背景

对于需要保存超大状态（远超于内存容量）的流计算场景来说，目前 RocksDB [1] 是 Flink 平台上官方实现的唯一选择。业界也有使用 Redis 等其他服务作为状态后端的方案，但终究不够成熟，且已被社区否决 [2].

基于我们长期的状态调优经验，通过合理的资源分配，RocksDB 方案可以稳定支持上百 GB 甚至上 TB 的总状态量；但是众所周知的是，RocksDB 的可调参数非常繁琐，有上百个之多，且彼此之间还相互影响，非常难以调整。更致命的是，默认参数和配置不当的参数，读写性能会比较差，甚至会成为严重的性能瓶颈。

因此我们对 Flink 上的 RocksDB 的参数调优方法进行了梳理，希望能够帮助大家解决相关的问题。

概念

RocksDB 是一个基于 LevelDB 衍生的键值存储数据库，它内部的数据以 ColumnFamily（列族，亦有译为列簇）为逻辑单位进行存储，类似于关系数据库中的“表”（Table）。所有的 ColumnFamily 共享同一份 WAL（Write-Ahead Log，用于崩溃恢复的流水日志），但是独享自己的 MemTable（可以认为是内存中的写入缓存，默认是基于 Skip-List 跳表实现的），以及 SST 文件组（MemTable 进行 Flush 操作以后，保存到磁盘上的有序数据文件，按 Key 排序并划分为若干 Data Block，并附加有索引和可选的布隆过滤器等内容，以方便快速查找）。

当 MemTable 写满了以后，会变成不可变的 MemTable，然后后台定期刷写（Flush）到磁盘上；而在磁盘上后，还有定期的 Compaction 定时整理机制，对重复的数据进行去重和压缩重整，然后分层存储（Level 0 简称 L0、L1、L2... 以此类推，每层的大小不一样，旧数据通过 Compaction 操作，逐步向下一层存储）。

当用户发出读取请求时，RocksDB 先从 MemTable 查找；如果没找到，再查找不可变的 MemTable，随后再磁盘上进行逐级查找。加载数据块时，Block Cache 内存结构可以存储一些常用到的数据块，并可以把它的索引和布隆过滤器等结构也加载到内存，方便检索。RocksDB 还提供了参数可以把 L0 的索引等结构长期保留在 Block Cache 中，以加快查找速度；但这样也会长期占用 Block Cache 的空间，让能缓存的数据块个数变得更少。

因此，这些也体现了 RocksDB 调优的难点：容易顾此失彼，且需要在读放大（例如本来想读 1 KB 的数据，结果不得不读取 10 KB 的数据才能得到想要的，则读放大因子是 10）、写放大（例如本来想写 1 KB 的数据，结果写了 5 KB 的数据量，则写放大因子是 5）、空间放大（想存储 1 KB 的数据，结果实际占用了 100KB 等，那么空间放大的因子是 100）等效应之间权衡。

RocksDB 的 JNI API 提供了 DBOptions、ColumnFamilyOptions，以及 BlockBasedTableConfig 等类，来允许用户进行参数设置，而在 Flink 的 RocksDBStateBackend 类中，可以通过 setPredefinedOptions() 方法来传入这些配置。

如果想学习更多的概念知识，可以参考官方文档 [3].

调优参数说明

Block Cache 系列参数

Block 块是 RocksDB 保存在磁盘中的 SST 文件的基本单位，它包含了一系列有序的 Key 和 Value 集合，可以设置固定的大小。经过我们的长期验证，发现 Block Size 及其 Cache 的大小设置，对读写性能的影响最大。

Block Size

在 Flink 中，调整块大小的配置参数 [4] 是 state.backend.rocksdb.block.blocksize，底层是 ColumnFamily 对象的 setTableFormatConfig(new BlockBasedTableConfig().setBlockSize() 方法。

Block Size 默认值为 4KB，文档中建议生产环境调整到 16 ~ 32 KB；如果采用机械硬盘（HDD）来存放 SST 文件，那么在内存容量重组的情况下，可以调整为 128 KB，这样单次读取的数据量可以多一些；而且 Block 变大后，相同的总数据量下，索引所占用的内存会减少。

但是，通过增加 Block Size，会显著增加读放大（Read Amplification）效应，令读取数据时，吞吐量下降。原因是 Block Size 增加以后，如果 Block Cache 的大小没有变，就会大大减少 Cache 中可存放的 Block 数。如果 Cache 中还存处理索引和过滤器等内容，那么可放置的数据块数目就会更少，可能需要更多的磁盘 IO 操作，找到数据就更慢了，此时读取性能会大幅下降。反之，如果减小 Block Size，会让读的性能有不少提升，但是写性能会下降，而且对 SSD 寿命也不利。

