分类号_____ 学号 M201476152

学校代码 10487 密级_____

華中科技大學

硕士学位论文

基于 LSM-Tree 的键值存储引擎的优化研究

学位申请人：刘　楠

学 科 专 业：软件工程

指 导 教 师：吕泽华

答 辩 日 期：2016.12.19

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

Research on Optimization of a Key/Value Storage Engine Based on LSM-Tree

Candidate : Liu Nan

Major : Software Engineering

Supervisor : Lyu Zehua

Huazhong University of Science and Technology

Wuhan 430074, P. R. China

December, 2016

摘 要

面对数据的生成速度不断加快以及数据的组织形式日趋多样的现状，NoSQL 数据库逐步得到了市场的广泛认可和大量应用。其中，面向写性能优化的 LSM-Tree 数据结构以及基于其再开发的数据结构已经作为存储引擎的核心数据结构，并得到了广泛的应用。在 LSM-Tree 的相关研究中，如何降低写放大一直都是提升 LSM-Tree 整体性能的首要问题。

通过一系列科学严密的实验，发现了 LSM-Tree 原有合并调度机制会导致某些组件上的文件个数持续增长，从而引起写放大过大的问题。通过对该问题的进一步分析与研究，提出了基于时间片轮转的合并调度机制。时间片轮转调度机制在 LSM-Tree 原有的合并调度的基础上，增加了时间片的概念，使得 LSM-Tree 的组件在进行 Compaction 的同时，可以获取一个被称为是时间片的合并调度资源，拥有时间片的组件可以不受 LSM-Tree 原有的合并调度机制的约束，连续进行 Compaction，直至时间片的使用时间结束或该组件不再满足继续进行 Compaction 的条件。该机制通过公平分配合并资源，保证 LSM-Tree 上各层组件的文件个数增长平稳，从而缓解了由于某些组件文件个数过多而导致的写放大问题，以及由此产生的 LSM-Tree 写性能严重下降的问题。

在 RocksDB 的基础上实现了可以使用时间片轮转调度机制的 LSM-Tree 存储引擎——Slot。与 RocksDB 相比，Slot 可以在不影响读性能的前提下，将写性能提升 4 倍以上。时间片轮转调度机制是一种全新的合并调度机制，可以应用于任意基于 LSM-Tree 结构的 NoSQL 存储引擎，并大幅提升它们的写性能。

关键词：键值存储　　LSM-Tree　　写放大　　调度机制

Abstarct

In the face of the growing speed of the data and the increasingly diverse forms of data organization, NoSQL database has been widely recognized and used, with the characteristics of fast speed, good compatibility, easy to spread and so on. At the same time, the proportion of write operations is increasing, how to optimize the write speed of NoSQL database and make it provide better write performance has already become the focus topic in the field of data storage systems. Among all the popular KEY/VALUE storage models, the write-optimized data structure LSM-Tree (Log-Structured Merge-tree) and its variants becomes the core data structure and has been widely used in the mainstream KEY/VALUE storage databases. However, due to the defects of the compaction scheduling mechanism of LSM-Tree data structure, there is a serious problem, named write amplification, in the running process of the system, which greatly affects the write performance of LSM-Tree and throughout of overall system.

A optimization is found to solve the defects of the compaction scheduling mechanism of LSM-Tree data structure through a series of scientific and rigorous experiments, which is called the Time Slice Rotational Scheduling Mechanism (TSRS) and developed based on RocksDB. Through a series of experiments, it is proved that this optimization can make sure that the file number on each component of LSM-Tree grows smoothly through fair compaction and decrease write amplification caused by the excessive number of file on the component, and then improve the write performance of LSM-Tree.

TSRS has fundamentally sloved the problem of excessive write amplification of LSM-Tree. Compared with RocksDB, the write performance improvement can reach more than 4 times. TSRS can be applied to any NoSQL storage engine based on LSM-Tree.

Key words： Key-Value store　LSM-Tree　Write amplification　Scheduling mechanism

目 录

1 绪论

1.1 课题研究的背景及意义

随着通信技术及网络技术的发展与进步，中国提出了“互联网+”：通过使用移动智能设备，人们可以访问各大媒体门户网站、登录各种社交软件等方式从而快速获取最新资讯；通过网络商城就能便利购买完成各种日常商品；日益发展的快递、物流企业通过互联网跟踪统计并完成包裹的打包、分类、运输和派送等流程；平时生活中可以通过网上支付完成支付操作。鉴于通讯技术与网络平台连接方式在各个行业的普及应用，“互联网+”已经逐步成为社会和各个企业的常态，这也标志着中国经济的全互联网化[1]。

大数据（Big Data），就是随着各种 IT 设备以及通信技术发展而产生的新概念[2]。简单来说，大数据就是指无法在一定时间范围内用常规软件工具进行捕捉、管理和处理的数据集合[40,41]。同时，也是需要具备更强的决策力、探索力以及流程优化力的新型处理方式来处理这些海量、高增长率和多元化的信息资源[34]。IBM 将大数据特性总结为 5V，分别：Volume、Velocity、Variety、Value 和 Veracity[39]。

（1）Volume 表示的是数据量大。当前形势下，各企业所存储的数据集已经从 TB 级增长到了 PB 级，甚至已经到达了 EB 和 ZB。根据 IDC（Internet Data Center）的调研报告，2015 年全球产生的数据总量已经达到了 7.7ZB，预计到 2020 年全球数据总量将达到 40ZB[38]。数据的容量决定了待分析数据的价值及其隐含的信息。

（2）Velocity 表示生成速度快。首先是网络速度大幅度提升，以中国为例，在 2015 年 05 月 16 日，国务院办公厅发布了《关于加快高速带宽网络建设推进网络提速降费的指导意见中》，并在其中明确提出“提速带宽”的要求，同时，中国已大规模推广 4G 通讯服务，4G 时代的飞速发展使得数据流量急速增长[3]。另一方面，越来越多的行业与互联网络对接，更多的用户习惯进行网上购买和网上支付，2016 年，天猫商城在双十一当天的成交量已经到达了 1207 亿元，与 2015 年相比增加了 32.6%，由此可以说，选择在网上购物的用户数量正在快速增长。

（3）Variety 表示数据形式多种多样。一方面，随着越来越多的企业与互联网络

对接，因此所生成的数据在结构和形式上各不相同；另一方面，由于传感器在人们的生活中不断推广和普及（如：智能家电、智能手机、智能手表等），由它们所产生的数据格式类型也很多（如：图像、文本、音频、视频等）。据 IDC 的一项调查报告中指出：企业中 80%的数据都是非结构化数据，而且，这些数据每年都按 60%的速度在不停增长[5]。

（4）Value 表示数据价值。“大数据”中潜在的巨大使用价值已经在实际生活中起到了很大作用。例如，通过分析大量的信息访问数据，企业或者应用工具就能知道用户的喜好、阅读习惯、年龄段等信息，根据数据挖掘得到的这些信息，应用软件会定期向用户推荐其所关心的资讯，从而提高用户体验[4]。

（5）Veracity 表示的是数据的真实性。Veracity 这个词由在 Express Scripts 担任首席数据官(Chief Data Officer, CDO)的 Inderpal Bhandar 在波士顿大数据创新高峰会(Big Data Innovation Summit)的演讲中提出，他认为应该将真实性纳入大数据分析中，判断并筛除资料中有误差、伪造、异常的数据，防止脏数据损害到资料系统的完整性和正确性，从而避免其影响最终决策[6]。

面对当前形势下的数据量庞大、数据结构多样、数据产生速度快、数据潜在的价值以及数据的真实性等特点，各种相关技术面临着多重挑战：存储、分析、能耗、隐私、安全、易用性等[7]。数据的高效能存储是后期进行数据分析处理的基础，而数据挖掘的目的就是发掘出其中潜在的使用价值[45]。为了能有满足大数据的数据庞大、结构多种多样、数据产生快等多重特性，NoSQL[36]数据库逐渐被认可且迅速发展。

键值数据库是当前应用十分广泛的分布式 NoSQL 数据库之一，例如：BigTable[8]、PNUTS[9]、Dynamo[10]、Cassandra[11]、HBase[44]、HyperDex[12]等，其特点就是采用键值对来存储数据，优势在于容易部署和简单性。除此之外，应用广泛的分布式 NoSQL 数据库还有基于文件存储的 MongoDB[43]，同时具备图结构以及文档结构的数据库 OrientDB[42]，以及图形数据库 Neo4j[25]。与传统 SQL 数据库以及和其他类型的 NoSQL 数据库相比，键值对数据库具有较高的可扩展性以及时空代价低等优势[13]。其中，BigTable、HBase 这两个数据库虽然在逻辑结构是针对列存储的，但是实际上底层的存储模式仍然是键值对存储。

另外，根据一项研究表明：针对数据的写操作比例正在逐渐上升，同时增长速度持续增加。在 2010 年前后用户对数据的读操作比例是 80%-90%，而到了 2012 年对数据的读操作比例下降到了 50%左右。不同的系统将用户的所有点击操作或者智能移动设备的访问等操作进行连续不间断地记录并形成日志文档，数据分析师则以不同方式对日志数据进行发掘、分析（例如：长时间扫描或者定点查询等）。在实际使用中，绝大多数的数据读操作都会在多层缓存系统中被找到或命中[27]，例如：web 浏览器的 Cache，CDN（Content Delivery Network，内容分发网络），Redis[15]以及各种操作系统的缓存等等，可是更新的数据必须被存储到存储介质上，目的是确保数据的持久化。所以优化多级缓存系统就被用来提升系统的读操作性能，基于写优化的数据结构就被更普遍的使用到了存储系统当中，提升系统的写操作性能。按照 Facebook 对本公司的图片数据缓存系统的分析发现：大约只有 9.9%的数据是通过存储引擎直接在磁盘上读取获得的，而剩余的 90.1%的数据会在多级的数据缓存系统中被命中[16]。因此，存储系统写性能优化已经逐渐成为存储系统性能优化的主要问题。

对于一个数据存储系统来说，存储引擎的性能高低对整个系统性能都起到了关键作用。文章主要侧重于当前流行的 NoSQL 存储引擎，针对其存在的写放大问题进行优化研究，进而提升存储系统的整体性能。

1.2 相关研究概况

近年来，优化键值存储引擎是存储系统领域的研究热点。科技的持续发展和进步，使得存储设备也在不断地更替创新，主要表现为存储容量增大、速度不断加快。由于存储设备的快速更新，对于如何有效利用新设备来提升系统性能等相关研究也在不断增加，例如：针对 SSD（Solid State Drives，又称固态硬盘）优化的 LSM-SDF[17] (Log Structured Merge-Software Defined Flash)，针对闪存优化的 FlashStore[18]、SkimpyStash[19]等。与此同时，针对存储引擎的核心数据结构优化等相关研究也再持续增加；其中，最为热门的研究对象就是 LSM-Tree[20]（Log Structured Merge Tree，又称日志结构合并树）。

LSM-Tree 数据结构体分为两部分，内存组件和磁盘组件；其中，内存组件只有

一个，而磁盘组件则是多个（或者说是多层）。前端数据通过写请求传给 LSM-Tree 后，首先会被写入到内存组件中，当内存组件的文件量达到其阈值时，内存组件会将这些数据批量刷写入磁盘组件中。数据被持久化到磁盘之后会进行数据合并操作，目的是控制磁盘组件上的数据量和数据有序性，同时提升系统的读性能。下面将具体介绍一些基于 LSM-Tree 实现的经典系统以及针对其进行的优化与改进方法。

LevelDB[35]。它是 LSM-Tree 数据结构的经典实现，该存储系统是 Google 开源的一个键值对存储引擎库。RocksDB 是由 Facebook 开源的键值对存储引擎库，它是基于 LevelDB 并在其基础上做了一些改进和优化，最主要的就是改变了处理数据的 Compaction 操作的方式，使得 Compaction 从单线程操作变成了多线程操作，并且还同时支持列存储。

bLSM[14]。bLSM 的主要是将 B-Tree 和 LSM-Tree 的特性进行结合以此提升系统性能。bLSM 基于 LSM-Tree 同时在其内存组件中加入 B-Tree 结构来组织数据，并且更换了后台进行数据 Compaction 的选择算法，从而减低了数据 Compaction 的进行频率，提高了 LSM-Tree 的整体性能。

LSM-Trie[21]。LSM-Trie 的主要思想是通过使用前缀树来存储 LSM-Tree 的元数据信息，从而降低 LSM-Tree 在进行读操作时的 I/O 次数，进而提升了系统的读写性能。

PCP[22](Pipelined Compaction Procedure，又称流水线合并操作)。PCP 的主要思想是采用流水线的数据合并操作，利用底层硬件设备的并行操作来加速数据合并过程，同时，避免不必要的数据移动，最终达到减少数据 I/O，提升系统性能的目的。

VT-Tree[23](Variable Transmission Tree)。VT-Tree 的主要思想是通过缝合技术（stitching technique）来减少数据的无效的移动和排序时无效的磁盘 I/O，以及减少并发症的频率。但缝合技术可能产生碎片，这些碎片会降低数据扫描和合并的效率；

LSM-SDF。LSM-SDF 的主要思想是通过对 SSD 的多通道特点进行改善，进而保证 LSM-Tree 能更好的体现 SSD 在存储方面的高效性能，从而进一步提升 LSM-Tree 在 SSD 上的读写性能。

大多数研究热点都在如何降低数据合并的频率，提高数据合并的速率以及控制

在热数据上的数据合并等，而忽视了对 LSM-Tree 数据结构的写放大（write Amplification）[24]问题的关注和优化。

除了上述所提到的各种研究之外，更多的是利用 LSM-Tree 进行元数据管理优化，其中包括：Tair（The Arabidopsis Information Resource)、Atlas[26]、SSDB、BabuDB[28]、TableFS[29]等。这主要就是利用了 LSM-Tree 能够高效地处理小对象数据这一特性。例如，淘宝基于 LevelDB 开发了一款高性能键值存储引擎——Tair，中国人自己开发的 SSDB 是一款具有高性能的开源的 NoSQL 数据库，并被奇虎 360 用于替换原来使用的 Redis。SSDB 支持丰富的数据结构，并且具有很高的性能。由于 SSDB 可以持久化数据存储，所以其数据存储的总量要比 Redis 大得多。

1.3 主要研究内容

LSM-Tree 数据结构的主要思想是批量处理和延迟操作，为了保障系统的读写性能，以及磁盘组件上的数据量和数据有序性，需要在后台进行数据 Compaction 操作。当系统进行数据 Compaction 时，需要将之前的数据从磁盘中读出，处理并排序后再将数据重新写会磁盘中，这就带来了严重的问题，即写放大问题。写放大（Ratio of Write Amplification，简称 RWA）的定义为：

$$Ratio\ of\ Write\ Amplification = \frac{Input\ of\ C_i + Input\ of\ C_{i+1}}{Input\ of\ C_i} \qquad (1\text{-}1)$$

写放大代表了系统进行的所有 I/O 中无效 I/O 的占比。写放大越大，无效 I/O 的占比就越大。那么，当系统处于不断写入数据状态时，后台的数据 Compaction 操作会与写入操作产生冲突，此时系统会停止响应前端的写入操作，全力进行后台的数据 Compaction 操作，即系统处于写等待状态。此时数据迁移的总 I/O 量决定了写等待的时延长度，而写放大则决定了迁移相同数据量时需要消耗的总 I/O 量，因此，写放大越大，迁移相同数据量所消耗的时间会越长，进而系统等待时间也就越长，导致系统整体性能降低。

