[V2.0 compaction性能改进](http://172.16.1.168:8090/pages/viewpage.action?pageId=20266790)

[Skip to end of metadata](http://172.16.1.168:8090/pages/viewpage.action?pageId=20266790#page-metadata-end)

* Added by [徐江河](http://172.16.1.168:8090/display/~jianghe.xu), last edited by [徐江河](http://172.16.1.168:8090/display/~jianghe.xu) on Oct 16, 2019  ([view change](http://172.16.1.168:8090/pages/diffpages.action?pageId=20266790&originalId=22686242))

[Go to start of metadata](http://172.16.1.168:8090/pages/viewpage.action?pageId=20266790#page-metadata-start)

根据一些POC的经历，团队发现StellaDB的compaction性能不足，在bulk load过程中会有大量的数据卡在Level 0而无法compaction至下层。所以需要进行compaction性能提升的改进。

1. 首先，先后进行了两次性能测试：[StellaDB 1.1性能测试报告](http://172.16.1.168:8090/pages/viewpage.action?pageId=20266573)      [StellaDB 2.0客户端性能测试报告 - 2019.5](http://172.16.1.168:8090/pages/viewpage.action?pageId=20266620)   
   但如后一次测试中所提到的，这样的bulk load性能测试设计还不够完善。应当改良测试设计，建立易操作、度量易获取的测试流程。
2. 在进行代码评审的时候发现了性能改进点与bug，修改的同时也改善了compaction的性能
   1. [WARP-33866](http://172.16.0.244:8080/browse/WARP-33866)
   2. [WARP-33880](http://172.16.0.244:8080/browse/WARP-33880)
3. 根据之前的测试，刘伟提出了一些改进建议。参见： [StellaDB 1.1 BulkLoad性能测试与底层存储优化建议](http://172.16.1.168:8090/pages/viewpage.action?pageId=20270394)
4. 提出性能改进方案：[Compaction性能改进策略 - Flush文件分片](http://172.16.1.168:8090/pages/viewpage.action?pageId=20270823)  方案设计、实施计划等参见页面

[[临时]其他将会使用的compaction改进手段](http://172.16.1.168:8090/pages/viewpage.action?pageId=20270871)

[Skip to end of metadata](http://172.16.1.168:8090/pages/viewpage.action?pageId=20270871#page-metadata-end)

* Added by [徐江河](http://172.16.1.168:8090/display/~jianghe.xu), last edited by [徐江河](http://172.16.1.168:8090/display/~jianghe.xu) on Jun 26, 2019  ([view change](http://172.16.1.168:8090/pages/diffpages.action?pageId=20270871&originalId=21005244))

[Go to start of metadata](http://172.16.1.168:8090/pages/viewpage.action?pageId=20270871#page-metadata-start)

* 目前的配置项只有GRAPH\_COMPACTION\_MIN\_FILES\_PER\_LEVEL对于开始进行compaction的时机做了规定、GRAPH\_COMPACTION\_INI\_PICK\_FILES\_PER\_LEVEL对于每次pick上一层的segment数目做了规定，这两项配置项并不能很好地帮助CompactionHandler计算应当优先挑哪一层去compact、可以挑多少segment去compact。根据LevelDB和RocksDB的经验，应该配置每一层的预期总文件大小，每下一层单个文件大小等比增大（比如10倍扩大），根据每层实际大小、文件数、预期大小等数据计算各层的compaction优先级，更灵活地选择上一层进入compaction的文件数目（上一层适当地选多一些，尤其是在不增加下层选择的文件时，可以降低下层文件占比，减少由此带来的读写扩大）。

# [Compaction性能改进策略 - Flush文件分片](http://172.16.1.168:8090/pages/viewpage.action?pageId=20270823)

[Skip to end of metadata](http://172.16.1.168:8090/pages/viewpage.action?pageId=20270823#page-metadata-end)

* [Attachments:4](http://172.16.1.168:8090/pages/viewpageattachments.action?pageId=20270823&metadataLink=true)
* Added by [徐江河](http://172.16.1.168:8090/display/~jianghe.xu), last edited by [徐江河](http://172.16.1.168:8090/display/~jianghe.xu) on Jun 25, 2019  ([view change](http://172.16.1.168:8090/pages/diffpages.action?pageId=20270823&originalId=21004946))

