研 究 生 毕 业 论 文

（申请工程硕士学位）

论　文　题　目　图数据库StellarDB存储模块性能优化的设计实现

作　者　姓　名

学科、专业名称　工程硕士（软件工程领域）

研　究　方　向　软件工程

指　导　教　师

2020 年 2 月 27 日

学　　号：**MF1832198**

论文答辩日期：**20xx 年 x 月 xx 日**

指 导 教 师：　　（签字）

The Design and Implementation of a Performance Optimization on Storage Module of a Graph Database

By

(Author Name)

Supervised by

(Supervisor's position)**(Supervisor's Name)**

A Thesis
Submitted to the XXX Department
and the Graduate School
of XXX University
in Partial Fulfillment of the Requirements
for the Degree of
Master of Engineering

Feb 2020

研究生毕业论文中文摘要首页用纸

毕业论文题目：图数据库StellarDB存储模块性能优化的设计实现

工程硕士（软件工程领域） 专业 2020 级硕士生姓名：

指导教师（姓名、职称）：

摘 要

今天，图数据库由于其在现实生产生活场景中的建模优势、在关系查找方面的性能优势，称为越来越炙手可热的软件产品。国内外许多软件厂商都在推出自己的图数据库产品。StellarDB是星环公司的图存储引擎，可以与其他组件配合，实现图存储、图分析、图可视化等重要功能。

StellarDB存储引擎采用日志结构合并树（Log-Structured Merge-Tree，下称LSM树）数据结构作为底层数据存储方式。但是，在实际部署过程中，遭遇了“高并发写入之后读请求响应超时”的问题。本文介绍了对问题的分析过程，对flush算法与compaction算法的联合创新性优化方案的设计与实现，并展示了优化过程中的多次性能测试结果，从中解释算法逐步优化的根据与效果。另外，本文通过对StellarDB的存储模块进行抽象简化，实现了一个存储模块模拟程序，用于再次测试、验证优化算法的有效性。

经过优化，StellarDB在高并发写入情况下compaction遭遇性能瓶颈的问题已经解决，数据文件可以在LSM树种顺畅流动到底层。目前优化后算法已经在StellarDB中稳定运行约半年，证明了其稳定性。

关键词： 图数据库，StellarDB，日志结构合并树，flush，compaction

研究生毕业论文英文摘要首页用纸

THESIS: The Design and Implementation of a Performance Optimization on Storage Module of a Graph Database

SPECIALIZATION: Software Engineering

POSTGRADUATE: (Author Name)

MENTOR: (Supervisor's position)(Supervisor's Name)

Abstract

Nowadays, graph database is becoming more and more popular because of its advantage in modeling real life and performance advantage in relationship search. Many software companys, countrywide or worldwide, are building their own graph database systems. StellarDB is the graph storage engine of Transwarp Inc. Combined with other Transwarp products, it can support various big data analysis requirements, including graph storage, graph analysis, graph visualization, etc..

StellarDB storage engine adopted Log-Structured Merge-Tree(i.e. LSM-tree) as its underlying data structure for record storage. However, during deployment and tests on clients' hosts, performance issues were encountered that read requests timeouted after high concurrency write operation. The thesis introduces the analyze process on the issue and the design and implementation of a combined innovative optimization on flush algorithm and compaction algorithm. Then the thesis demostrated results of performance tests during the optimization process, explaining the evidences on which the optimization was based. Moreover, the thesis implemented an simulator of the storage module for more testing and verification of the influence of the new algorithms.

After this optimization, the performance bottleneck of StellarDB performing compaction under high concurrency write operation was successfully solved. Now the data files can be compacted to the bottom level of LSM-tree easily. The optimized algorithm has been running in deployed StellarDB instances for about 6 months, which proved its steadiness.

Keywords: Graph database, StellarDB, Log-Structured Merge-Tree, flush, compaction

目　录

表 目 录

图　目　录

第一章　引言

1.1　项目背景

图是一种由点边组成的半结构化数据，用于映射事物之间的关系，如人际关系、交易往来、交通道路等模型。属性图（Property Graph）是近年来兴起的一种图模型，在点、边上可以自由定义属性和类型，从而形成社交网络、交易网络等复杂图。根据著名数据库统计网站DB-Engines的统计，图数据库在短短数年内获得了远超其他类型数据库的关注。大数据时代不仅带来了数据量的增长，也带来了数据类型的多样化。随着5G时代的临近，智能终端、无线传感器组成的数据网络会带来更复杂的数据类型，给数据存储和分析带来巨大压力。

传统关系型数据库擅长处理拥有固定结构的表格型数据，通过一些JOIN操作来得到数据之间的关联关系。在数据量增长或数据类型复杂时，关系型数据库会存在几个瓶颈。一、为了获得数据之间的连接信息，关系型数据库不得不通过JOIN的方法来取得“下一跳”节点。大量的JOIN操作不仅对计算资源造成极大浪费，也无法快速返回数据结果。二、图数据在应用场景中可能频繁地修改数据模型，关系型数据库在应对这种场景时，对用户的模型设计能力要求极高。关系型数据库由于数据模型限制而无法适配图场景，图数据库因此孕育而生。相较于关系型数据库，图数据库在这些方面具有优势：拥有灵活可变的数据结构；充分利用图的内联信息，可存储规模庞大的关系；实时返回查询结果。

图数据库把边视作数据的一种，关系型数据库的JOIN操作转换为图数据库的一次普通查询。在数据量增加时，JOIN操作会急剧增加查询的开销，但对于图数据库仅会增加少量开销。随着图数据数量增加，单机系统在计算和存储上存在明显瓶颈，分布式图数据库是未来的趋势。

Transwarp StellarDB是一款为企业级图应用而打造的分布式图数据库，用于快速查找数据间的关联关系，并提供强大的算法分析能力。StellarDB克服了海量关联图数据存储的难题，通过自定义图存储格式和集群化存储，实现了传统数据库无法提供的低延时多层关系查询，在社交网络、公安、金融领域都有巨大应用潜力。 [1]

1.2　图数据库发展现状

1.2.1　主流图数据库

目前的主流图数据库都是国外公司的产品。有些是开源/商业数据库产品，如Neo4j, Janus Graph；有些则是大公司开发自用，如Facebook的RocksDB、Google的LevelDB。

Neo4j由Neo4j公司开发，是一个嵌入式的、基于磁盘的、具备完全的事务特性的Java持久化引擎。它提供同时提供社区免费版和商业版，是“DB-Engines Ranking”上流行度排名第一的图数据库。

JanusGraph是Linux基金会名下的一款开源、分布式图数据库，其开源遵从Apache 2.0协议。这款产品的的特点在于与Apache旗下众多项目连结紧密，可以良好协作。

RocksDB、LevelDB则分别是Facebook与Google的内部产品。虽然两个项目在GitHub上都有开源版本，但是开源项目更新缓慢，而且性能水平明显不是成熟版本的水平，所以不能代表这两个项目的内部版本实际效果。

1.2.2　图数据库存储常用技术

图数据库的概念只是限定了它给用户呈现的数据模型抽象应当是图，而对于内部存储的实现没有限制。所以图数据库的存储可以使用传统的表来实现；也可以使用键值对存储，强调NoSQL优势。StellarDB所采用的LSM树就属于一种键值对存储，这也是主流图数据库的共同选择。因为LSM树能够很好地满足大多数图场景对频繁写的需求：不论是新增、更新还是删除操作，时间复杂度与数据量都是线性关系。因为compaction操作与客户的写入是异步的，而且只要compaction性能不过差，就基本不会影响到客户的读写请求。

根据需求不同，图数据库还可以在磁盘读写为主和全内存缓存之间进行选择。StellarDB由于要应对众多客户的复杂场景，选择了依靠磁盘读写与内存的部分缓存。而一些目标场景明确、硬件资源充足的图数据库，如领英自用的GraphDB，就采用了数据内存全缓存的方案，在性能上明显高于依赖磁盘的数据库。

1.3　论文主要工作

本文主要介绍了StellarDB的存储模块的设计，描述了对于其写入性能瓶颈的一次优化的设计与实现。在本次优化之前，StellarDB在对于LSM树数据结构

进行compaction处理时，会遭遇“第0层的数据向下流动速度极慢”的严重性能问题，极大程度导致了在写入频繁的应用场景中StellarDB容易超负荷、读写操作响应超时的缺陷。通过本文介绍的对于flush处理模块与compaction处理模块的联合优化，StellarDB的存储模块极大程度上解决了写入性能问题，进而获得了在写入频繁的场景中读写性能的全面提升。

1.4　论文组织结构

本文的组织结构如下：

第一章，引言部分。介绍了StellarDB的项目背景及图数据库国内外研究现状，简要的介绍了本文的主要工作。

第二章，技术综述。介绍了StellarDB系统实现过程中所涉及的技术，包括LSM树数据结构、Raft一致性协议等技术的相关介绍。

第三章，StelllarDB存储模块的分析与设计。首先，分析了该模块的地位、与其他模块的协作关系。然后，大致说明本模块内的组件协作与划分，详细介绍模块对LSM树数据结构的维护方式。最后，简要介绍优化模拟测试程序的设计目标。

第四章，StellarDB存储模块局部及其模拟程序的详细设计与实现。其中通过源代码等方式详细说明了存储模块的flush、compaction部分的设计与实现细节，重点展示在写入处理方面的优化；展示了优化模拟程序的详细设计。性能测试部分展现了在高性能集群下进行使用实际数据的严格控制变量的StellarDB写入性能对比测试的结果，以及优化模拟程序的测试结果。

第六章，总结与展望。总结本论文所做的工作，并对StellarDB的未来发展方向做进一步的展望。

第二章　技术综述

本章介绍StellarDB存储引擎系统在设计与开发过程中用到的相关理论和技术。由于StellarDB是数据库，在应用中位于底层，所以使用的技术偏向数据结构与算法，而开源框架一类的较少。

2.1　Log-Structured Merge-Tree数据结构

日志结构合并树（Log-Structured Merge-tree，下文称LSM树）是基于磁盘的数据结构，旨在在长期运行过程中在文件上提供记录的高频率插入和删除，以及低成本索引。LSM树使用延迟和批量索引更改的算法，将更改从内存中的组件通过一个或多个磁盘组件进行级联。这种高效的方式类似于到合并排序。在此过程中，所有索引值都可以通过内存组件或磁盘组件连续访问。相比B树等传统访问方法，该算法大大减少了磁盘臂的移动。LSM树还可以推广到其他操作，比如插入和删除。 [2]

包括Google Bigtable、Apache Cassandra等数据库都采用了LSM树作为数据存储结构。 [3] StellarDB也采用LSM树作为数据记录的存储结构。StellarDB目前没有支持使用多种磁盘组件的级联，数据仅在逻辑上有额外的层次划分。StellarDB将数据从内存组件向磁盘的流动操作称为flush，将数据在磁盘中多个逻辑分层之间的流动称为compaction。这两种操作是维护LSM树的主要操作，其具体算法将在本文第三章详细说明。

