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与实现

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：

论文答辩日期：**20xx 年 x 月 xx 日**

指 导 教 师：　　（签字）

The Design and Implementation of Performance Optimization on Storage Module of StellarDB

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与实现

工程硕士（软件工程领域） 专业 2018 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，成为越来越重要的基础软件。国内外许多软件厂商都推出了自己的图数据库产品。StellarDB是星环公司的图数据库，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，StellarDB遭遇了严重的性能的问题。在进行大数据集高并发写入之后，StellarDB性能急剧下降。虽然数据库节点还处于活跃状态，但客户端发出查询请求之后，无法在预期时间内接收到回复。原定的响应超时时间为30秒，但此时的StellarDB实际需要数分钟才能返回查询结果。

本文介绍了对问题的分析过程：通过对性能数据的观察，可以看到数据堆积在LSM树的最顶层，向下流动速度极慢，导致读请求搜索数据条目主键的时候需要访问多达上千个数据文件，从而使系统无法在预期时间内回复。这是因为在数据流动过程中，由于LSM树结构本身的固有维护方式，存在大量对旧数据的重新读写，造成了磁盘读写的浪费。而此情景下磁盘已经发挥最大读写速度，所以需要提升对磁盘读写的利用效率，减少浪费。

据此思路，本文实现了减少读写浪费的优化方案。首先，通过修改flush算法对内存缓冲区进行强制分片，优化文件中的数据分布，使得单个L0数据文件主键范围缩小，方便compaction调度。其次，实现更精细compaction算法，根据下层数据文件的主键范围选择尽量多的上层数据文件，并计算I/O浪费最低的方案，能降低compaction过程中的磁盘I/O浪费，加速数据在LSM树向下流动。本文详细说明了对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，用于测试、验证优化算法的有效性。

最终测试结果显示，从L0到L1的compaction读扩大率从419%下降至119%，数据不再堆积于顶层，各层compaction过程顺畅，数据能够较快流动到底层。

经过优化，StellarDB在高并发大数据集写入情景下的性能问题已经解决，系统能够及时正常响应读请求。目前优化后算法已经在StellarDB中运行，性能稳定。

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 Performance Optimization on Storage Module of StellarDB

SPECIALIZATION: Software Engineering

POSTGRADUATE: (Author Name)

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

Abstract

Nowadays, graph databases are becoming increasingly important as basic software due to their modeling advantages in real-life production scenarios and their performance advantages in relationship finding. Many software companys, countrywide or worldwide, are building their own graph database systems. StellarDB is the graph database of Transwarp Inc., which can be combined with other components to achieve important functions such as graph storage, graph analysis, graph visualization, etc. StellarDB storage engine uses Log-Structured Merge-Tree (LSM-tree) data structure as the underlying data storage method.

However, during the actual deployment, StellarDB encountered serious performance issues. StellarDB performance dropped dramatically after high concurrency writes of large data sets. Although the database node is still active, the client cannot receive a response within the expected time after sending a query request. The original response timeout was 30 seconds, but StellarDB actually took several minutes to return the query results.

The thesis describes the process of analyzing the problem: by looking at the performance data, it can be seen that the data piles up at the top of the LSM-tree and flows downwards very slowly, resulting in the need to access up to thousands of data files when reading requests to search for the primary key of a data entry, which prevents the system from responding in the expected time. This is because in the data flow process, due to the inherent maintenance of the LSM-tree structure itself, there is a lot of re-write of old data, resulting in the waste of disk reading and writing. And in this scenario disk has played the maximum read and write speed, so it need to enhance the utilization efficiency of the disk read and write, and reduce the waste.

With this method, the thesis implemented the following optimization. Firstly, the new flush algorithm forces memory buffer to be sliced, making the record key range of data files in L0 become narrow. By optimizing files' data distribution, scheduling compaction tasks became easier. Secondly, the new compaction algorithm can choose files in upper level basing on lower level key range and calculate the compaction schedule plan with least I/O waste. Then the thesis demostrated results of performance test 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.

Based on this idea, the thesis achieves an optimization scheme to reduce read and write waste. First, by modifying the flush algorithm to force slicing of the memory buffer and optimize the data distribution in the file, the primary key range of a single L0 data file is narrowed, facilitating compaction scheduling. Secondly, to achieve a more refined compaction algorithm, select as many upper-level data files as possible according to the primary key range of the lower-level data files, and calculate the scheme with the lowest I/O waste. So StellarDB can reduce the disk I/O waste in the compaction process and accelerate the data flow down the LSM tree. The thesis details the design and implementation of a joint innovative optimization scheme for the flush algorithm and compaction algorithm, and presents the results of several performance tests during the optimization process to explain the basis and effect of the algorithm step-by-step optimization. In addition, the thesis implements a storage module simulation program for retesting and verifying the validity of the optimization algorithm by abstracting and simplifying the storage module of StellarDB.

The final test results showed that the compaction read amplification rate from L0 to L1 dropped from 419% to 119%, the data no longer piled up on the top layer, the compaction process at each layer was smooth, and the data could flow to the bottom layer relatively quickly. After optimization, StellarDB's performance issues in high concurrent large data set write scenarios have been resolved and the system is able to respond to read requests in a timely and normal manner. The optimized algorithm is currently running in StellarDB with stable performance.

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

随着信息技术的发展，人类采集、分析数据的能力越来越强，逐渐迈进大数据时代。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。图数据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，具有极强的现实意义。如图 1.1，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。

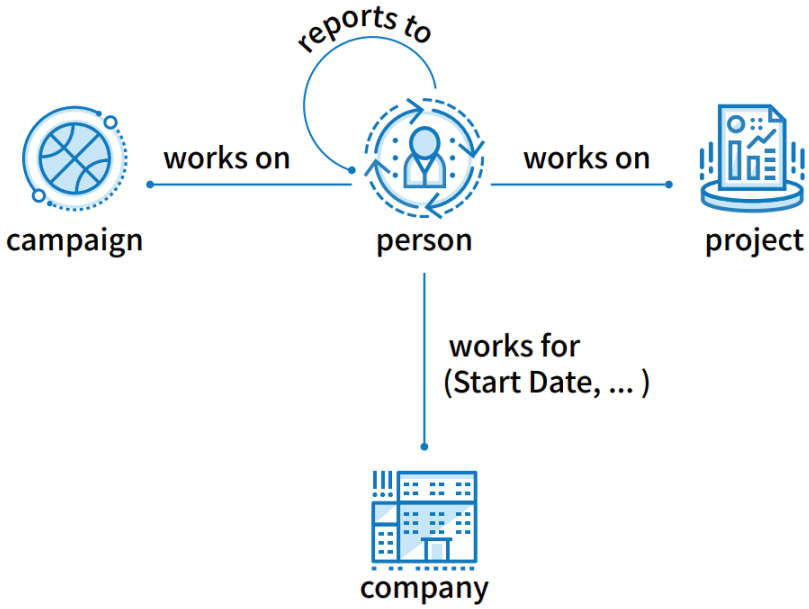

图 1.1: 属性图模型示例 [1]

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。其一，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。其二，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。

如图 1.2，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 [2]。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。

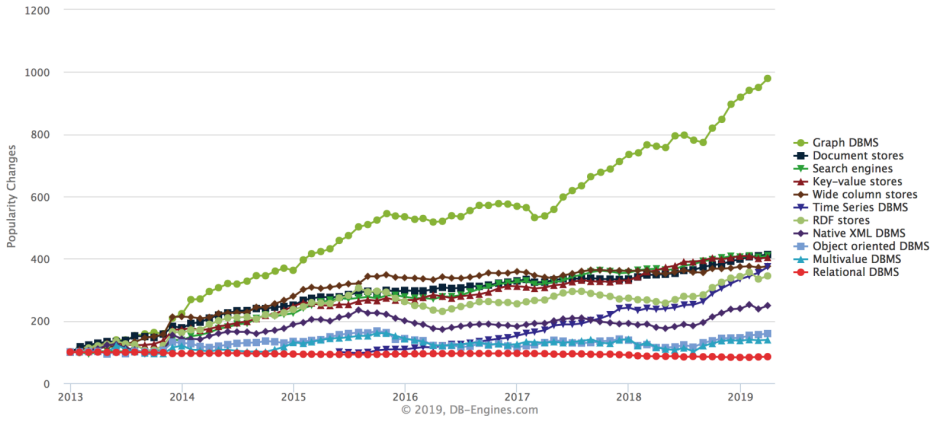

图 1.2: 图数据库流行度趋势上升迅速 [2]

Transwarp StellarDB是星环公司为企业级图应用打造的一款分布式图数据库，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。结合原生存储引擎和计算引擎，StellarDB可以轻松实现数千亿边规模的海量图存储，实时数据插入更新，10层以上深度链路查询，以及复杂算法分析 [1]。