本文主要对 LSM-Tree 本身存在的写放大问题进行优化研究，具体的工作内容如下：对 LSM-Tree 进行分析和优化，提出了基于时间片轮转的合并调度机制，该机制

可以更准确地判断和选择需要合并的组件，通过公平地分配合并资源，从而有效调节各层组件上的数据量，避免由于某些组件上的数据量过大造成的写放大问题，进而有效提升系统的写性能。

1.4 论文的结构

通过六个章节阐述相关研究工作，每个章节的相关内容如下：

第一章为绪论。本章节首先介绍了科技的不断进步、NoSQL 数据库逐渐被认可和广泛使用以及随着各个行业不断接入互联网络所带来的“大数据”存储问题等方面介绍了本课题的研究背景。随后系统阐述了当前研究领域，针对 LSM-Tree 数据结构的研究热点以及研究状况。最后说明了论文的工作内容。

第二章为相关技术介绍。本章节先以 LevelDB 键值存储引擎为例，详细地介绍了 LSM-Tree 的主要思想与整体架构。接着重点介绍了 LSM-Tree 的合并机制。最后介绍了在本文测试部分所使用的主要测试工具——YCSB。

第三章为 LSM-Tree 的写放大问题分析。本章首先分析并介绍了 LSM-Tree 的 Compaction 调度机制，接着介绍该调度机制造成了 LSM-Tree 的 C_2 层组件上文件个数严重滞留以及带来 C_1 组件的写放大严重，最后通过各项实验验证了该调度机制对 C_1 组件以及对系统整体性能的影响。

第四章为时间片轮转调度机制的实现。本章首先提出了时间片轮转调度机制并且结合运行过程中的几个状态详细阐述了该机制的主要流程以及主要思想，最后详细介绍了在 Facebook 的开源存储引擎 RocksDB 实现时间片轮转调度机制，包括实现的基础类和方法。

第五章为优化方案测试及对比分析。本章节首先介绍了测试的软硬件环境、测试的核心思想以及测试过程中的主要参数设置。并且对优化方案进行了参数选择测试、性能对比测试，同时对测试结果进行分析说明。

第六章为总结与展望。本章节系统地总结了研究工作和优化成果，同时指出该优化方案未来的优化改进方向。

2 键值存储相关技术介绍

在众多键值存储系统中，基于 LSM-Tree 结构的键值存储系统最为流行，同时因 LevelDB 是 LSM-Tree 的经典实现而被广泛应用。本章选择 LevelDB 系统，详细介绍 LSM-Tree 数据结构的整体架构。

2.1 LSM-Tree 以及 LevelDB 简介

LSM-Tree，又叫做日志结构合并树，由 Patrick O'Neil 等人在 1996 年提出。该数据结构现已被广泛使用在数据管理系统中，如 BigTable、HBase、LevelDB 等。

2.1.1 核心思想与整体结构

当前的数据存储系统在更新数据方面通常使用两种方式进行操作，即 In-Place Update 与 Out-of-Place Update。In-Place Update 又称为原地更新方式，对数据的修改和删除操作都会即时生效， 例如：B-Tree 结构和其变体最能体现该思想，数据结构的实现可以参考 Berkeley DB[30]；Out-of-Place Update 又称为延迟更新，对数据的修改和删除操作不是即时生效，而是以标签的形式写入到系统中，并在后续操作中进行数据删除或者版本合并等。与 In-Place Update 相比，Out-of-Place Update 因写时延更小，同时缩短了数据写入时中对旧版本的查找时间等特点，使用于写时延敏感的操作。

LSM-Tree 就是一种典型的延迟更新数据结构。用户对 LSM-Tree 进行的写入、删除、更新等操作都先保存在其内存组件中，当内存组件的数据量到达规定的阈值后，系统会将数据以及对数据的相关操作一起从内存组件移到磁盘组件上，然后通过一个叫做 Compaction 的后台线程对磁盘组件不断地进行 merge（合并）和 sort（排序）操作，目的是重新组织这些数据，以保持 LSM-Tree 磁盘组件的数据在各层组件上均匀分布。其中，合并操作将判断同一个键的不同版本值是否还应该继续保存，这是因为 LSM-Tree 的延迟操作会使得同一个键的不同版本值都被保存，这时系统就要将最新的数据保留同时删除旧数据；排序操作根据规定的键的顺序进行排序，以

此来确保磁盘上各个组件的所有数据具有一致的有序性。

通过上述的工作机制，将实现以下效果：首先，把内存数据批量地写入磁盘组件，缩短来自前台的每次操作所需的查询时间，使系统能够立即响应前台的各种数据操作请求，降低了立即更新以及立即排序而引起的写延迟；其次，对前台应用插入的数据进行合并和排序操作，降低前台应用进行查找、读取等操作的延时，同时避免过期版本的数据占用磁盘组件的空间。

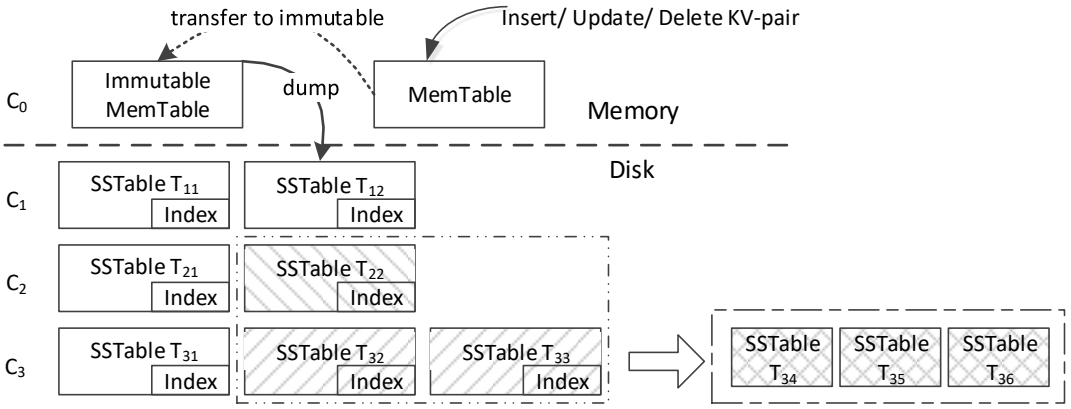


图 2-1　LevelDB 架构图

图 2-1 是 LevelDB 架构图。如图所示，LevelDB 的内存组件（又称 C_0 组件）有 MemTable 和 Immutable MemTable 两部分；LevelDB 磁盘组件是层级结构，一共有 7 层组件，图中使用 C_i（i=1，2，...，7）进行表示。

如图 2-1 所示，首先，所有数据以及对数据的操作（包括修改和删除等）都保存在 C_0 组件的 MemTable 中，由于 MemTable 使用的是 SkipList，因此数据在内存中是有序的。当 MemTable 的数据量到达系统阈值时，MemTable 就会自动转化为 Immutable MemTable，此时 MemTable（即 Immutable MemTable）将成为只读状态，不能接接收新的数据和数据操作。Immutable MemTable 上的数据会通过系统 dump 操作，转变成一个 SSTable 文件存储在磁盘组件上。SSTable 文件是有序的，其数据根据 key 从小到大有序存储，当 SSTable 产生后就不会再接受任何的数据修改操作，只有以查询方式的读操作。LevelDB 的磁盘组件是由多层组件组成，其中，各层组件都是由多个 SSTable 组成。因每个 SSTable 存储的数据是有序的，所以能使用

SSTable 文件中的 Key 的最小值、最大值来标记 SSTable 的键范围（key-range）。LevelDB 把磁盘组件上的每一个 SSTable 的 key-range 保存在内存里，在进行读操作时，先利用内存中缓存的 SSTable 的 key-range 对读操作进行筛选，当 SSTable 的 key-range 中包含读请求的 Value 时，再进一步去磁盘上读取对应的 SSTable 文件内容。需要特殊说明的是磁盘上的 C_1 组件，因为 C_1 组件上的 SSTable 是由 C_0 组件直接通过 dump 方式生成的，因此 C_1 组件上的 SSTable 文件之间可能存在 key-range 重叠的情况。

通过前端应用对 LevelDB 进行读操作时，利用 SSTable 缓存在内存中的 key-range 筛选出需要读的 SSTable 文件集合。因 SSTable 文件存储在磁盘组件上，读取 SSTable 文件时要对磁盘进行访问操作。为了可以有效地提升读操作的性能，LevelDB 限制了 C_1 组件上 SSTable 文件的数量，同时保证 C_i（i=2, …, n）组件上各个 SSTable 不存在 key-range 重复。LevelDB 每隔一定时间会对 C_1 组件上的 SSTable 文件进行合并和排序操作，同时将重组后的数据写入到 C_2 组件。当数据从 C_1 组件向 C_2 组件上移动时，为了确保迁移到 C_2 组件上的 SSTable 和 C_2 组件原有的 SSTable 文件之间的 key-range 不重叠，需要读取 C_2 组件中覆盖 C_1 组件 key-range 的所有 SSTable 文件与 C_1 组件上的 SSTable 文件进行合并和排序操作。当新的 SSTable 产生后写入到 C_2 组件上同时将之前从 C_2 组件读出的 SSTable 文件删除。在 LevelDB 中，将这个在相邻两个组件之间进行 merge、sort 操作的过程称之为 Compaction 操作。

每一次的 Compaction 操作的完成时间应当限制在一定范围内，长时间的 Compaction 会降低整个系统的性能。单次 Compaction 的时间会随着数据量的增大而延长。由于 C_1 组件上的 SSTable 存在 key-range 重叠，当 C_1 进行 Compaction 操作时，会出现 C_1 组件的 key-range 覆盖了 C_2 组件上所有的 SSTable 文件，使得参与 Compaction 操作的数据量过大，那么就应当通过控制 C_2 组件上整体数据量来限制参与 Compaction 操作的数据量。因此，需要将 C_2 组件上的数据移动到 C_3 组件上，以此类推产生了 C_i（i=4,5,6,7）组件。这就形成了 LSM-Tree 磁盘组件的层级结构。

LevelDB 通过两种方式提供高效的写性能：第一，将数据写入到内存组件中的 MemTable 上来缩短写延时；第二，将 Immutable MemTable 直接 dump 到磁盘来减少

因 Immutable MemTable 导致的写等待。通过两种方式提供高效的读性能：第一，控制 C_1 组件上的 SSTable 文件数量；第二，保持 C_i（i>=2）组件上 SSTable 文件内部有序以及整体有序。LevelDB 通过设定相邻两组之间的数据容量比为 10（例如：C_2 组件的阈值设置为 10MB，那么 C_3 组件的阈值就为 100MB，由此类推，C_7 组件的阈值为 1TB），来确保系统可以存储大量数据。

LSM-Tree 数据结构的主要特点是层级组件，这样设计的主要目的是减少系统的写延迟。其中，多层组件指的是多个磁盘组件以及一个内存组件。图 2-1 为 LSM-Tree 数据结构的架构图。通过数据连续进行上层组件向下层组件上迁移来确保 LSM-Tree 具有高效的读、写性能。当某个组件的数据容量大小超出阈值时，该组件就会触发 Compaction 操作。每次参与 Compaction 操作的为相邻两层组件，Compaction 过程中将两个组件上参与 Compaction 操作的数据进行排序，并删除无效的和旧版本的数据，以确保数据的有序性和准确性。

LSM-Tree 通过 Round-Robin 方式在进行 Compaction 操作的组件上选取一个 SSTable 文件参与 Compaction。当通过 Round-Robin 方式选取了一个 SSTable 文件时，以该文件的 key-range 作为相邻下一层组件上选取 SSTable 文件的 key-range。选取该文件以及下层组件上包含该 key-range 的所有 SSTable 文件作为输入进行 Compaction，Compaction 的结果再写回到下层组件上，形成新的 SSTable，而作为输入参与 Compaction 的所有 SSTable 将被删除。

如图 2-1，当 C_2 组件上的数据大于其阈值，那么在 C_2 组件上选取一个 SSTable T_{22}，从 C_3 组件上选取与 C_2 组件上的 T_{22} 在 key-range 存在重叠的 SSTable T_{32} 和 T_{33}，将选择的 T_{22}、T_{32}、T_{33} 这三个文件重新合并以及排序产生三个新的 SSTable 文件，称之为 T_{34}、T_{35}、T_{36}；将这三个产生的 SSTable 文件写到 C_3 组件上。通过这种方式，实现了将数据 T_{22} 从 C_2 组件迁移至 C_3 组件上。

2.1.2 LSM-Tree 的基本操作

基于 LSM-Tree 数据结构的数据存储系统对外提供了 5 个基本的数据操作，如下所示：

（1）数据写入操作（Insert）：向数据库中写入一条 Key/Value 记录。该条记录

首先被存入内存组件（即就是 C_0 组件）中，当数据总量超过内存组件的数据总量阈值时，内存组件触发 Compaction 操作将数据迁移到磁盘组件上。

（2）数据更新操作(Update)： 在 LSM-Tree 数据结构并没提供直接进行数据更新操作的接口，该数据结构采用了延迟更新操作，通过直接写入 Key 相同的记录，在后续的 Compaction 过程中进行数据更新；或者先通过删除操作删除旧的记录，在写入新的记录完成数据更新操作。

（3）数据删除操作(Delete)：在 LSM-Tree 中，不做立即删除，同样以写入的方式对待删除的数据添加删除标记，在后续的 Compaction 过程中进行真正的数据删除操作。需要说明的是，如果待删除的数据正在被读取或被占用时是不能被删除的。LSM-Tree 是通过 Version 标记对读取或者解除占用操作进行相应的计数加减，当 Version 的计数等于 0 时，那么相应的数据就可以被删除了。

（4）数据查询操作(Get)：对一个精确查询或者区间查询操作，系统查找的原则是，第一步会去内存组件中匹配查询条件，如果找到则返回对应结果；否则接着对磁盘组件进行从上到下顺序查找，直到在某层组件上到与查询条件匹配的结果并返回，或者找到 C_7 层组件查询失败。

（5）数据扫描操作(Scan)：在 LSM-Tree 中，通过既定的顺序（从小到大或者从大到小的顺序或者随机等）对待扫描的 Key-Value 所有版本进行扫描，并找到其中最新版本记录。

2.1.3 逐层流动机制

图 2-2 记录了键值对从上层组件向下层组件流动过程中的数据读写情况。在整个数据流动过程中，即使是在同一层组件上，也需要将一个键值对进行多次地读出、写入操作。这是因为在 LSM-Tree 数据结构中，数据的 Compaction 操作是在两个相邻组件间进行，同时因为上层组件触发数据 Compaction 操作的次数远远大于下层组件触发数据 Compaction 操作的次数。上层组件上的数据向下层组件移动之前就已经多次参与了前一层组件上的 Compaction 操作。这也就是导致 LSM-Tree 的写放大的原因之一。