因此我们的调优经验是，如果需要增加 Block Size 的大小来提升读写性能，请务必一并增加 Block Cache Size（接下来要介绍）的大小，这样才可以取得比较好的读写性能。如果内存已经吃紧，那么不建议继续增加 Block Cache Size，否则会有 OOM 的风险（如果在容器环境下限定了使用的内存总将的话，会更明显），那相对来说，也就不建议继续增加 Block Size 了，可以适当减少 Block Size 来提升读取吞吐量，同时关注写性能是否可以接受。

Block Cache Size

Block Cache Size 对于读性能来说至关重要，在 Flink 中的对应参数是 state.backend.rocksdb.block.cache-size. 默认情况下，缓存清除算法用的是 LRU（Least Recently Used），这是一种有锁的算法。RocksDB 还提供了 Clock 算法可选。在我们的常见测试场景下，算法的影响并不大。

以我们的经验来看，增加 Block Cache Size，可以明显增加读性能。默认大小为 8 MB，但是通常在内存富余的情况下，建议设置到 64 ~ 256 MB。

在 Flink 1.10 等高版本，可以启用 state.backend.rocksdb.metrics.block-cache-usage 监控指标，以实时观察 Block Cache 的用量情况，并作出针对性的优化。

Index 和 Bloom Filter 系列参数

每个 SST 都可以有一个索引（Index）和 Bloom Filter（布隆过滤器），他们可以提升读性能，因为有了索引，不必顺序遍历整个 SST 文件，就可以定位具体的 Key 在哪里，因为已经保存了所有的 Key、Offset、Size 等元数据；而通过布隆过滤器，可以在假阳（False Positive）率很低的情况下，迅速判断某个 Key 是否在这个 SST 文件中，如果返回 False 就不再继续找索引了。

Max Open Files

这个参数决定了 RocksDB 可以打开的最大文件句柄数，在 Flink 的参数里是 state.backend.rocksdb.files.open. 如果没有启用 cache_index_and_filter_blocks 参数（下面会讲）时，Max Open Files 可以表示内存中容纳的 Index 和 Filter Block 的数目。默认值是 5000，如果进程的 ulimit 没有限制的话，建议改为 -1（无限制）。

这个参数如果过小，就会出现索引和过滤器 Block 无法载入内存的问题，导致读取性能大幅下滑。

Cache Index And Filter Blocks

cache_index_and_filter_blocks 这个参数（blockBasedTableConfig.setCacheIndexAndFilterBlocks）很有趣，默认是 false，表示不在内存里缓存索引和过滤器 Block，而是用到了载入，不用就踢出去。如果设置为 true，则表示允许把这些索引和过滤器放到 Block Cache 中备用，这样可以提升局部数据存取的效率（无需磁盘访问就知道 Key 在不在，以及在哪里）。但是，如果启用了这个选项，必须同时把 pin_l0_filter_and_index_blocks_in_cache（blockBasedTableConfig.pinL0FilterAndIndexBlocksInCache）参数也设置为 true，否则可能会因为操作系统的换页操作，导致性能抖动。

需要注意的是，对于此参数，一定要注意 Block Cache 的总大小有限，如果允许 Index 和 Filter 也放进去，那么用来存放数据的空间就少了。因此，我们建议在 Key 具有局部热点（某些 Key 频繁访问，而其他的 Key 访问的很少）的情况下，才打开这两个参数；对于分布比较随机的 Key，这个参数甚至会起到反作用（随机 Key 时，读取性能大幅下降）。

Optimize Filter For Hits

optimize_filters_for_hits 这个参数（columnFamilyOptions.setOptimizeFiltersForHits）如果设置为 true，则 RocksDB 不会给 L0 生成 Bloom Filter，据文档中描述，可以减少 90% 的 Filter 存储开销，有利于减少内存占用。但是，这个参数也仅仅适合于具有局部热点或者确信基本不会出现 Cache Miss 的场景，否则频繁的找不到，会拖累读取性能。

对于 Cache 和 Filter 等内存用量，可以通过 Flink 的 state.backend.rocksdb.metrics.estimate-table-readers-mem 监控指标来估计。

MemTable 系列参数

MemTable 是 RocksDB 重要的内存结构，也是 LSM 树的核心实现，通常使用 Skip List 跳表来实现。它可以看作是一个内存中的 Write Buffer（写缓冲），影响着写性能。