StellarDB的主要优势包括原生图存储、灵活数据模型、快速数据导入、存储计算融合和实时更新与查询。StellarDB为数据存储设计了专有的图存储结构，

并通过高效的压缩算法减少磁盘和内存的使用量。根据分区策略，图数据被分散于集群各节点，拥有良好的可扩展性。此外，StellarDB通过星环分布式存储引擎Shiva为每份图数据创建多个副本，以保证数据的容错性、一致性和高可用性。StellarDB允许用户构建类型丰富的属性图，可以给图中不同的边、点实体添加不同的标签，每个标签下的数据有独立的属性和索引，允许对数据模型和索引字段进行修改和更新。StellarDB拥有强大的数据导入功能，通过交互式界面可以快速配置和导入任务，同时也支持调用Java API插入数据，导入速度可达60GB/小时。StellarDB将存储引擎和计算引擎结合，使计算引擎可以利用数据locality提升计算性能。StellarDB支持数据的实时更新和查询，支持按照边点主键或者属性条件批量更新或批量删除。

StellarDB在很多行业都有广泛的应用。StellarDB可以存储客户购买历史记录，好友列表和感兴趣商品等关联信息，支持多层次商品推荐的计算方式。通过构建商品与兴趣标签的知识图谱，可构建客户的兴趣画像并关联商品；或通过关联具有相似购买记录的人群，计算人物相似度并关联相似人群的高评分商品。通过关联查询、可视化图分析、图挖掘、机器学习和规则引擎的结合使用，可快速检索关联关系数据，并挖掘隐藏关系并模型化业务经验，帮助金融机构的建立一个可持续、经济可行的反洗钱合规框架。通过筛选分析合作关系、集团关系、投资关系、社团分类关系以及资产与负债等业务数据，来识别风险客户和风险集团，降低人力成本消耗并大幅提升反欺诈能力。

1.2　图数据库发展现状

1.2.1　主流图数据库

如图 1.3，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 [4]。它提供同时提供开源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，在所有数据库中流行度排名为第22位 [2]。

Neo4j使用Java开发，而且支持通过Cypher查询语言在其他编程语言进行调用。对Neo4j的操作既可以使用HTTP协议进行请求，也可以使用Neo4j自创的二进制的Bolt协议调用 [5]。

作为流行度最高的图数据库，Neo4j各方面示例都强大、均衡：Neo4j性能

Rank			DBMS	Database Model	Score		
Apr 2020	Mar 2020	Apr 2019			Apr 2020	Mar 2020	Apr 2019
1.	1.	1.	Neo4j	Graph	50.81	-0.97	+1.32
2.	2.	2.	Microsoft Azure Cosmos DB	Multi-model	32.05	+0.41	+5.77
3.	3.	↑ 4.	ArangoDB	Multi-model	4.88	-0.06	+0.59
4.	4.	↓ 3.	OrientDB	Multi-model	4.52	-0.35	-1.68
5.	5.	5.	Virtuoso	Multi-model	2.62	-0.24	-0.70
6.	6.	6.	Amazon Neptune	Multi-model	1.81	-0.01	+0.42
7.	7.	7.	JanusGraph	Graph	1.78	-0.01	+0.40
8.	8.	↑ 10.	GraphDB	Multi-model	1.16	-0.01	+0.19
9.	9.	9.	Dgraph	Graph	1.12	+0.03	+0.04
10.	↑ 14.	↑ 12.	Stardog	Multi-model	0.96	+0.17	+0.14
11.	↓ 10.	↓ 8.	Giraph	Graph	0.94	-0.01	-0.26
12.	12.	↑ 13.	TigerGraph	Graph	0.88	-0.02	+0.07
13.	↓ 11.	↑ 18.	FaunaDB	Multi-model	0.87	-0.07	+0.50
14.	↓ 13.	↓ 11.	AllegroGraph	Multi-model	0.83	+0.01	-0.06
15.	15.	↑ 16.	Blazegraph	Multi-model	0.64	0.00	+0.08
16.	16.	↓ 15.	Graph Engine	Multi-model	0.55	+0.01	-0.01
17.	17.	↑ 25.	Grakn	Multi-model	0.42	-0.07	+0.31
18.	18.	↓ 17.	InfiniteGraph	Graph	0.39	+0.01	-0.01
19.	19.	19.	FlockDB	Graph	0.27	+0.00	0.00
20.	20.	↑ 23.	HyperGraphDB	Graph	0.23	+0.02	+0.02

图 1.3: 主流图数据库流行度趋势 [3]

强大，可以在对数据量大、连通度高的图数据进行多跳查询时保障稳定、即时返回的性能。Neo4j基于Raft技术实现自管理群功能，能够滚动更新、提供实时切换的热备份，保障了高可用性。Neo4j的属性图模型使对图数据的构建、查询易于实施，提供了使用上的灵活性。Neo4j实现了细粒度的安全管理，包括LDAP/目录服务，安全登录等 [6]。

JanusGraph是一款Linux基金会管理下的开源分布式图数据库。其开源协议是Apache License 2.0 [7]。

JanusGraph的开发背后还有有诸多公司的支持，如IBM和Google [8]。所以JanusGraph在与其他项目的协作上灵活度很高。用户可以选择多种存储引擎作为JanusGraph的底层支持，包括Apache基金会的Cassandra、HBase，Google公司的云端BigTable，Oracle的BerkeleyDB，Scylla等等。

JanusGraph尤其与Apache基金会的各种大数据项目协作紧密：JanusGraph的支持图数据全局分析报告，可以与其他大数据平台，如Apache Spark、Apache Giraph、Apache Hadoop等进行ETL集成。JanusGraph支持外部索引存储，如Elastic Search，Apache Solr和Apache Lucene等。通过外部的索引存储，JanusGraph可以对地理坐标、数值范围、文本类型等数据进行搜索匹配。JanusGraph与Apache TinkerPop的图技术栈有着底层集成，包括Gremlin图查询语言、Gramlin图服务

器和Gremlin应用。

TigerGraph是一款高性能、高伸缩性的图数据库，通过对图数据结构的底层实现，它可以为了容量或速度的不同考量进行轻松缩放。TigerGraph通过其独特的“Native Parallel Graph”技术，可以支持实时大规模数据即的完全分布式并行分析。

TigerGraph的关键优势包括其查询语言GSQL，以及其对大数据集的支持[9]。TigerGraph所使用的查询语言GSQL不同于Neo4j的Cypher，更加相似于传统SQL语言，所以对SQL用户更友好、更易学。TigerGraph在大数据集支持方面，导入速度可以达到单机每小时导入50GB至150GB，而对点、边的遍历查询速度可达单机每秒上亿条 [10]。

1.2.2　图数据库存储特点

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 [11]。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（见2.1节）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 [12]。

1.3　论文主要工作

本文首先介绍了StellarDB产品背景，说明了其主要特点与优势，然后介绍了StellarDB的存储模块的设计，解释了它对LSM树数据结构的实现与维护方法。

本文描述了StellarDB在“大数据集高并发写入”之后的性能问题：与处于空闲状态下不同，此时数据库系统无法在预期时间内响应客户端的查询请求，导致客户端请求超时失败。

之后本文对性能问题进行了分析：性能数据显示，StellarDB在对于LSM树数据结构进行compaction处理时，第0层的数据向下流动速度极慢，导致L0数据文件数目严重膨胀。在LSM树数据结构下，读数据需要搜索每一个L0数据文件。这就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。

本文描述了上述性能问题的分析与优化的设计与实现。由于磁盘读写速度已达瓶颈，所以合理的优化方式是减少compaction过程中的读写浪费，也就是在第0层的数据与第1层的数据合并时，降低第1层的数据所占比重。本文实现了对于flush算法与compaction算法的联合优化：首先，在flush算法中加入“内存缓冲区强制分片”，提供“单个L0文件的数据主键范围较小”的条件。然后，在compaction算法中通过“文件选取顺序倒置”与更精细的文件选择算法，将I/O浪费率降至最低。这样的优化能够减少对同等数据进行compaction所消耗的磁盘I/O量。

最后，本文展示了StellarDB原测试数据与新的优化模拟程序的测试结果，证明了算法的有效性：L0至L1的compaction读扩大率从419%下降至119%，数据在LSM树中不再堆积于上层。经过这样的优化，StellarDB的存储模块提高了compaction效率，消除了读请求超时失败的性能问题，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。

第五章，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。

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 [13]。其设计目的在于获得比传统的B+树与索引顺序存取方法更好的写入速度。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 [14]。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 [15]。LSM树还可以推广到其他操作，比如插入和删除 [16]。

LSM树的基本设计比较简明。各批写入操作会会被按顺序保存到一组较小的索引文件中，每个文件都包含涵盖一段时间内的变更。文件按照写入时间排序以保证之后能快速搜索。文件本身完成写入之后就不会被更新，对数据的更新总是被写到新创建的文件中去。读操作会检查所有的数据文件。数据文件需要被周期性地合并，来保证其数量不过度膨胀。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的数据文件之间的流动称为compaction。这两种操作是维护LSM树的主要操作。

当处理数据更新操作时，系统首先应将其加入内存缓冲区，并维持缓冲区的数据主键顺序。通常，内存缓冲区会有预写式日志（write-ahead-log）支持，以保障系统可以恢复写入内存缓冲区的数据。当内存缓冲区被数据填满，就会通过flush操作写到磁盘上的一个新文件中。随着数据的不断写入，flush操作也不断重复。如图 2.1，新写入的数据条目只会创建新的顺序的不可变的数据文件，而非修改包含此主键条目的老数据文件。

当需要从LSM树种读取数据条目时，系统首先需要搜索内存缓冲区，然后以从旧到新的时序搜索各个数据文件，然后将搜索到的条目按时序合并，得到此条目的当前状态。显然，数据文件的个数会影响读取时间，所以需要减少数据文件的数目。