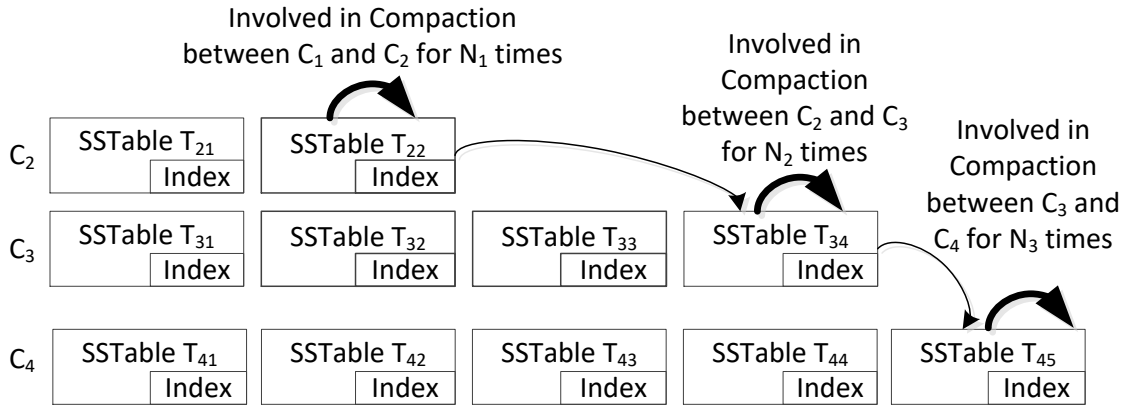


图 2-2　LSM-Tree 中键值对的迁移过程

2.2　YCSB

YCSB（Yahoo! Cloud Serving Benchmark），是一款基础测试工具，是 Yahoo 公司专门针对云服务基础测试而开发的[33]。这款测试工具的主要目的是针对云存储服务平台进行一些基本的测试，同时可以提供统一的测试性能比较。该测试工具为多个分布式系统定制了直接进行测试接口和结果报告，比如：Cassandra、HBase、PNUTS 等。除了分布式存储系统之外，YCSB 还对单机版的数据或者存储引擎提供了一个名为 MapKeeper 的测试接口，目前 MySQL、LevelDB、LMDB[37]（Lightning Memory-Mapped Database Manager）、Berkeley DB 等系统中都已经接入 MapKeeper 测试接口。

YCSB 在进行测试时需要与数据库进行通信完成测试工作。在测试过程中，存储系统充当服务端，YCSB 是存储系统的客户端。图 2-3 为 YCSB 的架构图。

根据图 2-3 所示，可以看出 YCSB Client 的内部结构分为：Workload Executor、Client Threads、Stats、DB Interface Layer 四部分。YCSB Client 接受外部参数的方式有两种：一种是 Workload file，另一种为命令行参数。

Workload Executor 将 Workload file 定义的参数进行解析，然后生成对应的测试数据；Client Threads 的主要工作是负责管理 YCSB Client 同时进行多个云存储系统访问的线程；Stats 主要工作则是对测试过程中云存储系统的性能数据进行统计。DB Interface Layer 是 YCSB Client 对云存储系统进行访问的统一接口。

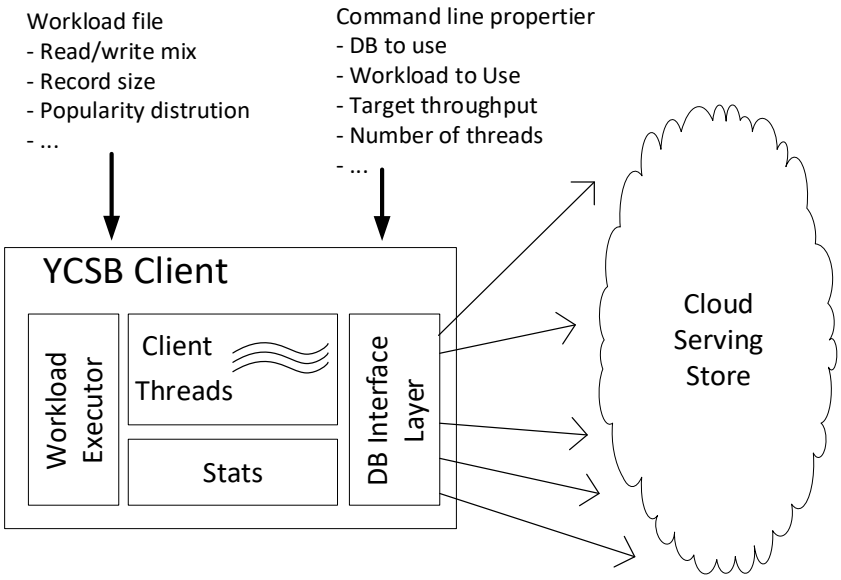


图 2-3　YCSB 架构图

Workload file 主要定义了 YCSB 生成数据的几种方式，例如：每条记录长度、读写比例、数据的分布方式等等；命令行参数则是在 YCSB Client 的运行过程中给 YCSB Client 提供参数，其中包括待测试对象、Workload file 的位置、需要启用的线程个数、以及湿度将测试进度反馈到控制台等等。

2.3　本章小结

本章节首先详细介绍了十分流行的 LSM-Tree 数据结构的核心思想与整体结构；然后以 Google 公司开源的键值存储引擎 LevelDB 为例，全面介绍了 LSM-Tree 数据结构的具体实现，以及 Compaction 操作过程；最后介绍了存储系统测试工具 YCSB。

3 LSM-Tree 的写放大问题分析

本章主要通过详细分析 LSM-Tree 的合并调度机制，指出该机制中存在的问题，并通过一系列的测试，逐步验证了该问题对系统性能的影响。

3.1 LSM-Tree 写放大问题分析背景

随着现代互联网络的高速发展，互联网络数据的生成速度也日渐加快，用户进行网络写请求比例也越来越大；因此，后台数据库来源于前台应用对数据访问的延时的要求也越来越严格。在这样的大数据环境下，数据存储服务商为了提供更低时延的写操作服务，放弃了传统的数据库事务，而选择使用能更好适应数据多样性的键值存储系统。当前的键值存储系统，越来越注重数据一致性、系统服务能力以及系统可靠性。面对这种需求，如何提高系统的写性能，已经成为键值存储系统优化的主要方向。

LSM-Tree 数据结构主要是面向于写优化，该数据结构的主要思想是延迟操作和批量处理。LSM-Tree 数据结构典型的实现是 LevelDB 键值存储系统引擎，因其在存储方面表现出来的高性能而被广泛应用在生产实践中。图 2-1 是 LevelDB 整体结构图。LevelDB 为了实现延迟操作，把对系统的插入、删除、更新等操作都转换为对系统的插入操作，用操作类型（Action Type）来区别不同的操作。为了确保系统的读写性能，通过后台线程对数据进行合并与排序处理，即就是前文提到的 Compaction。由于本文主要是针对 Compaction 的调度机制进行优化，为了便于理解，首先重点介绍 Compaction 的调度机制。

在 LevelDB 中，Compaction 是进行数据持久化，以及整理磁盘数据分布的重要操作。

触发 Compaction 操作的条件主要有以下几点：第一，当 C_0 组件达到阈值时，触发 Compaction 操作，将内存数据转化为 SSTable 文件并写入到 C_1 组件上；第二，当磁盘上某一个组件的数据总量达到或超过阈值时，触发 Compaction 操作，将该组件上的部分数据移动到下一层组件中；第三，当磁盘上处于较上层组件的某个 SSTable

文件被多次访问且没有找到有效数据时，可以认为该 SSTable 文件处在不优的状态，而进行 Compaction 操作后会倾向于均衡状态，所以主动触发 Compaction 操作，将该 SSTable 向下层移动；第四，用户主动调用 Compaction 函数，触发 Compaction。

在同一时间如果有多个组件均满足上述条件，则需要启用 Compaction 的调度机制对这些组件进行优先度计算并排序，当前优先度最高的组件进行 Compaction 操作；因为优先度越高，表示该层组件越不均衡。

在 LSM-Tree 中，各个组件的优先度计算原则主要有以下三种情况：

（1）首先，C_0 组件进行 Compaction 的优先度最高，无论什么时候，一旦 C_0 组件需要进行 Compaction 操作，系统就会立刻暂停当前正在进行的其它 Compaction 操作，直到 C_0 组件的 Compaction 操作执行完毕，再恢复之前的 Compaction，否则 C_0 组件的数据量达到其阈值后，将暂停外部应用的写请求，降低系统性能；

（2）其次，除 C_1 组件比较特殊之外，其他各层组件的优先度均可通过如下公式计算得出：优先度=该层当前总数据量/该层数据阈值；

（3）最后，C_1 组件的优先度计算较为特殊，由于 C_1 组件是磁盘上的第一个组件，直接与内存组件 C_0 进行数据交互，加之 C_0 组件进行 Compaction 操作的优先度最高，使得 C_1 组件的文件个数增长速度快于其他各层组件，如果不加以限制，则可能导致 C_1 组件上文件个数太多，进而导致 key-range 重叠的文件个数增加，使得过多的文件参与 Compaction，最终造成 Compaction 时间过长，降低系统性能。为了避免因为过量的写操作降低系统性能，因此 LSM-Tree 在 C_1 组件上设置有三个阈值来限制写操作，分别为 N_1，N_2 和 N_3（$N_1<N_2<N_3$）：当 C_1 组件上的数据量达到 N_1 时，C_1 组件就可以触发 Compaction 了，此时 C_1 组件的优先度计算通过前面提到的公式得出；当 C_1 组件上的数据量达到 N_2 时，系统会主动提高 C_1 组件自身的 Compaction 优先度，此时 C_1 组件的优先度将被直接赋给一个较大的值，默认为 10000，同时减缓 C_0 组件的数据写入速度；当 C_1 组件上的数据量达到 N_3 时，系统会极大地提高 C_1 组件的 Compaction 优先度，此时 C_1 组件的优先度将被直接赋给一个更大的值，默认为 100000，同时暂停 C_0 组件的数据写入操作，直至 C_1 组件上的数据量小于 N_3。其中，N_1 表示开始进行 Compaction 的阈值；N_2 表示限制写操作的阈值；N_3 表示暂停写操作的阈值。

3.2 C_1 的写放大问题与分析

在 LevelDB 中，为了保证尽快地将数据从内存迁移到磁盘上，C_0 组件和 C_1 组件进行 Compaction 操作时，C_0 组件只是将一个 Immutable Memtable 中的数据转化成一个 SSTable 文件同时写入到 C_1 组件中，并不与 C_1 组件上的其他 SSTable 文件进行合并，这就导致了 C_1 组件上的多个 SSTable 文件之间有可能存在 key-range 重叠。因此，当 C_1 组件进行 Compaction 时，不能简单地从 C_1 组件上选择一个 SSTable 文件，还需要检查被选择的 SSTable 是否与 C_1 组件上其他 SSTable 文件存在 key-range 重叠，如果有重叠，则要将有重叠的 SSTable 文件全部选上，然后再从 C_2 组件中选择 SSTable 文件进行 Compaction。由于 C_1 组件上选择的 SSTable 文件个数较多，覆盖的 key-range 相对较大，那么，一旦 C_2 组件上的 SSTable 文件个数过多，就会导致从 C_2 组件上选择过多的 SSTable 文件参与 Compaction 操作，从而导致此次 Compaction 的写放大很大。写放大表示一次 Compaction 操作的有效率，其计算公式为：写放大=写操作 I/O 总量 / 有效 I/O 总量，可以看出，写放大越大，表明此次操作的无效 I/O 比例越大，则效率就越低。为了避免在 Compaction 操作中，因为从 C_2 组件上选择太多的 SSTable 文件而导致的写放大过大问题，LevelDB 将 C_2 组件的默认阈值设置的很小，这个值与 C_1 组件的阈值几乎相同，而不同于其他层组件那样是上层组件的 10 倍。

表 3-1　LevelDB 各层组件文件个数阈值

组件	C_2	C_3	C_4
阈值	5	50	500

然而，在实际测试中发现，C2 组件的阈值并没能有效地限制 C2 上的文件个数。图 3-1 是 LevelDB 写入 5GB 数据时 C2、C3 和 C4 层组件的文件个数变化曲线图，表 3-1 是 LevelDB 的各层组件上文件个数的阈值。事实上，在 LevelDB 中各层组件的阈值是用文件大小表示的，而并非文件个数，但由于每个 SSTable 文件大小固定为 2MB，因此为了方便起见，本文中将组件阈值换算成了文件个数。

从图 3-1 中可以看到，在测试开始的几秒内，C2 组件上的文件个数便已经达到了阈值，并且在 insert 过程结束之前一直呈上升趋势，最高达到约 1500 个，大约是

其阈值的 300 倍。

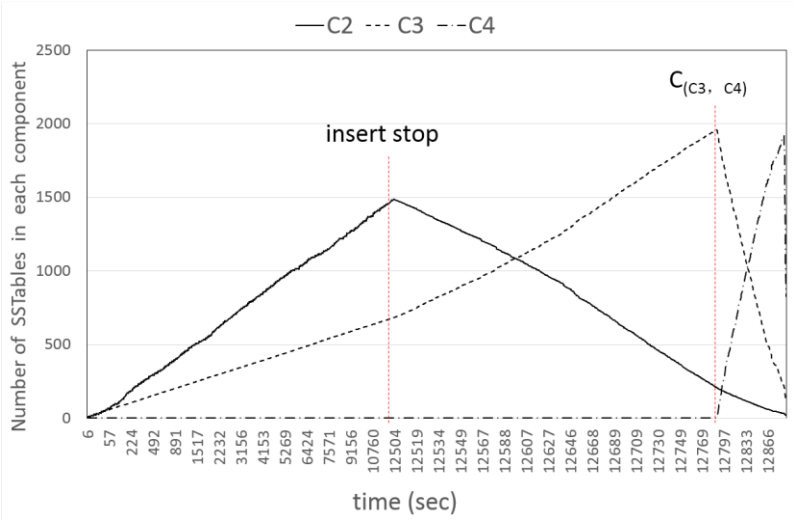


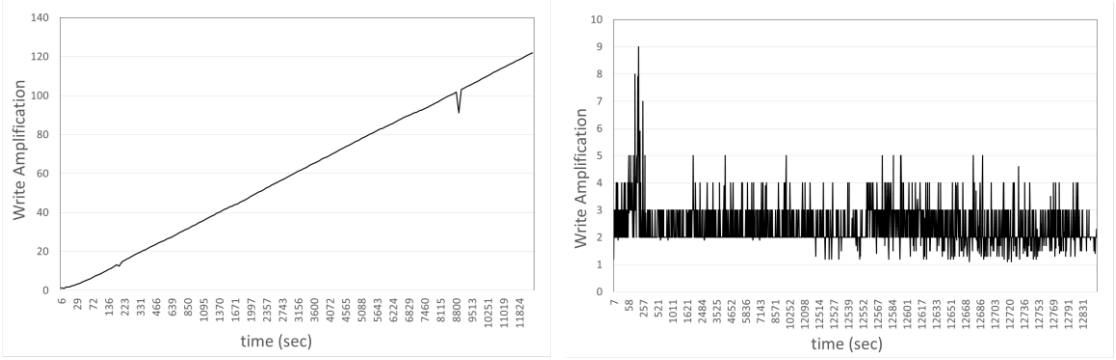
图 3-1　LevelDB 各层文件个数变化趋势

造成上述现象的原因主要有以下三点：第一，C_0 是内存组件，数据写入速度快，达到阈值所用时间短，同时 C_0 组件的 Compaction 优先度最高，并且又是直接进行 dump 操作，因此数据从 C_0 迁移至 C_1 组件上的速度很快；第二，受 C_0 组件影响，C_1 的文件个数增长很快，很容易到达 N_2 阈值，一旦 C_1 的文件个数达到 N_2，C_1 的 Compaction 优先度就会超过 C_2，从而优先进行 C_1 组件的 Compaction 操作；第三，C_1 组件的一次 Compaction 操作会将 C_1 上的多个 SSTable 文件移动到 C_2 组件上，而 C_2 组件的一次 Compaction 操作通常只能将一个 SSTable 文件移动到 C_3 组件上。以上三点导致了数据从 C_1 组件迁移至 C_2 的速度远大于数据从 C_2 组件迁移至 C_3 组件上的速度，从而使得数据很容易滞留在 C_2 组件上，导致 C_2 组件上的数据量逐渐增大。随后，当数据插入结束后（insert stop），C_0 组件上不再有新的数据插入，因此不会再向 C_1 层组件输出数据，进而 C_1 组件也无法达到触发 Compaction 操作的条件，没有了 C_1 层组件的 Compaction 操作，C_2 组件上的文件个数也就不再增加，随着 C_2 上 Compaction 操作的继续，C_2 组件上的文件个数逐渐减少。