2.2　Raft一致性协议

为了保证多副本数据最终状态一致，StellarDB使用Raft一致性协议作为多副本的协调机制，并利用Raft的日志（log）来短期备份写入的数据。

2.2.1　Raft概念

Raft是由Diego Ongaro与John Ousterhout的一篇论文 [4]中所提出的分布式存储一致性算法。在分布式系统中，为了防止服务器数据由于只存一份而导致在服务时由于一个存储节点故障就产生服务完全不可用或数据丢失的严重后果，数据的存储会有多个备份副本，分别存储于不同的存储服务器上，都可以提供服务。这样一来，如果有合适的算法能保障各服务器对同一份数据存储的内容

一致，并且在一台服务的存储服务器故障时，这个集群能以适当的逻辑切换到其他正常服务器提供服务，那么就可以实实在在地保障分布式存储服务的质量。Raft就是为这样的系统服务的。

2.2.2 Raft的特点

简单易学。Paxos算法由Leslie Lamport发表于1990年，是当时最实用的分布式存储一致性算法。 [5]而Raft是由Paxos简化得来。Diego Ongaro与John Ousterhout指出：经过对比，学生学习Paxos所需时间明显长于学习Raft所需时间。

最终一致性。存储一致性根据对于各个服务器的同一份数据之间允许差异的严格程度不同，可以分为强一致性、最终一致性、弱一致性。根据“CAP定理”，一个分布式系统不能同时保障一致性（Consistency）、可用性（Availability）与分区容错性（Partition tolerence）。但是经过权衡，系统可以达到“BASE”效果：基本可用（Basically available）、软状态（Soft state）、最终一致性（Eventually consistent）。Raft所达到的最终一致性，是指各节点上的数据在经过足够的时间之后，最终会达到一致的状态。

2.2.3 Raft的选举

Raft集群各机之间的RPC报文可分为两种：添加条目RPC（AppendEntries RPC，以下简称AE）与请求投票RPC（RequestVote RPC，以下简称RV）。AE是leader用来向follower加entry使用的。RV是“候选人”（candidate，是除了leader、follower以外的第三种状态，只出现于选举时）用来向其他follower要求给自己投票的。

如果超过一定时间，follower检测不到来自leader的周期性心跳消息（leader将不含实际entry的AE作为心跳使用），就会变为candidate状态。此时，Raft集群就会开始选举leader。在Raft中，时间上有着term的概念，表示一个leader的统治期；每个term有着独特的递增的序号，称为term ID。leader会在自己发送的AE中都附上自己的term ID。当新的candidate产生，它会在上一个leader的term ID基础上加1，作为自己的term ID，并在广播给所有其他机的RV中也附上新的term ID。任何非candidate的节点收到了带有比自己已经见过的任何ID更大的term ID的RV时，就会回复这个RV，并更新“自己已经见过的最大ID”。如果同时这个RV中说明的candidate含有的日志足够新（详细说明见下一小节），follower就会为这个candidate投票。这样一来，任何节点就都不会为同一个term ID投两票。如果一个candidate收到了足够的票数（票数加上自己的一票能够占集群的多数），就会开始发送心跳，宣告自己在这个term内的leader地位，开始服务。

由于在一个leader失效时，可能有多个follower超时的时刻相同，发出RV广播的时刻也大致相同，结果在新term都得不到足够的票数，所以candidate存在等票超时与随机等待机制，避免一致冲突，选不出leader。candidate在等足够的票时当然也会看有没有其他candidate宣布胜利。如果都没有发生的话，在一段超时后，candidate们会再开启新term ID，但会随机等待一段时间，然后才广播RV请求投票（广播RV前相当于follower，可以投票）。由于随机等待的时间有长有短，最终一定会由率先结束等待的candidate获胜。

2.2.4　Raft的日志复制

前面提到，Raft集群由leader统一处理所有客户端请求，会将写请求转换为log entry，然后用AE向follower发送来复制log。Leader创建日志条目时，会给它附上两个属性：term ID与log index。term ID即为自己统治期的ID，在当前term的日志条目都用这个ID；而log index也是一种递增序号，但它是在集群的整个运行期间连续的。跨term时，log index会在前面的的基础上递增1，而非归零重计。如此一来，考虑到选举机制保证了一个term ID一定对应确定的一个leader节点，我们由term ID+log index这个组合就一定可以确定唯一的一条日志。

Leader生成了写操作的日志之后，就通过AE将日志条目发送到各个follower处，让它们把日志按早晚顺序加入自己的日志存储中。如果有集群多数的节点（包含leader自己）都成功存储了一条日志（follower会回复自己的存储情况），那么leader就认为这条日志及更早的日志的存储都是安全的，会回复客户端写操作成功，并会告知集群已经可以将到此条的所有日志中的写操作实际进行，修改自己的存储数据。

当leader上台时，会以自己自己的日志存储为准，使其他节点与自己对齐。如果比自己快，就截掉多的；比自己慢，就用自己的日志存储给它慢慢补足。但是在复制日志的过程中，各个follower可能因各种原因速度差异较大。那么如果leader突然故障，而一个复制的特别慢的follower选举为leader，那么不就会导致大量写操作失效吗？之所以要保证日志已经复制到了多数机上，才可认为写操作成功，就是为了避免这种情况。之前在讲解选举机制时提到：RV要包含candidate的日志存储版本消息，也就就是最后一条日志的term ID+log index。如果投票的follower发现这candidate的版本消息比自己的还要旧，就会拒绝给它投票。由于只有复制到了多数节点的日志的写操作才被回报为“成功”，而选举时必须要获得多数票才能当选，所以最终当选的leader一定拥有上一个leader所回报为成功的操作的所有日志，不会缺失。

2.3　本章小结

本章介绍了StellarDB存储模块主要使用的几种技术：用于保存数据、提高写入性能的LSM树数据结构，与用于保障多副本数据一致性的Raft一致性协议。对于LSM树的详细介绍会与系统设计一起包含于下一章。

第三章　StellarDB存储模块的分析与设计

StellarDB是一款包含了图可视化、API外部调用、自我监控管理等丰富功能的数据库系统，可以与星环科技的多款大数据分析软件协作。本章，我们主要分析其存储模块（即图 3.1中的Distributed Graph Storage Engine），并介绍其核心设计：采用LSM树数据结构进行数据存储。

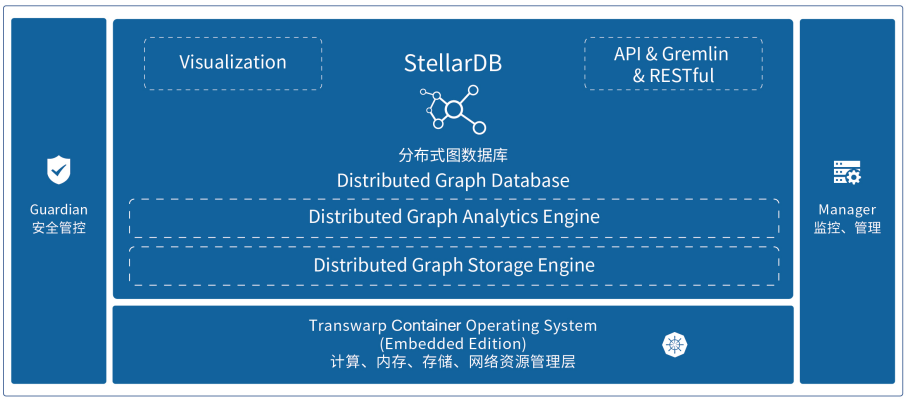


图 3.1: StellarDB宏观架构图

3.1　StellarDB存储模块功能与协作

StellarDB使用Java与Scala语言开发，其存储模块位于io.transwarp.graphsearch.storage程序包。存储模块承接消息处理模块的读写请求，完成异步处理之后发送回复。存储模块同时可以向一些星环产品提供越过正常请求机制的改动存储的接口，以实现一些提高计算速度的底层优化。

存储模块内部包含了基于Raft一致性协议的多副本热备份功能，使得一份写入请求可以应用到多台主机上。当一台主机失效，上层的消息处理模块可以检测到失效的发生，从而将Master地位的主机进行切换，使得客户的读写请求处理不受影响。

在底层，存储模块直接通过文件系统控制对磁盘的读写。存储模块拥有对多块磁盘的使用率平衡功能，使各块磁盘的负载比较均匀，避免负载不均带来的请求处理瓶颈。

3.2　StellarDB存储模块内部设计简述

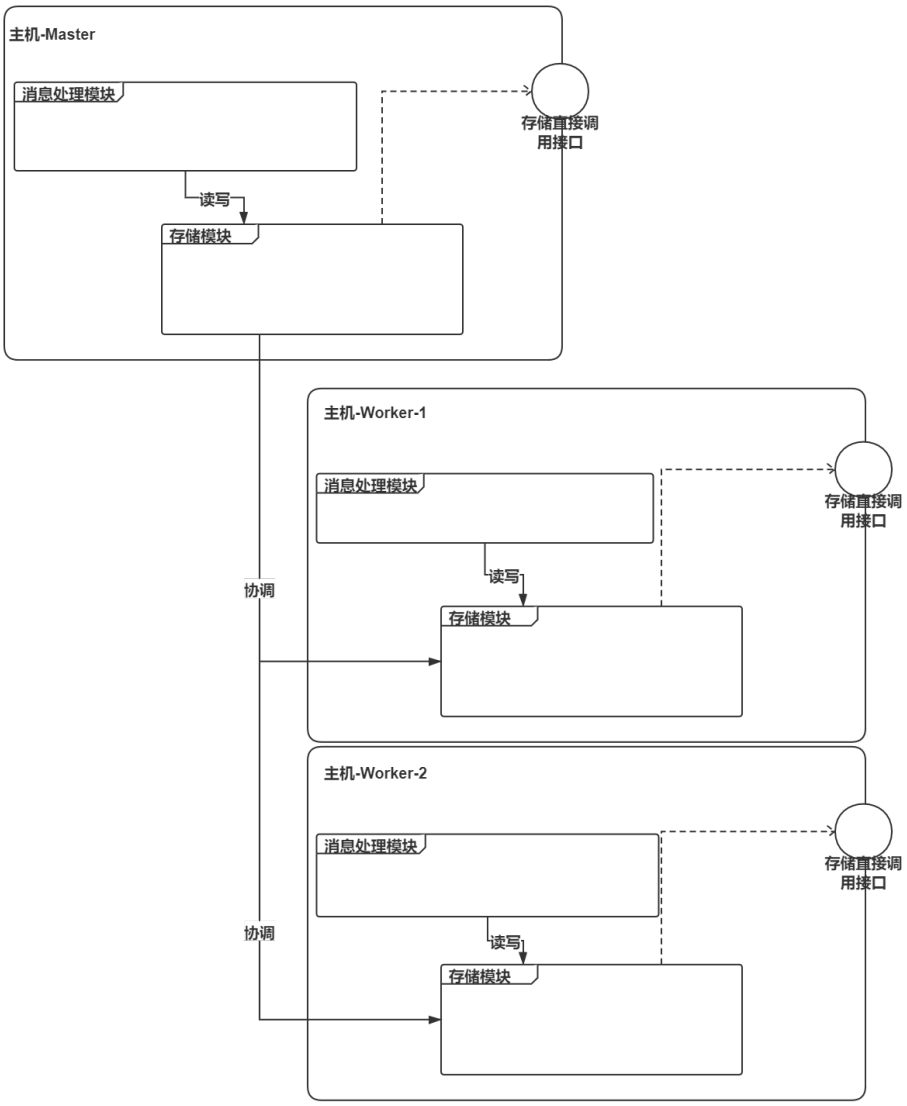


图 3.2: StellarDB存储模块外部协作关系

图 3.3是经过简化后的存储模块核心类设计。图中的箭头表示了单台主机写入数据过程中的数据流动过程。下面以写入过程为例说明StellarDB存储模块内部各类交互过程。