[Go to start of metadata](http://172.16.1.168:8090/pages/viewpage.action?pageId=20270823#page-metadata-start)

### 背景

 由之前的bulk load性能测试可知，目前compaction的主要问题是Level0文件积压。也就是说，L0文件生产得快（写入memtable与flush的速度写得）、消费得慢（L0->L1的compaction很慢），性能瓶颈是L0至L1的compaction性能。

 根据使用的测试数据（LDBC数据集），我们发现，L0中的文件的条目rowkey，很可能基本分布于整个的rowkey“定义域”。例如：LDBC中的userid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rowkey的条目，第2个文件里面基本都是“2”打头的rowkey的条目… … 但是，一个L0文件却同时拥有1~9打头的rowkey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：

1. 每次L0->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；

2. 虽然compaction thread可能开了许多，但是L0->L1的compaction任务每次只能有一个thread执行（因为所有L1文件都被它占用、上锁），并行度差。

### 解决思路与方案

 我们希望通过避免任务占用所有L1文件，降低L0→L1 compaction的读写扩大、提高并行度。所以，L0→L1 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rowkey定义域限制为仅包含一小部分rowkey的范围。

也就是把：（此为“图1”）

变成：（此为“图2”）

 在flush生成L0文件的时候，根据RK1、RK2等rowkey分划，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→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 thread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 task使用的L0输入size固定为1个mentable的大小，那么在理想情况下：

 原来图1将Segment 1、2 、3compact进入Level1，每个task会使用的输入的size为（memtable size \* 1 + level-1 size），输出的size为(level-1 size)，**串行**执行3遍，

 改进后每个task会使用的输入size**约**为(memtable size \* 1 + level-1 size / 3)，输出的size**约**为(level-1 size / 3)，**并行**执行3遍。

 由于以上数值（包括分片数为3而非其他值）都是假想，所以并不方便计算读写、时间的节约比例。不过很容易看出，如果认为L0 pick size <<L1 size的话，同样的compaction任务对于L0进行N个分片，总读写量变成了约1/N，时间有潜力变为1/N^2.

### 方案细节讨论

1. 分片的片数选择

 这一点需要实际测试才能得到较好的原则。分片太细的话，要么每次compaction pick的L0小文件太多，比较与merge效率降低，要么同步增大memtable size，导致L0总size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shard可用的compaction thread数，达到thread充分利用。（其实thread数的设定目前也是比较随意，这两个参数都有待讨论）

2. 分片的rowkey分割点问题

 显然，我们需要能够把原来的L0文件比较均匀的分划开的分割点，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userid是均匀随机分布于定义域中（用户可能使用整形，也可能用英文等等），那么第一个生成的immutable memtable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。

 也有观点认为，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。

 如果上面的假设不成立，那么我们需要使用一种策略持续计算合适的分割点并且动态调整实际应用的分割点。有关如何调整还有待商榷。一种简单的思路是：调整的频率如果较低的话，可以暂时停止L0的写入（阻塞flush，积压immutable memtable），L0全部完成compaction之后，调整分割点并开启flush。

3. 是否必须有确定的分割点

 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pick L0的被切细的文件，也能达到上面的严格分片的效果。本人认为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，Seg1+2是更早的memtable flush的产物，如果compaction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rowkey区间，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。

 毫无疑问，这种策略在编码修改上是相当简单的，也避免了前面选择分割点的一系列讨论。这个方案可以作为实验方案，作为前面的复杂方案的先行验证。在高度随机的数据集中，本方案缺点就能够被尽量避免，减少输入的扩大。

### 方案优缺点

1. 从上面的计算来看确实有很大的性能提升的帮助；

2. 已经不是LevelDB或者RocksDB已验证的成熟方案，有风险；

3. 带来了额外的调优参数；

4. 一定程度上依赖Bulk load时rowkey的随机分布。如果load时都是从rowkey 1打头的开始，按顺序到9打头的，那么本策略将没有帮助（实际上，为了让这个策略生效，先shuffle输入数据也不是不行）。

以上内容包含于 [改进策略.pdf](http://172.16.1.168:8090/download/attachments/20270823/%E6%94%B9%E8%BF%9B%E7%AD%96%E7%95%A5.pdf?version=1&modificationDate=1560326621000&api=v2)