如图 3-1 所示，除了 C_2 层组件的文件超过其文件个数阈值以外，C_3 组件上的文件个数也在测试开始不久，便超过了自身的文件个数阈值，即便是在写入操作结束以后，仍然保持上升趋势。而这一现象，也是由 C_2 层组件上文件个数过多造成的。根据前文可以得知，当多个组件同时满足 Compaction 条件时，需要根据组件优先度

选择哪层组件进行 Compaction 操作。在 C_3 组件上文件个数逐渐增大的整个过程中，C_3 组件的 Compaction 优先度始终低于 C_2 组件的 Compaction 优先度，因此数据无法从 C_3 层移动到 C_4 层组件，这一点从 C_4 层的文件个数一直为 0 也可以得知。而随着写入操作的结束，C_2 组件上的文件个数逐渐减少，C_3 组件的文件个数逐渐增加，最后当 C_3 组件的文件个数增加到 2000 时，其 Compaction 优先度为 40，而此时 C_2 组件上的文件个数不到 200 个，其 Compaction 优先度不足 40，C_3 组件的 Compaction 优先度大于 C_2 层组件的 Compaction 优先度，因此 C_3 组件开始进行 Compaction，即图中的 $C_{(C3,C4)}$位置，数据开始从 C_3 组件向 C_4 组件移动，C_3 层组件上的文件个数开始减少，C_4 组件上文件个数开始上升。

图 3-2 是 C_1 组件以及 C_2 组件在进行 Compaction 操作时的写放大变化趋势图。一方面，从(a)可以明显看到，C_1 组件的 Compaction 写放大随着时间逐渐增大，与图 3-1 中 C_2 组件上的文件个数随时间增加的趋势相吻合，这就说明了 C_2 组件上的 SSTable 文件个数增加会导致 C_1 组件的写放大增大。另一方面，从(b)中可以看出，尽管 C_3 上的 SSTable 个数也在不断增加，但 C_2 组件的 Compaction 写放大相对比较稳定，基本在 1 到 5 之间。这主要是由于在 C_2 组件进行 Compaction 时，从 C_2 组件上选择的 SSTable 个数为 1 个，而 C_2 组件上过多的 SSTable 个数导致了其上面的每个 SSTable 所覆盖的 key-range 较小，因此从 C_3 组件上选择的要覆盖 C_2 上选中的 key-range 的 SSTable 文件个数不多，从而 C_2 组件的 Compaction 写放大稳定在较小的范围内。



(a) C_1 组件 Compaction 写放大　　(b) C_2 组件 Compaction 写放大

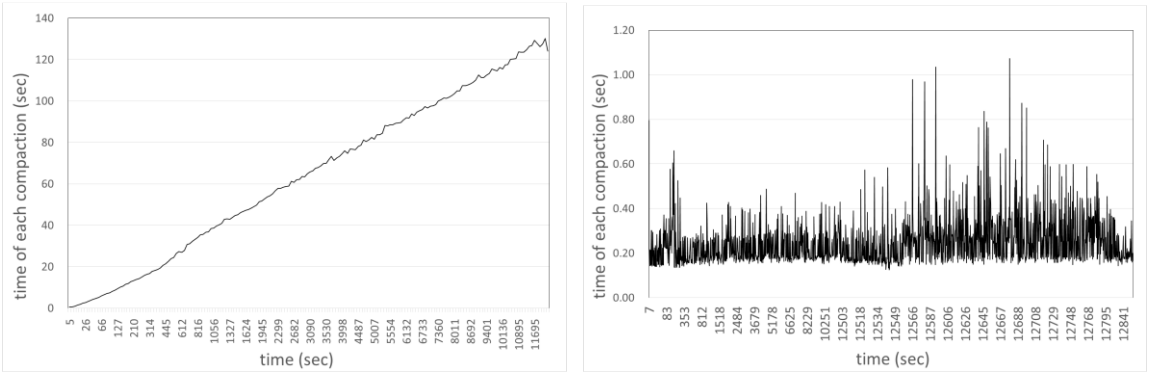
图 3-2　各层组件 Compaction 写放大统计

综上所述，造成 C_1 组件的 Compaction 写放大过大的原因主要有以下两点：

（1）C_1 组件在进行 Compaction 时，从 C_1 上会选择多个 SSTable 文件参与合并，从而合并的 key-range 较大；

（2）C_2 组件上的 SSTable 文件个数较多，导致每个 SSTable 文件包含的 key-range 较小。因此，在 C_1 组件进行 Compaction 时，从 C_2 组件上会选择大量的 SSTable 文件参与合并，导致写放大过大。

图 3-3 是 C_1 与 C_2 组件的单次 Compaction 耗时随时间变化曲线图。随着 C_1 组件的写放大逐渐增加，造成 C_1 组件的 Compaction 时间也越来越长，最高可达 130 秒左右，如图(a)所示；而相应的，由于 C_2 组件的 Compaction 写放大相对稳定，使得 C_2 组件的 Compaction 时间也保持在 1.2 秒之内，远远小于 C_1 组件的 Compaction 时间，如图(b)所示。



(a) C_1 组件单次 Compaction 耗时　　(b) C_2 组件单次 Compaction 耗时

图 3-3　各层组件单次 Compaction 操作耗时

因此，可以得出结论：一旦 C_1 组件的 Compaction 时间过长，就会导致 C_1 上的数据无法及时移动到 C_2 组件上，而由于 C_0 组件的 Compaction 具有最高优先度，并且会持续进行，最终使得 C_1 组件上的 SSTable 文件个数到达 N3 阈值，使得 LevelDB 对写请求停止响应。

图 3-4 展示了在运行过程中 LevelDB 的实时吞吐量。如图所示，随着 LevelDB 的持续运行，其性能逐渐出现上下波动，且波动幅度越来越大，直至出现系统停止响应写操作的情况，即图中吞吐量为 0 的区间；而且随着系统继续运行，停止响应

写操作所持续时间也越来越长，从 10 秒逐渐增加到 100 秒，这一情况与 C_1 组件的 Compaction 时间的逐渐增长相吻合。

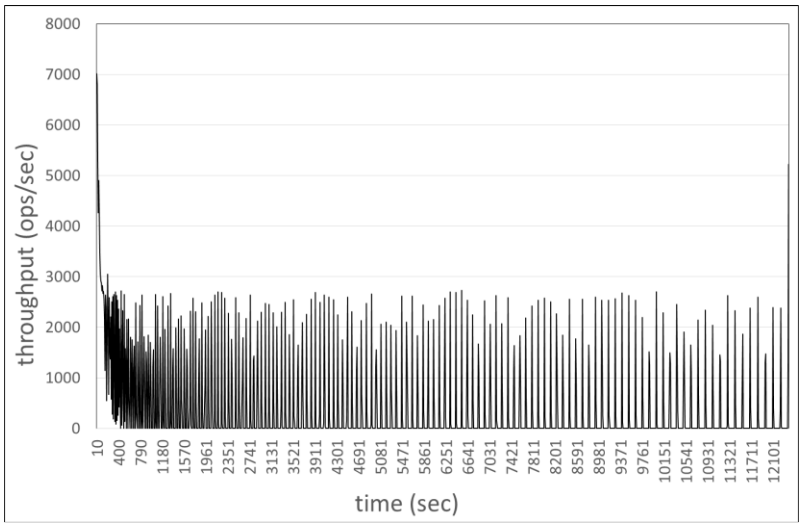


图 3-4　LevelDB 实时吞吐量

为了更加清晰地说明 C_1 单次 Compaction 耗时与系统写性能的关系，将 LevelDB 运行过程中 5000 秒到 8000 秒的实时吞吐量与这期间 C_1 组件进行 Compaction 操作的时间点进行了映射，如图 3-5 所示。从图中，可以清楚地看到，每次 C_1 组件进行 Compaction 操作， LevelDB 的吞吐量都会大幅降低了。这足以说明，C_1 组件的 Compaction 时间过长是影响 LevelDB 整体吞吐量下降的重要因素。

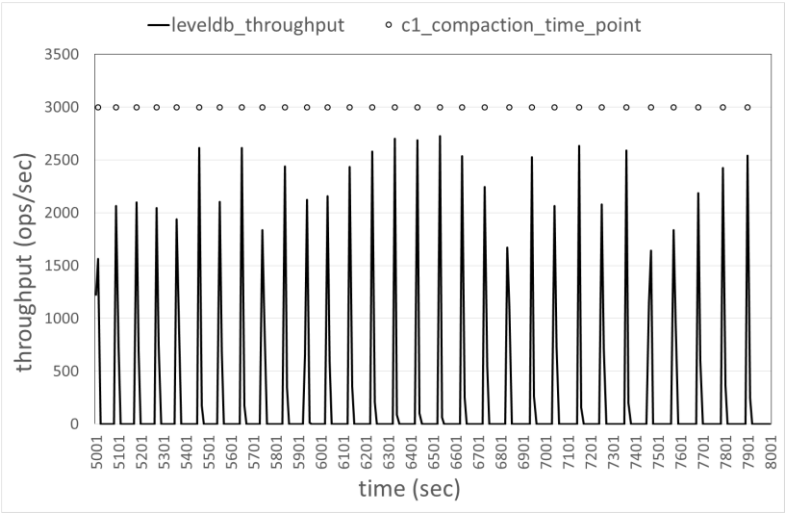


图 3-5　LevelDB 实时吞吐量与 C1 组件 Compaction 操作时间点

综上所述，在运行过程中，由于 LevelDB 没有有效地限制 C_2 组件上的 SSTable 文件个数，从而导致了 C_2 组件上 SSTable 文件个数增长过快，进而使得 C_1 组件的 Compaction 写放大过大，增加了 C_1 组件的单次 Compaction 时间，最终影响了 LevelDB 的整体性能。

为了从根本上解决由于 C_2 组件上文件个数过多导致的 LevelDB 的整体性能降低这一问题，提出了一种更加公平的 Compaction 调度机制——时间片轮转调度机制。时间片轮转调度机制旨在通过公平分配 Compaction 资源，保证各层组件都有充分的时间进行 Compaction 操作，从而有效控制各层组件的文件个数，进而保证其上层组件在进行 Compaction 操作时不会从本层读取过多的文件，从而避免了由于写放大过大而导致写性能严重下降的情况。

3.3 本章小结

本章首先介绍了 LSM-Tree 的 Compaction 调度机制；接着指出 C_2 组件上文件个数过多的问题，并通过一系列实验逐步验证了由于 LSM-Tree 的 Compaction 调度机制存在的缺陷，使得 Compaction 资源分配不均，进而导致 C_2 组件上文件个数过多，进一步造成了 C_1 组件写放大过大以及 Compaction 时间过长，最终降低了系统整体性能。

4 时间片轮转调度机制的实现

上一章节主要讲述了的 LSM-Tree 合并调度机制存在的问题。本章主要针对该问题，提出基于时间片轮转调度机制的优化方案，并详细介绍如何基于 RocksDB 进行实现。

4.1 时间片轮转调度机制的提出与设计

时间片轮转调度机制是一种公平调度机制，用于补充 LevelDB 原有的调度机制。通过设定的合并选择机制，选择待合并的组件，赋予其占用时间片（一种合并所需的系统资源）的权限，保证该组件在占用时间片的过程中连续进行若干次的合并操作，从而有效地控制组件上的文件个数，降低写放大，提升系统整体吞吐量。

时间片轮转调度机制的主要流程如图 4-1 所示。

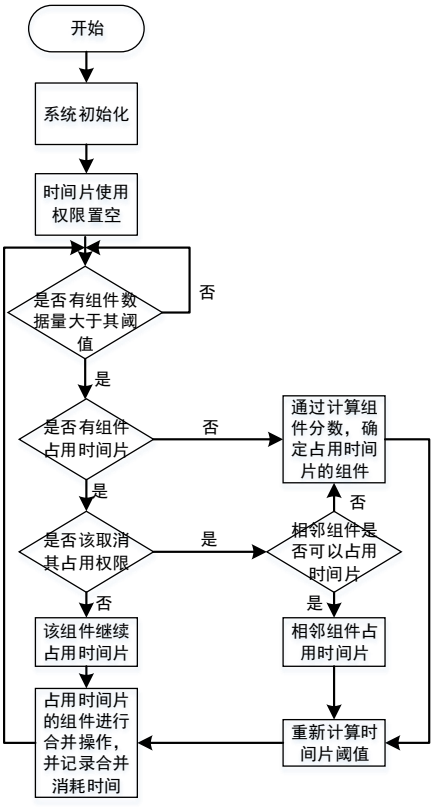


图 4-1 时间片轮转调度机制流程图

时间片轮转调度机制的流程主要分为三个阶段：

1）分配时间片占用权限

时间片在时间片轮转调度机制中是组件进行 Compaction 操作的必要条件，只有获取到时间片占用权限，才能进行 Compaction 操作。系统在为 LSM-Tree 的某个组件分配时间片占用权限时，待分配组件需要满足以下条件：

（1）被分配组件必须满足该组件的文件个数超过了该组件的文件个数阈值。

（2）优先选择上一次被分配时间片占用权限的组件的相邻下层组件进行分配。例如：C_2 组件刚刚解除时间片占用权限，那么优先选择 C_3 组件进行分配；

（3）如果根据条件（2）选择的组件不满足条件（1），又或者当前时间片的占用权限为空，则计算所有组件的当前文件个数与各个组件阈值的比值，满足条件（1）并且比值最大的组件将被分配时间片占用权限。

（4）如果所有组件均不满足条件（1），那么时间片占用权限暂时置为空，即没有组件拥有时间片占用权限，等待有组件满足条件（1）时再进行分配。

2）使用时间片

当某层组件获得时间片占用权限后，就可以在一定时间范围内多次进行 Compaction 操作。其中，该时间范围称为时间片阈值。

随后，在每一次 Compaction 时，会将单次 Compaction 所消耗的时间进行累加，一旦总的 Compaction 时间超过了时间片的阈值，则取消该组件占用时间片的权限。