首先，上层的消息处理模块把封装为写入事件的数据发送给GraphDB。在StellarDB中，数据库的每一个图对应一个GraphDB实例。为了实现上文提到的磁盘读写均衡功能，每一个图被划分成多个GraphShard，而数据记录会被按照其哈希值分配到对应的GraphShard中。同一张图的不同GraphShard可以拥有

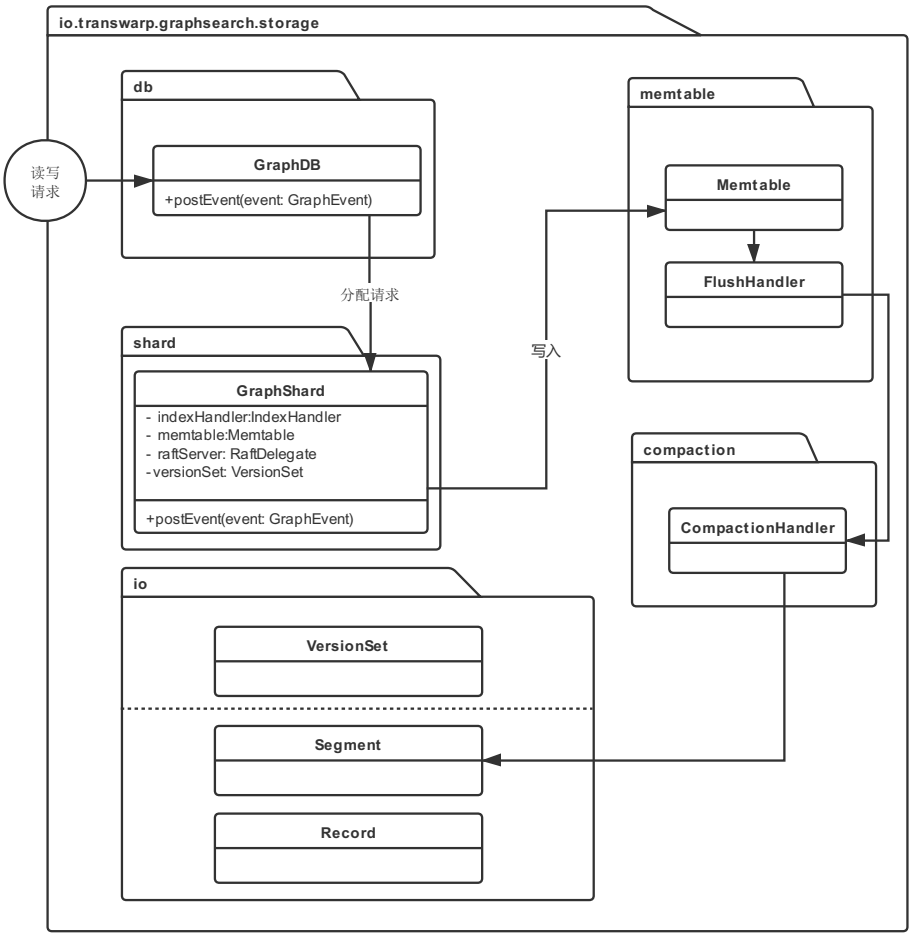


图 3.3: StellarDB存储模块内部协作关系简述

不同磁盘上的的数据存储路径。这样就通过数据的分桶实现了磁盘的充分利用。

StellarDB的多副本备份的单位就是GraphShard。在GraphShard类中，包含了Raft消息同步机制。Raft帮助GraphShard将从GraphDB收到的写入事件在多个主机的同一个GraphShard之间进行同步。所以GraphShard并不会在收到GraphDB传来的写入事件之后立即将事件解析为写入数据。写入动作实际上是由Raft master在集群间同步的写入事件触发的。

通过Raft协议收到了写入事件之后，GraphShard会将数据写入自己的Memtable。虽然LSM树存储是基于磁盘的，但StellarDB选择先将写入数据缓存在内存中，也就是Memtable类，之后通过flush过程将缓存的数据写到磁盘，将其纳入LSM树的维护机制中。CompactionHandler通过定期检查机制与flush、compaction的自动触发对于本GraphShard的LSM树状态进行即时检查。如果发现了进行compaction的

需要，CompactionHandler就会启动compaction任务，使数据流向LSM树下层，或者使LSM树最下层的文件合并。

LSM树存储结构帮助StellarDB实现了读写事件的处理与数据维护过程的异步化。这使得频繁写入操作对于读取操作的性能变得较小。如果能够实现高效的flush、compaction算法，充分利用硬件的性能维护LSM树存储，数据库便可以同时应对高并发的写入与读取请求。

3.3　LSM树的基本维护方式

StellarDB将图的点或边都以键值对的形式作为记录存储，其中主键是字节数组，可以排序；对于数据的额外写入操作，比如修改点或边的属性值、删除某点或边，StellarDB也会把它作为拥有对应的主键的键值对来存储。这样一来，每一条数据的所有写入历史都能够对应LSM树的一个节点。在StellarDB中，数据存储文件称为Segment；在Segment中，数据按照其主键依序存储。所以，Segment就成为了LSM树中的一棵子树。

如图 3.4是StellarDB某GraphShard的数据在磁盘上的分布示例。逻辑上，数据文件被划分在4层中，L0为上层，拥有最新写入的数据；L3为下层，拥有时间上最早写入的数据。如果对于某一条记录A在创建后，又将其删除，那么就可能在下层有“创建A”的一条记录，在上层有“删除A”的一条记录。

记录被划分在L0到L3的不同层次中，对应着节点分布在LSM树的不同高度。最下层的记录，位于LSM树的根；而刚刚写入的记录，则是LSM树的叶节点。需要从LSM树中搜索、读取某一条记录时，就从下层开始查找其主键，一直找到上层，将找到的所有写入记录按照时间顺序合并，就可以得到最新状态的这条记录。这就是从LSM树中读取数据的方法。

显然，在数据库系统的长期运行中，对于某个主键对应的记录，可能会有许多次写入操作。一方面，如果把所有操作记录都存储起来，在查找、合并的时候时间开销会很大，也会占用大量磁盘空间；另一方面，数据的逻辑分层也应该有一定的限制，无限制扩充分层也会给LSM树的维护带来麻烦。所以需要将对于一条记录的操作过程压缩合并，删除无用的历史状态，仅保留最新状态。这种压缩合并的过程便称为compaction。

如图 3.5 ，在L1与L2各有两个数据文件，分隔开的区域表示其中包含的记录，数字表示主键。对于许多主键，L1与L2中都包含与之对应的写入记录。比如对于主键为3的查询，（暂时不考虑可能存在的其他层次）就需要从Segment-3与Segment-1中分别查询到它的写入记录，合并得到最终结果。

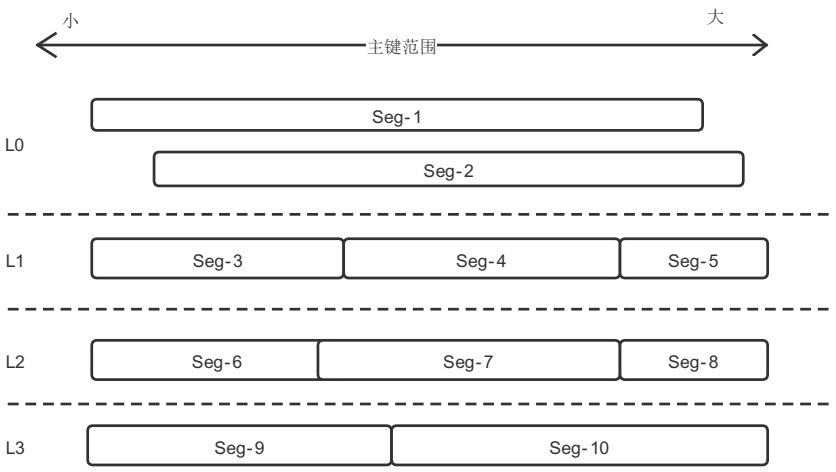


图 3.4: 某Shard内数据文件分布示意图

如果我们想要减少文件数目，压缩存储空间，就需要用新的写入记录去合并或覆盖旧的记录。我们希望将Segment-2消除，将其中的记录全部合并进入下层，就需要找到所有包含有重叠的主键的L2数据文件，把他们当成数据源，在L2中生成一个新的文件Segment-5. 如图 3.6，对于L2中原有的旧条目，我们用来自L1的新纪录将其覆盖，同时保持记录在文件中按主键排序。这样，我们就可以删除三个源文件，只保留一个生成文件。值得注意的是，StellarDB会保持同一层（L0除外）内的各个数据文件彼此记录的主键互不重叠，这样，每次从上下两层取用进行compaction操作的数据文件也在逻辑上是“相邻的”，生成的新的数据文件也就不会与没被取用的数据文件有主键上的范围重叠。

3.4　优化模拟测试程序

对于下一章描述的算法优化，除了在StellarDB中实际的实现与验证，本文还实现了一个对于优化的模拟程序，用于在更严格的条件控制之下测试优化算法效果。

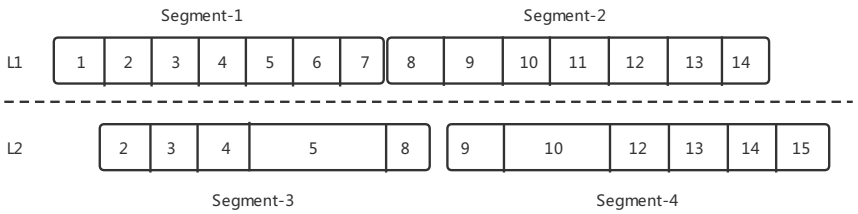


图 3.5: Compaction操作-原状态

3.4.1　目标

StellarDB使用模拟数据在集群环境上进行测试虽然可以展示最接近实际生产环境下的优化效果，但是由于系统内部高度异步化，实际的数据处理能力会受到集群负载的显著影响。当集群同时有其他数据库实例运行，CPU与磁盘负载较大时，性能测试获取的数据容易出现波动的情况。所以为了验证“新实现的flush算法与compaction算法的优化确实能够降低写入过程中的读写扩大”，应当使用事件处理具有确定性的系统模拟StellarDB的算法优化，在一个稳定的计算环境中进行测试，观察读写扩大是否真的发生了改变。

