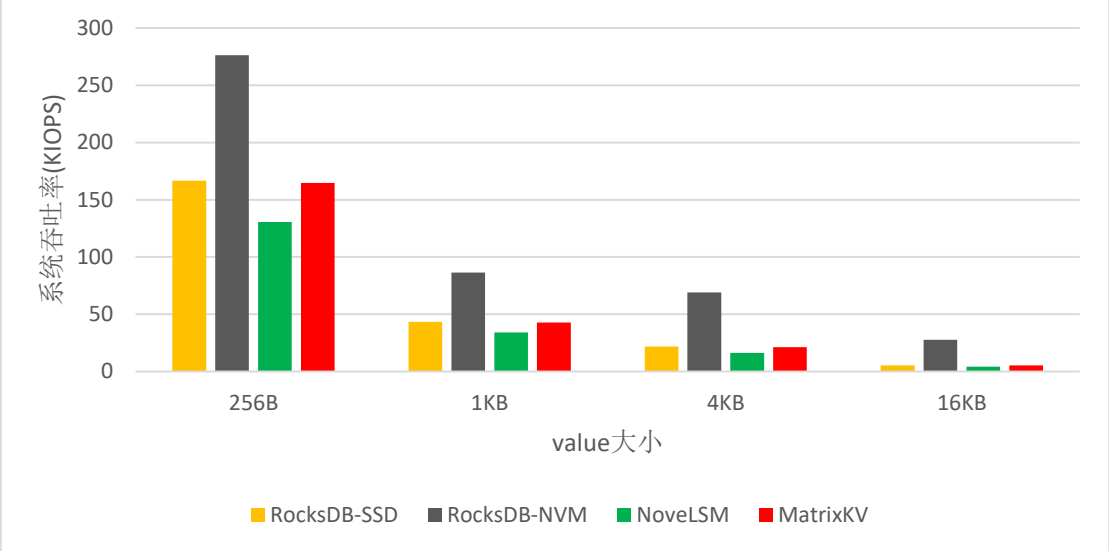

图 5-8 系统顺序写测试 IOPS

由图可以明显看出，在系统顺序写性能测试中，由于设备的高速特性，RocksDB-NVM 的速度远高于其他三种系统方案，而其他三种系统速度之间除 NoveLSM 性能稍低外基本区别不大。这是由于 RocksDB 与 LevelDB 等系统针对

于 Compaction 操作的一项优化方案——当需要进行 Compaction 操作的 SSTable 在 Compaction 过程中时，没有找到任何 key 范围与其重叠的 SSTable，则只需将该 SSTable 的元数据(Table Cache)进行修改转变为更底层的 SSTable，不需要对 SSTable 进行读取操作，因此基于 RocksDB 的 MatrixKV 和 RocksDB-SSD 与基于 LevelDB 的 NoveLSM 三种系统在 SSD 内部没有任何 Compaction 操作，NoveLSM 与 MatrixKV 相对于 RocksDB-SSD 多了写 80GB 数据进入 NVM 的过程，但由于 MatrixKV 的 NVM 中拥有 8GB 的数据空间大小，因此写入到 SSD 中的数据量稍少，因此 MatrixKV 的顺序写性能与 RocksDB-SSD 基本一致；而 NoveLSM 在 NVM 中不进行数据持久化，且其基于基础性能相对较低的 LevelDB 实现，因此性能相对其他两种系统较低。

5.3.4 顺序读性能测试

针对于四种系统的顺序性能，课题同样对其进行了测试。与随机读类似，测试过程采用 db_bench 测试工具，对前一小节测试中顺序写入到数据库中的 80GB 数据进行顺序读取其中的 100w 条数据内容，计算平均速度。测试结果如图 5-9 与图 5-10 所示。