3）取消时间片占用权限

为了避免不能进行 Compaction 操作的组件继续占用时间片，导致系统资源空闲，因此特别规定，满足以下条件的组件将被取消时间片占用权限：

（1）组件在使用时间片过程中，进行 Compaction 操作的总时间大于时间片阈值。

（2）组件当前文件个数已小于该组件的文件个数阈值。

当某层组件被取消占用时间片的权限后，将无法继续进行 Compaction 操作，直到下一次获得时间片占用权限；同时，系统重新进入分配时间片占用权限阶段，为新的组件分配时间片占用权限。

为了便于进一步理解时间片轮转调度机制，选取了系统运行过程中的六个阶段进行详细说明，如图 4-2 所示。其中，TS 表示时间片，时间片的阈值是 T_0。需要特殊说明的是，本图为示意图，在实际的 LSM-Tree 数据结构中，各层组件的文件个数阈值是以 10 倍逐层递增的。

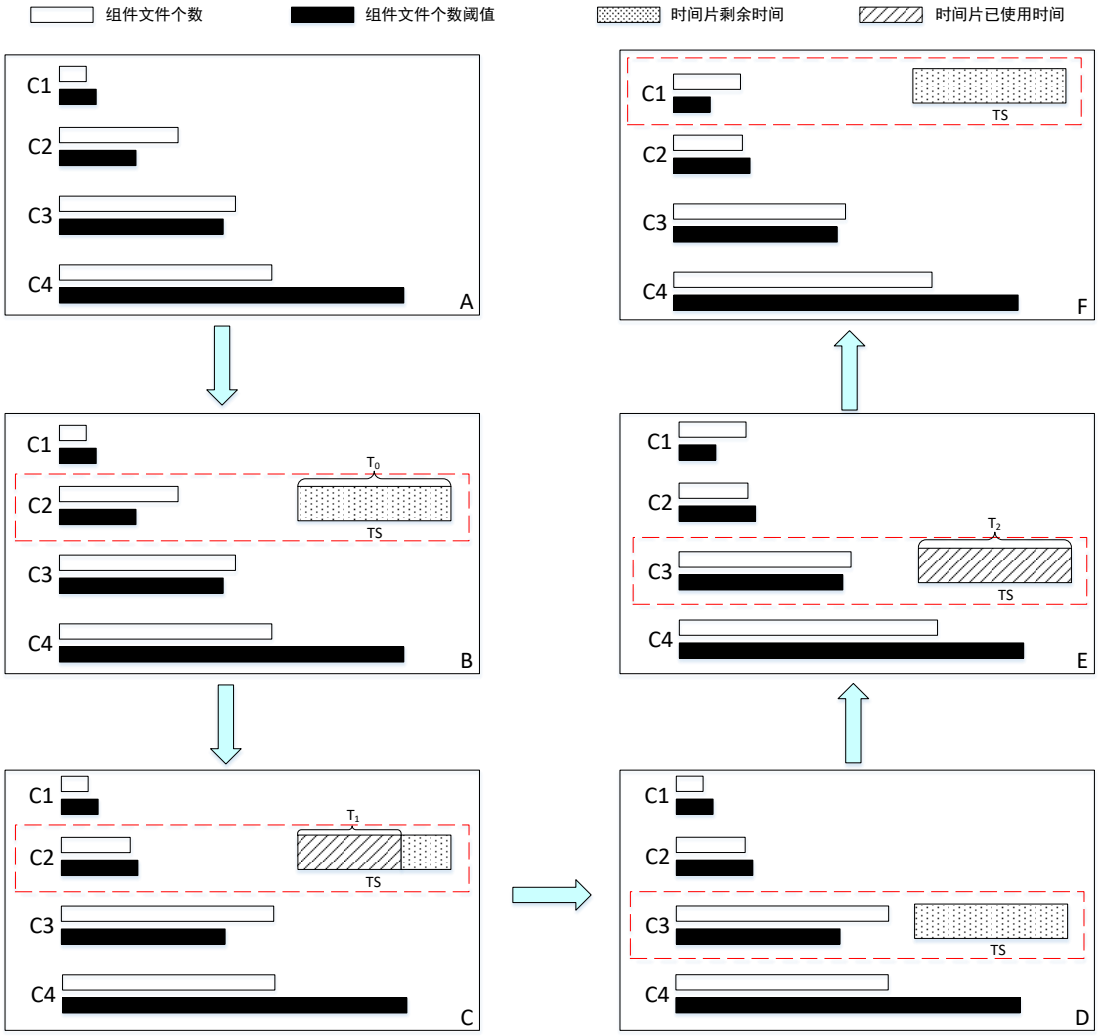


图 4-2　时间片轮转调度机制实例图

A 阶段，系统触发 Compaction，此时没有组件占用时间片，C_2 和 C_3 组件的文件个数都大于其文件个数阈值，根据时间片轮转调度机制中分配时间片的条件，C_2 组件的优先度大于 C_3 组件的优先度，因此 C_2 组件获得时间片 TS 占用权限。

B 阶段，C_2 组件开始使用时间片进行 Compaction 操作。

C 阶段，是 B 阶段经过 T_1 时间以后进入该阶段。此时，C_2 组件的文件个数小于其文件个数阈值，不满足进行 Compaction 操作的条件，即满足了取消时间片占用权限的条件，因此 C_2 组件需要释放占用的时间片。此时 C_2 组件的相邻下层组件 C_3 组件可以优先获得时间片 TS 占用权限，由于 C_3 的文件个数大于组件文件个数阈值，即满足时间片占用权限的条件；因此，C_3 组件获得时间片 TS 占用。

D 阶段，C_3 组件开始使用时间片进行 Compaction 操作。

E 阶段，是 D 阶段进过 T_2 时间以后进入该阶段。此时，C_3 组件的时间片使用时间已经达到时间片的阈值，即 $T_2=T_0$，满足了取消时间片占用权限的条件，C_3 组件释放占用的时间片。此时 C_3 组件的相邻下层组件 C_4 可以优先获得时间片 TS 占用权限，但是 C_4 组件上的文件个数没有超过其文件个数阈值，不满足进行 Compaction 操作的条件，无法占用时间片，因此时间片 TS 的占用权限通过重新计算优先度进行判断。经过计算与判断，C_1 组件获得时间片 TS 占用权限。

F 阶段，C_1 组件开始使用时间片进行 Compaction 操作。

以上是时间片轮转调度机制的详细介绍。时间片调度机制通过时间片的分配、时间片的使用和时间片的取消来完成一个时间片轮转调度机制的生命周期。

4.2 时间片轮转调度机制的实现

接下来，本节将详细介绍时间片轮转调度机制的实现环境和具体实现方法。

4.2.1 实现环境

为了便于后续的测试以及对比分析，基于 RocksDB 实现时间片轮转调度机制。

RocksDB 是 Facebook 基于 Google 的 LevelDB 进行多次优化后的开源的键值存储引擎。我们选择 RocksDB 而不选择 LevelDB 进行时间片轮转调度机制实现的原因有两点：第一，RocksDB 与 LevelDB 都是基于 LSM-Tree 实现的键值存储引擎，在本质上是一致的；第二，RocksDB 在性能方面相对于 LevelDB 有了很大的提升，很多方面都优于 LevelDB。

表 4-1 是时间片轮转调度机制具体的实现环境参数表。如表所示，RocksDB 是一个键值存储引擎，主要功能是给存储系统提供底层的存储服务，因此时间片轮转调度机制的实现选择了 C++语言。在实现过程中使用到了 C++的 C11 特性，所以选择了 GCC4.8.2 版本；选择了 Makefile 工具来管理系统的代码编译工作和安装；选择 Valgrind 工具对系统进行内存泄漏情况进行检测以及性能分析。选择了 Gflags 依赖库进行接收功能测试过程中命令行参数；选择 snappy 依赖库对系统进行压缩。

表 4-1　实现环境参数

项	内容
操作系统	Red Hat Enterprise Linux Server release 6.5
开发工具	GCC 4.8.2，Valgrind 3.8.1、Makefile
开发语言	C++
依赖库	snappy 1.1.0、 Gflags 2.0

4.2.2　时间片轮转调度机制的基础类及方法

时间片轮转调度机制的基础类为 CompactionSlot，表 4-2 是其主要的成员变量，表 4-3 是其主要方法。

表 4-2　CompactionSlot 主要成员变量

变量名	变量含义	变量类型
total_time_	表示时间片的阈值，在系统初始化阶段设定	uint64_t
compacted_time_	当前组件占用时间片期间所进行的Compaction的总时间	uint64_t
current_level_	当前占用时间片的组件的编号	int
level_num_	当前系统中组件的总个数	int
score_	当组件占用时间片时，给组件额外增加的优先度值，默认为1000,000	double
compacted_num_	当前组件占用时间片期间所进行的Compaction的总次数	int

表 4-3　CompactionSlot 主要方法

方法名	方法功能概述	主要代码实现
ChangeLevel()	当某一组件触发取消时间片占用权限的条件以后，自动将时间片的占用权限交给其相邻下一层组件，若前一组件为最后一层组件，则交给第一层组件。	current_level_ = (current_level_ + 1) % level_num_

FinishOneCompaction(uint64_t compaction_time)	当组件在占用时间片期间完成一个Compaction后，累加本次Compaction所用的时间，并判断是否超过时间片阈值。	compacted_time_ += compaction_time;
		if(compacted_time_ > total_time_) {
		ChangeLevel();
		}

ChangeLevelManual (int level)	指定某个组件占用时间片	current_level_ = level

4.2.3　时间片轮转调度机制的实现

图 4-3 是时间片轮转调度机制实现流程图。如图所示，时间片轮转调度机制主要通过改造和补充 RocksDB 的原始的 Compaction 过程。需要特殊说明的是，正常字体为 RocksDB 原始 Compaction 流程，斜体部分是为实现时间片轮转调度机制而新添加的流程。

下面将结合图 4-3 对时间片轮转调度机制的实现进行具体介绍。

首先，在实现过程中，将分配时间片占用权限和取消时间片占用权限两个功能进行了合并，即在每一次取消时间片占用权限之后立即进行下一次时间片占用权限的分配。在图 4-3 中为流程③和流程⑥两部分。具体的实现如下：如果是由于触发了取消时间片占用的条件(1)，则通过调用 CompactionSlot 的 ChangeLevel 方法，先由当前组件的相邻下一层组件获取时间片占用权限；如果是由于触发了取消时间片占用的条件(2)，则通过计算每层组件文件个数与文件个数阈值的比值，即就是各层组件的优先度，选出当前各层组件优先度最大且大于 1 的组件，通过 CompactionSlot 的 ChangeManual 方法，将时间片占用权限分配给该组件。

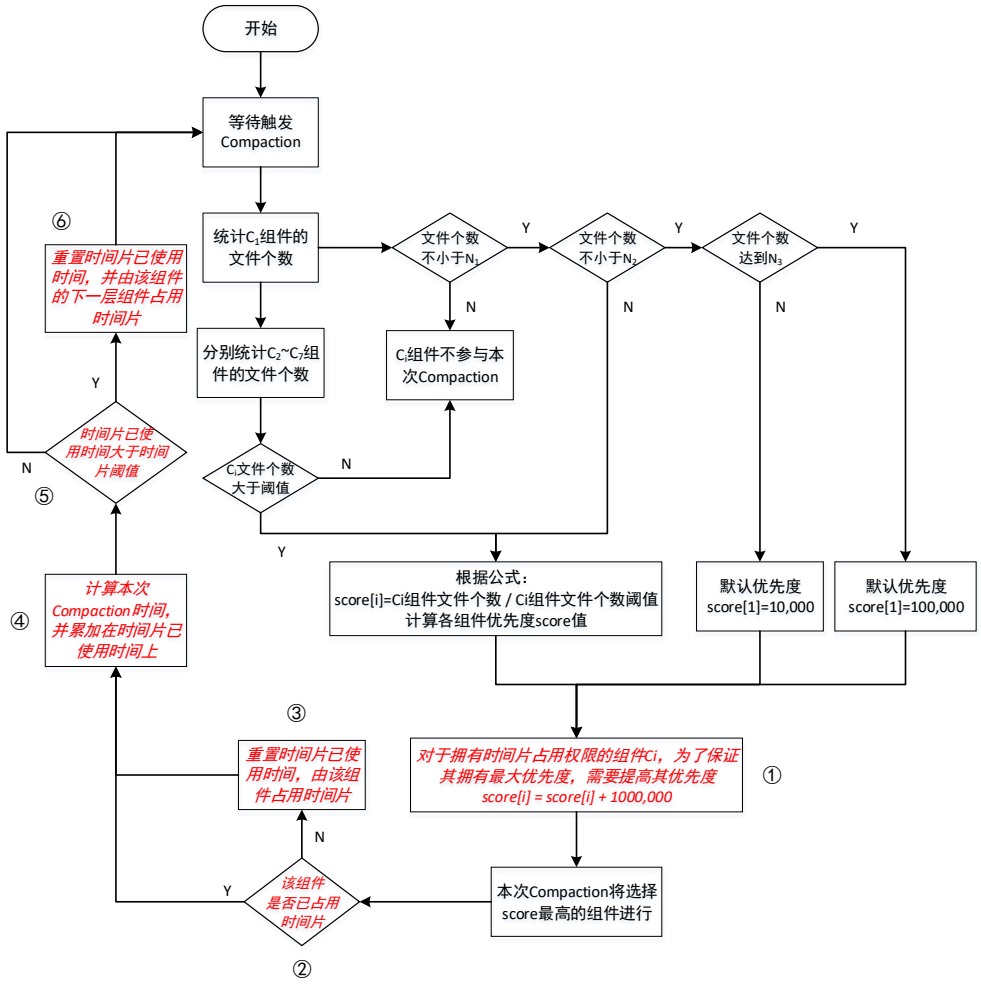


图 4-3　时间片轮转调度机制实现流程图

其次，在每次 Compaction 结束以后，需要对本次 Compaction 的操作时间进行统计，并累加在时间片的已使用时间中，以便在时间片用完的时候及时取消时间片占用权限。这个功能即为图 4-3 中的流程④。具体实现是通过调用 CompactionSlot 的 FinishOneCompaction 方法。

最后，是使用时间片进行 Compaction 的具体实现。这里通过借助 RocksDB 本身的优先度计算方法，并在原有优先度计算结果的基础上，增大当前占用时间片权限的组件的优先度计算结果，使得当前占用时间片权限的组件拥有最高的优先度，从而保证占用时间片权限的组件可以继续进行 Compaction 操作。在图 4-3 中，该实现流程即为流程①。

4.3 本章小结

本章首先提出了时间片轮转调度机制以及该机制的主要思想，即轮转分配时间片的方式保证公平的分配 Compaction 资源，从而弥补了原始调度机制的缺陷；接着选取了该机制运行过程中的几个状态，进一步阐述了时间片轮转调度机制的主要流程，即分配时间片占用权限、使用时间片和取消时间片占用权；最后详细介绍了如何在 RocksDB 的基础上实现时间片轮转调度机制。

5 优化方案测试及对比分析

本章主要介绍了时间片轮转调度机制的测试方案、测试过程以及测试结果。通过一系列测试，验证时间片轮转调度机制的优化效果。

5.1 测试环境及方法

5.1.1 测试环境

表 5-1 是测试对比对象和测试的软硬件环境。测试对比对象是 RocksDB，为了方便起见，使用 Slot 作为基于 RocksDB 实现的时间片轮转调度机制的简称，RocksDB 作为 RocksDB 原始调度机制的简称。