3.4.2　功能设计

程序应对StellarDB存储模块进行抽象简化，去除：

1. 多副本备份功能
2. 多图多GraphShard管理功能
3. 异步响应客户读写请求功能
4. 对于LSM结构进行持久化的功能

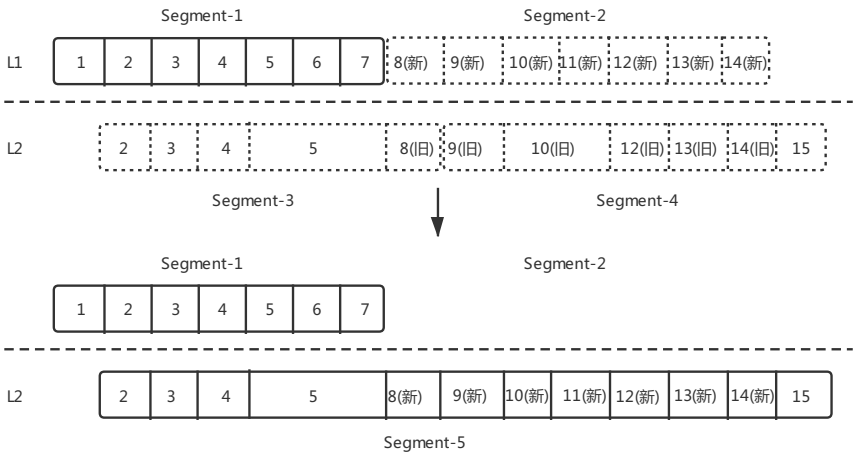


图 3.6: Compaction操作-操作结果

保留：

1. 对LSM树的基本读写功能
2. LSM树的版本管理功能
3. flush功能，包括实现基于跳表的内存缓冲区
4. compaction功能
5. compaction性能统计功能，包含读写数据量的详细统计

3.5　本章小结

本章介绍了StellarDB的存储模块的设计，包括模块功能、内部协作、LSM树的维护方式。这为下一章的性能问题分析铺垫了背景。本章也介绍了对于优化模拟测试程序的设计目的与预期效果的说明。

第四章　优化详细设计实现与结果展示

本章介绍在StellarDB存储模块的一次针对产品在生产环境中遭遇的实际性能问题的优化。

4.1　问题说明

根据同事在客户现场进行产品展示时的经历，团队发现StellarDB在高并发写入操作之后，很长时间内磁盘与CPU占用居高不下，而且读请求响应超时，无法满足客户对系统性能的需求。经过简单排查，发现关键在于compaction性能不足。经过flush过程产生的L0的数据文件在L0累积，无法有效地通过compaction过程进入LSM树的下层。在极端条件下，一个GraphShard会同时管理近两千个L0的数据文件。

4.2　问题分析

在测试过程中，数据的积累只体现在了难以从L0流向L1，而从L1向L2的流动并无明显阻塞。这与L0的特殊性质有关。StellarDB使用内存缓冲区（即Memtable）临时保存写入的记录，等到缓冲区满，再将缓冲区的记录通过flush过程直接写为L0的数据文件。这样的操作使得L0的数据文件不能满足“彼此之间记录的主键范围不重叠”的条件，而LSM树的设计又要求L1的数据必须比L0旧才能保障正确性，所以在从L0向L1的compaction过程中，StellarDB每次会取几个（根据具体参数设置有一个上限）L0的最旧的数据文件，计算他们总共的主键范围，再从L1中找出所有与此范围有交集的数据文件，将它们一起作为源数据进行compaction，生成新的L1数据文件。

但是，根据使用的测试数据（模拟真实社交网络的LDBC数据集），我们发现，一个L0中的文件的记录主键，很可能均匀分布于几乎整个的主键“定义域”。例如：LDBC中的主键都是随机生成的长整形数，首字符为1至9（此处暂不讨论各个字符出现频率），在合理的compaction之后，如果L1有9个文件，那么大致来说应该第1个的文件里面都是“1”打头的主键的条目，第2个文件里面基本都是“2”打头的主键的条目……但是，一个L0文件却同时拥有1至9打头的主键的条目。这就导致每一次的L0到L1的compaction，都需要把所有的L1文件涵盖在输入中。

这样有两个后果：一、每次L0向L1的compaction任务，不论取了几个L0的文件，都会把L1的所有内容重新读一遍、重新写一遍，造成极大的资源浪费；二、虽然compaction线程池足够大，但是单个GraphShard的L0向L1的compaction任务每次只能有一个线程执行（因为所有L1文件都被它占用、上锁），并行度差。

4.3　问题解决思路

Compaction过程中，数据从上层流向下层，上下层的文件都需要被读取、解析、排序、合并，然后重新输出。此时上层的新数据被归入新的层次，可称为有效数据；而下层数据被重新读写一遍，依然回到本层，是一种无效操作。“读写扩大”（Read/Write Amplification）指这种数据流动过程中，“无效操作”所占比重。我们希望通过避免任务占用所有L1数据文件，降低L0向L1的compaction的读写扩大、提高并行度。所以，L0向L的1 compaction任务不应该使用所有的L1文件。在不破坏分层LSM树数据约束的前提下，我们可以把L0文件从“平摊”整个主键定义域限制为仅包含一小部分主键的范围。也就是把图 4.1的情况变成图 4.2。

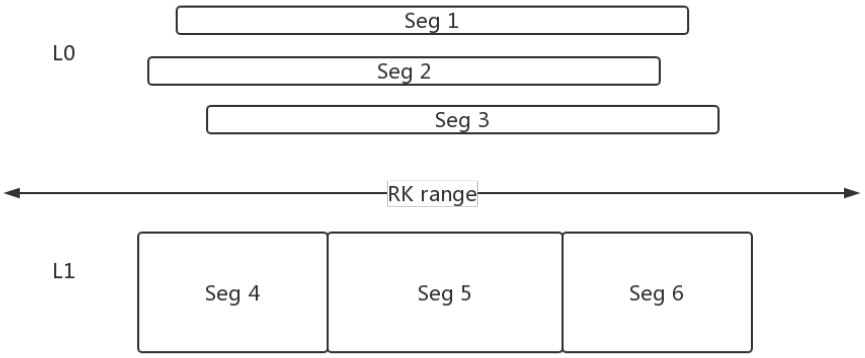


图 4.1: Compaction算法优化前L0与L1结构

如图 4.2,在flush生成L0文件的时候，根据RK1、RK2等主键划分，生成多个L0文件，纳入L0的不同的区间管理。而当需要进行L0向L1的compaction时，可以只取Seg 1+4+7+10这3个文件（注意：区间内L0文件依然有序，可以取1+4+10但不可以取1+7+10），而避免牵扯Seg11、12。这样就减少了由于将L1文件纳入输入而引起的读写扩大。同时，其他compaction线程能够并行做Seg 2+5+8+11、Seg 3+6+9+12的compaction，大大提高了并行度。如果每个compaction任务使用的L0输入数据量上限固定为1个mentable的大小，那么在

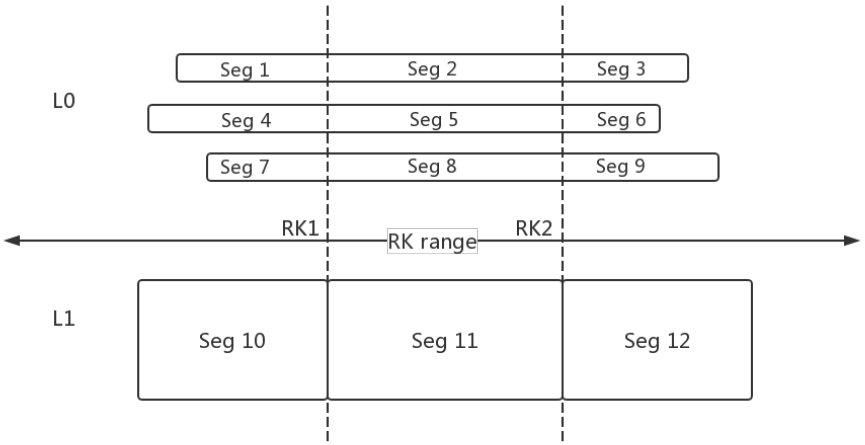


图 4.2: Compaction算法优化后L0与L1结构

理想情况下：

原来图 4.1将Segment 1、2 、3 做compaction进入L1，每个任务会使用的输入的大小为（Memtable大小×1 + L1大小），输出的大小为(L1大小)，串行执行3遍，改进后每个任务会使用的输入大小约为(Memtable大小×1 + L1大小/ 3)，输出的大小约为(L1大小/ 3)，并行执行3遍。很容易看出，如果认为L0数据取用上限远小于L1大小的话，同样的compaction任务在L0分片树为N时，总读写量变成了约1/N，时间开销有潜力变为1/(N ×N).

