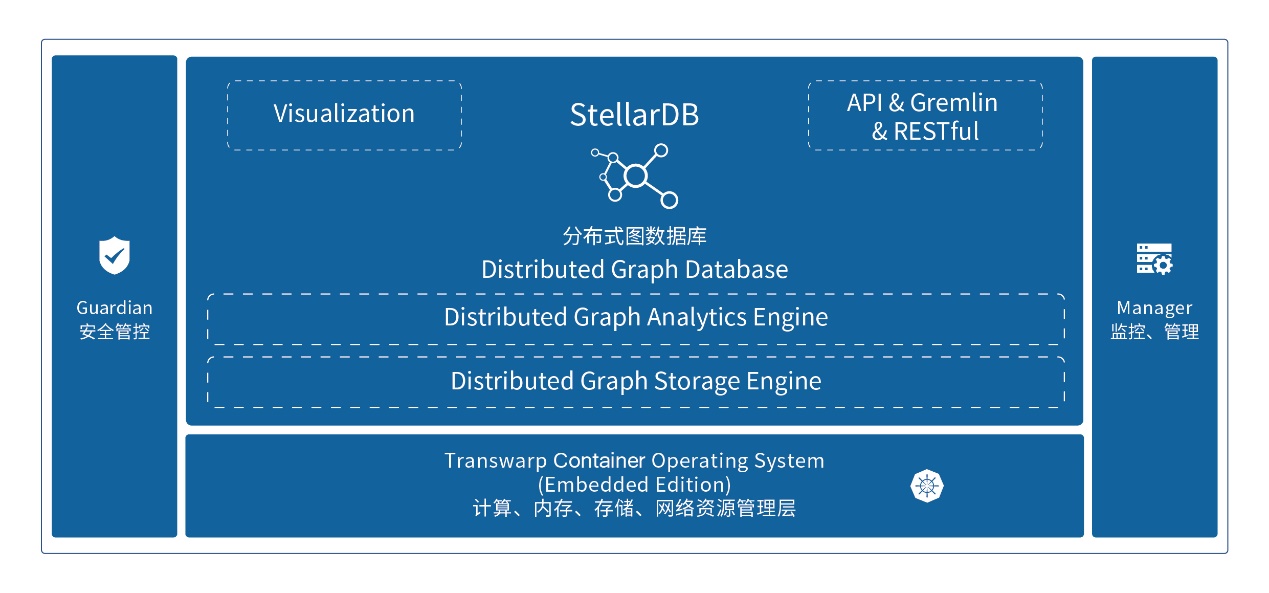
# 第三章 概略设计与算法简介

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（即下图中的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。



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包。模块承接消息处理模块的读写请求，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。

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。

图片：storage-overview.pdf

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。

2. StellarDB存储模块内部设计

图。。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。

图片：inner

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. 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。。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：compaction-basic-sample

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。S1。 ，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：compaction-detail-sample-1

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。S2 。，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。

# 第四章 优化详细设计与实现

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。

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。

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1~9（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1~9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。

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。。。。的情况变成图。。。

图Algo-Before

图Algo-After

 如图。。After。。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，那么在理想情况下：

 原来图。。Algo-Before。。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^2.

4. 解决方案细节研究

①. 分片的片数选择

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。

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。

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Algo-Problem

 如图。。Algo-Problem。。，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。

5. StellarDB优化实现

①flush算法前后对比

Flush算法为优化所作的修改比较简单。如图。。before。。，优化前，flush使用Memtable.flush(Segment)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。

图flush-before

如图。。flush-after。。，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。

图flush-after

②compaction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

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。。compaction-after。。，新算法按照“选用0个L1文件”->“选用1个L1文件”->“选用2个L1文件”->...->“选用所有L1文件”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。

图compaction-after

L0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。

图Algo-Pick

如图Algo-Pick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∅，也就没有必要再搜索walID=103的情况了。

图l0-pick-1,2

# 第五章 测试

1. 测试环境与方案

①测试环境

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。

②测试方法

测试方法是使用io.transwarp.graphsearch.tools.utils.ClientLevelPerformanceTest进行LDBC数据集。。。引用。。。（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。然后利用监控页面以及StellarDB的性能监测RESTful接口收集测试数据。

2. 对比测试结果

表~\cite{table:testResults}展示了算法优化前后以及调节部分参数前后的性能测试结果。

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  |  |  |  |  |  |  |  |  |
|  | 是否优化算法 | Memtable大小 |  |  |  |  |  |  |
| 1 | 否 |  |  |  |  |  |  |  |
| 2 |  |  |  |  |  |  |  |  |
| 3 |  |  |  |  |  |  |  |  |
| 4 |  |  |  |  |  |  |  |  |
| 5 |  |  |  |  |  |  |  |  |
| 6 |  |  |  |  |  |  |  |  |