表 5-1 测试环境

项	内容
对比对象	RocksDB v2.5
测试工具	Yahoo! Cloud Serving Benchmark (YCSB)
操作系统	Red Hat Enterprise Linux Server release 6.5
CPU	Intel(R) Xeon(R) CPU E5-2620 v2 @ 2.10GHz
Memory	1024 MB
HDD	Seagate ST2000NM0033-9ZM（容量2TB，7200转/秒，缓存128M）两块，一块做系统盘，一块做数据测试

5.1.2 测试工具与实验方法

在测试实验，使用 YCSB 完成测试负载的生成和测试结果的统计。测试主要分为参数选择测试、性能对比测试和探究性测试。

5.2 参数选择测试

在时间片轮转调度机制中，时间片阈值的大小是决定其优化效果的关键参数。为此，进行了如下测试，以便确定在后续测试过程中 Slot 所选择的时间片大小。

测试所用负载是由 YCSB 生成的随机 KEY/VALUE 对，其中，KEY 值的平均大

小为 24B，VALUE 值的平均大小为 1000B，KEY/VALUE 对总长度为 1KB。测试分为十组，每组的时间片大小从小到大依次为 200ms、500ms、1000ms、2000ms、4000ms、8000ms、16000ms、24000ms、32000ms 和 64000ms。

测试主要针对写性能，因此，主要统计的是 YCSB 的数据加载阶段。具体方法为：设置 YCSB 加载的数据量为 5000000 条，每条数据的大小为 1KB，共计 5GB 数据。然后分别统计每个组写入所有数据所消耗的总时间，通过总数据量和总时间的比值，计算每个组的数据写入性能，测试结果如图 5-1 所示。

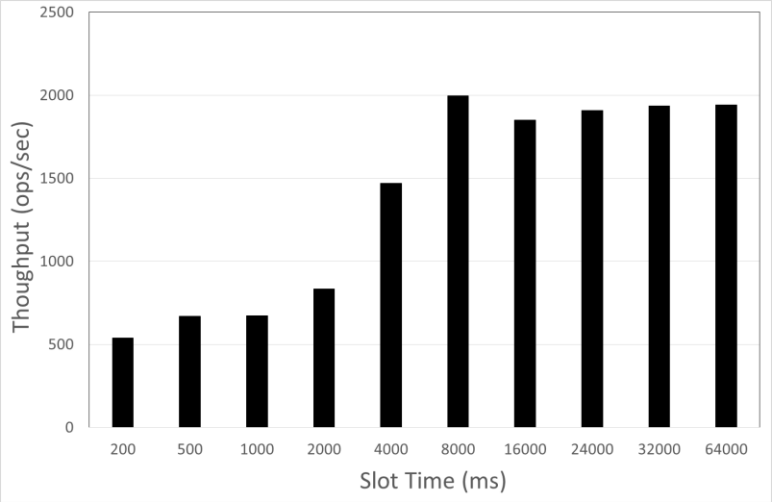


图 5-1　时间片阈值大小择优测试

如图 5-1 所示，在时间片从 200ms 到 8000ms 的变化过程中，系统写入性能逐渐增加，而时间片到达 8000ms 以后，系统的写入性能则不随时间片大小的增加而提升。出现这一现象的主要原因是：当时间片阈值的较小时，获取到时间片占用权限的组件可以连续进行的 Compaction 次数也较少，从而很难有效地解决 RocksDB 的 C_2 组件上数据滞留导致的写放大问题，因此系统的写性能提升幅度有限；而随着时间片阈值的不断加大，占用时间片的组件可以连续进行 Compaction 的次数也逐渐增加，C_2 组件上的数据滞留问题得到进一步解决，系统的写性能就进一步得到提升。而当时间片阈值到达 8000ms 以后，由于 C_2 组件上的数据滞留问题已经得到充分解决，进而系统的写性能不再随时间片阈值的不断增大而继续提升，因此，进一步增加时间片阈值大小也不会对系统写性能产生更加明显的效果。

为了进一步验证上述原因，分别统计了时间片大小为 200ms、1000ms 和 8000ms 时，各层组件文件个数的变化情况，如图 5-2 至图 5-4 所示。

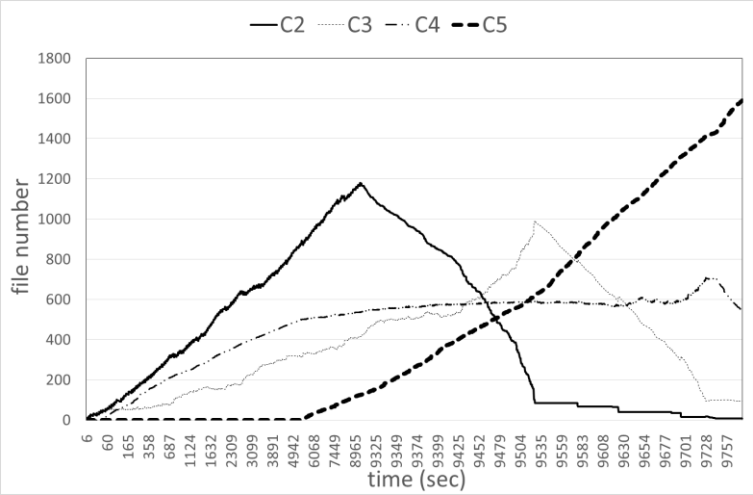


图 5-2　时间片阈值 200ms 各层组件文件个数

图 5-2 是时间片阈值为 200ms 时各层组件上文件个数随时间变化曲线图。如图所示，当时间片阈值为 200ms 时，C_2 组件上的文件个数仍然不能得到有效控制，最高时可达到 1200 个左右，因此系统的写性能较低。

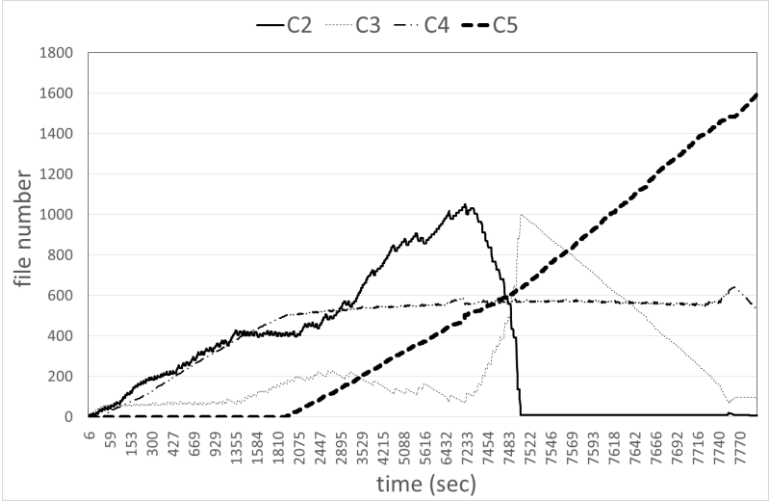


图 5-3　时间片阈值 1000ms 各层组件文件个数

图 5-3 是时间片阈值为 1000ms 时各层组件上文件个数随时间变化曲线图。如图所示，C_2 组件上的文件个数最高为 1000 个左右，相比于时间片阈值为 200ms 时有所

降低，但是依然很高，系统的写性能并没有很大提升。

图 5-4 是时间片阈值为 8000ms 时各层组件文件个数随时间变化曲线图。如图所示，C_2 组件上的文件个数被控制在 100 个左右，此时与 C_2 组件的文件个数阈值 50 比较接近，因此系统的写性能也相比于其他两个时间片阈值下的写性能两个更高。

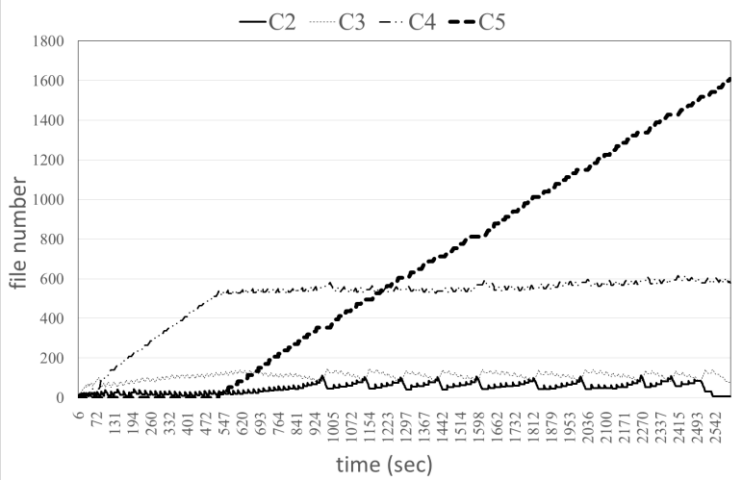


图 5-4　时间片阈值 8000ms 各层组件文件个数

与此同时，如图 5-4 中所示，当时间片阈值为 8000ms 时，不止 C_2 层组件，其他各层组件上的文件个数都得到了有效控制，基本都维持在其文件个数阈值附近，使得系统处于相对稳定的运行状态。

根据上述测试结果，为了达到比较好的优化效果，在后续的测试中，时间片的阈值大小统一设置为 8000ms。

5.3　性能对比测试

接下来，主要通过实验对比 RocksDB 和 Slot 的性能，来验证 Slot 的性能优化效果。

首先是写入性能的对比测试。与 5.2 类似，这里主要统计的是 YCSB 的数据加载阶段。为了保证测试的全面性，这里使用四组不同的数据量进行测试，分别为 5GB、10GB、20GB 和 50GB。

图 5-5 是四组数据集下 RocksDB 和 Slot 写性能对比图。如图所示，Slot 的写入性能远远高于 RocksDB，最高可达到 2017%。然而，随着数据量的增加，RocksDB 和 Slot 的写性能出现了不同程度的下降。

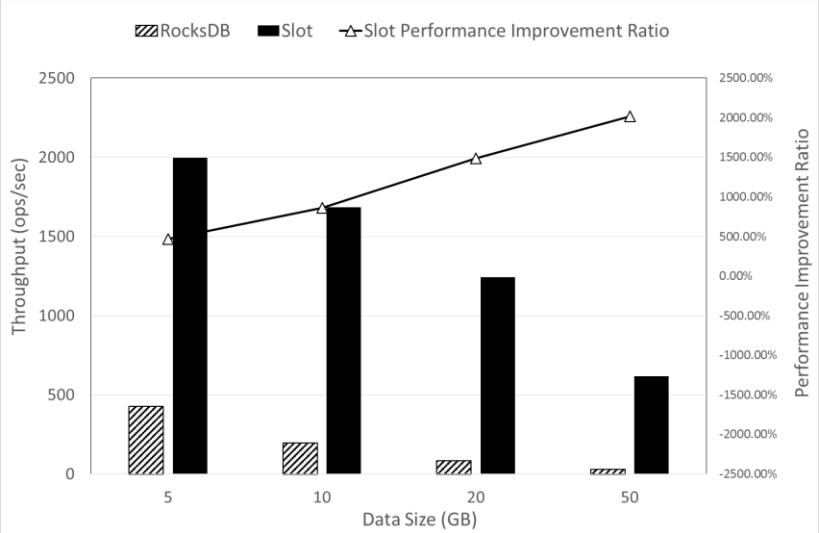


图 5-5　RocksDB 和 Slot 写性能测试对比图

造成上述现象的主要原因分为两个方面：一方面，对于 RocksDB 来说，如图 3-1 所示，在写操作停止之前，C_2 文件个数会持续增加；因此，当写入的数据量增加时，C_2 的文件个数会进一步增加，如图 5-6 所示。另一方面，对于 Slot 来说，当数据量增加时，8000ms 时间片大小很难再继续有效地控制 C_2 组件的文件个数，导致文件个数进一步增加，如图 5-7 所示。因此随着写入数据量的增加，RocksDB 和 Slot 的写入性能都呈下降趋势。尽管如此，Slot 相对于 RocksDB 性能的提升比例仍然是逐渐上升的，从 486%到 2017%，可见，随着数据量的增加，Slot 的优化效果更加明显。

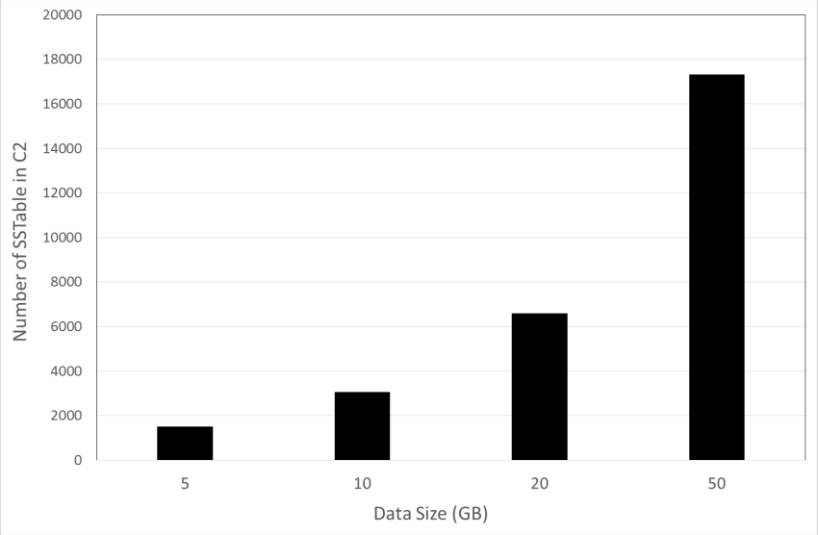


图 5-6　RocksDB 的 C_2 组件文件最大个数随插入数据量变化图

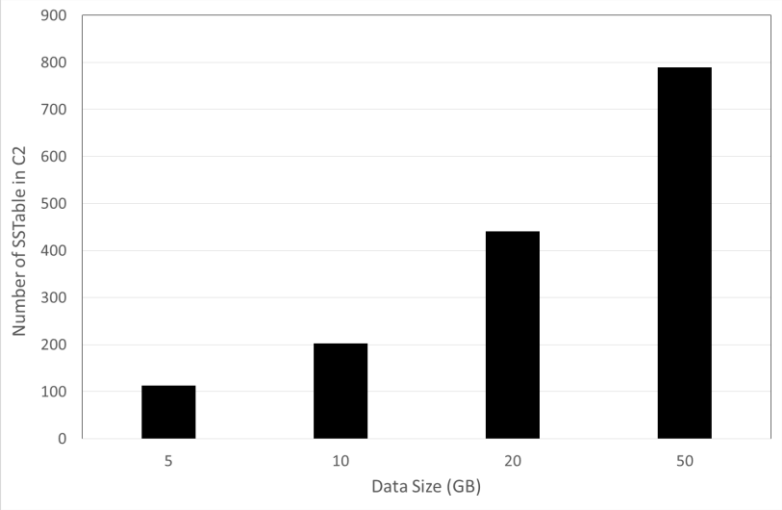


图 5-7　Slot 的 C_2 组件文件最大个数随插入数据量变化图

然后是读性能对比测试。由于时间片轮转调度机制并未对 LSM-Tree 的读性能做任何的优化，因此，该测试主要目的是说明该机制没有对 LSM-Tree 的读性能造成任何负面影响。