4.4　算法细节研究

4.4.1　分片数的设置

L0分片数需要实际测试才能得到性能较好的默认值。分片太细的话，要么每次compaction取用的L0小文件太多，排序与合并效率降低；如果要么同步增大Memtable大小，导致L0总大小变大，也影响从L0的查询效率。不过，至少应该保证分片数不少于本GraphShard可用的compaction线程数，使线程得到充分利用。

4.4.2　分片的主键分割点问题

显然，我们需要能够把L0文件比较均匀的划分开的分割点，来达到良好的效率提升效果。那么，分割点的设定值、分割点是静态还是动态就值得研究。

如果能够假设：插入的数据顺序总是随机的，即主键是均匀随机分布于定义域中，那么第一个生成的Memtable的记录就是在定义域内均匀的，我们可以直接按照设定的分片数把这个Memtable按照条目数量等分，就自然获得了一组静态的分割点。这是第一种分割点选择方式。第二种方式是，可以在第一次compaction之前都不使用本分片策略，直到这次compaction完成，拥有了L1数据，这些数据综合了多个Memtable的数据，便可以认为从这些数据计算分割点更为可靠。

最终的优化实现采用了第一种方式。原因主要是第二种方式编码、测试难度高；而根据测试结果，这种方式就能够达到优化效果。

4.4.3　compaction是否需要针对分割点具体优化

有观点认为，只要每次flush时按长度均匀分片，compaction策略完全不变，依然按原来的方式自由地选取L0的被切细的文件，也能达到上面的严格分片的效果。但实际上此种方式在运行中由于种种数据不均衡而可能退化成原来的“每次用上L1全部数据”的情况。

图 4.3: Compaction算法需要针对分割点具体优化

如图 4.3，Seg1、2是更早的flush的产物，如果想要使用Seg 1+3+10+11进行compaction，那么也必须加入Seg 2与Seg 12.因为Seg 3与Seg 2有重合的主键范围，而Seg 2的数据更旧，所以Seg 3的compaction不能早于Seg 2，然后，compaction也自然要包含Seg 12.虽然这种“向旧数据的扩展”的方向不会多米诺骨牌一样连续环环传播（不会引入Seg 4以及Seg 4所依赖的旧数据），但仍有读写扩大的风险。最终的优化实现遵从了“按照分片范围针对性选取数据文件”进行compaction的策略。

4.5　算法优化实现

4.5.1　Flush算法前后对比

Flush算法为优化所作的修改比较简单。如图4.4，优化前，flush使用Memtable的flush()方法，每个Memtable只能固定写入一个数据文件中，导致了之后的读写扩大问题。

如图4.5，经过优化，flush操作根据Memtable大小与分片数，计算每个L0数据文件应该拥有的记录数目，作为一种动态自调节的分割策略。

4.5.2　Compaction算法前后对比

Compaction算法为优化所作的修改相对复杂，主要可以分为两个部分：L0、L1文件选取顺序倒置，新增L0内部文件选取算法。

4.5.2.1　L0、L1文件选取顺序倒置

旧算法先按照时间顺序从L0中取出最旧的文件，计算其主键范围，再从L1中选取所有与此范围重叠的数据文件。实际上，由于原有的并行化设计，L1中的被选取的文件有可能正在进行L1向L2的compaction。此时就要从刚选取的L1文件中剔除正在被占用的文件。根据剔除掉的主键范围再反过来剔除不能被compaction的L0文件。这样会降低文件选取效率。

如图4.6，新算法使用L1文件数量尽量少，会按照“选用0个L1文件”、“选用1个L1文件”、“选用2个L1文件”这样逐渐增加的顺序尝试，计算出选取L0文件时可以使用的主键范围，根据范围尝试计算合适的L0文件选择计划，从中选择读扩大率最小的进行调度。由于是从L1文件开始选取的，就省去了重新检查、剔除L0文件的步骤，使得逻辑大大简化。

4.5.2.2　L0内部文件选择算法

由于在新的flush算法中，一个Memtable（其中的记录视为同一时刻写入的）会被分到多个L0文件中，所以我们给L0文件附上属性“walID”。同一个Memtable生成的L0文件拥有相同的walID；walID越小，数据越久。选取L0文件时，要从旧Segment（walID小）向新Segment（walID大）选。因为每选取完一组walID相同的segment之后，要根据选取的情况缩小可用的主键范围。

如图4.7用英文字母表示数据文件里面包含的条目主键，我们想要尝试计算“在L1中只选取Seg85时，能够包含进compaction任务的L0文件”。首先，Seg85的主键范围时[I,K]，但是实际上可以扩大到(G,N)这个开区间。然后根

据walID（同一个Memtable生成的L0文件享有相同的walID）顺序选择L0文件并缩小主键范围。首先对于walID=100，可以选择I与M，同时没有被选取的L0文件无法影响主键范围；然后处理walID=101，只能选择K+L这一个文件，而且由于没被选入的文件影响，主键范围被缩小到了(J,M)；再看walID=102，发现Segment J+M已经不能选择，因为可选主键范围是(J,M)而不是[J,M]，而且，Segmemt J+M直接把可选主键范围缩减到空集，也就没有必要再搜索walID=103的情况了。

4.6　优化模拟测试程序的设计与实现

本节介绍优化模拟测试程序的详细实现。由于StellarDB的具体实现一定程度上涉及保密制度，上文中的介绍并没有展示详尽的模块局部设计。优化模拟测试程序抽取了设计算法优化的StellarDB存储模块的局部结构。本节的介绍是上文对StellarDB存储模块结构的一项补充。

如图 4.10为简化后的优化模拟测试程序类图，为了信息清晰，其中没有展示抽象接口、算法多版本实现、工厂类等辅助设计，仅包含核心类与其主要的公共方法。

4.6.1　Driver类

Driver类包含了测试数据构造过程与向DB类写入的过程，是程序的入口。这两者的具体实现都与具体的测试用例有关，详见下文的测试用例说明。

4.6.2　DB类

DB类对应着StellarDB的GraphShard，由于不考虑多图多shard管理，所以在使用模拟程序进行测试的时候，所有数据处理都在同一个GraphShard之内，也就是同一棵LSM树之内。

DB类拥有对各关键类的引用，方便各功能类初始化时获取互相之间的引用。DB暴露了一个写入接口，支持向本图（单一GraphShard）内写入一条记录（Record）。它会使用内存缓冲区RecordList，如果缓冲区满，则发送异步事件，请求FlushHandler对于这个内存缓冲区进行flush操作。

4.6.3　Record类与Segment类

作为基本的事务模型，多条数据记录Record顺序排列组成一个数据文件Segment。Record由两个字节数组构成：key与value。在本系统中，Record直

接以“key长度+key+value长度+value”的形式直接编码，存储于Segment之中。Segment类则作为读写时的中介，掌握如何编码、解码数据文件的知识。

4.6.4 FileMeta类

FileMeta类是在系统中，尤其是在flush与compaction过程中对Segment的代理与增强。由于数据文件在LSM树中流动时，会拥有所处层数、文件排序ID、起止记录范围等对于compaction调度很重要的元信息，所以系统使用FileMeta类来存储系统运行时数据文件的元信息。

4.6.5 VersionSet类与Version类

Version类表示其对象被构建出的那一时刻（及之后一段时间）系统的LSM树中的数据文件状态，其中保存了一个FileMeta列表，确保系统能持续追踪管理的数据文件。Version对象从构建完成之后状态就不会改变。VersionSet对于一个DB是唯一的，它引用了系统最新的Version。系统LSM树中的数据文件变动的时候，需要以VersionMod对象描述自己的修改内容，调用VersionSet的修改方法。而VersionSet对于修改请求的处理是同步的且会检查修改是否合法，这就避免了“异步的flush与compaction过程共同修改LSM树”可能带来的多种问题。

4.6.6 Bus类

Bus类负责将系统中的事件分发到其对应的处理器。本系统中总共有3种事件：

1. FlushEvent:当内存缓冲区满，由DB发出的flush请求。事件由FlushHandler处理。

2. CompactionEvent：由FlushHandler或者CompactionHandler发出，提示LSM树某一层有新增文件，可能需要对此层进行compaction。事件由CompactionHandler处理。

3. MetricEvent：由FlushHandler或CompactionHandler发出，提供自己处理的数据处理任务性能统计信息。事件由MetricHandler处理，性能信息在这里汇总并通过日志进行打印。

4.6.7　FlushHandler接口与FlushExecutor接口

FlushHandler响应并调度FlushEvent，为其分配FlushExecutor作为flush操作的实际执行类。优化前后的flush算法分别处于不同的FlushExecutor中。为了确定性，本系统中的Handler都只拥有一个Executor，flush事件彼此之间是串行处理的。

4.6.8　CompactionHandler接口与CompactionExecutor类

与flush操作类似，CompactionHandler的不同版本实现包含优化前或优化后的compaction算法。但由于具体的读写操作是相同的，所以不同版本的CompactionHandler使用了同一个CompactionExecutor类。

4.7　StellarDB对比测试

4.7.1　测试环境

测试环境为星环公司内部开发用集群。使用三台主机，每台主机有24核CPU，使用6块机械硬盘。

4.7.2　测试目标与测试方案

本次测试希望能够测出flush、compaction算法的优化给系统带来的性能提升大小，并且寻找在使用测试数据集条件下的各项参数最优值，以为生产环境中的调参提供参考。

测试方案是使用某StellarDB测试程序进行LDBC数据集 [6]（点边数分别为36485769、231371311）的导入，其效果相当于使用StellarDB客户端进行高并发写入。然后利用监控页面、StellarDB的性能监测RESTful接口与通过log4j打印的日志中的性能信息收集测试数据。在测试中，通过改变配置项：包括代码（配置算法为优化前或优化后）与调节参数验证性能提升并找出最优参数。

如图 4.11是取自对比测试中的一次实验的单机数据，StellarDB的RESTful接口返回的性能信息包含了详细的compaction过程对于数据文件的I/O量、消耗时间等数据。具体的各项参数意义为：

1. ”BeforeLevelIds”：compaction过程是对于LSM树的哪层进行的。图 4.11中的值“[0,1]”指compaction是消耗L0与L1的数据，生成L1数据文件。

2. ”maxTotalBeforeFileSize(MB)”：调度的compaction任务中，最多一次有多少数据量作为输入，以MB为单位。此数据展示了单个compaction任务的最大负载，由于单个compaction任务数据量过大会长时间占用I/O线程。如果大粒度的任务数量过多，就会使得其它I/O任务无法得到即时调度，影响读写请求的响应时间。所以此参数应当控制在合理的上限。

3. ”maxTotalBeforeFileNumber”：调度的compaction任务中，最多一次有多少个数据文件作为输入。它与”maxTotalBeforeFileSize(MB)”影响类似：如果同时读取的文件数过多，会影响系统I/O性能，还会因为需要对于各个文件的记录进行归并排序而占用大量CPU资源。所以此参数也需要控制在合理上限之内。

4. ”totalCompactionCount”：总共进行的compaction任务数。此参数结合总计数目可以计算compaction任务粒度大小。

5. ”totalCompactionCost(ms)”：用于compaction任务的总CPU时间统计，不含调度时间，仅含对数据文件进行读写的时间，单位为毫秒。每个线程分别统计CPU时间，最后累加，所以在多核CPU环境下，这个数比compaction经过的自然时间要长很多倍，但能更精确地体现compaction消耗的CPU资源量。

6. ”totalBeforeFileSizes(MB)”、”totalDeltaFileSize(MB)”、”totalAfterFileSize(MB)”：表示compaction任务读取、输出的数据文件总大小以及过程中减少的数据量。其中读取量按照数据来源层进行了区分。本示例中，compaction过程总计读取了约18733MB的L0数据文件与27797MB的L1数据文件，输出了32305MB的L1文件，减少了14225MB。

7. ”conclusion”中的”avgFileConsume”：展现了按照“消耗的下层数据文件数目”划分后的compcation数据，包含compaction任务数、任务平均使用上层数据文件个数、任务平均用时、读扩大率等。这项统计是专门为了此次性能优化而增加，因为优化算法的中心思想就是减少单次compaction使用的下层数据文件个数。