通常情况，Flink 里每个 State 都对应了一个 ColumnFamily，而每个 ColumnFamily 都会有自己的 MemTable。因此，在计算内存占用时，一定要把 Flink State 的个数算进去。如果状态数很多的话，内存用量就会飞速增长，从而导致 OOM 或者不稳定。

在最新的 Flink 1.10 中提供了全自动托管的 RocksDB 内存管理方案，可以对不同 ColumnFamily 之间的内存进行一定程度的共享，从而大大降低内存占用，这些在后续文章中会有更详细的介绍。

Write Buffer Size

Flink 的 state.backend.rocksdb.writebuffer.size 参数，可以控制 Write Buffer 即 MemTable 在内存里的空间占用情况。默认大小是 64 MB（RocksDB 官方文档写错了，写着只有 4 MB，与实际不符）。在 RocksDB 5.6 版本之后，这些空间还可以算进 Block Cache Size，通过自定义 Write Buffer Manager 的方式，进行统一的大小管理。对于 Flink 的 RocksDB 衍生版，FRocksDB 而言，就是通过这种方式来实现内存容量的上限管理的。

通常来说，Write Buffer 越大，写放大效应越小，因而写性能也会改善。不过这个参数的调整，必须随着下面的几个参数一起来做，否则可能会达不到预期的效果。

Max Bytes For Level Base

Flink 对应的参数是 state.backend.rocksdb.compaction.level.max-size-level-base. 需要特别注意的是，如果增加 Write Buffer Size，请一定要适当增加 L1 层的大小阈值（max_bytes_for_level_base），这个因子影响非常非常大。如果这个参数太小，那么每层能存放的 SST 文件就很少，层级就会变得很多，造成查找困难；如果这个参数太大，则会造成每层文件太多，那么执行 Compaction 等操作的耗时就会变得很长，此时容易出现 Write Stall（写停止）现象，造成写入中断。

Max Bytes For Level Multiplier

max_bytes_for_level_multiplier 参数决定了每层级的大小阈值的倍数关系。如果不考虑其他因子的影响，如果 max_bytes_for_level_base = 1GB，max_bytes_for_level_multiplier = 5，那么 L1 的大小阈值是 1GB，L2 的大小阈值是 5GB，L3 的大小阈值是 25GB... 以此类推，所以 RocksDB 的默认分层存储叫做 Leveled 结构，有点类似于阶梯（如果启用 state.backend.rocksdb.compaction.level.use-dynamic-size 参数，则更加复杂）。

除了 Leveled 结构以外，还有 Universal 和 Tiered 等其他结构安排，这里不再深入展开。

这个 max_bytes_for_level_multiplier 参数对写入性能影响也是非常大的，请根据实际情况进行调整，没有一个统一的规则。

Write Buffer Count

Flink 的 state.backend.rocksdb.writebuffer.count 参数（也可以通过 columnFamilyOptions.setMaxWriteBufferNumber 设置）可以控制内存中允许保留的 MemTable 最大个数，超过这个个数后，就会被 Flush 刷写到磁盘上成为 SST 文件。

这个参数的默认值是 2，对于机械磁盘来说，如果内存足够大，可以调大到 5 左右，以令 MemTable 的大小减小一些，降低 Flush 操作时造成 Write Stall 的概率。

Min Write Buffer Number To Merge

Flink 的 state.backend.rocksdb.writebuffer.number-to-merge 参数（columnFamilyOptions.setMinWriteBufferNumberToMerge）决定了 Write Buffer 合并的最小阈值，默认值为 1，对于机械硬盘来说可以适当调大，避免频繁的 Merge 操作造成的写停顿。

根据我们的调优经验来看，这个参数调小、调大都会造成性能下滑，它的最佳值会在某个中间值附近，例如 3 等。

对于 MemTable 所占用的内存大小估算指标，可以启用 Flink 的 state.backend.rocksdb.metrics.cur-size-all-mem-tables 参数来实时监控。

Flush 和 Compaction 相关参数

RocksDB 的后台进程中，有持续不断的 Flush 和 Compaction 操作。前者将 MemTable 的内容刷写到磁盘的 SST 文件中；后者则会对多个 SST 文件做归并和重整，删除重复值，并向更高的层级（Level）移动。例如 L0 -> L1 等。

频繁的 Flush 和 Compaction 操作，在写数据量大时，会严重影响性能，甚至造成写入的完全停顿，即 Write Stall，因此这里也需要进行细致的调优。

Target File Size

Flink 参数为 state.backend.rocksdb.compaction.level.target-file-size-base（ColumnFamilyOptions 的 setTargetFileSizeBase 方法），表示上一级的 SST 文件达到多大时触发 Compaction 操作，默认值是 2MB（每增加一级，阈值会自动乘以 target_file_size_multiplier）. 为了减少 Compaction 的频率，可以适当调大此参数，例如调整为 32MB 等，此参数对性能的影响也比较大。