测试工具依然选用 YCSB。首先，在 YCSB 的数据加载阶段，分别向 RocksDB 和 Slot 加载 5000000 条数据。然后，在 YCSB 的运行阶段，分别从 RocksDB 和 Slot 中随机读取 5000000 条数据，并统计读取数据所消耗的总时间，计算读性能。

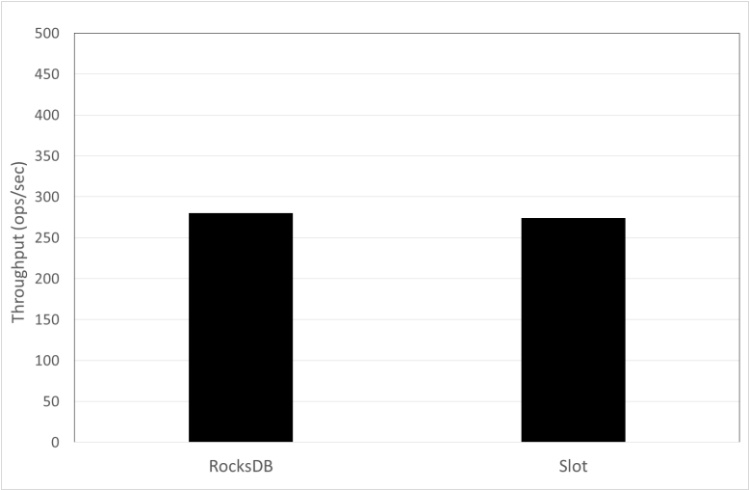


图 5-8　RocksDB 和 Slot 读性能测试对比图

图 5-8 是 5GB 数据集下 RocksDB 与 Slot 的读性能对比图。从图中可以看到，Slot 和 RocksDB 的读性能几乎相同。因此可以得出如下结论：时间片轮转调度机制在不影响 LSM-Tree 读性能的前提下，有效地提升 LSM-Tree 的写性能，从而提升了 LSM-Tree 的整体吞吐量。

5.4 优化原理探究分析

本节将统计和分析测试过程中的各项指标，逐步揭示 Slot 写性能较 RocksDB 有大幅提升效果的根本原因。为了方便起见，选取测试数据集的总数据量为 5GB 及时间片阈值为 8000ms 的测试结果进行分析。

5.4.1 写放大对比分析

写放大是影响 LSM-Tree 写性能的一个重要因素，为了验证 Slot 在写放大方面的优化效果，分别统计了 RocksDB 和 Slot 在数据插入的过程中的写放大情况。

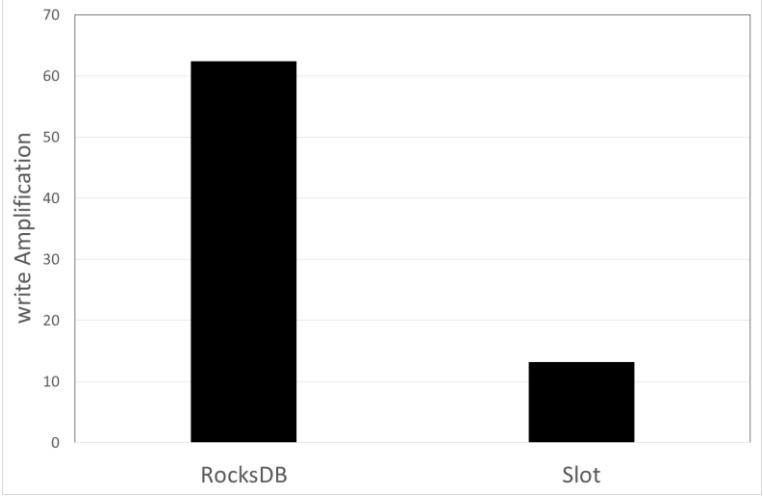


图 5-9　写放大对比图

图 5-9 是 5GB 数据集下 RocksDB 和 Slot 的写性能对比图。如图所示，RocksDB 的平均写放大为 62 左右，而 Slot 的平均写放大为 13 左右，相比于 RocksDB 降低了将近 80%。也就是说，如果需要通过 Compaction 操作将 2MB 的数据从组件 C_i 移动到组件 C_{i+1}，RocksDB 需要读写 124MB 的数据，而 Slot 只需要读写 26MB 的数据，在同样的磁盘读写速度情况下，Slot 的速度是 RocksDB 的 4 倍左右。

这里，对 LSM-Tree 的写放大进行更详细的阐述。前文提到，在 LSM-Tree 中，

下层组件的文件个数阈值是上层组件文件个数阈值的 N 倍（在 RocksDB 中，这一值默认为 10）。在各层组件数据均匀分布（即各层组件的文件个数接近其文件个数阈值，同时各层组件的文件所覆盖的 key-range 几乎相同）的情况下，下层组件文件个数大致是上层组件文件个数的 N 倍，因此下层组件每个文件所覆盖的 key-range 为上层组件每个文件所覆盖的 key-range 的 1/N。因此，通常情况下，在每次进行 Compaction 过程时，上层组件每选择 1 个文件，下层组件就会覆盖 N 个文件，而在 Compaction 过程中，只有上层组件选择的文件是需要移动到下层组件上，称之为有效移动，下层组件选择的文件仍然停留在下层，称之为无效移动；因此，在正常情况下，每次 Compaction 的写放大为 N+1，这一数值在 RocksDB 中为 11，即当 RocksDB 运行稳定后，最小的写放大为 11。而根据测试，Slot 的写放大为 13，与 RocksDB 的最小写放大十分接近；因此可以认为，时间片轮转调度机制可以有效地控制 Compaction 的写放大，调节 RocksDB 在运行过程中的稳定性，使系统达到近似最优的运行状态。

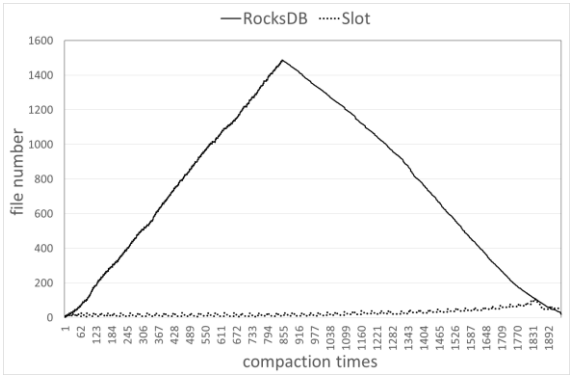
5.4.2 文件个数对比分析

前文提到，LSM-Tree 的写放大问题主要来源于 C_2 组件上的文件滞留，而时间片轮转调度机制的目的也在于解决 C_2 组件上文件的滞留问题，因此，为了验证时间片轮转调度机制是否真正解决了 C_2 组件上文件滞留的问题，使用 YCSB 进行数据插入的同时，分别统计了 RocksDB 和 Slot 各层组件上文件个数的变化情况，如图 5-10 所示。

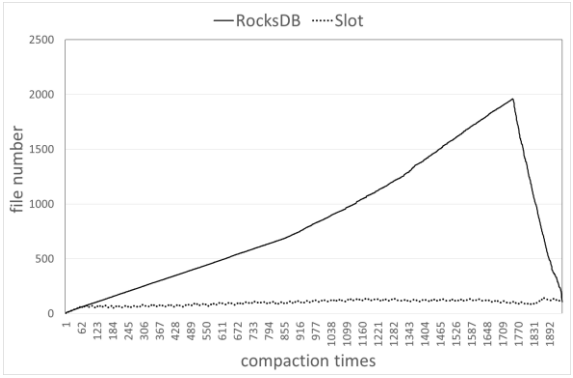
需要特殊说明的是，该测试实验统计的是每次 Compaction 后各组件上的文件个数，因此横轴选用的是 Compaction 次数。由于 Slot 的总 Compaction 次数大于 RocksDB（具体原因见 5.4.3），因此，图 5-10 选择 RocksDB 和 Slot 的前 2000 次 Compaction 作为横轴坐标，这样就可以看到在相同的 Compaction 次数情况下，RocksDB 各层组件的文件个数与 Slot 各层组件文件个数的对比情况。

图 5-10（a）是 RocksDB 和 Slot 的 C_2 组件上文件个数变化曲线图。可以明显看出，原始的 RocksDB 中 C_2 组件上文件个数呈直线增长，直至数据插入操作结束（具体介绍见 3.2）。而 Slot 则从测试开始阶段就将 C_2 组件上的文件个数有效地控制在在其文件个数阈值附近，并且随着测试的继续，C_2 组件上的文件个数也没有明显增加，同时稳定在其文件个数阈值附近。

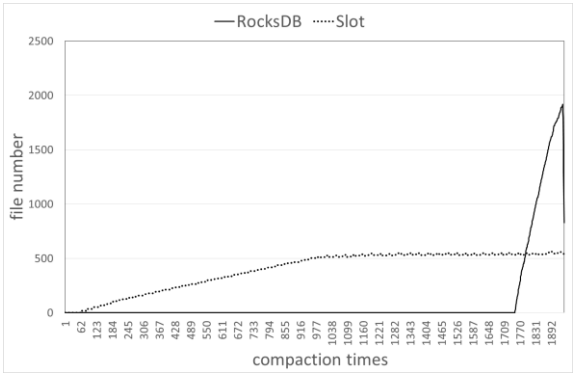
图 5-10 中的（b）、（c）和（d）分别是 RocksDB 和 Slot 的 C_3、C_4 和 C_5 组件文件个数变化曲线图。从图中可以清楚看到，从测试开始，Slot 有效地控制了 C_3、C_4 和 C_5 组件上的文件个数增长，并且随着测试的进行，这三层组件上的文件个数都没有明显增加。



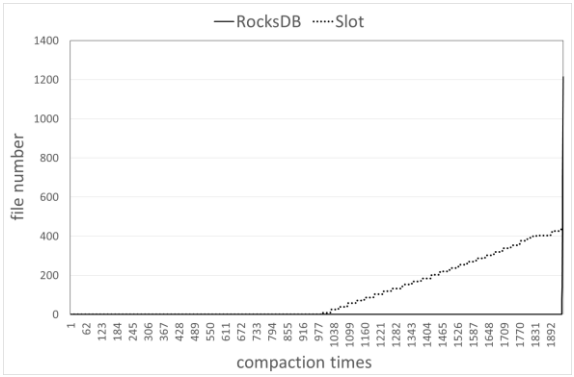
(a) C_2 组件文件个数变化图



(b) C_3 组件文件个数变化图



(c) C_4 组件文件个数变化图



(d) C_5 组件文件个数变化图

图 5-10　RocksDB 和 Slot 各层组件文件个数变化图

由此可以得出结论，正是由于 Slot 有效地控制各层组件上的文件个数，从而保证系统在数据写入的过程中，不会出现由于某层组件文件个数过多而导致的写放大过大的问题。

5.4.3　Compaction 次数和 Compaction 时间分析

正如前文所讲，RocksDB 写性能下降的一个主要表现就是系统停止响应写操作的持续时间越来越长，而进一步地，通过测试验证了系统停止响应写操作的持续时间正是 C_1 组件进行 Compaction 的时间。接下来，将通过统计 RocksDB 和 Slot 上的

C_1 组件的 Compaction 时间，进一步验证 C_2 组件上文件个数对 C_1 组件的 Compaction 时间的影响。

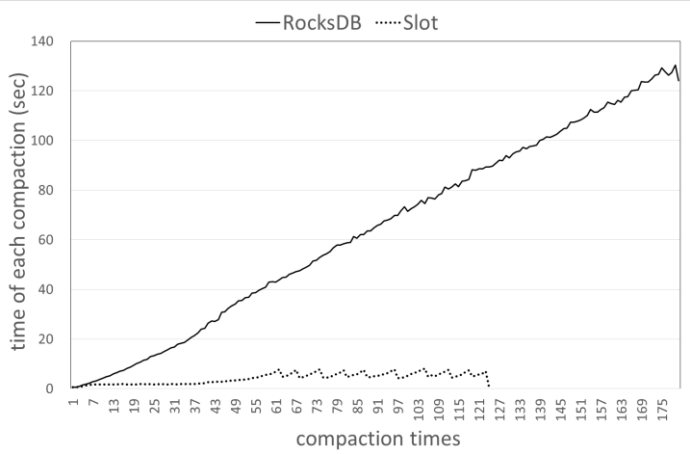


图 5-11　C_1 组件 Compaction 时间变化图

图 5-11 为 C_1 组件 Compaction 时间变化图。其中，纵坐标为每次 Compaction 的持续时间，横坐标为 Compaction 的次数。从图中可以清晰看到，在 RocksDB 中，C_1 组件的 Compaction 时间随 Compaction 的持续进行在逐渐增加，而在 Slot 中，C_1 组件的 Compaction 时间则基本稳定在 10 秒左右。因此可以得出结论，当 C_2 组件上的文件个数得到有效控制以后，C_1 组件的 Compaction 时间就不会由于 C_2 组件上文件个数的不断增加而延长，系统停止响应写操作的情况就可以得到改善，从而系统的整体写性能就能得到提升。

除此之外，通过实验对 RocksDB 与 Slot 的各层组件的 Compaction 次数也进行了相应的统计。

图 5-12 为 RocksDB 与 Slot 的各层组件 Compaction 次数对比图。从图中可以看到，一方面，Slot 的 C_1 组件的 Compaction 次数是 RocksDB 的 70%，小于 RocksDB，这就减缓了 C_2 组件上文件个数的增长速度，从而有效地缓解了由于 C_2 组件上文件个数过多而导致的 C_1 组件的写放大过大问题；另一方面，C_3 组件和 C_4 组件的 Compaction 次数远大于 RocksDB，其中 RocksDB 的 C_4 组件的 Compaction 次数为 0。这主要是由于 RocksDB 的 C_1 组件的 Compaction 时间过长，长期占用 Compaction 资源，导致其他层组件无法进行 Compaction，而 Slot 采用时间片轮转调度机制，各层

组件可以更加公平的共享 Compaction 资源，从而 C_3 组件和 C_4 组件可以更加有效地进行 Compaction。这也是为什么 Slot 可以有效控制 C_2 组件以及其他层组件上文件个数的原因。

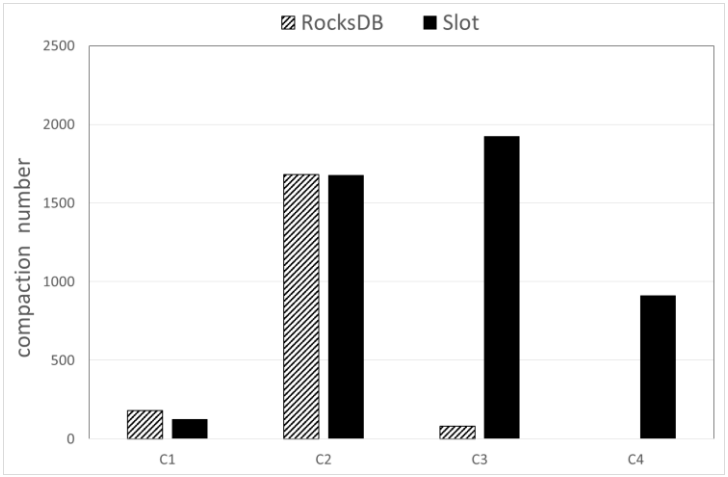


图 5-12　各层组件 Compaction 次数对比图

5.5　参数选择深入测试

在 5.2 节中，通过实验已经证实了，当测试数据量为 5GB 时，选择 8000ms 大小的时间片可以使 Slot 的写性能达到最大。但是，从 5.3 节中也可以看到，随着数据量的增加，Slot 的组件文件个数仍然呈上升趋势。为了进一步研究时间片阈值大小的选择与测试数据量之间的联系，本节又进行了一系列测试。