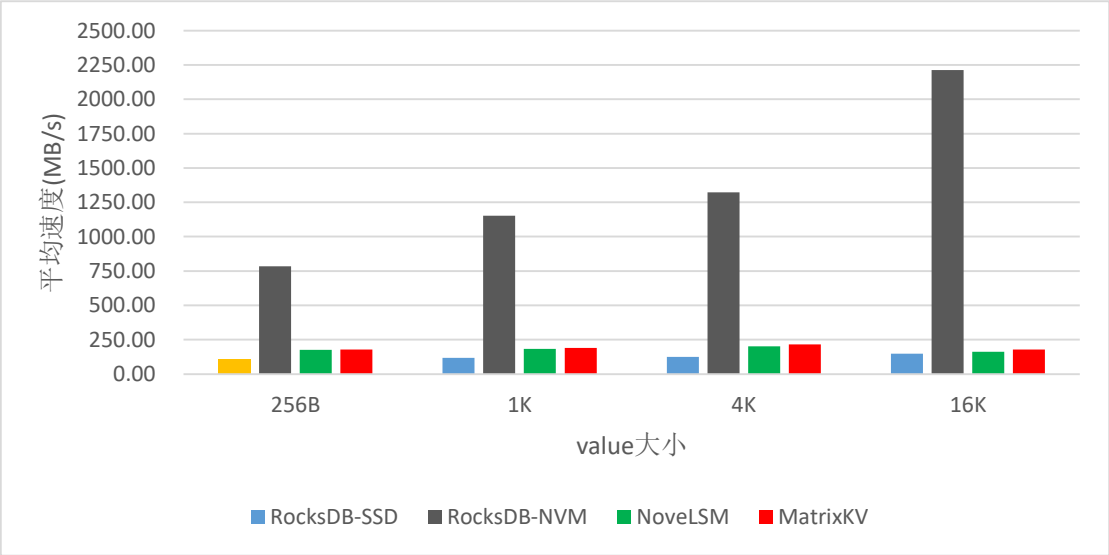

图 5-9 系统顺序读测试速度

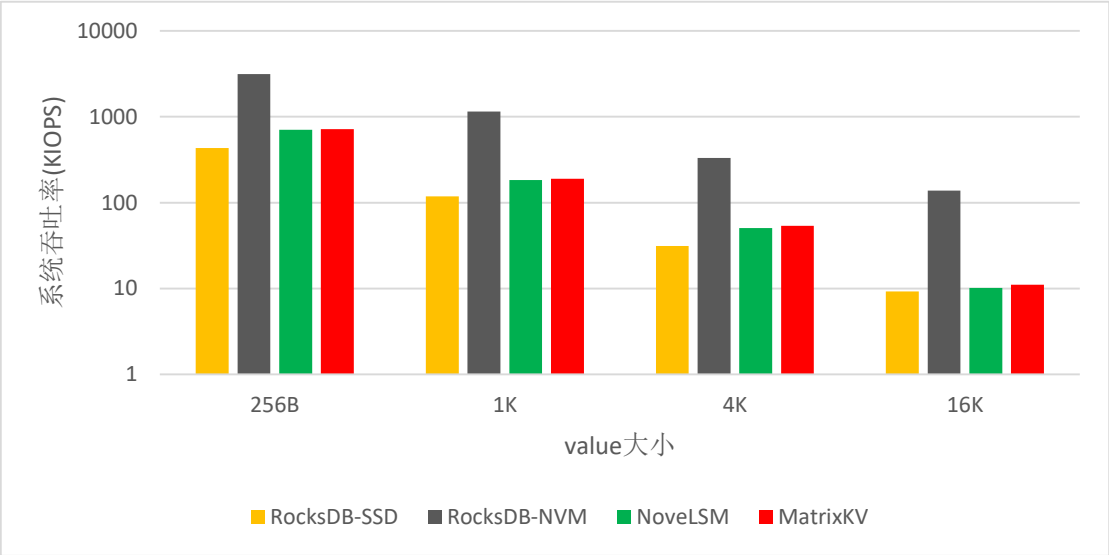

图 5-10 系统顺序读测试 IOPS

与顺序写结果基本类似，RocksDB-NVM 由于设备的高速特性，读取速度远高于其他三种系统，而同样使用了 8GB NVM 的 NoveLSM 与 MatrixKV 却相较于 RocksDB-SSD 的顺序读差距并不大，这是由于以下原因：

① NoveLSM 采用 LevelDB 实现，性能相较于 RocksDB 有一定的性能差距，且其 LSM 树结构均存储于 SSD 中，这导致了 NVM 的作用并不明显性能相较于 RocksDB-SSD 性能提升了 1.10~1.64 倍；

② MatrixKV 由于 Matrix-Table 的索引结构问题，顺序写入的数据在 Matrix-Table 中无法构成有效的索引指针，导致了在整个大容量的 Matrix-Table 中每个 table 均采用了全部范围的二分查找，极大地影响了性能，虽然采用了高速的 NVM 设备，但带来的性能提升只有 1.20~1.73 倍。

5.3.5 YCSB 测试

为了更好地分析系统在实际应用中的负载性能问题，课题采用了被 NoSQL 领域广泛使用的 YCSB 基准测试工具来对系统进行了性能测试，测试对比了四种测试方案的性能。YCSB 测试工具设置采用 7 个工作负载，每个工作负载的负载内容如下所示：