”conclusion”中“L0至L1读扩大率”计算方式为：

$$RA_{L_0L_1} = \frac{(L1 - total - read - size)_{L_0L_1}}{(L0 - total - read - size)_{L_0L_1}} \tag{4.1}$$

如图 4.12是StellarDB在日志文件中周期性打印的compaction性能信息。由于过去性能测试仅保留了RESTful接口性能数据，没有保留日志，所以此处为近期的某次系统功能集成测试之后的日志示例。日志功能实际与RESTful接口等价，只是能够长久保存，记录从系统启动以来的每一分钟的compaction进度变化。日志以表格形式打印出各层compaction的I/O量、任务量、消耗时间等等，并计算、展示出读写扩大率，使性能监控更直观。此例中由于功能集成测试的写入数据量极为微小，只在几KB的数量级，所以没有体现出读写数据量等数据。

4.7.3 测试结果

测试包含多次对比测试，过程中包含算法修改、参数调整。具体的测试变量与结果如下。

4.7.3.1 测试：LEG

如表 4.1，本次测试使用未优化的算法。这是优化之前StellarDB的默认参数，包括生产环境也使用这些参数。

表 4.1: 测试条件：LEG

	参数值	解释
使用算法	旧	-
Memtable size	2MB	内存缓冲区大小
Compression level	-,-,-,-	是否对各层的数据文件进行压缩，本值表示全部不压缩
Bloom level	-,-,-,-	是否在各层数据文件中加入Bloom过滤器以辅助读操作，本值表示全部不使用Bloom过滤器
Pick file	4,4,8,2	各层compaction时最多使用文件数
File size	16,16,2048,8192	各层单个数据文件最大容量，单位为MB

测试LEG在55分钟自然时间内完成了3/4的L0文件compaction任务，由于耗时过长，不再等待剩下的compaction完成。从L0、L1分别读取文件18718MB、78472MB，读扩大比为419%。如表 4.2展示了按照“使用L1文件个数”分类的compaction任务统计数据。平均每个compaction任务使用3.97个L0文件与2.60个L1文件。419%的读扩大率明显无法满足我们对于高效I/O的要求。

表 4.2: 测试LEG的L0至L1 compaction统计数据

使用L1文件个数	compaction任务数	compaction总用时(ms)
0	51	14413
1	316	891891
2	713	4903831
3	677	7753863
4	554	9264321
5	4	77820

4.7.3.2　测试：EXP1

如表 4.3，本测试开始使用优化后算法。测试条件描述中相较测试LEG缺失的项表示值与LEG中的值相同，不再重复。

本测试使用的优化算法是开发过程中的初步设计：仅包含flush优化而不包含compaction对应优化。测试意图是与之后的包含compaction优化的测试进行对比，说明专门为flush分片开发compaction优化是否有必要。

表 4.3: 测试条件：EXP1

	参数值	解释
使用算法	优化-仅flush	-
Memtable size	10MB	内存缓冲区大小
Flush split	5	单个Memtable被强制分片为5片，这样可以保持单个L0数据文件大小与LEG测试中相同，都为2MB

如表 4.4，对于仅修改flush算法的改动，compaction任务分布并没有明显向“少使用L1文件”的方向倾斜。但是EXP1测试仅花费25分钟即完成了所有L0文件的compaction。平均每个compaction任务使用7.88个L0文件与2.10个L1文件，总容量分别为18702MB和31372MB。168%的读扩大率相比LEG大大改善。

4.7.3.3　测试：EXP2

如表 4.5，同时翻倍分片数与内存缓冲区大小（从而L0文件大小不变），与EXP1对比，观察分片是否越细越好。

EXP2消耗自然时间为23分钟。如表 4.6，在分片数增加之后，compaction的调度更倾向于仅使用较少的L1文件，“仅使用1个L1文件”的compaction任务占到了51.6%。总计消耗L0与L1文件分别18702MB、27468MB，此时读扩大率

表 4.4: 测试EXP1的L0至L1 compaction统计数据

使用L1文件个数	compaction任务数	compaction总用时(ms)
0	40	18393
1	383	1362706
2	365	2821458
3	286	3249375
4	137	2311579
5	4	75456
6	2	57541

表 4.5: 测试条件：EXP2

	参数值	解释
使用算法	优化-仅flush	-
Memtable size	20MB	内存缓冲区大小
Flush split	10	保持单个L0数据文件大小为2MB

为147%。虽然从读扩大率的角度，EXP2相比EXP1改善不明显，但本着最初针对“减少L1文件利用数”的方向，此次调参在之后的测试中也保留。

表 4.6: 测试EXP2的L0至L1 compaction统计数据

使用L1文件个数	compaction任务数	compaction总用时(ms)
0	178	45924
1	948	3242160
2	453	2438499
3	201	1556544
4	51	524535
5	7	104650

4.7.3.4　测试：EXP3

如表 4.7，在EXP2的基础上，引入compaction对应优化，但也并非最终方案：没有对于单次compaction任务中使用L0数据文件的量进行限制。

如表 4.8，引入compaction优化之后，“减少L1文件用量”的效果明显，“仅使用1个L1文件”的compaction任务占到了92.5%。总计消耗L0与L1文件分别18718MB、15324MB，此时读扩大率降低到82%。测试中，数据文科可以顺畅地从L0流动到L3，而没有观察到在L0的大量积压。可以说，此时优化效果已经相当优秀，

表 4.7: 测试条件：EXP3

	参数值	解释
使用算法	优化-flush与compaction-1	加入compaction初步优化
Memtable size	20MB	内存缓冲区大小
Flush split	10	保持单个L0数据文件大小为2MB

目标已初步完成。

但是，相比之前的测试，此次测试中出现了巨大粒度的compaction任务：有的任务总读取量多达278MB，甚至造成了统计数据更新缓慢。这个现象如果出现在生产环境中，会给运维人员带来困惑。所以需要修改算法，增加最大任务粒度限制。

表 4.8: 测试EXP3的L0至L1 compaction统计数据

使用L1文件个数	compaction任务数	compaction总用时(ms)
0	24	2496
1	1260	4654440
2	64	190912
3	11	56309
4	3	13935

4.7.3.5　测试：EXP4

如表 4.9，在EXP3的算法基础上，加入单次任务容量的间接限制：限制读取上层数据文件与下层数据文件的容量比例。由于算法追求尽量少读取下层数据文件，所以上层数据文件的用量也得到了控制。这个比例被固定限制为不大于5.0，不小于0.5.

表 4.9: 测试条件：EXP4

	参数值	解释
使用算法	优化-flush与compaction-2	为compaction任务读取的上下层数据文件大小比例增加限制
Memtable size	20MB	内存缓冲区大小
Flush split	10	保持单个L0数据文件大小为2MB

加入限制条件之后，最大compaction任务规模下降至90MB，而任务分布基

本保持不变。L0与L1的文件消耗分别为18569MB与16470MB。88.7%的读扩大率虽然略微高于EXP3的结果，但依然优秀。

本次测试中，观察到存在少量极小的L0文件一直滞留L0，无论经过多长时间都不会被compaction，这是因为增加的限制使得过小的L0文件无法被调度。

4.7.3.6　测试：EXP5

如表 4.10，在EXP4的基础上，进一步优化L0文件选择策略：“从仅使用一个L1文件开始搜索调度方案”改为“从不使用L1文件开始搜索调度方案”。这样可以进一步降低总读扩大率。

表 4.10: 测试条件：EXP5

	参数值	解释
使用算法	优化-flush与compaction-3	尝试优先搜索不使用L1数据文件的compaction调度方案
Memtable size	20MB	内存缓冲区大小
Flush split	10	保持单个L0数据文件大小为2MB
Compaction thread	15,15,4,6	保持单个L0数据文件大小为2MB

如表 4.11，此次优化后，“不使用L1文件”的compaction任务数目大大增加。L0、L1文件消耗量分别为17470MB、13815MB，读扩大率为79%。相对EXP4，优化效果更进一步，但也没有解决EXP4中体现出的文件滞留L0的bug。

表 4.11: 测试EXP5的L0至L1 compaction统计数据

使用L1文件个数	compaction任务数	compaction总用时(ms)
0	767	71606
1	9353	5079851
2	252	272166
3	94	98361
4	67	40101
5	3	6555

4.7.3.7　测试：EXP6

如表 4.12，在EXP5的基础上，在compaction任务调度时再次引入“单次任务使用L0文件数目”限制，与最初的算法不同，这个限制既包括上限也包括下

限。这样既能解决EXP4、EXP5中展现出来的部分L0文件compaction受阻的问题（EXP4中的条件需要与本条件同时不满足时才会停止L0文件的选取），也可以顺便对于compaction任务粒度做出限制。

表 4.12: 测试条件：EXP6

	参数值	解释
使用算法	优化-flush与compaction-4	引入新版本的L0文件读取数目限制
Memtable size	20MB	内存缓冲区大小
Flush split	10	保持单个L0数据文件大小为2MB
Compaction thread	15,15,4,6	各层compaction可用线程数

在综合修复了各个算法缺陷之后，最终算法的性能略有折损：L0与L1文件分别读取量为18633MB、22225MB，读扩大率为119%。但是由于已经消除了compaction的性能瓶颈，此优化结果也得到了组内的认可。

4.7.4　分析总结

根据上面的对比测试结果，可以看出：

1. Compaction优化算法有效降低了L0到L1的compaction过程的的读扩大率。这意味着在数据总量一定的情况下，LSM树的维护所需的磁盘I/O大幅减少，从而为各种形式的请求处理节省了计算资源。
2. 清空L0用时大幅缩短，这意味着记录能够在更短时间内以整体有序的状态存在于L1以及更低层中。这是高效响应读请求的基础。Compaction性能的改善会同时提高高并发写入情境下StellarDB对读请求的处理能力。
3. 需要额外的compaction任务粒度限制。一方面保障读扩大率的低水平，另一方面避免过大任务导致处理阻塞。
4. 内存缓冲区分片数设为10可以较好地配合优化算法，降低读扩大率。
5. 应该为compaction过程提供足够的I/O线程，保障其顺利进行。

4.8　模拟程序对比测试

4.8.1　测试环境

测试环境为私人笔记本电脑，使用单块机械硬盘与16核心2.6GHz主频CPU。

4.8.2　测试目标与测试方案

测试关注在compaction过程中，从L0到L1的读写数据量，从而计算不同的算法的读写放大效果，验证优化后算法可以降低读写放大，降低磁盘负载。

测试使用控制变量的方式，对“是否采用优化算法”与“模拟数据插入顺序”这两个变量进行交叉测试，同时保持其他变量不变，总共进行了4次测试。测试结果关注L0到L1的compaction过程的读写放大率。