系统周期性地对LSM树进行compaction操作。Compaction操作会选择几个

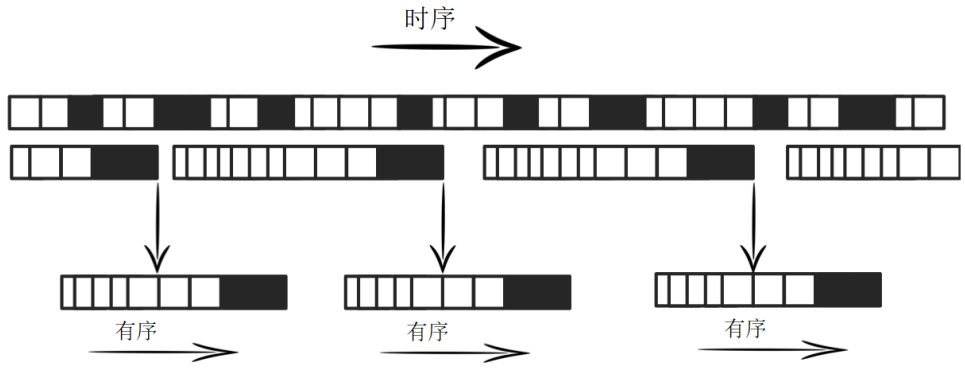

图 2.1: LSM树通过flush操作增加数据文件

数据文件，把合并到一起输出，并删除原来的文件。其中的具有重复主键的条目会被合并成一条。这个操作的意义不仅在于合并条目，更重要的是减少数据文件的数目，避免数据文件数目过多导致读取性能下降。由于数据文件中的数据条目都是按照主键排序的，所以compaction操作相当高效，就像合并排序，对每个输入文件只需要遍历一遍即可 [17]。为了使LSM树的数据文件更易管理，LSM树的具体实现可以给数据文件标注逻辑上的层次。如图 2.2，层数标识数据文件的新旧，刚刚写入的数据文件都处于第0层（Level 0，简称L0），也就是最高层。除了L0以外，每层的数据文件都可以看作一个整体，层内各文件保持有序，数据主键不重叠，就像是一个巨大的数据文件被切分不同区域。通过规定compaction只能从高层向低层写（或最底层文件之间互相合并），旧的数据文件之间的时间顺序与主键顺序更加清晰。这样做还就控制了读数据时需要搜索的数据文件个数：少于L0文件数与分层数的和。

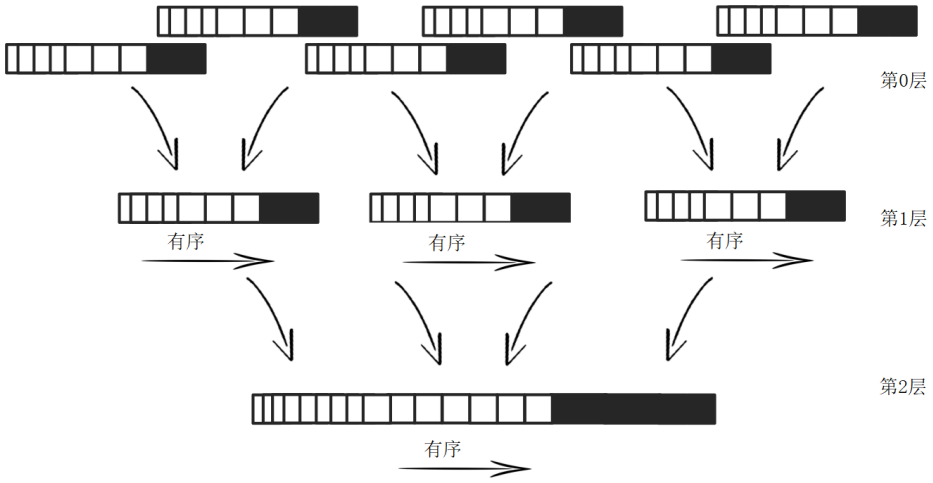

图 2.2: 分层compaction示意图

包括Google Bigtable、Apache Cassandra等数据库都采用了LSM树作为数据存储结构 [18]。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。

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 [19]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 [20]。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 [19]。

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 [21]。

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，就可能会导致大量写操作失效。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 Zookeeper

StellarDB客户端使用Zookeeper对服务器端进行感知，通过与服务器端共享Zookeeper集群获取集群IP等关键信息。ZooKeeper是一个分布式的，开放源码的分布式应用程序协调服务。Zookeeper 能够保证其数据的可靠性和高可用性，并且提供了多种编程语言的API。

Zookeeper 由多台服务器组成，每台服务器中都复制了一份元数据，程序通过访问任意一台服务器都可以获得完整且一致的数据。通过这种方式，Zookeeper 实现了高可用性、高可靠性和高性能 [22]。Zookeeper 服务器集群中的多台服务器分为两种角色：其一是leader（主节点），正常情况下，只有一台服务器作为leader；leader 是由所有服务器自主选举产生的 [23]。当有写入数据操作时，由leader 负责将写入操作传播至其他服务器，并由leader 返回写入结果。其他服务器的角色是follower，读数据的操作可以由follower 直接处理。leader 接收来自follower 的写入数据操作，并将结果返回follower [24]。

Zookeeper 中，使用一种结构类似于文件存储系统的树状层次模型来存储数据。在该模型中，节点被称为Znode，每个节点都有其唯一的标志路径。Znode 既可以直接存储数据，又可以在其中新建Znode 子节点。Znode 是有版本的，每个节点中可以存储多个版本的数据。Znode 可以被其他程序注册监听，当该节点发生变化（更新，删除）时，Zookeeper 将会自动通知监听该节点的程序

[25]。

2.4 Docker

由于基于虚拟机（Virtual Machine）进行的传统应用部署方式所占用的资源较多，操作步骤较冗杂繁多，且启动速度非常缓慢，无法很好地适应当前业务流量的高度弹性化以及微服务架构的技术环境。另外一种虚拟化技术——容器（Container）技术应运而生，容器技术因为其启动速度快、资源占用少、体积小的诸多优点迅速得到主流软件开发业界的追捧。

Docker是一个开源的轻量级应用容器引擎，基于Go语言实现，并且遵从Apache 2.0开源协议 [26]。Docker是目前业界最为流行的容器解决方案，可以看做是对容器的一种封装，为开发者提供了方便易用的容器使用接口。

Docker能够帮助开发者对于应用以及依赖打包到一个轻量级、可移植的容器镜像（Docker Image）中。打包完成之后，一方面可以将打包后的容器镜像发布到Windows或者主流的Linux机器上；另外一方面也可以实现虚拟化。Docker容器完全基于沙箱机制，相互之间不存在任何接口。除此之外，相比于虚拟机，容器性能开销也非常低。Docker的接口非常简单，因此开发者可以极为方便地创建和使用容器，以及将应用程序放入Docker容器中，并对Docker容器进行版本管理以及修改。 [27]

在云计算时代，应用的弹性伸缩能力至关重要。而Docker容器可以随开随关，因此非常适合用于弹性扩容和缩容。此外，对于微服务架构而言，借助Docker等容器技术，可以使多个服务实例在同一台物理机上运行，能够有效地降低资源消耗。此外，Docker使得开发者能够将应用程序和基础架构进行分离，并使用与管理应用程序相同的方式来管理基础架构，从而实现应用程序的快速交付和快速部署，大大减少应用程序从开发环境的编写到在生产环境中实际运行之间的延迟。StellarDB的部署方式便是以Docker容器的形式运行于服务器。 [28]

2.5 Kubernetes

Kubernetes是一个自动化容器操作的项目，由Google于2014年启动，目前由开源社区维护。此技术旨在解决集群中容器的部署、更新、维护等一系列问题。Kubernetes源自于Google内部的Brog容器管理系统，本身并不是容器的实现，其核心是将分布在不同主机的应用容器，通过网络组件连接成一个网络，从而实现服务的分布式计算。StellarDB部署采用Kubenetes管理Docker容器，以利用其

对容器的自动创建、故障重启、网络共享等功能降低运维难度 [29]。

Kubernetes包括以下几个重要的概念：Cluster、Master、Node、Pod、Service。一个Kubernetes 系统通常称为一个集群，集群主要包括一个Master节点和若干Node节点。Master节点包括API Server、Scheduler、Controller Manager、etcd等，主要负责管理和控制。其中API Server是整个系统的对外接口，可供客户端和其他组件调用；Scheduler负责对集群内部的资源进行调度；Controller Manager负责管理控制器。Node节点包括Docker、Kubelet、Kube-proxy、Fluentd、Kube-dns以及Pod。Pod是管理，创建，计划的最小单元，是一个容器环境下的“逻辑主机”。它可能包含一个或者多个紧密相连的应用，这些应用可能是在同一个物理主机或虚拟机上。 [30]Service可以看作一组提供相同服务的Pod 的对外访问接口。Docker用于创建容器；Kubelet主要负责监视指派到它所在Node上的Pod，包括创建、修改、监控、删除等操作；Kube-proxy主要负责为Pod对象提供代理；Fluentd主要负责日志收集、存储与查询 [31]。