负载 load：对数据库写入 80GB 的数据；

负载 A：50%的读操作，50%的更新操作；

负载 B：95%的读操作，5%的更新操作；

负载 C：100%的读操作；

负载 D：95%的读操作，5%的热数据插入；

负载 E：95%的范围查询，5%的数据插入；

负载 F：50%的读操作，50%的读修改写（read-modify-write，RMW）操作。

YCSB 测试结果如图 5-11 所示。

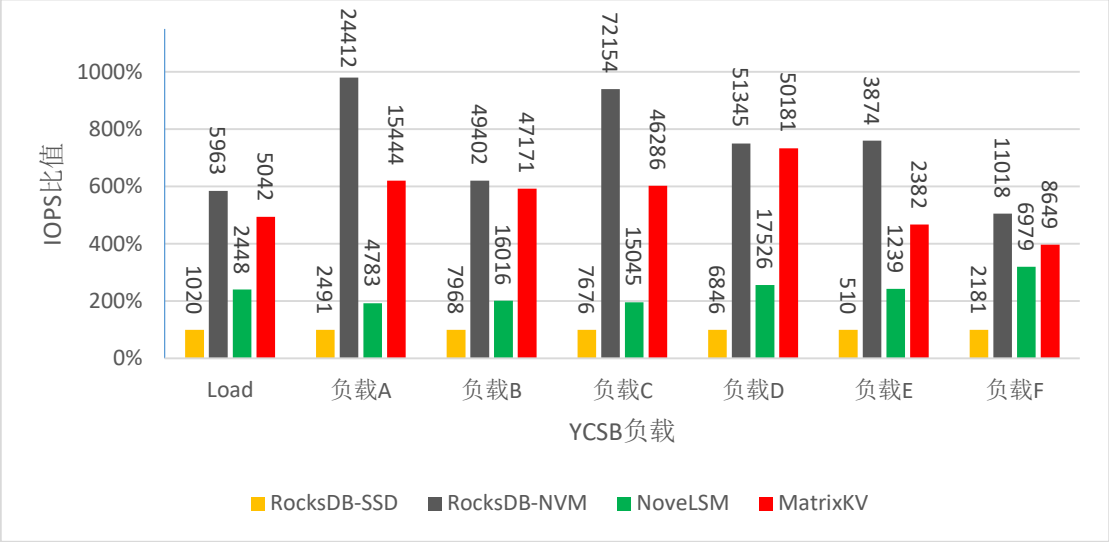

图 5-11 YCSB 测试

由于不同的负载中各个系统的 IOPS 相差较大，为了较好地显示系统性能优异，图中纵坐标设置为与 RocksDB-SSD 的比值，图中标签对应了各个系统在不同负载下的实际 IOPS。由图可知，在数据库构建的过程中，MatrixKV 的性能要优于 RocksDB-SSD 与 NoveLSM，是 RocksDB-SSD 的 5.94 倍，是 NoveLSM 的 2.06 倍。而采用了全 NVM 存储的 RocksDB-NVM 性能仅超出 MatrixKV 的 18%。负载 A-F 分别在 Load 负载构建的数据库中进行相应的读写修改等操作，MatrixKV 系统的性能为 RocksDB-SSD 的 4.67~.33 倍，是 NoveLSM 的 1.23~3.23 倍。在负载 B 与负载 D 上，MatrixKV 系统的性能接近于 RocksDB-NVM，尤其是负载 D 中 MatrixKV 的性能达到了 RocksDB-NVM 的 97.7%，原因在于一个采用 LSM 树结构的存储系统，LSM 树越上层的数据越热，绝大多数的热数据都存

储在 L0 层中，而 MatrixKV 拥有着大容量且存储在 NVM 上的 L0 层(Matrix-Table)，因此对热数据的访问拥有较好的性能。

5.4 系统写放大性能测试

由文章第三章问题分析小节可以了解，一个基于 LSM 树结构的键值存储系统，LSM 树的层次是影响系统性能的重要因素之一。层次越多意味着系统在数据写入到存储设备中时经历的 Compaction 次数越多，且越下层的空间大小越大是 LSM 的结构特性之一，这意味着层次越多带来的写放大越多，对系统的性能影响也越大。本小节对几种系统在随机写 80GB 数据到系统中的写放大与数据分布等结果进行了测试分析。

5.4.1 系统数据层次分布

测试参数设置与性能波动测试相同，针对于 16B 大小的 key，4KB 大小的 value 进行随即写入 80GB 数据，等待系统所有数据写入完成后，统计各个系统 LSM 树结构中的数据量占比。