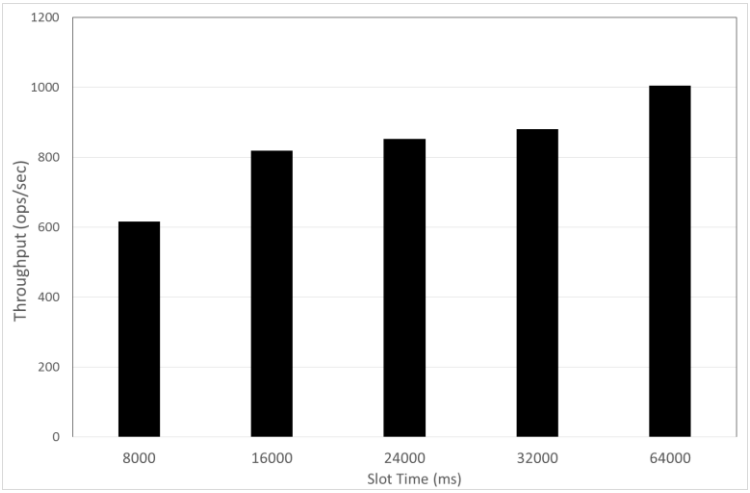


图 5-13　50GB 数据量时不同时间片大小性能对比图

测试分为五组，每组的时间片大小从小到大依次为 8000ms、16000ms、24000ms、32000ms 和 64000ms。设置 YCSB 加载的数据量为 50000000 条，每条数据的大小为 1KB，共计 50GB 数据；分别统计每个组写入所有数据所消耗的总时间，通过总数据量和总时间的比值，计算每个组的数据写入性能，测试结果如图 5-13 所示。

对比图 5-13 和图 5-1，不难发现，当测试数据量从 5GB 增加到 50GB 后，8000ms 大小的时间片不再是最优时间片了，随着时间片大小从 8000ms 增加到 64000ms，Slot 的性能得到了进一步的提升，提升比例达到 63%。

由此，可以得出如下结论：在不同的数据量情况下，使得 Slot 达到最大性能的最优时间片阈值也是不同的，因此，在实际生产应用中，时间片阈值的选择，需要根据不同的业务特点来进行决定。当业务的数据规模比较小时，设置相对较小时间的片阈值；相反，当业务的数据规模相对较大时，应当设置较大的时间片阈值。

5.6 本章小结

本章首先介绍了测试环境以及测试方法；然后通过参数选择测试确定了在 5GB 数据集下最优的时间片阈值为 8000ms；接着通过性能对比测试验证了时间片轮转调度机制可以将 RocksDB 的写性能提升 486%至 2017%；随后通过一系列的探究性分析解释了时间片轮转调度机制的优化原理，即时间片轮转调度机制使得各层组件可以更加公平地共享 Compaction 资源，避免了由于 C_1 组件的 Compaction 优先度过高导致的 C_2 组件上文件滞留问题，以及因此而导致的其他组件文件个数异常问题，从而降低了 Compaction 过程中的写放大过大的问题，进而减少了 Compaction 的时间，从而在根本上提高了写性能；最后通过分析在不同数据集下时间片大小对性能的影响，指出了在实际生产应用中需要根据不同的业务特点来设定时间片阈值大小注意。

6 总结与展望

6.1 全文总结

随着大数据时代的到来，NoSQL 数据库、ETL、数据挖掘、机器学习等技术得到了飞速发展，面对互联网数据的指数增长、分布式数据管理需求的日益增加、数据组织形式日益多样化等重大挑战，受事务有效性限制的传统关系型数据库，已经逐渐无法满足用户对数据处理和数据分析的业务需求。

在这种情况下，具有高可扩展性和易用性的 NoSQL 数据库逐渐广泛应用于生产环境，并且拥有大量的开源系统可供用户选择和使用，其中以 LSM-Tree 数据结构为基础的键值存储引擎最为流行。LSM-Tree 借助其延迟更新、批量写入等特性，在数据插入速度上具有很大优势，顺应于当前数据存储发展的趋势——写操作相比于读操作所占比例越来越大并且保持一个快速增长的趋势。对于数据库而言，存储引擎可以说是其最核心的组件之一，存储引擎性能的优劣直接决定了一个数据库的整体性能和可用性。

基于当前流行的 LSM-Tree 结构的键值存储引擎的进行详细分析，发现了 LSM-Tree 自身的 Compaction 调度机制所存在的严重问题，通过一系列科学严密的实验验证了该问题使得 LSM-Tree 中 C_1 组件抢占 Compaction 资源，引起 C_2 组件上文件个数异常，导致 C_1 与 C_2 组件进行 Compaction 期间写放大严重（即系统无效 I/O 比例增加）以及 Compaction 时间过长，进而 C_1 组件文件个数上涨直至到达 N3 阈值，造成系统停止响应写操作，最终影响系统整体性。针对这一问题，提出了新的 Compaction 调度机制——时间片轮转调度机制，该机制用于弥补原始的调度机制存在的 Compaction 资源分配不均的缺陷，避免了由此造成的某层组件文件个数过多导致的系统写放大严重以及写性能下降等问题。接着将时间片轮转调度机制实现在 LSM-Tree 结构的键值存储引擎——RocksDB 上。最后，通过测试对比，证实了时间片轮转调度机制可以在 RocksDB 的基础上，将写性能进一步提升 4 倍以上。

整个基于 LSM-Tree 的键值存储引擎的优化研究工作主要有以下几个方面：

（1）深入分析 LSM-Tree 结构的 Compaction 调度问题，以及由此问题带来的写放大过大和性能下降等现象。

（2）提出了时间片轮转调度机制，以解决 LSM-Tree 结构的 Compaction 调度问题。

（3）在 RocksDB 的基础上实现了时间片轮转调度机制。

（4）通过测试和进一步分析，验证了时间片轮转调度机制的优化效果和优化原理。

6.2 展望

时间片轮转调度机制是首次提出，由于研究时间以及相关条件的限制，仍然有很大的提升空间和研究价值。以下几个方面可以作为进一步改进和优化时间片轮转机制的方向：

（1）可以将时间片的阈值可以由静态改为动态。由于在实际运行过程中，可以根据组件获得时间片占用权限时文件个数与文件个数阈值的比值，以及该组件以往进行 Compaction 的效率，确定组件通过 Compaction 操作将超出阈值部分的组件移动到下层组件所需要的时间，从而确定时间片的阈值。这样就可以更加合理地分配时间片。

（2）组件在占用时间片进行 Compaction 的过程中，不一定要将本组件超过阈值的文件全部移动到下层，可以设置一个比例，比如将超过阈值的 70%的文件移动到下层组件即可放弃时间片。这样既可以有效控制各组件的文件个数不至于太大，也可以及时地将时间片交给更需要做 Compaction 的组件。

由于时间仓促以及研究水平有限，论文中难免存在不足，请各位老师及同学批评指正。

致 谢

首先，衷心的感谢我的导师吕泽华老师。在研究生期间，吕老师给我们学生留下和蔼可亲，兢兢业业的印象。在研究方面非常有经验，吕老师经常给同学们共享一些有研究参考价值的文章以及耐心指导同学们如何以科学并准确的凡是进行科研工作。在我毕业开题期间，吕老师给了我很对有益有价值的意见和建议；在我撰写毕业论文期间，吕老师十分耐心地审阅同学们的论文，并且认真指导同学们修改论文，让大家对自己的论文方向有了进一步的认识和理解。

我要感谢我的实习单位——中国科学院信息工程研究所第二工程部，感谢给了我十分难得的实习机会、优良的研究环境以及浓厚的研究氛围，让我可以在实际的科研项目中不断地锻炼和进步，在各种困难和挑战面前不退缩，勇往直前。感谢我的企业指导老师岳银亮副研究员，在我实习期间给我的细心指导，让我认识到实验与实践的重要性和科学性。

感谢我实习单位的同事们，在平时的学习、工作和生活中给了我很大的指导，促进我不断进步。

感谢我的同学和舍友们，在生活上给我莫大的关心和帮助。

感谢我的父母，你们是我永远的后盾，给了我支持和信心。

最后，感谢我的论文评阅老师和答辩委员会的各位老师，感谢所有在工作学习和生活上帮助过我的人！真诚祝愿你们，身体健康，工作顺利！

参考文献

[1] 戴德宝, 刘西洋, 范体军. “互联网+”时代网络个性化推荐采纳意愿影响因素研究. 中国软科学, 2015(8): 163-172

[2] 王元卓, 靳小龙, 程学旗. 网络大数据: 现状与展望. 计算机学报, 2013, 36(6): 1125-1138

[3] 吴列宏, 张瑜. 试析大数据双向驱动电信运营商流量经营. 现代电信科技, 2015, 45(3): 71-74

[4] 严霄凤, 张德馨. 大数据研究. 计算机技术与发展, 2013(4): 168-172

[5] 马建光, 姜巍. 大数据的概念、特征及其应用. 国防科技, 2013, 34(2): 10-17

[6] 纪慧蓉, 拾祎春. 大数据信息存储应用. 中国高新技术企业, 2008(16): 129-129

[7] 孟小峰, 慈祥. 大数据管理: 概念、技术与挑战. 计算机研究与发展, 2013, 50(1): 146-169

[8] Chang F, Dean J, Ghemawat S, et al. Bigtable: a distributed storage system for structured data. Acm Transactions on Computer Systems, 2008, 26(2): 205-218

[9] Cooper B F, Ramakrishnan R, Srivastava U, et al. PNUTS: Yahoo!'s hosted data serving platform. Proceedings of the Vldb Endowment, 2008, 1(2): 1277-1288

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

[11] Lakshman A, Malik P. Cassandra: a decentralized structured storage system. Acm Sigops Operating Systems Review, 2010, 44(2): 35-40

[12] Escriva R, Wong B. HyperDex: a distributed, searchable key-value store. Acm Sigcomm Computer Communication Review, 2012, 42(4): 25-36

[13] Beaver D, Kumar S, Li H C, et al. Finding a Needle in Haystack: Facebook's Photo Storage. OSDI, 2010, 10: 1-8

[14] Sears R, Ramakrishnan R. bLSM: a general purpose log structured merge tree. Acm Sigmod International Conference on Management of Data. ACM, 2012: 217-228

[15] Zawodny J. Redis: Lightweight key/value store that goes the extra mile. Linux Magazine, 2009, 79: 12-56

[16] Huang Q, Birman K, Van Renesse R, et al. An analysis of Facebook photo caching. Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles. ACM, 2013: 167-181

[17] Wang P, Sun G, Jiang S, et al. An efficient design and implementation of LSM-tree based key-value store on open-channel SSD. Proceedings of the Ninth European Conference on Computer Systems. ACM, 2014: 16-45

[18] Debnath B, Sengupta S, Li J. FlashStore: high throughput persistent key-value store. Proceedings of the VLDB Endowment, 2010, 3(1-2): 1414-1425

[19] Debnath B, Sengupta S, Li J. SkimpyStash: RAM space skimpy key-value store on flash-based storage. Proceedings of the 2011 ACM SIGMOD International Conference on Management of data. ACM, 2011: 25-36

[20] O'Neil P, Cheng E, Gawlick D, et al. The log-structured merge-tree (LSM-tree). Acta Informatica, 1996, 33(4): 351-385

[21] Wu X, Xu Y, Shao Z, et al. LSM-trie: An LSM-tree-based ultra-large key-value store for small data items, 2015 USENIX Annual Technical Conference (USENIX ATC 15), 2015: 71-82

[22] Zhang Z, Yue Y, He B, et al. Pipelined Compaction for the LSM-Tree. Parallel and Distributed Processing Symposium, 2014 IEEE 28th International. IEEE, 2014: 777-786

[23] Shetty P, Spillane R, Malpani R, et al. Building workload-independent storage with VT-trees. Usenix Conference on File & Storage Technologies, 2013: 17-30

[24] Moshayedi M, Wilkison P. Enterprise SSDs. Queue, 2008, 6: 32-39

[25] Webber J. A programmatic introduction to Neo4j. Conference on Systems, Programming, and Applications: Software for Humanity. 2012:217-218

[26] Lai C, Jiang S, Yang L, et al. Atlas: Baidu's key-value storage system for cloud data. Mass Storage Systems and Technologies (MSST), 2015 31st Symposium on. IEEE, 2015: 1-14

[27] Katsaros K, Xylomenos G, Polyzos G C. MultiCache: An overlay architecture for information-centric networking. Computer Networks the International Journal of Computer & Telecommunications Networking, 2011, 55(4):936-947

[28] Stender J, Kolbeck B, Högqvist M, et al. BabuDB: Fast and efficient file system metadata storage. Storage Network Architecture and Parallel I/Os (SNAPI), 2010 International Workshop on. IEEE, 2010: 51-58

[29] Ren K, Gibson G A. TABLEFS: Enhancing Metadata Efficiency in the Local File System. USENIX Annual Technical Conference, 2013: 145-156

[30] M. A. Olson, K. Bostic, et. al. Berkeley DB. in USENIX ATC'1999

[31] Rosenblum M, Ousterhout J K. The design and implementation of a log-structured file system. ACM SIGOPS Operating Systems Review. ACM, 1991: 1-15

[32] William Pugh. Skip lists: A probabilistic alternative to balanced trees. Lecture Notes in Computer Science, 1989, 33(6): 668-676

[33] Cooper B F, Silberstein A, Tam E, et al. Benchmarking cloud serving systems with YCSB. Proceedings of the 1st ACM symposium on Cloud computing. ACM, 2010: 143-154

[34] 金懿. 大数据下的广告营销战略发展趋势. 中国传媒科技, 2013(14): 39-40

[35] Dent A S. Getting Started with LevelDB. Packt Publishing Ltd, 2013

[36] Pokorny J. NoSQL databases: a step to database scalability in web environment. International Journal of Web Information Systems, 2013, 9(1): 278-283

[37] Chu H. MDB: A Memory-Mapped Database and Backend for OpenLDAP. 2011.

[38] 李国杰, 程学旗. 大数据研究: 未来科技及经济社会发展的重大战略领域——大数据的研究现状与科学思考. 中国科学院院刊, 2012(6): 45-77

[39] Ahuja S P, Moore B. State of Big Data Analysis in the Cloud. Network & Communication Technologies, 2013, 2(1): 14-15

[40] Merv Adrian. Big Data: it’s going mainstream and it’s your next opportunity. Teradata Magazine, 2011(1): 3-5

[41] Manyika J, Chui M, Brown B, et al. Big Data: The Next Frontier for Innovation, Comptetition, and Productivity. Analytics, 2011

[42] Tesoriero C. Getting started with OrientDB. Packt Publishing Ltd, 2013

[43] Chodorow K. MongoDB: The Definitive Guide 2nd Edition. O'Reilly Media, Inc, 2014

[44] George L. HBase : the definitive guide. Andre, 2011, 12(1): 1-4

[45] 姜丰. 大数据环境下技术创新管理方法研究. 中国新通信, 2016, 18(1): 27-27