4.8.2.1　变量：是否采用优化算法

由于已经拥有在StellarDB上进行算法调优的丰富经验，本模拟程序直接实现了两个版本的优化算法：StellarDB最初的flush、compaction算法，和已经整合了各种任务粒度控制方式、细节优化手段的最终版flush、compaction算法。

4.8.2.2　变量：模拟数据插入顺序

优化算法的优化效果基于“数据随机无序插入”的假设。本次实验有两种模拟数据插入方式。首先，模拟数据记录的key与value值相同，都是长度为7的字节数组。将数组分为长度为4的前缀与长度为3的后缀，前缀取值为“0001”至“5000”，后缀取值为“001”至“300”，总计为1500000条模拟记录。一种插入顺序为对每一个后缀遍历所有前缀，这样不论从整体看还是从内存缓冲区对记录进行排序的视角看，插入是高度无序的；另一种插入顺序为对每一个前缀遍历所有后缀，这样的插入方式就是完全按照key顺序插入的。

4.8.2.3　不变量

1. 内存缓冲区容量：5000条记录
2. Flush分片数：4
3. 触发各层compaction的文件数量：5,5,10,4
4. 各层单个数据文件最大容量（单位为MB）：2,2,4,8

4.8.3　测试结果

如表 4.13，在完全有序的插入方式下，不论是否进行优化，由于L0新加入记录一定与L1内容不重叠，所以compaction不会使用到L1文件。而当无序插入时，优化算法带来了I/O效率的提升。由于本测试中数据量少、内存缓冲区设置小，每个L0文件大小仅为76KB（分片后为19KB），相比2MB的L1文件体积相当

小，所以总体读扩大率高于StellarDB性能测试中的数据。但在这样的条件下，读扩大率的变化更为明显。

优化算法中L0消耗略低于未优化算法是因为其实现仅当L0文件数不少于5个才会触发。这个差异不影响算法正确性与测试结论。

表 4.13: 优化模拟程序测试结果

是否采用优化	插入方式	L0消耗（KB）	L1消耗（KB）	读扩大率
是	无序	21848	142394	652%
否	无序	23602	435642	1845%
是	有序	21848	0	N/A
否	有序	23602	0	N/A

4.9　本章小结

本章详细介绍了StellarDB一次针对compaction性能瓶颈的问题分析、解决思路、算法设计实现，以及优化算法模拟程序的设计实现。本章也通过详尽的测试说明了算法的逐步优化过程，证明了优化的有效性。

```
void flush(Segment segment) throws IOException {
  SkipIterator<GraphRecord> iter = skipList.toIterator(-1);

  while (iter.hasNext()) {
    segment.put(iter.next());
  }

  segment.flush();
}
```

图 4.4: Flush算法优化前代码

```
private long expectedFileSizeLimit = memtableSize / flushSplitCount;
…
void doExperimentalFlushing(GraphMemTableFlushEvent event) throws IOException {
    …
    SkipIterator<GraphRecord> flushingIterator = memTable.getFlushingIterator();
    while (flushingIterator.hasNext()) {
      long newSegmentId = versionSet.getNextSegmentId();
      String newSegmentPath = StorageUtils.getSegmentPath(segmentDirectory, newSegmentId);
      Segment segment = SegmentFactory.getWritableSegment(…);

      long flushedCount = 0;
      while (flushedCount < expectedFileSizeLimit && flushingIterator.hasNext()) {
        GraphRecord merged = flushingIterator.next();
        segment.put(merged);
        flushedCount += merged.getRecordSize();
      }
      segment.flush();
      segments.add(segment);
    }
    flushingIterator.dispose();
    …
}
```

图 4.5: Flush算法优化后代码

```
basicFiles.sort((f1, f2) -> { // L0 文件实质上内部根据 flush 时间分成了多层
    int majorCampare = f1.getMajorId() - f2.getMajorId();
    if (majorCampare != 0) {
        return majorCampare; // 先按照 flush 时间排序
    } else {
        return (int) (f1.getFileId() - f2.getFileId()); // 再按照主键顺序排序
    }
});

if (level1Files.size() == 0) { // 当 L1 为空，则可以将所有 L0 文件同时当作输入源
    upperLevelFiles = basicFiles;
    lowerLevelFiles = null;
} else { // 否则就根据 L1 的文件主键范围，拟定从 L0 中选取的文件
    double ra = Double.MAX_VALUE;

    for (int viceCount = 0; viceCount <= level1Files.size(); viceCount++) {
        for (int ind = 0; ind <= level1Files.size() - viceCount; ind++) {
            int leftInd = ind, rightInd = ind + viceCount - 1;
            byte[] minRowKey = ···; // 根据选取的 L1 文件，计算出 L0 文件主键最小值
            byte[] maxRowKey = ···; // 根据选取的 L1 文件，计算出 L0 文件主键最大值

            long viceSize = 0L;

            // calcPlan 实现了“L0 文件选择算法”
            List<FileMeta> thisPlan = calcPlan(minRowKey, maxRowKey, basicFiles);
            if (thisPlan.size() > 0) {
                long currentLevelSize = 0L;
                for (FileMeta l0Pick : thisPlan) {
                    currentLevelSize += l0Pick.getFileSize();
                }
                // 每次计算选取计划的读扩大率，比较以找到能最小化读扩大率的选取方式
                double thisRA = viceSize * 1.0 / currentLevelSize;
                if (lowerLevelFiles == null || thisRA < ra) {
                    lowerLevelFiles = new ArrayList<>();
                    for (int j = leftInd; j <= rightInd; j++) {
                        lowerLevelFiles.add(level1Files.get(j));
                    }
                    upperLevelFiles = thisPlan;
                    ra = thisRA;
                }
            }
        }
    }
}
```

图 4.6: Compaction算法优化：L0、L1文件选取顺序倒置

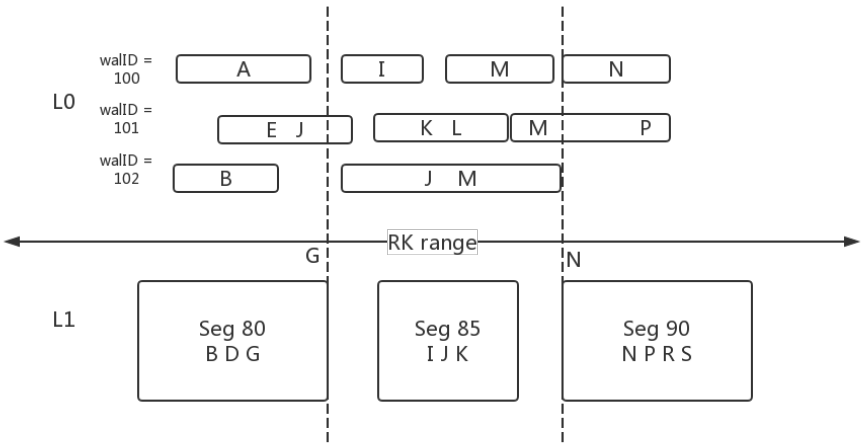


图 4.7: Compaction算法优化：L0内部文件选择情景示例

```java
/**
 * 根据允许选取的主键范围，筛选出可以参与 compaction 的 L0 文件
 * @param minRowKey 主键最小值
 * @param maxRowKey 主键最大值
 * @param level0Files 所有 L0 文件
 * @return 筛选出的可进行 compaction 的 L0 文件
 */
private List<FileMeta> calcPlan(byte[] minRowKey, byte[] maxRowKey, List<FileMeta>
level0Files) {
    List<FileMeta> ret = new ArrayList<>();
    if (level0Files.size() == 0 || CommonUtils.compareByteArray(minRowKey, maxRowKey) >= 0)
return ret;

    long totalSize = 0L;
    byte[] lBorder = minRowKey, rBorder = maxRowKey;

    for (int ind = 0; ind < level0Files.size(); ind++) {

        long currentMajorId = level0Files.get(ind).getMajorId();

        // Process level0Files with same walId and find the segments that can be added
        List<Integer> addedInd = new LinkedList<>();
        while (ind < level0Files.size() && level0Files.get(ind).getMajorId() ==
currentMajorId) {
            FileMeta l0File = level0Files.get(ind);
            if (CommonUtils.contains(lBorder, rBorder,
                    l0File.getStartRecord().getKey(),
                    l0File.getEndRecord().getKey(),
                    true, true)) {
                totalSize += l0File.getFileSize();
                addedInd.add(ind);
                ret.add(l0File);

            }
            ind++;
        }
        ind--;

        // Update lBorder and rBorder
        if (addedInd.size() > 0) {
            Integer firstAdded = addedInd.get(0);
            Integer lastAdded = addedInd.get(addedInd.size() - 1);

            if (firstAdded > 0) {
                FileMeta preAdd = level0Files.get(firstAdded - 1);
                if (preAdd.getMajorId() == currentMajorId) {
```

图 4.8: Compaction算法优化：L0内部文件选择算法源码

```
                    lBorder = CommonUtils.max(lBorder, preAdd.getEndRecord().getKey());
                }
            }
            if (lastAdded < level0Files.size() - 1) {
                FileMeta postAdd = level0Files.get(lastAdded + 1);
                if (postAdd.getMajorId() == currentMajorId) {
                    rBorder = CommonUtils.min(rBorder, postAdd.getStartRecord().getKey());
                }
            }
        } else { // No file with this walId is usable.
            // Iterate from right to left
            for (int i = ind; i >= 0 && level0Files.get(i).getMajorId() == currentMajorId; i--) {
                FileMeta fileMeta = level0Files.get(i);
                // If there is a segment fully covers the row key range including the border, then the range is narrowed to 0.
                if (CommonUtils.contains(fileMeta.getStartRecord().getKey(),
                        fileMeta.getEndRecord().getKey(),
                        lBorder, rBorder)) {
                    return ret;
                }

                int startKeyCompRBorder = …;
                int startKeyCompLBorder = …;
                int endKeyCompRBorder = …;
                int endKeyCompLBorder = …;
                if (startKeyCompRBorder >= 0 || endKeyCompLBorder <= 0) {
                    // Such segment doesn't overlap with row key range, and can't influence the row key range.
                } else { // And these ones overlap with (lBorder, rBorder). If it fully covers the row key range including the border, then the range is narrowed to 0.
                    if (startKeyCompLBorder <= 0 && endKeyCompRBorder >= 0) {
                        return ret;
                    }
                    if (endKeyCompRBorder < 0) { // So startKeyCompLBorder is guaranteed to be <=0, because this segment is not usable.
                        lBorder = fileMeta.getEndRecord().getKey();
                    } else {
                        // startKeyCompLBorder > 0
                        rBorder = fileMeta.getStartRecord().getKey();
                    }
                }
            }
        }
        if (CommonUtils.compareByteArray(lBorder, rBorder) >= 0) {
            break;
        }
    }
    return ret;
}
```