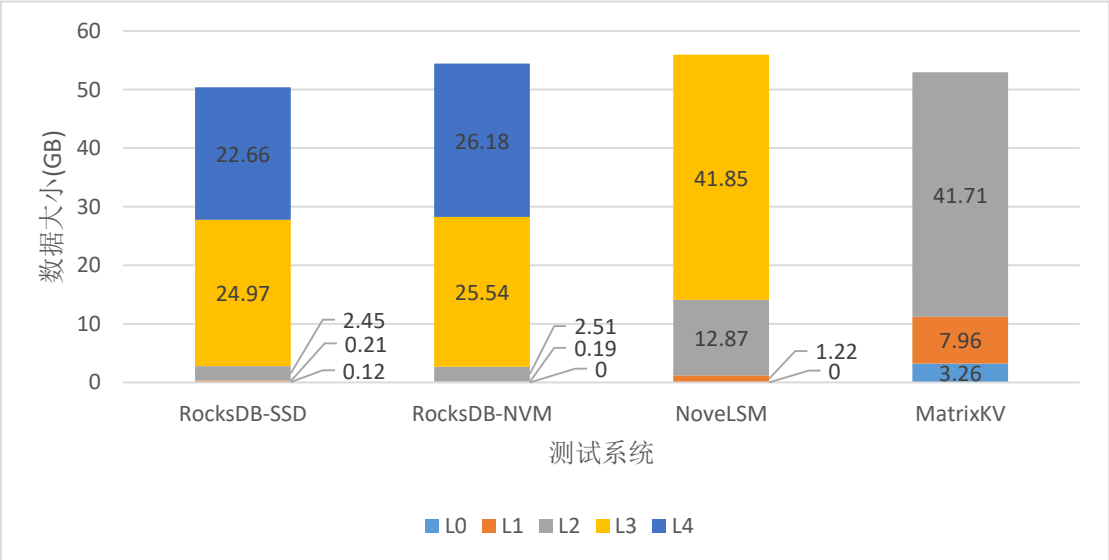

图 5-12 系统数据层次分布

四种系统完成 80GB 的数据写入后，LSM 的各个层次数据总量如图 5-12 所示。四种系统的最终写入了 LSM 树结构中的数据量大小相近约为 50GB，但明

显可以看出，MatrixKV 的最终最大层次为 L2，而其余系统的层次均相对高，如 RocksDB-SSD 和 RocksDB-NVM 达到了 L4 层，NoveLSM 达到了 L3 层。由于 MatrixKV 的每一层空间相对较大，导致了 L0、L1 等层可以容纳相对较多的数据，使得其最终的数据分布层次较少，NoveLSM 虽然拥有着较大的 NVM memtable，使得其在 L0 层的 SSTable 的大小可以达到 4GB，但由于其是 DRAM memtable 与 NVM memtable 双线流程，因此可能导致了 L0 层的 SSTable 虽然数量多，但是实际上有一部分 SSTable 大小只有 64MB，导致了存储数据量相对较少，造成了一定量的 Compaction，使得 LSM 层数变多。四类系统的数据分布基本符合越下层数据量越多的特点。

5.4.2 系统 compcation 次数分析

课题对上一小节中的测试过程中各个系统的 Compaction 的次数与数据量进行了统计与分析，结果如图 5-13 所示。

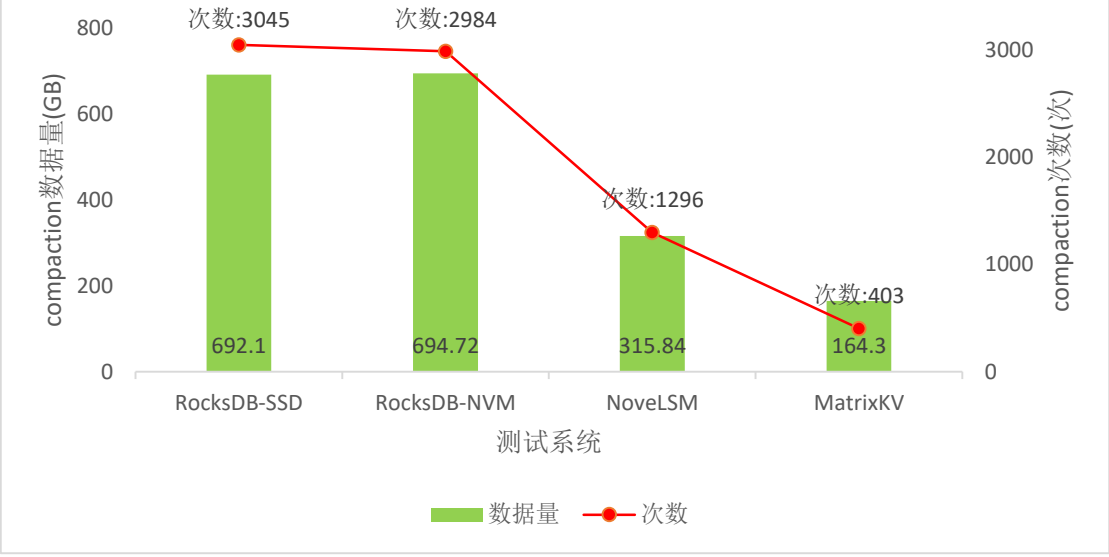

图 5-13 系统 Compaction 数据量与次数

从图中可以明显看出两种不同存储介质的 RocksDB 在 Compaction 次数与数据量上均远高于 NoveLSM 和 MatrixKV，MatrixKV 的次数和数据量最少，只有 NoveLSM 的一半左右，这一数据结果很好地呼应了前一小节中四种系统的 LSM 树结构中的数据分布结果。由于 RocksDB 中 LSM 树每一层的空间远小于