### 方案具体实施计划

1. 从目前的测试环境配置的“每个shard分配到的compaction thread数一般不超过1”的情况来看，WARP-34457虽然在本策略下理论上会变得更容易出现，但实际工作中仍然基本不会出现的修复；WARP-33837同理。所以虽然本次compaction改进应当修复这两个缺陷，但是可以在带有着两个不紧急的缺陷的同时进行有效的性能测试验证。

2. 将会先较快速地实现“不适用分割点，而是根据memtable长度分割”的策略，并进行测试。因为预计这种策略在随机导入的过程中（即测试用LDBC数据集）性能提升接近于使用动态分割点的策略。（本周内，即最晚6.14，可以完成）

3. 如果2带来了显著性能提升，则探索动态分割点的实现方法（可以通过结合数据集特征先设静态分割点的方法，找出最优静态分割点的特征，推测动态分割点计算方法），否则，悲伤地探索动态分割点的实现方法。

4. 最后，进行对WARP-33837/34457的检查。3中的修改有可能涉及到这两个bug的代码而顺带修复。并且，改进compaction metrics，使其充分详细，方便观测compaction的过程与性能。

# [Graphene](http://172.16.1.168:8090/display/GRAP)

# ****…****

# 

# [V2.0 compaction性能改进](http://172.16.1.168:8090/pages/viewpage.action?pageId=20266790)

# [Compaction性能改进策略 - 测试数据](http://172.16.1.168:8090/pages/viewpage.action?pageId=21005239)

[Skip to end of metadata](http://172.16.1.168:8090/pages/viewpage.action?pageId=21005239#page-metadata-end)

* [Attachments:1](http://172.16.1.168:8090/pages/viewpageattachments.action?pageId=21005239&metadataLink=true)
* Added by [徐江河](http://172.16.1.168:8090/display/~jianghe.xu), last edited by [徐江河](http://172.16.1.168:8090/display/~jianghe.xu) on Jun 27, 2019  ([view change](http://172.16.1.168:8090/pages/diffpages.action?pageId=21005239&originalId=21005568))

[Go to start of metadata](http://172.16.1.168:8090/pages/viewpage.action?pageId=21005239#page-metadata-start)

测试方法是使用io.transwarp.graphsearch.tools.utils.ClientLevelPerformanceTest进行LDBC数据集（sf10，点边数分别为36485769、231371311）的导入，利用监控页面以及RESTful接口（/metrics/compaction/0）收集测试数据。

在测试中发现：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。

[Compaction性能改进策略 - 输入文件选择](http://172.16.1.168:8090/pages/viewpage.action?pageId=21004945)

[Skip to end of metadata](http://172.16.1.168:8090/pages/viewpage.action?pageId=21004945#page-metadata-end)

* [Attachments:1](http://172.16.1.168:8090/pages/viewpageattachments.action?pageId=21004945&metadataLink=true)
* Added by [徐江河](http://172.16.1.168:8090/display/~jianghe.xu), last edited by [徐江河](http://172.16.1.168:8090/display/~jianghe.xu) on Jun 26, 2019  ([view change](http://172.16.1.168:8090/pages/diffpages.action?pageId=21004945&originalId=21005241))

[Go to start of metadata](http://172.16.1.168:8090/pages/viewpage.action?pageId=21004945#page-metadata-start)

上一个策略说明实质只包括了flush策略，本页面说明与其配合的输入文件选择策略。

设计思路：降低读扩大率

我们在挑选compaction的输入文件时，应当多选高层的（指L0），少选低层的（指L1）：低层的文件数据被读出、merge后，又要写回低层，是一种读扩大，浪费了磁盘与CPU资源。使用旧有的的compaction策略（配合flush不分片），读扩大率（read amplification ratio，以下简称RA）约为420%，也就是每消费100MB的L0文件，compaction要同时消费420MB的L1文件与其merge。我们希望在“L0各个文件内部row key相对聚集”的前提下，通过灵活的文件选取策略，既保障compaction并行度，又显著降低RA，也不要把每次的输入总量搞得过大导致“伪卡死”。

方案

综合考虑并行问题与效率问题，我们选择：

1. 每个shard的每个level同时最多只能有一个compaction线程；
2. 先选尽量少的低层segment，再根据低层的segment的row key range去选尽量多的高层segment；
3. 限制输入规模的参数不再使用原来的GRAPH\_COMPACTION\_INI\_PICK\_FILES\_PER\_LEVEL，而是另行设计。