测试中发现：新的compaction策略下，对于一个有15个shard（active+standby）的图，compaction峰值时会同时使用15个L0 compaction线程与约8个L1 compaction线程。如果配置线程上限少于此值，则有可能会出现compaction starvation。

以下是持续优化策略过程中所做的改进内容以及性能测试的简要说明。

详细内容以及原数据见附件 [compaction\_metrics.tar.gz](http://172.16.1.168:8090/download/attachments/21005239/compaction_metrics.tar.gz?version=1&modificationDate=1561537021000&api=v2)

#### leg:

legacy  
stream control不限  
Memtable.size = 2097152 （2MB）——每个L0文件与experimental一致  
compression level = -,-,-,-  
bloom level = -,-,-,-  
pick.file=4,4,8,2  
file.size=16,16,2048,8192 (MB)

#### exp1:

experimental  
stream control不限  
Memtable.size=10485760 (10MB)  
split count = 5  
compression level = -,-,-,-  
bloom level = -,-,-,-  
pick.file=4,4,8,2  
file.size=16,16,2048,8192 (MB)

#### exp2:

配置同exp1，改进了Metrics，增加一些自动计算项目

#### exp3：

在exp2的基础上，memtable.size与split.count翻倍，即memtable为20MB， 分10片，每个L0文件预期为2MB

#### exp4:

在exp3的基础上，改用2号策略

#### exp5: sf1单shard压力测试（仅本次测试为sf1，其他均为sf10）

加入单次输入容量限制：限制mainWrappers总大小 / viceWrappers总大小的比例（可以直接对应为设置了RA的下限）  
CompactionTimeout保持与之前一样为5000ms，replica=1  
可以观察到：  
L0 compaction调度经常被L1的所阻塞。现实用时28min，但是L0的单线程compaction只调度了18.4min  
从memtable全部flush完成后，在被L1阻塞时，L0的compaction调度动力主要就靠CompactionTimeoutEvent  
我认为单shard测试随机性较大，可能遭遇多shard中L0 compaction被迫等待L1，也可能不遭遇。所以此测试意义不大。

#### exp6: sf10

使用exp5中加入的策略，重新进行性能测试.  
可以看到在leader shard基本可以保持没有L0 Segment积压。L0 compaction速度与写memtable的写入和flush速度差不多。这与exp4是一样的。  
只是由于导入程序是在lev1上运行的，还是多线程，导致lev1的worker没有线程可用，lev1严重滞后。  
但即使不考虑lev1的情况，也发现lev3上的一部分leader的L0由于L1阻塞而调度很不积极，300+的segment数与旁边shard的个位数对比鲜明。  
RA从exp4的80%变为90%，仍可以接受。

#### exp7:

在exp6的基础上，调度时先检查L1相邻文件间的row key间隔能否直接放入L0 compaction task  
从metrics观察到“使用0个viceWrapper”的compaction task数目依然稀少，此变更影响很有限，但有帮助性能提升的潜力，值得保留；  
同时依然可以观察到大约1/3的shard存在L0调度受阻的情况（当然是由于L1的compaction所致）

#### exp8:

在exp7基础上，最后一层改为row key不重叠；而且在handleCompactionTimeoutEvent中，加入了一厢情愿的shuffle，有可能有帮助。  
测试数据与exp7相同，被明显阻塞的shard数目没有减少，于是不保存此次测试性能数据。  
将GRAPH\_COMPACTION\_LEVELED\_THREAD\_NUMBERS改为15,4,4,6重新测试，看能否在前中期尽量提高L0 compaction容量。  
过程中观察到L0同时使用到超过8个线程的时间极少，原来的线程应该没有被线程数所限制。所得的测试数据与exp基本相同。

#### exp9：

GRAPH\_COMPACTION\_LEVELED\_THREAD\_NUMBERS改为15,15,4,6；  
改良输入大小控制条件，在避免过大输入的同时，避免由于偶尔单个L1文件过小导致不能生成compaction plan  
可以观察到不再存在“一小部分shard L0 compaction受阻”的情况，compaction线程利用率较高；  
随着导入脚本运行完成，L0 compaction也完成（compaction速度比导入脚本后期写入速度快），其他Level的文件数也正常，一直在使用旧式compaction策略  
至此可以确定“L0 compaction受阻”是由不合理的输入限制+随机出现的L1小文件导致的，受阻后一直等待L1的LeftTimeout与L0的更多数据。  
从性能数据上面看，总体RA率有所升高，在导入脚本降速之前（导入脚本是每个线程parse一个csv，但是一些较短的csv一段时间后消耗完，就只有几个线程还在parse剩的长csv），  
RA可以保持80~85%，降速之后，RA逐步上升，最终变为将近120%。不过RA并不是目的，当数据变少，RA上升是意料中的，只要compaction速度跟得上就没有问题。

# 第六章 总结与展望

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。

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。

# 摘要&英文

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，并简单展示了优化前后性能对比测试结果。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。