NoveLSM 与 MatrixKV，数据空间的不足导致需要向更下层进行合并，所以导致了更多的 Compaction 操作次数与数据量，浪费了过多的系统资源。两种系统由于结构相同，只有存储介质的区别，因此 Compaction 的数据量与次数基本一致。RocksDB-SSD 由于 SSD 的 I/O 速率低，过多的 Compaction 数据必然给系统带来了较大的停顿延时，而 RocksDB-NVM 因为 NVM 的高吞吐速度，694.72GB 的数据量可以在较短的时间内写入，所以 Compaction 给系统带来的性能浮动相对较小。

NoveLSM 系统由于层次相对于 MatrixKV 较高，所以 Compaction 的次数要高于 MatrixKV，但远低于 RocksDB-SSD 与 RocksDB-NVM。

5.4.3 系统写放大数据分析

为了更好地分析由于 LSM 树结构而导致的写放大性能问题，课题针对四种系统的 80GB 数据随机写测试的写放大相关的数据信息进行了统计与分析，数据包括有最终有效地保留在存储设备（SSD 或 NVM）中的数据总量及系统运行中总的写入数据量。四种系统的写放大比例结果如图所示。

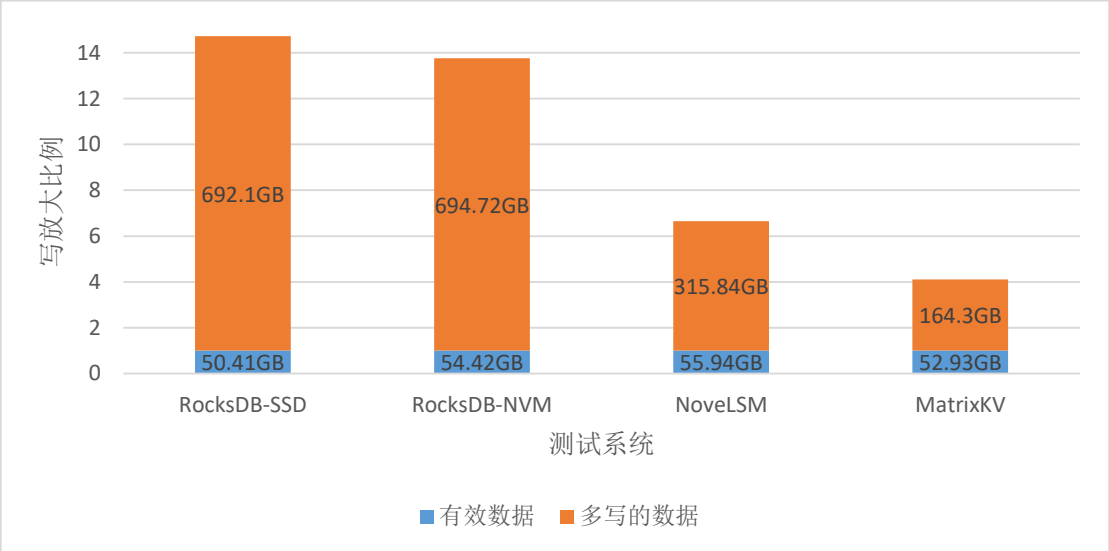

图 5-14 系统写放大情况

图中纵坐标为两种数据量与有效数据数据量的比值，以此来更为直观地显示写放大的比例。由图可知，LSM 树结构层次最深的两种 RocksDB 系统的写放大

比例最高，分别为 14.73 和 13.77，而 NoveLSM 与 MatrixKV 相对较少分别为 5.65 和 3.10，这进一步佐证了前文的分析——LSM 树层次越深，给系统带来的写放大越大。NoveLSM 由于增加了 L0 层的空间进而使得 LSM 树的层次变少，避免了过多的写放大。而 MatrixKV 除却增大了 L0 层的空间大小外，还在 L0 层 Compaction 过程中进行了无范围覆盖的列块式切分进行细粒度的 Compaction，使得系统在每次 Compaction 中不会有过多的数据重复参与 Compaction，最大程度地避免了写放大问题。