图 4.9: Compaction算法优化：L0内部文件选择算法源码（续）

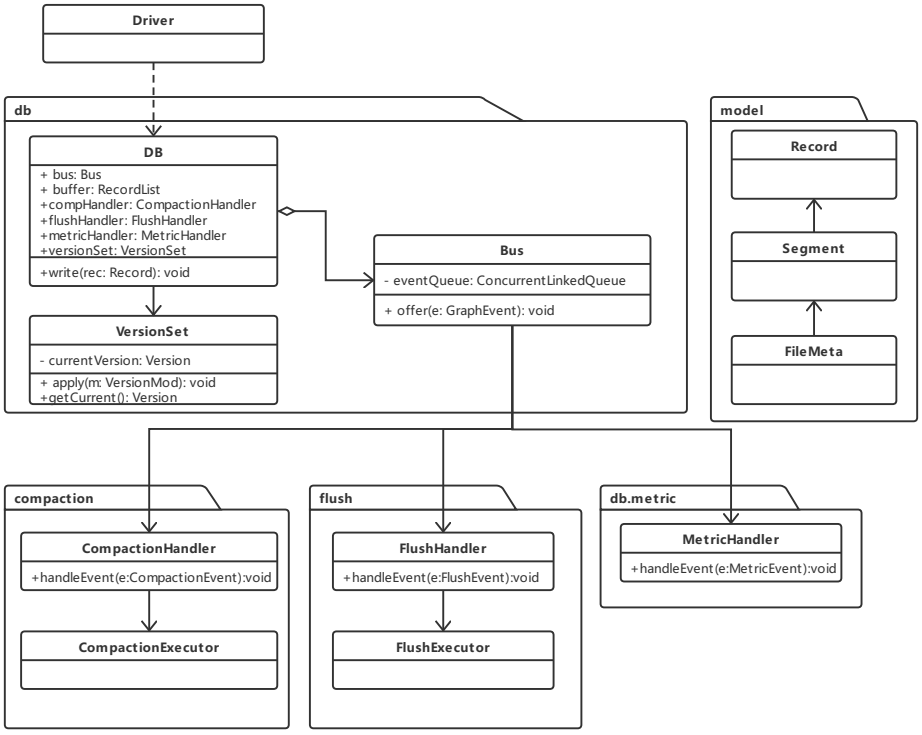
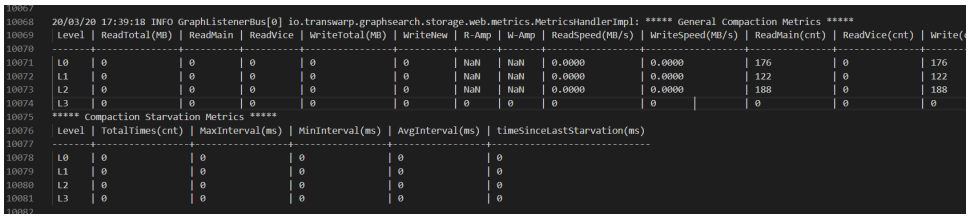


图 4.10: 优化模拟测试程序类图（简化版）

```
{
    "requestTime": 0,
    "data": {
        "resultDataType": 3,
        "value": {
            "lastUpdateTimestamp": "2019-06-18 03:52:30",
            "singleMetric": {
                "BeforeLevelIds": [
                    0,
                    1
                ],
                "maxTotalBeforeFileSize(MB)": 142.0456304550171,
                "totalDeltaFileSize(MB)": -14225.996609687805,
                "maxDeltaFileSize(MB)": 0,
                "maxCompactionCost(ms)": 26503,
                "compactionTimeByViceSegmentCount": {
                    "0": 52849,
                    "1": 3318137,
                    "2": 2811514,
                    "3": 1820163,
                    "4": 719644,
                    "5": 41170
                },
                "consumeMainSizeByViceSegmentCount": {
                    "0": 1370.1176319122314,
                    "1": 7986.782918930054,
                    "2": 5502.717299461365,
                    "3": 2733.948058128357,
                    "4": 1077.5102272033691,
                    "5": 62.908509254455566
                },
                "maxTotalBeforeFileNumber": 59,
                "AfterLevelId": 1,
                "totalBeforeFileSizes(MB)": [
                    18733.98464488983,
                    27797.365659713745
                ],
                ...
            }
        }
    }
}
```


图 4.11: StellarDB的RESTful接口返回的性能信息示例



```
20/03/20 17:39:18 INFO GraphListenerBus[0] io.transwarp.graphsearch.storage.web.metrics.MetricsHandlerImpl: ***** General Compaction Metrics *****
| Level | ReadTotal(MB) | ReadMain | ReadVice | WriteTotal(MB) | WriteNew | R-Amp | W-Amp | ReadSpeed(MB/s) | WriteSpeed(MB/s) | ReadMain(cnt) | ReadVice(cnt) | Write(
|-------+---------------+----------+----------+----------------+----------+-------+-------+-----------------+------------------+---------------+---------------+-------
| L0    | 0             | 0        | 0        | 0              | 0        | NaN   | NaN   | 0.0000          | 0.0000           | 176           | 0             | 176
| L1    | 0             | 0        | 0        | 0              | 0        | NaN   | NaN   | 0.0000          | 0.0000           | 122           | 0             | 122
| L2    | 0             | 0        | 0        | 0              | 0        | NaN   | NaN   | 0.0000          | 0.0000           | 188           | 0             | 188
| L3    | 0             | 0        | 0        | 0              | 0        | 0     | 0     | 0               | 0                | 0             | 0             | 0
***** Compaction Starvation Metrics *****
| Level | TotalTimes(cnt) | MaxInterval(ms) | MinInterval(ms) | AvgInterval(ms) | timeSinceLastStarvation(ms)
|-------+-----------------+-----------------+-----------------+-----------------+----------------------------
| L0    | 0               | 0               | 0               | 0               | 0
| L1    | 0               | 0               | 0               | 0               | 0
| L2    | 0               | 0               | 0               | 0               | 0
| L3    | 0               | 0               | 0               | 0               | 0
```

图 4.12: StellarDB的使用Log4j以表格形式打印性能信息

第五章　总结与展望

5.1　总结

1. 本文介绍的compaction算法是通过对最初的“读请求响应超时问题”反复测试、仔细分析后，发现性能瓶颈从而针对性能瓶颈做优化而得到的。这个过程中，离不开本人预先对StellarDB系统的性能监测功能的完善。所以，处理性能优化问题时，需要重视系统的性能统计功能。如果现有的统计能力不能满足问题分析的需求，则应当现场开发统计功能。

2. “通过分割Memtable给compaction降低读扩大率提供条件”的方案看起来并不直接，像是一种“曲线救国”，但实际上也是根据“读扩大率过高导致大量浪费的I/O”这一现象层层推导得到：为了节省I/O，降低读扩大率，就要减少compaction任务中对L1文件的使用，增加对L0文件的使用。但在随机写入的环境下，现有的flush策略使得所有L1文件都会被使用，所以要先修改flush策略，使调度仅包含少量L1文件的compaction任务变成可能。所以在flush的时候就要控制单个L0文件的主键范围。优化性能需要用逻辑推导解决方案，并做好推翻一些看似理所应当的设计的准备，这样才能发现创新性的解决方案。

3. 性能优化最终要回归性能数据才有说服力。这次写入性能优化除了compaction算法，实际上还包含了许多参数调节，伴随着多次对比性能测试。这些没有在本文中提及。正是详实的性能数据才能证明优化的有效性。尤其是调参带来的性能变化的数据，对于系统部署时的性能调优更是一笔宝贵的财富。

5.2　展望

本文所介绍的优化算法在约半年前便已经完成开发、测试，开始在StellarDB的稳定版本内发挥作用。客户的使用反馈证实了此优化确实解决了之前的高并发写入后无法进行读请求的问题。但是对于LSM树写入维护的优化工作从未停止。本优化中的“L0与L1文件选取顺序倒置”的方法应当也可以应用于更底层的compaction过程，从而在底层也实现降低读扩大率的效果。进行性能测试的过程中，有时会出现下层compaction或最底层文件互相合并的操作造成大

量I/O浪费的情况，虽然不会对读请求处理产生显著影响，但也是未来优化的方向。

参考文献

[1] 星环科技, Stellardb产品白皮书, http://transwarp.io/transwarp/product-StellarDB.html?categoryId=21.

[2] P. E. O'Neil, E. Cheng, D. Gawlick, E. J. O'Neil, The log-structured merge-tree (lsm-tree), Acta Inf. 33 (4) (1996) 351–385.
URL https://doi.org/10.1007/s002360050048

[3] J. Ellis, Leveled compaction in apache cassandra, https://www.datastax.com/blog/2011/10/leveled-compaction-apache-cassandra.

[4] D. Ongaro, J. K. Ousterhout, In search of an understandable consensus algorithm, in: G. Gibson, N. Zeldovich (Eds.), 2014 USENIX Annual Technical Conference, USENIX ATC '14, Philadelphia, PA, USA, June 19-20, 2014, USENIX Association, 2014, pp. 305–319.
URL https://www.usenix.org/conference/atc14/technical-sessions/presentation/ongaro

[5] L. Lamport, The part-time parliament, ACM Trans. Comput. Syst. 16 (2) (1998) 133–169.
URL https://doi.org/10.1145/279227.279229

[6] ldbc, Synthetic data generator for the ldbc social network benchmark and ldbc graphalytics, https://github.com/ldbc/ldbc_snb_datagen.

简历与科研成果

基本情况 徐江河，男，汉族，1996 年 3 月出生，山东省淄博市人。

教育背景

2018.9～2020.6 南京大学软件学院 硕士

2014.9～2018.7 南京大学软件学院 本科

致　　谢

在本篇论文完成之际，也就意味着我在南京大学软件学院的学习生涯也将要结束了。我要向曾经帮助过我的导师、同学和朋友致以最诚挚的感谢。首先要感谢我的论文指导老师邵栋老师。在整个论文写作的过程中，邵老师对我耐心帮助，在论文不足之处他总是及时耐心地提出修改和优化的建议。每一次和邵老师交谈，我都能感受到邵老师作为师长的责任与担当。邵老师对待学术严谨的态度和平易近人的生活作风深深的影响着我，使我受益终身。

同时，我要感谢实习期间给予我帮助的”buddy”王志平和公司给予我帮助的所有同事们。他们在项目完成和技术指导上给了我很大的帮助，感谢他们在忙于工作的同时还能抽出时间对我进行指导。这段经历，我相信在我以后的工作生活中一定会有很大的借鉴意义。

最后我要感谢南京大学软件学院和南京大学软件学院所有的教职工人员。是他们为能来到南京软件学院读研的同学们提供了一个开放的包容的成长环境，他们高远的学术视野和严谨的工作作风深深的影响着每一个南京大学软件学院的学子们。在南京读硕士的这两年，让我对自己的认识更清晰，对自己的未来更有信心。谢谢大家!