下面对每个设计进行详细说明。

1. 每个shard的每个level同时最多只能有一个compaction线程  
   这个决策是主要为了解决 [WARP-34457](http://172.16.0.244:8080/browse/WARP-34457) ， 同时，根据测试，实际运行环境中并不存在一个Level需要占用多个线程的情况。所以这样设计简化了同步逻辑，能减少潜在的bug。
2. 先选尽量少的低层segment，再根据低层的segment的row key range去选尽量多的高层segment  
   这个决策一方面是希望解决 [WARP-33837](http://172.16.0.244:8080/browse/WARP-33837)； 另一方面，根据测试数据，发现选的低层segment越少，总体数据吞吐越高（消费L0文件容量/消耗时间 ），而且，减少打开的segment数目也有助于减轻系统与磁盘压力。所以，只使用1个L1文件的方案优先于使用2个L1文件的方案。另外，使用0个L1文件的方案最优先：当L1没有文件时，或者L1文件间的row key间隙能够放的下某些（walID最小的）L0文件的row key range时，便可以这样调度。
3. 限制输入规模的参数不再使用原来的GRAPH\_COMPACTION\_INI\_PICK\_FILES\_PER\_LEVEL，而是另行设计  
   本方案实现初期，想要彻底抛开输入规模限制，测试时发现可能会调度出同时用数百个文件作为输入的compaction任务，此任务耗费时间极长，让系统看起来像是卡死，而且实际上由于打开文件太多，读写效率也低，数据吞吐率差。所以，本方案新增了配置项GRAPH\_COMPACTION\_MAX\_MAIN\_VICE\_RATIO与GRAPH\_COMPACTION\_L0\_MAX\_PICK。前者规定了选取L0文件总大小与L1文件总大小的最大比值（也是任务的RA的下限的倒数），后者规定了一次调度所能够选取的L0文件数目的最大值。在选取L0文件时，只有同时违反了这两个限制，才会停止选取更多L0文件。由于compaction输出可能生成容量极小的“零头”L1文件，所以当L1如果只有1个这样的零头而没有正常的较大的L1文件，GRAPH\_COMPACTION\_MAX\_MAIN\_VICE\_RATIO便会阻止一切L0文件被选入调度（直到这个零头由于L1的level life timeout被compact到L2）；由于压缩算法，对于某些很“有规律”的数据集，L0也有可能出现大量容量极小的文件，GRAPH\_COMPACTION\_L0\_MAX\_PICK会阻止这些小文件被快速compact。因此，我们需要这两个限制条件共同作用。

文件选择算法详细说明

算法按照“选用0个低层文件”->“选用1个低层文件”->“选用2个低层文件”->...->“选用所有低层文件”的顺序，计算出可以使用的row key range，根据range尝试计算合适的高层文件选择计划，从中选择RA最小的进行调度。

如果是要用多个低层文件，那么底层文件要按照start row key排序，然后只尝试选择相邻的低层文件。这样才能使它们构成的row key range全部可用。

计算row key range不仅可以包含尝试选取的低层文件，还可以包括它（们）与相邻文件之间的row key range空隙。这样可以最大化用于选取高层文件的row key range。

选取高层文件时，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row key range。如图，英文字母表示segment里面包含的条目row key：

我们想要尝试计算只选取Seg85时，能够包含进compaction任务的L0文件。首先Seg85的row key range是[I,K]，但是实际上可以扩大到(G,N)。注意：这个区间是开区间。然后根据walID顺序选择L0文件并缩小row key range。首先对于walID=100， 可以选择I与M，同时没有被选取的segment无法影响row key range；然后处理walID=101，只能选择K+L这一个segment，而且由于没被选入的segment影响，row key range被缩小到了(J,M)；再看walID=102，发现segment J+M已经不能选择，因为区间是(J,M)而不是[J,M]，而且，segmemt J+M直接把row key range缩减到∅，也就没有必要再搜索walID=103的情况了。

方案效果

在持续处理高频写入时，RA保持在80%左右；若写入频率低，RA会少量上升。对于现有的解析CSV的导入脚本来说，在开20个shard+leviathan1~4集群的条件下，L0的compaction与峰值写入速度基本同步，不会积压L0文件。

详细效果可以参考 [Compaction性能改进策略 - 测试数据](http://172.16.1.168:8090/pages/viewpage.action?pageId=21005239)