5.5 本章小结

本章节首先针对课题研究所提出的问题——采用 LSM 树结构的存储系统会因为三种停顿延时而导致系统性能产生较大的波动，对 RocksDB-SSD、RocksDB-NVM、NoveLSM、MatrixKV 四种系统在 80GB 数据随机写状态下每半分钟的系统平均速度进行了统计，得出了四种系统在运行过程中的性能波动情况，分析 RocksDB、NoveLSM、MatrixKV 的优缺点，并分析 MatrixKV 的性能相对平稳以及 NoveLSM 波动幅度较大的原因；然后对于同样采用了 8G 的 NVM 设备构建异构存储系统的 NoveLSM 与 MatrixKV 进行了细粒度的性能波动情况进行了分析，分析了两种系统性能波动曲线各个阶段不同状态的原因，并从系统结构层次对其进行了分析；随后对四种系统的读写性能进行了相应的测试，测试结果显示 MatrixKV 要优于 RocksDB-SSD 与 NoveLSM，接近于 RocksDB-NVM，这说明了课题方案针对于停顿延时的优化起到了不错的作用；最后对 LSM 树结构的存储系统常见的 LSM 树数据深度与写放大情况进行了测试与分析，测试结果证明了 LSM 树写放大越大，LSM 树数据深度越深，MatrixKV 相对于其他四种系统 LSM 树数据深度最低，很好地降低了 LSM 树的数据层次，而两种 RocksDB 系统拥有着最高且近乎一致的写放大情况，说明了写放大与数据存储设备的种类无关，这与第三章节中的问题分析得出的结果相呼应。

6 总结与展望

6.1 全文总结

针对于现今时代的数据爆炸场景，非易失性存储设备与非关系数据库系统，这软硬件两种技术均可以较好地应对当前环境对数据存储的高速需求。本文首先对这两种技术的原理与国内外研究现状进行了相应的分析介绍，并针对于非关系数据库系统的典型代表之一——基于 LSM 树结构的键值存储系统进行了分析，分析得出由于 LSM 树的结构特性，系统在运行过程中，随着数据地不断写入，需要定期地完成数据导入到持久化存储设备（如 SSD）中，且需要定期对持久化设备中的数据文件进行合并，频繁的合并给系统的性能带来了极大的开销，造成系统产生了周期性的停顿延时，导致系统性能降低，并给用户带来较差的性能体验。

针对上述的系统性能延时的问题，本文利用非易失存储设备 NVM 的非易失性与高速性，提出了一种基于 NVM 与传统存储设备 SSD 结合的混合键值存储系统 MatrixKV。MatrixKV 基于 LSM 树结构，采用高速的 NVM 设备来代替 SSD 作为 L0 层数据的存储设备，对 L0 层的数据结构进行修改，采用 table 堆叠，并在相邻层的 table 之间构建关系索引指针，形成矩阵结构的 Matrix-Table；同时修改 L0 层的 Compaction 策略，将传统的按 table 进行 Compaction 的过程，转变为按列切分 Matrix-Table，进行以列为单位的细粒度 Compaction 操作。方案设计的主要特点在于以下几个方面：

① 利用 NVM 设备，扩大 L0 层的空间，并通过按列进行 Compaction 的方式，缩小了每次 Compaction 操作中涉及的数据范围，减少了所设计的下层 SSTable 的个数，减少了 Compaction 的数据量，缓解了 Compaction 过程中的写放大问题，因而降低了 Compaction 操作对系统性能的影响，较好地缓解了系统性能波动的问题；

② 通过扩大 Matrix-Table 的空间，使得 LSM 树结构中每一层的空间均相应扩大、LSM 树层次变少，缓解了 LSM 树内部的数据 Compation 带来的数据写放

大问题，提升系统的性能；

③ 由于传统的 LSM 树结构系统 L0 层 table 之间无序的特性，系统对 L0 层中数据的读访问需要遍历 L0 层所有的 table，扩大 L0 层的空间会一定程度上给系统的读操作带来较大影响，因此，MatrixKV 针对 Matrix-Table 的数据进行了组织，table 以堆叠的形式进行数据存储，并在相邻的 table 之间建立关系索引指针，使得在二分查找完上层 table 后，可以通过索引指针直接确定下层 table 中包含目标数据的数据段的位置，减少查找的范围，加速查找过程。

文章针对于以上设计方案，在 RocksDB 系统的基础上进行了实现，并在 Intel 公司真实的 NVM 设备 Apache Pass 上进行了测试。测试结果表明，MatrixKV 系统可以有效地缓解系统性能波动的问题，带来了一定的系统性能提升。其随机写性能是采用 SSD 作为存储设备的 RocksDB 的 3.91~4.99 倍，是 NoveLSM 的 1.66~2.27 倍。同时系统在写放大问题上也有了一定的优化，写放大比例为 3.10，RocksDB 为 14.73，Nove LSM 为 5.65。

6.2 研究展望

MatrixKV 系统在系统性能波动与写放大等问题上较传统基于 LSM 树结构的键值存储系统有较大的性能改进，但目前该系统仍然存在着一些不足和可以改进的地方，具体如下：

① 在针对于 table 之间毫无范围重叠的情况，即完全顺序读写的情况下，L0-table 中的索引结构不能起到任何读取加速作用，反而会因为插入时的开销，导致性能下降；

② 由于 LSM 树结构的特性，因此对现今 NoSQL 系统的常见需求场景范围查询请求支持较差，性能相对不如其他不采用 LSM 树结构的存储系统。

③ MatrixKV 解决系统写放大问题的方式是增加 LSM 树每层的大小，这并不能从根本上解决写放大问题，后续的研究方案可以从 LSM 树的结构入手，通过修改结构来减少 SSTable 的合并次数。