2.6　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，用于保障多副本数据一致性的Raft一致性协议，用于集群间同步与服务器探查的Zookeeper，用于StellarDB部署的 Docker 和 Kubernetes 。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及其存储模块设计说明

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（如图 3.1）。本章介绍了StellarDB的模块划分，分析了其存储模块，包括存储模块核心设计：采用LSM树数据结构进行数据存储。本章末尾介绍了作为简化版StellarDB存储模块的优化模拟测试程序的设计。

图 3.1: StellarDB宏观架构图

3.1　StellarDB架构

如图 3.2，作为星环大数据平台的重要组件，StellarDB架构主要由以下部分组成：

3.1.1　存储引擎

图数据以高效的压缩格式存储于星环分布式存储引擎Shiva中，借助图分区算法，图数据可按策略分散存储于集群中，拥有良好的可扩展性，并具备存储任意规模图的理论能力。

存储引擎架构为Master-Worker结构，多个Master组成Master Group负责元信息管理、任务调度、负载均衡等功能；Worker存储图数据，并提供数据读取、更新和删除功能。存储引擎通过Raft协议来保证数据一致性和高可用性。

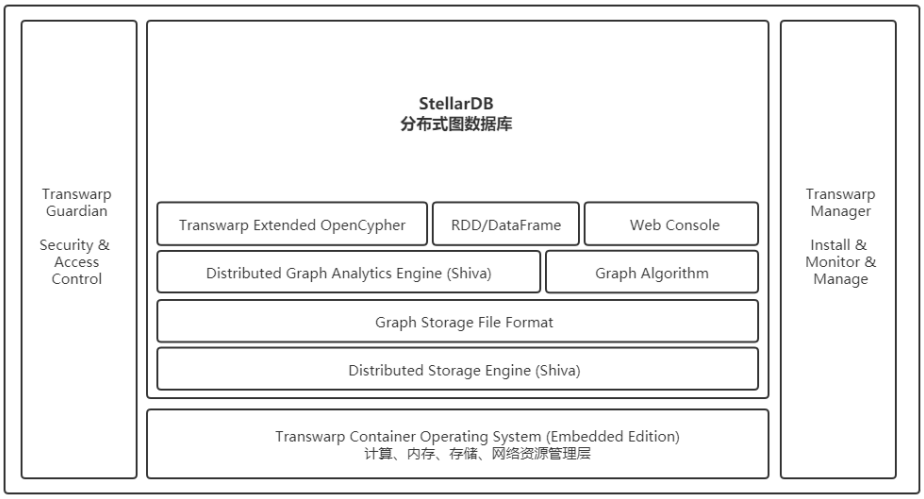

图 3.2: StellarDB模块划分

3.1.2　计算引擎

借助星环分布式计算引擎Inceptor的计算分析能力，计算能力随着节点数目增长线性扩展。StellarDB可同时为用户提供实时图查询和离线算法分析，支持海量边点的大图分析。

计算引擎和存储引擎同机部署，利用数据locality特性加速图计算和分析任务。计算引擎内置了部分常用图算法，并以RDD的方式提供数据和计算的接口。

3.1.3　扩展OpenCypher

OpenCypher在历经了多年的产业界验证后，成为了当前最主流的开源图数据库访问语言。StellarDB实现了OpenCypher，在提供标准功能之外，还提供了一些扩展语言，以满足图计算和复杂查询流程的需求。

3.1.4　可视化引擎

用户通过查询语法，可以完成图数据的查询和分析。StellarDB提供网页可交互分析工具，用户可基于查询结果做进一步的数据分析，或者通过业务数据和图谱模型来构建新图谱。

3.1.5　其他模块

除了上述模块，StellarDB还借助星环其他产品提供丰富的功能，如通过Transwarp Guadian提供用户认证、权限管理功能，Transwarp Manager提供安装和资源监控服务。

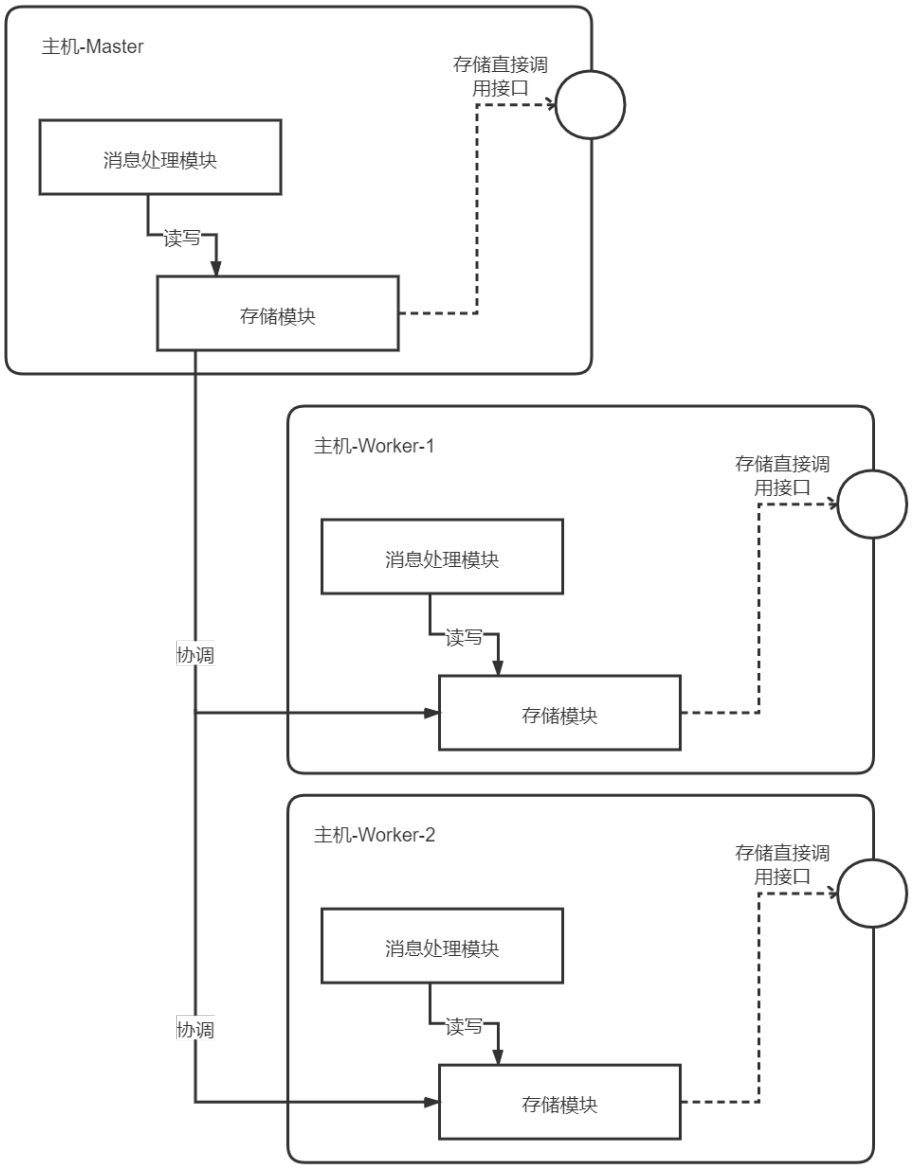

图 3.3: StellarDB存储模块外部协作关系

3.2　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. graph-search. 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。

3.3　StellarDB存储模块内部设计简述

图 3.4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Mem-table。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

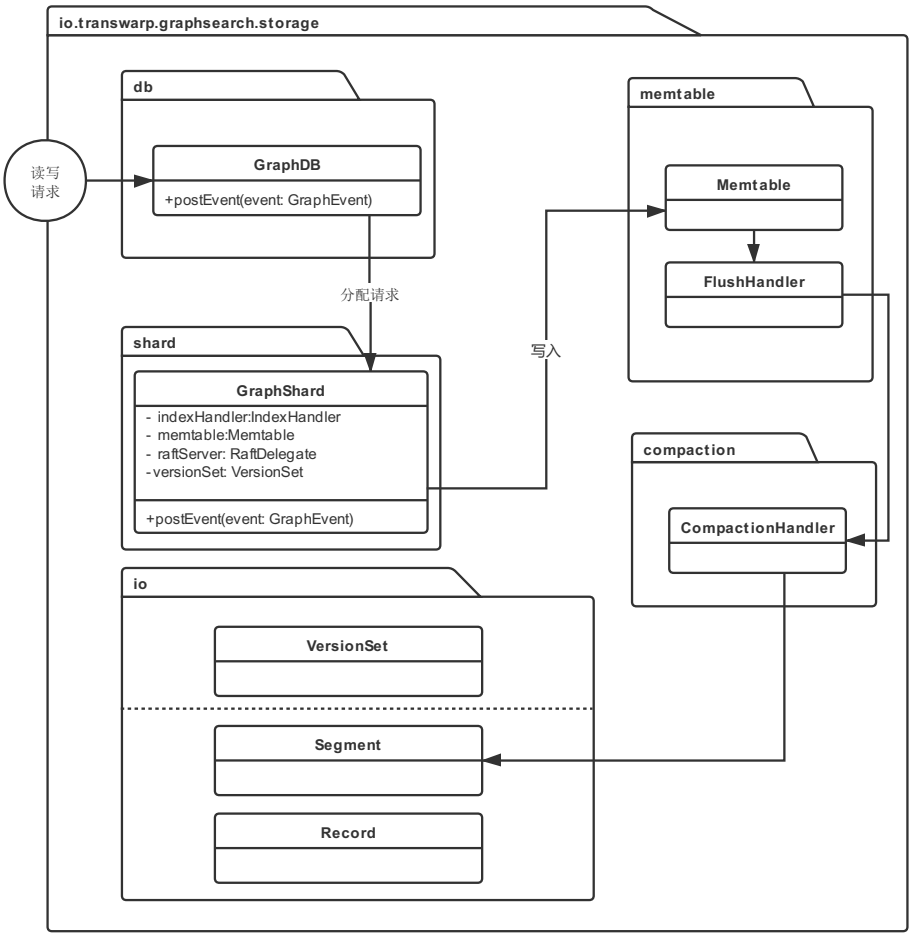

图 3.4: StellarDB存储模块内部协作关系简述

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。

3.4　LSM树的基本维护方式

本节介绍StellarDB对LSM树数据结构的实现与维护方式。

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 3.5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。

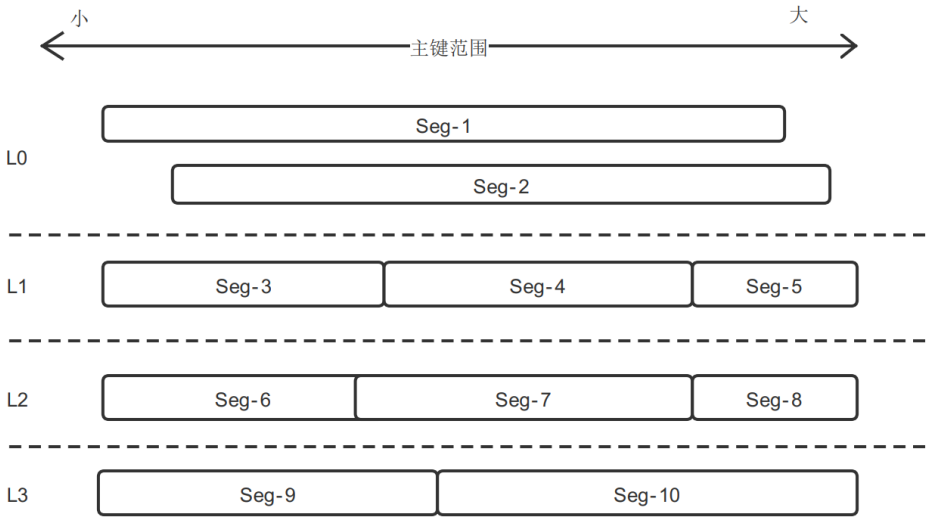

图 3.5: 某Shard内数据文件分布示意图

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 3.6 ，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。

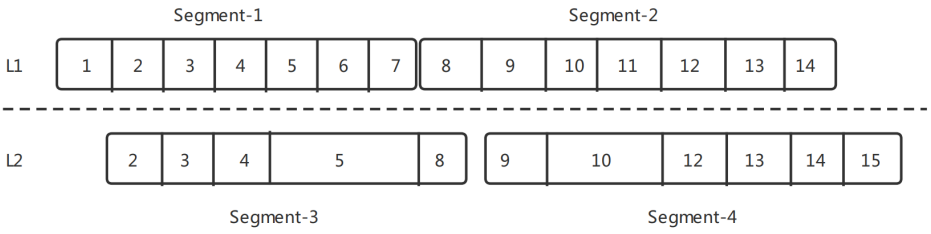

图 3.6: Compaction操作-原状态

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 [32]。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 3.7，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。

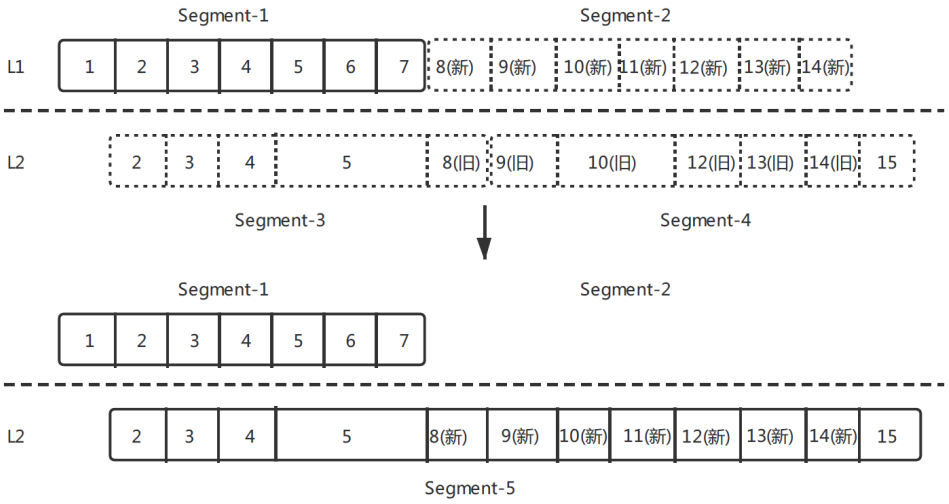

图 3.7: Compaction操作-操作结果

3.5　优化模拟测试程序功能

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。

3.5.1　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。

3.5.2　功能设计

程序应对StellarDB存储模块进行抽象简化，去除多副本备份功能、多图多GraphShard管理功能、异步响应客户读写请求功能和对于LSM结构进行持久化的功能。程序保留对LSM树的基本读写功能、LSM树的版本管理功能、flush功能（包括实现基于跳表的内存缓冲区）、compaction功能、compaction性能统计功能（包含读写数据量的详细统计）。

3.6　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，本文中的介绍不会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，作为对StellarDB存储模块结构的一项补充。

如图 3.8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。

3.6.1　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。本优化模拟测试程序由于功能简略，并不是以持续服务的形式运行。Driver类

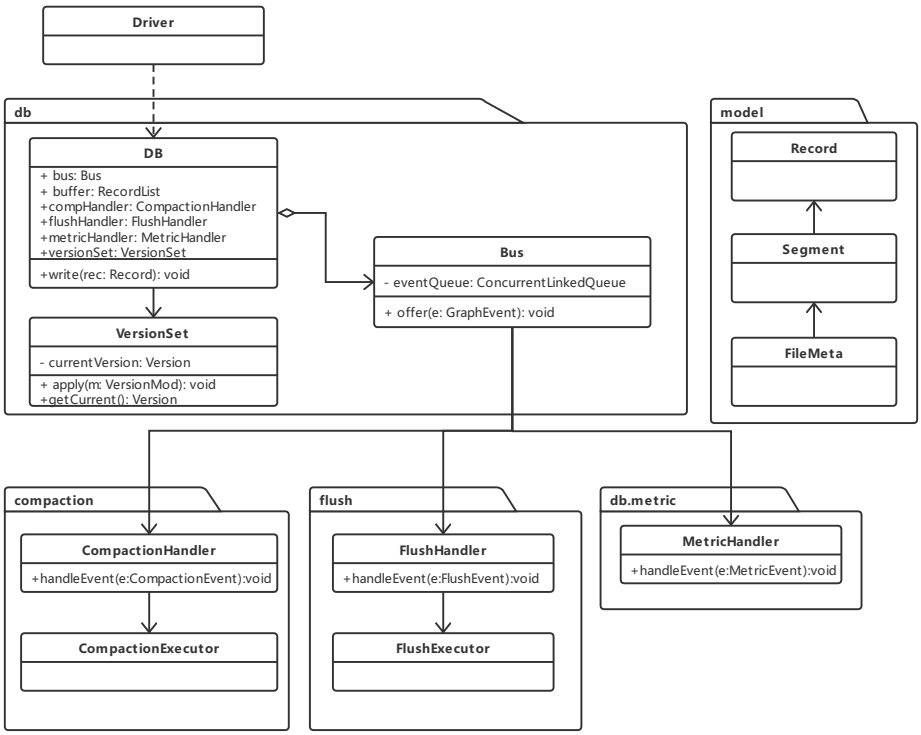

图 3.8: 优化模拟测试程序类图（简化版）

包含了Java中的main方法，方法中包含了在内存中构造测试数据、创建DB类实例、启动DB服务、实时调用DB类的写入接口的过程，其间DB类会异步地对写入数据进行操作。不同Driver类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。

3.6.2　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。

3.6.3　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。

3.6.4　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。

3.6.5　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。

3.6.6　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Compaction Handler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。

3.6.7　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。

3.6.8　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。

需要注意的是，由于本模拟程序中LSM树不同的层次的compaction不会异步进行，所以模拟编写的“优化前”调度算法与StellarDB中实际的优化前调度算法略有不同。如图 3.9，在此实现中，由于L1文件在调度compaction任务时不会被其他compaction任务占用，所以采用了简单的“取用全部与L0文件有主键重叠的L1文件”的方式。这样的修改并不会影响测试结果得出的结论。因为StellarDB原算法在额外剔除一部分被占用的L1数据文件之后，所调度的compaction任务会剔除更多的L0文件，导致更严重的I/O浪费。