致谢

光阴似箭，日月如梭，三年的硕士生涯转眼就所剩无几，不知不觉我在母校已经度过了七年的时光，七年里我从原来的懂无知的大学新生，变成了如今即将要毕业的老学长。这七年里，母校给我提供了良好的学习与生活环境，帮助我不断地扩充学识与见识，让我受益颇深。同时也让我遇见了一群可爱的老师和同学，在此，我要向他们表示最真诚的感谢。

首先，我要感谢我的研究生导师万继光教授。是万老师带我走入了科研的道路，万老师为人和善、教学严谨，在平时的科研与生活上为我提供了极大的帮助；是万老师让我从刚开始进入实验室遇到大型项目代码就不知所措，到现在可以独当一面，负责课题组的一个项目；在我学术生涯里，老师不断地给我指引与教导，教会了我如何分析问题、理解问题、解决问题，给予我信心去面对一个又一个科研难题。

其次，我要感谢姚婷学姐、张艺文学弟、刘志文学弟、汤陈蕾同学、伍信一学弟、程志龙同学等，在我毕设期间对我课题方案设计等提供了建议与帮助；感谢李大平学长、徐鹏学长在三年的实验室生涯中给予我的照顾与帮助；感谢文多学弟在实验室项目中给予我的协助；感谢 B408 所有的同学，在我研究生期间大家一起团结友爱，互相帮助，一起营造了一个和谐的科研氛围，为我提供了一个良好的学术环境。

然后，我要感谢我的父母，非常支持我读研的决定，在我研究生期间提供良好的生活保障，不断地给予我关心与照顾，让我可以专心地完成科研任务。感谢我的女朋友，不断地督促我，让我得以顺利地完成学术任务。在我毕业论文撰写期间，我时常会有情绪低谷，是你给我不断地鼓励与支持，让我得以调整心态，重燃信心，完成最后的学术工作。

最后，感谢所有在我学生生涯为我提供过帮助的人。

参考文献

[1] 武延军. 大数据时代已经来临——人机物融合的大数据时代. 高科技与产业化, 2013, 9(5):46-49.

[2] David Reinsel, John Gantz, John Rydning. Data Age 2025. https://www.seagate.com/cn/zh/our-story/data-age-2025/.

[3] 逄健, 刘佳. 摩尔定律发展述评. 科技管理研究, 2015(15):46-50.

[4] Garnell Heather, Kocsis Matthew, Weber Matthew. A Survey of Phase Change Memory (PCM). 2007:121-134.

[5] 冒伟,刘景宁,童薇,冯丹,李铮,周文,张双武.基于相变存储器的存储技术研究综述.计算机学报,2015,38(05):944-960.

[6] Jeremy Zawodny. Redis: Lightweight key/value store that goes the extra mile. Linux Magazine, 2009, 24(1): 79-79.

[7] Nishtala Rajesh, Fugal Hans, Grimm Steven, et al. Scaling memcache at facebook. in: Proceedings of the 10th USENIX conference on Networked Systems Design and Implementation. lombard, IL , USENIX Association, 2013: 385-398.

[8] Andy Dent. Getting Started with LevelDB. Packt Publishing Ltd, 2013.

[9] Sanjay Ghemawat, Jeff Dean. Leveldb. https://github.com/google/leveldb.

[10]Facebook. RocksDB, a persistent key-value store for fast storage enviroments. http://RocksDB.org/, 2016.

[11]赵巍胜,王昭昊,彭守仲等.STT-MRAM 存储器的研究进展.中国科学:物理学 力学 天文学,2016,46(10):70-90.

[12]宋玲.RRAM 的阻变特性研究.微处理机,2014,35(04):24-25.

[13]黄寅,徐子亮.铁电存储器技术.半导体技术,2000(03):1-4.

[14]McGrath, Dylan, Intel, Numonyx claim phase-change memory milestone, www.eetimes.com

[15]Ranganathan, Chang. (Re)Designing Data-Centric Data Centers. IEEE

Computer Society Press, 2012.

[16]Yang Jisoo , Minturn Dave , Hady Frank . When poll is better than interrupt. Usenix Conference on File & Storage Technologies. USENIX Association, 2012.

[17]Ramos Luiz, Gorbatov Eugene, Bianchini Ricardo. Page placement in hybrid memory systems. Proceedings of the 25th International Conference on Supercomputing, 2011.Tucson. DBLP, 2011.

[18]Brad Dayley . The Definitive Guide to MongoDB: The NoSQL Database for Cloud and Desktop Computing. Springer Ebooks, 2010.