3.7　本章小结

本章介绍了StellarDB的架构，分析了各个模块地作用，然后详细介绍了存储模块的设计，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。

```
if (level < lastLevel) { // 非最后一层需要与下一层进行合并
    List<FileMeta> basicFiles = new ArrayList<>();
    for (FileMeta fileMeta : fileMetaList) {
        if (fileMeta.getLevel() == level) {
            basicFiles.add(fileMeta);
        }
    }

    if (level == 0) { // 第 0 层的数据必须先对旧的数据（fileId 小的）做 compaction
        if (basicFiles.size() > levelMinSize[0]) {
            basicFiles.sort(Comparator.naturalOrder());
            upperLevelFiles = basicFiles.subList(0, compactionMaxPickCount[0]);
        }
    } else { // 第 1、2 层则优先选择尺寸小的文件，减少零碎
        … //省略
    }

    if (upperLevelFiles != null) { //
        byte[] leftEnd = CommonUtils.smallestStartKey(upperLevelFiles);
        byte[] rightEnd = CommonUtils.largestEndKey(upperLevelFiles);
        //跟据上层文件的记录 rk 范围选取下层文件
        lowerLevelFiles = new ArrayList<>();
        for (FileMeta fileMeta : fileMetaList) {
            if (fileMeta.getLevel() == level + 1 && CommonUtils.overlapInRange(fileMeta, leftEnd,
rightEnd)) {
                lowerLevelFiles.add(fileMeta);
            }
        }
        lowerLevelFiles.sort(Comparator.naturalOrder());
    }
} else { // 最后一层与自己合并
    … //省略
}

//使用 upperLevelFiles 与 lowerLevelFiles 调度任务
if (upperLevelFiles != null && !upperLevelFiles.isEmpty()) {
    LOG.info("Scheduled a compaction task on level " + level);
    compactionExecutorV1.doCompactionV1(upperLevelFiles, lowerLevelFiles, level);
}
```

图 3.9: 优化模拟测试程序的优化前compaction调度算法代码

第四章　性能优化详细设计实现与结果展示

本章介绍在StellarDB存储模块的生产环境中遭遇的实际性能问题的优化。

4.1　性能问题说明

根据同事在客户现场进行产品展示时的经历，团队发现StellarDB在大规模数据集的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：虽然系统接收到了客户端的查询请求，但在默认的30秒超时时间内，甚至无法完成第一小批查询结果，导致客户端认为数据库查询失败，而实际上数据库在几分钟之后会向客户端端口发回第一小批查询结果。这样长的查询延迟无法满足客户对系统性能的需求。在近似场景的测试下重现这个问题，可以发现在经过25分钟的高压写入过程之后，各GraphShard的L0文件数均超过1000；在停止写入30分钟之后，仍然有超过四分之一的GraphShard的L0文件数超过1000。在这种条件下，这些GraphShard不能及时响应读请求。经过简单排查，发现关键在于数据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会同时管理约2000个L0的数据文件，而一个查询请求的处理需要对这上千个L0数据文件都进行搜索。这导致了I/O线程的短缺，使查询请求难以得到及时处理。

4.2　性能问题分析

在此次性能优化之前，公司技术人员已经对此性能问题进行了分析与优化尝试。但是限于可分析的性能数据有限，提出的分析与优化尝试效果不佳，未能解决查询请求超时问题。此次接手任务先完善了系统的性能监控能力，之后再全面进行了问题分析。

4.2.1　基于简略的性能数据的问题分析

在完善性能数据之前，StellarDB只可以报告各GraphShard的LSM树的各层文件数目，以及执行 compaction 任务的线程数目；通过查询日志，可以得知调度执行的flush任务与compaction任务是否完成。而性能测试的度量，只限于统计“写入开始”、“写入完成”、“compaction基本完成”等时间节点之间的间隔，无法精细地统计具体 compaction 任务的粒度、个数、执行效率等数据。

在这样的条件下，问题分析只包含了对系统现有可调节参数的讨论：

1. 数据文件是否开启压缩：为了节省存储空间，StellarDB的数据文件默认内部按数据块进行压缩。如果取消L0的压缩，可以提升Memtable的flush速度，而且在从L0向L1进行 compaction 时，也可以节省用于解压L0文件数据块的CPU资源。

2. Compaction文件限制数：为了避免过大规模的compaction任务长时间占用I/O线程，StellarDB为 compaction 调度限制了输入文件数。如果放宽此限制，就可以减少 compaction 总次数，有可能提高 compaction 并行度，从而提高效率。

3. Compaction超时时间：StellarDB会周期性自动触发 compaction 检查。如果此时间设置过短，可能会导致 compaction 线程竞争问题。

4. Compaction线程数：如果 compaction 任务之间存在线程竞争情况，那么增大线程池容量可以缓解竞争问题，提升 compaction 总体效率。

经过具体的参数调整，测试显示，Memtable的flush性能提升了3倍，但 compaction 的性能提升不明显。对LSM树的文件数目变动趋势的观察显示，compaction 依然阻塞于L0。从“写入开始”到“compaction基本完成”的时间也没有缩短。这说明flush性能并非此问题的性能瓶颈，调节现有参数无法解决 compaction 阻塞问题。

4.2.2　基于完善的性能数据的问题分析

在完善了StellarDB存储模块的性能监控功能之后（具体新增条目参见本章“StellarDB对比测试”一节），StellarDB重新进行了性能测试。测试发现，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。磁盘读写速度已经没有提升空间，而磁盘的总I/O量与写入StellarDB的数据量及其不匹配。进一步分析发现，compaction过程消耗的L0文件容量与数据集大小匹配，但同时compaction消耗了远远多于L0数据的L1数据。这说明compaction过程中存在对磁盘I/O能力的巨大浪费。

这个现象与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），可以发现，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，以此类推。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：其一，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；其二，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。

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

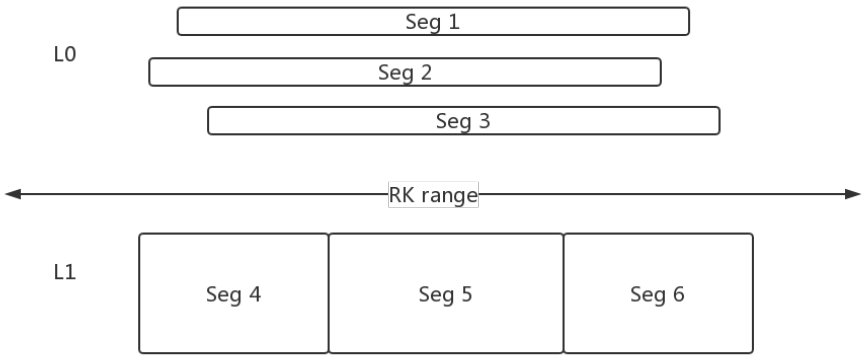

图 4.1: Compaction算法优化前L0与L1结构

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　算法优化设计

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。

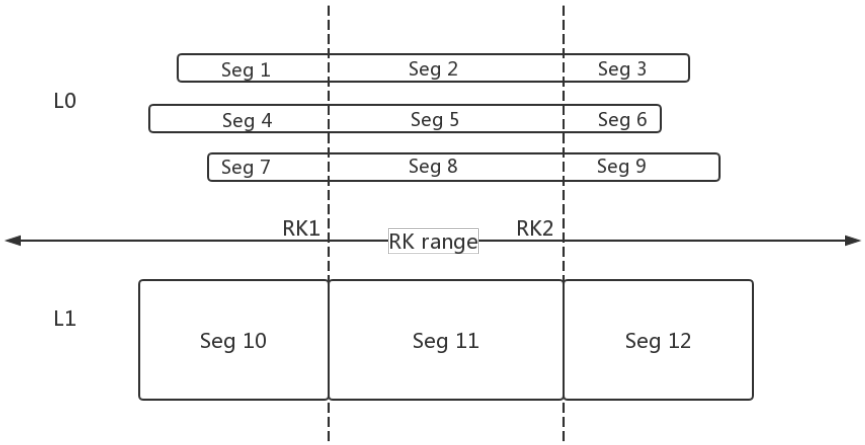

图 4.2: Compaction算法优化后L0与L1结构

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。

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

图 4.3: Compaction算法需要针对分割点具体优化

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。

```
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

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。

```
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

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。

L0、L1文件选取顺序倒置可以减少文件筛选步骤，提高选取率。

如图 4.6，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 4.9为改进后调度代码，图 4.10为算法流程图，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。

如图 4.7、图 4.8，新的L0内部文件选择算法不再使用死板的文件数目限制，

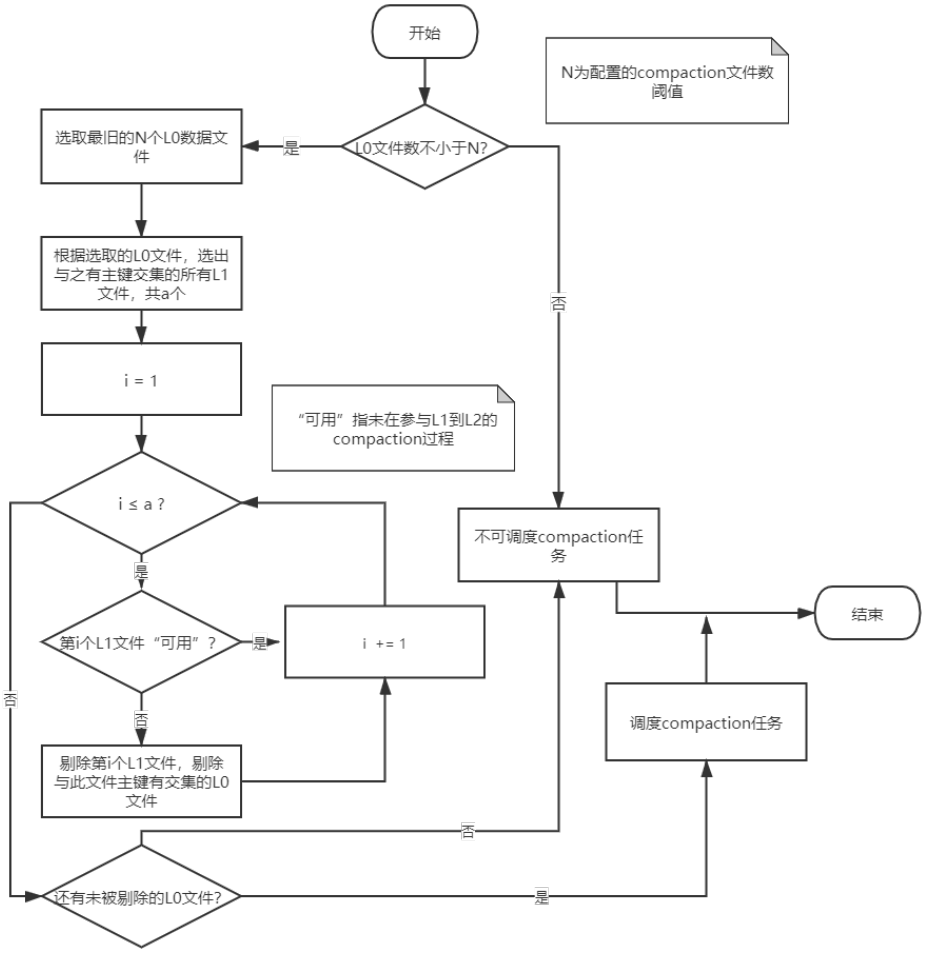

图 4.6: 优化前compaction算法流程图

将文件总容量、读扩大率等条件都纳入考量，更精细地选取尽量多的L0文件。

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。

```
/**
 * 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 4.7: 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 4.8: Compaction算法优化：L0内部文件选择算法源码（续）

```
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 4.9: Compaction算法优化：L0、L1文件选取顺序倒置