[19]Xia Fei, Jiang Dejun, Xiong Ji, et al. HiKV: A Hybrid Index Key-Value Store for DRAM-NVM Memory Systems. In 2017 USENIX Annual Technical Conference, 2017

[20]Kannan Sudarsun, Bhat Nitish Gavrilovska Ada. et al. Redesigning lsms for nonvolatile memory with novelsm. In 2018 USENIX Annual Technical Conference, pages 993–1005, 2018.

[21]Lu Lanyue, Thanumalayan Sankaranarayana Pillai , Hariharan Gopalakrishnan et al. WiscKey: Separating Keys from Values in SSD-Conscious Storage. Acm Transactions on Storage, 2017, 13(1):1-28.

[22]Praveen K. Create a Persistent Memory-Aware Queue Using the Persistent Memory Development Kit (PMDK). Software.intel.com, 2018

[23]O'Neil Patrick, Cheng Edward, Gawlick Dieter, et al. The log-structured merge-tree (lsm-tree). Acta Informatica, 33(4):351–385, 1996.

[24]Intel. Intel-micron memory 3d xpoint.

[25]Arpaci-Dusseau Remzi, Arpaci-Dusseau Andrea. Operating systems: Three easy pieces, volume 151. Arpaci-Dusseau Books Wisconsin, 2014.

[26]Arulraj Joy, Pavlo Andrew. How to Build a Non-Volatile Memory Database Management System. Acm International Conference on Management of Data.

ACM, 2017.

[27]Caulfield Adrian, De Arup, Coburn Joel, et al. Moneta: A High-performance Storage Array Architecture for Next-generation, Non-volatile Memories. IEEE/ACM International Symposium on Microarchitecture. IEEE, 2010.

[28]Chen Shimin, Jin Qin. Persistent B + -trees in non-volatile main memory. Proceedings of the VLDB Endowment, 2015, 8(7):786-797.

[29]Balmau Oana, Didona Diego, Guerraoui Rachid, et al. TRIAD: creating synergies between memory, disk and log in log structured key-value stores. Proceedings of the 2017 USENIX Conference on Usenix Annual Technical Conference. USENIX Association, 2017.

[30]Kim Wook-Hee, Nam Beomseok, Park Dongil, et al. Resolving journaling of journal anomaly in android I/O: multi-version B-tree with lazy split. Proceedings of the 12th USENIX conference on File and Storage Technologies. 2014.

[31]Lakshman A. Cassandra-a decentralized structured storage system. 3rd ACM SIGOPS International Workshop on Large Scale Distributed Systems and Middleware (LADIS 09), Oct. ACM, 2009.

[32]Driskill-Smith A. Latest advances and future prospects of stt-ram. In Non-Volatile Memories Workshop, pages 11–13, 2010.

[33]Beyah Raheem, Corbett Cherita, Copeland John. The case for collaborative distributed wireless intrusion detection systems. IEEE International Conference on Granular Computing. 2006.

[34]Condit J, Nightingale E B, Frost C, et al. Better I/O through byte-addressable, persistent memory. Acm Symposium on Operating Systems Principles. DBLP, 2009.

[35]Kawahara T. Scalable Spin-Transfer Torque RAM Technology for

Normally-Off Computing. IEEE Design & Test of Computers, 2011, 28(1):52-63.

[36]Ahn, Sang Jung, Mayuram R, et al. ForestDB: A Fast Key-Value Storage System for Variable-Length String Keys. IEEE Transactions on Computers, 2016, 65(3):902-915.

[37]Dulloor Subramanya, Kumar Sanjay, Keshavamurthy Anil, et al. System software for persistent memory. European Conference on Computer Systems. ACM, 2014.

[38]Shetty Pradeep, Spillane Richard, Malpani Ravikant, et al. Building workload-independent storage with VT-trees. Usenix Conference on File & Storage Technologies. USENIX Association, 2013.

[39]Yang Jun, Wei Qingsong , Chen Cheng, et al. NV-Tree: Reducing Consistency Cost for NVM-based Single Level Systems. 13th USENIX Conference on File and Storage Technologies (FAST '15). USENIX Association, 2015.

[40]Yao Ting, Wan Jiguang, Huang Ping, et al. A light-weight compaction tree to reduce i/o amplification toward efficient key-value stores. In Proceedings of the 2017 IEEE 33th Symposium on MSST, 2017.

[41]Brian Cooper, Adam Silberstein, Erwin Tam, et al. Benchmarking cloud serving systems with YCSB. ACM symposium on Cloud computing (SoCC), 2010: 143-154.

附录 1　攻读硕士学位期间发表论文目录

[1] 詹玲,**朱承浩**,万继光.Ceph 文件系统的对象异构副本技术研究与实现.小型微型计算机系统,2017,38(09):2011-2016.

[2] 詹玲,吴畏,王方,**朱承浩**,万继光.一种基于 SSD 缓存的 RAID5/6 写优化技术研究[J].小型微型计算机系统,2018,39(10):2226-2232.