如图 4.11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

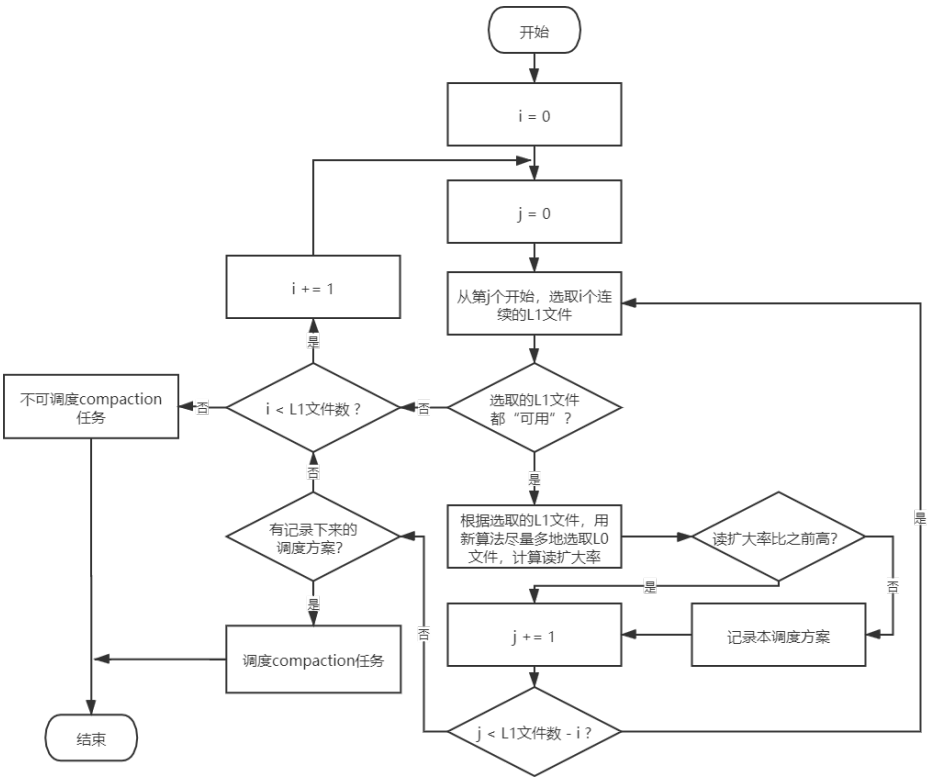

图 4.10: 优化后compaction算法流程图

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 StellarDB对比测试

4.6.1 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。

4.6.2 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

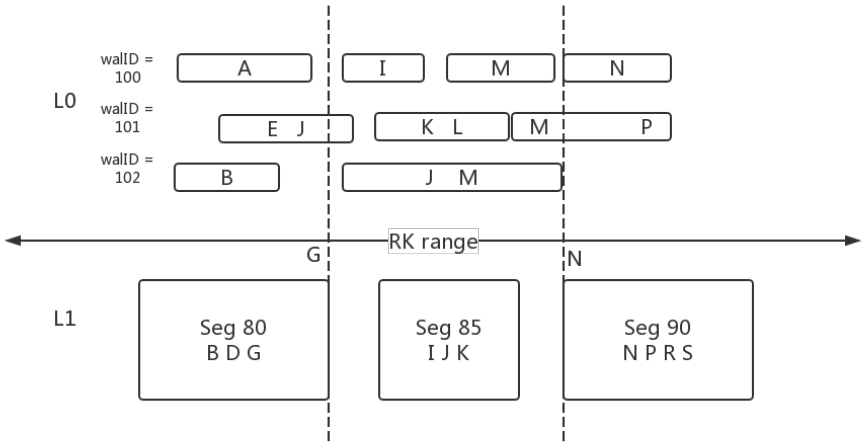

图 4.11: Compaction算法优化：L0内部文件选择情景示例

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 [33]（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 4.12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 4.12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 4.13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。

```
{
    "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 4.12: StellarDB的RESTful接口返回的性能信息示例

4.6.3　测试结果：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。

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。

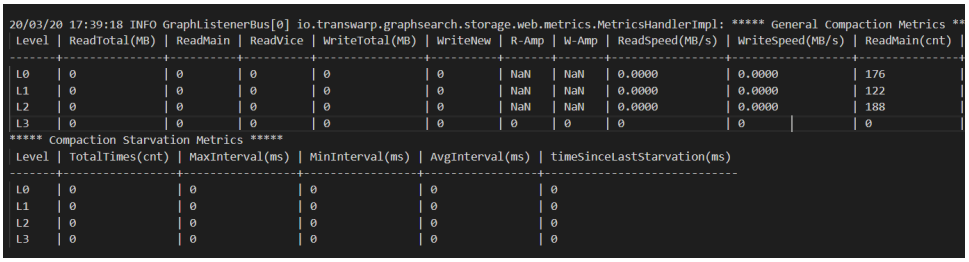

图 4.13: StellarDB的使用Log4j以表格形式打印性能信息

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LSM树各层的数据文件进行压缩，各层设值用逗号分开，“-”值表示不压缩
Bloom level	-,-,-,-	是否在LSM树各层数据文件中加入Bloom过滤器以辅助读操作，各层设值用逗号分开，“-”值表示本层不使用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

4.6.4　测试结果：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。

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

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

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

“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。

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

4.6.5　测试结果：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。

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

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

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

4.6.6　测试结果：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。

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

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18718 MB、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，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。

4.6.7　测试结果：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

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

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

如表 4.10，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。

表 4.10: 测试EXP4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	35	24175
1	1371	5081404
2	119	354093
3	16	93159
4	1	3984

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。

4.6.8　测试结果：EXP5

如表 4.11，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 4.11: 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 4.12，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 4.12: 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

4.6.9　测试结果：EXP6

如表 4.13，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。

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。

4.6.10　分析总结

根据上面的对比测试结果，可以看出：

表 4.13: 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

1. Compaction优化算法有效降低了L0到L1的compaction过程的的读扩大率：最初从419%的水平，最好情况下优化到了88.7%，最终选择的综合方案效果为119%。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。在测试过程中观察到的现象则是数据文件不再堆积于L0，在持续写入过程中，L0层文件数始终保持在一个较低的水平。而写入过程结束后，L0的compaction也可以在3分钟之内完成。

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。

4.7　模拟程序对比测试

4.7.1　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。

4.7.2　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。

4.7.3　测试变量：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。

4.7.4　测试变量：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。

测试不变量为：

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

4.7.5　测试结果

如表 4.14，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 4.14: 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

4.8　本章小结

本章首先介绍了StellarDB的compaction性能瓶颈的问题分析与解决思路。由于compaction的I/O浪费导致数据文件在L0堆积，读请求的处理需要扫描上千个数据文件。磁盘I/O的利用效率低下导致了最终的读请求超时。所以应该通过优化compaction以及相关过程减少I/O浪费，有效利用磁盘I/O，避免数据堆积，避免读请求超时。

然后本章介绍了优化算法设计实现，通过详尽的测试说明了算法的逐步优化过程，验证了优化效果。通过在flush过程对内存缓冲区强制分片，并在compaction过程精细计算不同调度方式可以实现的读扩大率，StellarDB在L0的compaction读扩大率从419%降低到了119%。这样的进步突破了compaction在L0的性能瓶颈，避免了数据堆积，消除了读请求超时的业务问题。

最后，本章介绍了优化算法模拟程序的设计实现以及其测试结果，进一步证明了优化算法的有效性。

第五章　总结与展望

5.1　总结

大数据时代对数据库性能提出了很高的要求。数据库在承载较高的写入负载的同时，也需要提供充足的查询性能。本文介绍了对星环科技StellarDB的存储模块性能的优化过程，主要进行了以下工作：

1. 介绍StellarDB产品背景与StellarDB存储模块的所用到的核心技术，包括LSM树数据结构与Raft一致性协议等，并说明了这些技术在StellarDB中所起到的作用。

2. 解释了StellarDB存储模块的概略设计与它对LSM树的实现、维护方法。这为后文说明算法优化的设计提供了基础。

3. 说明了性能问题在StellarDB的具体业务表现，并对其进行深度分析：在实际业务应用中，StellarDB经过大数据集高并发写入后，无法在预期时间内完成查询请求的处理。最终性能瓶颈被定位在L0的compaction过程。这是由于对磁盘I/O的利用效率太低，有限的磁盘读写速度无法满足系统需要，所以需要减少I/O浪费。

4. 详细描述了算法的优化过程，包括在性能测试过程中对算法的逐步完善。最终通过在flush算法加入强制分片，在compaction算法对分片针对性优化调度方式，将读扩大率从419%降低至119%，消除了数据文件堆积，成功解决了业务上的性能问题。

5. 最后通过实现一个优化算法模拟程序，进一步证明了优化算法有效性。

经过对flush算法与compaction算法的联合优化，StellarDB的读超时的性能问题得到了解决。

工作过程中的一些算法优化经验如下：

1. 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。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。

2. 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。本文的首先优化了flush算法，通过分割Memtable改变了数据分布，为 compaction 的优化方案提供了前提条件。这样的flush改进方案看起来并不直截了当，很难在面对性能问题时一开始就想出。但实际上这种设计也是根据“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。

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，伴随着多次额外的对比性能测试。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有重大参考意义。

5.2 展望

本文所介绍的优化算法已经完成开发、测试，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并未停止。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] DB-Engines, 数据库流行度趋势, https://db-engines.com/en/ranking_trend.

[3] DB-Engines, 主流图数据库流行度, https://db-engines.com/en/ranking/graph+dbms.

[4] L. Wiese, Data analytics with graph algorithms - A hands-on tutorial with neo4j, in: H. Meyer, N. Ritter, A. Thor, D. Nicklas, A. Heuer, M. Klettke (Eds.), Datenbanksysteme für Business, Technologie und Web (BTW 2019), 18. Fachtagung des GI-Fachbereichs „Datenbanken und Informationssysteme" (DBIS), 4.-8. März 2019, Rostock, Germany, Workshopband, Vol. P-290 of LNI, Gesellschaft für Informatik, Bonn, 2019, pp. 259–261.
URL https://doi.org/10.18420/btw2019-ws-26

[5] Neo4j, Inc., Bolt协议主页, http://boltprotocol.org/.

[6] Neo4j, Inc., Neo4j产品主页, https://neo4j.com/neo4j-graph-database/.

[7] The Linux Foundation, JanusGraph产品主页, https://janusgraph.org/.

[8] D. Harris, Google, IBM back new open source graph database project, JanusGraph, https://architecht.io/google-ibm-back-new-open-source-graph-database-project-janusgraph-1d74fb78db6b.

[9] A. Deutsch, Y. Xu, M. Wu, V. Lee, Tigergraph: A native MPP graph database, CoRR abs/1901.08248.
URL http://arxiv.org/abs/1901.08248

[10] TigerGraph, Inc., Tigergraph主页, https://www.tigergraph.com/.

[11] J. L. Reutter, Current challenges in graph databases (invited talk), in: C. Lutz, J. C. Jung (Eds.), 23rd International Conference on Database Theory, ICDT 2020,

March 30-April 2, 2020, Copenhagen, Denmark, Vol. 155 of LIPIcs, Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2020, pp. 3:1–3:1.
URL https://doi.org/10.4230/LIPIcs.ICDT.2020.3

[12] GraphDB, Graphdb主页, http://graphdb.net/.

[13] 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

[14] N. Mittal, F. Nawab, C-LSM: cooperative log structured merge trees, in: Proceedings of the ACM Symposium on Cloud Computing, SoCC 2019, Santa Cruz, CA, USA, November 20-23, 2019, ACM, 2019, p. 481.
URL https://doi.org/10.1145/3357223.3365443

[15] T. Zhu, H. Hu, W. Qian, A. Zhou, M. Liu, Q. Zhao, Precise data access on distributed log-structured merge-tree, in: L. Chen, C. S. Jensen, C. Shahabi, X. Yang, X. Lian (Eds.), Web and Big Data - First International Joint Conference, APWeb-WAIM 2017, Beijing, China, July 7-9, 2017, Proceedings, Part II, Vol. 10367 of Lecture Notes in Computer Science, Springer, 2017, pp. 210–218.
URL https://doi.org/10.1007/978-3-319-63564-4_17

[16] N. Y. Song, H. Y. Yeom, H. Han, Efficient key-value stores with ranged log-structured merge trees, in: 11th IEEE International Conference on Cloud Computing, CLOUD 2018, San Francisco, CA, USA, July 2-7, 2018, IEEE Computer Society, 2018, pp. 652–659.
URL https://doi.org/10.1109/CLOUD.2018.00090

[17] B. Stopford, Log structured merge trees, http://www.benstopford.com/2015/02/14/log-structured-merge-trees/.

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

[19] 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

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

[21] F. D. Muñoz-Escoí, R. de Juan-Marín, J. García-Escrivá, J. R. G. de Mendívil, J. M. Bernabéu-Aubán, CAP theorem: Revision of its related consistency models, Comput. J. 62 (6) (2019) 943–960.
URL https://doi.org/10.1093/comjnl/bxy142

[22] P. Hunt, M. Konar, F. P. Junqueira, B. Reed, Zookeeper: Wait-free coordination for internet-scale systems, in: P. Barham, T. Roscoe (Eds.), 2010 USENIX Annual Technical Conference, Boston, MA, USA, June 23-25, 2010, USENIX Association, 2010.
URL https://www.usenix.org/conference/usenix-atc-10/zookeeper-wait-free-coordination-internet-scale-systems

[23] F. P. Junqueira, B. Reed, The life and times of a zookeeper, in: S. Tirthapura, L. Alvisi (Eds.), Proceedings of the 28th Annual ACM Symposium on Principles of Distributed Computing, PODC 2009, Calgary, Alberta, Canada, August 10-12, 2009, ACM, 2009, p. 4.
URL https://doi.org/10.1145/1582716.1582721

[24] R. Halalai, P. Sutra, E. Riviere, P. Felber, Zoofence: Principled service partitioning and application to the zookeeper coordination service, in: 33rd IEEE International Symposium on Reliable Distributed Systems, SRDS 2014, Nara, Japan, October 6-9, 2014, IEEE Computer Society, 2014, pp. 67–78.
URL https://doi.org/10.1109/SRDS.2014.41

[25] A. Frömmgen, S. Haas, M. Pfannemüller, B. Koldehofe, Switching zookeeper's consensus protocol at runtime, in: X. Wang, C. Stewart, H. Lei (Eds.), 2017 IEEE International Conference on Autonomic Computing, ICAC 2017, Columbus, OH, USA, July 17-21, 2017, IEEE Computer Society, 2017, pp. 81–82.
URL https://doi.org/10.1109/ICAC.2017.54

[26] Docker, Docker主页, https://www.docker.com/.

[27] K. J. Lidl, Understanding docker, login Usenix Mag. 42 (4).
URL https://www.usenix.org/publications/login/winter2017/lidl

[28] C. Boettiger, D. Eddelbuettel, An introduction to rocker: Docker containers for R, CoRR abs/1710.03675.
URL http://arxiv.org/abs/1710.03675

[29] Kubernetes, Kubernetes主页, https://kubernetes.io/.

[30] Kubernetes, Kubernetes中文文档, https://www.kubernetes.org.cn/docs.

[31] D. Haja, M. Szalay, B. Sonkoly, G. Pongrácz, L. Toka, Sharpening kubernetes for the edge, in: Proceedings of the ACM SIGCOMM 2019 Conference Posters and Demos, SIGCOMM 2019, Beijing, China, August 19-23, 2019, ACM, 2019, pp. 136–137.
URL https://doi.org/10.1145/3342280.3342335

[32] C. Gong, S. He, Y. Gong, Y. Lei, On integration of appends and merges in log-structured merge trees, in: Proceedings of the 48th International Conference on Parallel Processing, ICPP 2019, Kyoto, Japan, August 05-08, 2019, ACM, 2019, pp. 103:1–103:10.
URL https://doi.org/10.1145/3337821.3337836

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

致　　谢