但是，调大这个参数以后，只是推迟了 Compaction 的时机，并没有真正减少数据量，因此可能会造成重复数据不能及时清理（影响读性能），或者一次需要清理超多的数据（影响写性能），因此这个参数比较缺乏灵活性，我们正在设计一种按需 Compaction 的算法来改善这种场景。

Dynamic Level Bytes

state.backend.rocksdb.compaction.level.use-dynamic-size 参数允许 RocksDB 对每层的存储的数据量阈值进行动态调整，不再是简单的 Level Base 的倍数关系，这样生成的 LSM 树更稳定。

这个参数的默认值为 false，对于机械硬盘用户，建议设置为 true.

Compaction Style

state.backend.rocksdb.compaction.style 参数（ColumnFamilyOptions 的 setCompactionStyle 方法）允许用户调整 Compaction 的组织方式，默认值是 LEVEL（较为均衡），但也可以改为 UNIVERSAL 或 FIFO.

相对于默认的 LEVEL 方式，UNIVERSAL 属于 Tiered 的一种，可以减少写放大效应，但是副作用是会增加空间放大和读放大效应，因此只适合写入量较大而读取不频繁，同时磁盘空间足够的场景。

FIFO 则适合于将 RocksDB 作为时序数据库的场景，因为它是先入先出算法，可以批量淘汰掉过期的旧数据。

Compression Type

ColumnFamilyOptions 类提供了 setCompressionType 方法，可以指定对 Block 的压缩算法。

RocksDB 提供了无压缩、Snappy、ZLib、BZlib2、LZ4、LZ4HC、Xpress、ZSTD 等多种压缩算法支持，但是需要注意的是，启用前需要确认系统中是否已经装好了对应的压缩算法库，否则可能无法正常运行。

如果追求性能，可以关闭压缩（NO_COMPRESSION)；否则建议使用 LZ4 算法，其次是 Snappy 算法。启用压缩后，ReadOptions 的 verify_checksums 选项可以关闭，以提升读取速度（但是可能会受到磁盘坏块的影响）。

Thread Num（Parallelism）

state.backend.rocksdb.thread.num 这个参数允许用户增加最大的后台 Compaction 和 Flush 操作的线程数，Flush 操作默认在高优先级队列，Compaction 操作则在低优先级队列。

默认的后台线程数为 1，机械硬盘用户可以改为 4 等更大的值。

如果后台所有的线程都在做 Compaction 操作时，如果这时候突然有很多写请求，就会引发写停顿（Write Stall）。写停顿可以通过日志或者监控指标来发现。

Write Batch Size

state.backend.rocksdb.write-batch-size 参数允许指定 RocksDB 批量写入时占用的最大内存量，默认为 2m，如果设置为 0 的话就会自动根据任务量进行调整。这个参数如果没有特别的需求，可以不调整。

Max Background Compactions

DBOptions 类的 setMaxBackgroundCompactions 方法，可以设置最大的后台并行 Compaction 作业数。

如果 CPU 负载不高的话，建议增加这项的值，以允许更多的 Compaction 同时进行，减少读放大和空间放大，提升读取效率；但是如果设置的过大，可能会造成性能损耗，因为 Compaction 操作会带来停顿。

Set Max SubCompactions

DBOptions 类还允许用户通过 setMaxSubcompactions 选项，把 Compaction 操作拆分为并行的子任务。这个选项资料较少，我们测试来看，性能影响也不大，因此可以暂时忽略。

Set Level 0 File Num Compaction Trigger

ColumnFamilyOption 类的 setLevel0FileNumCompactionTrigger 选项，可以指定 L0 触发 Compaction 操作的文件个数阈值。默认值为 4，可以调大一些，以减少 Compaction 操作的频率（但是会带来 Compaction 时间的延长）。

Max Background Flushes

DBOptions 类的 setMaxBackgroundFlushes 可以设置后台最多同时进行的 Flush 任务数，默认值为 0. 如果设置为非 0 值，则表名把 Flush 任务放到高优先级队列。

我们建议把这项的值设置为 1 即可，不需要太大。

Set Use Fsync

DBOptions 也提供了 setUseFsync 选项，让用户可以启用 fsync 同步写入磁盘，避免数据丢失，默认值是 false。

我们建议 Flink 用户也设置为 false，因为 Flink 本身已经提供了 Checkpoint 机制来恢复状态，所以 WAL 和 fsync 等安全机制只会带来性能开销却不能带来好处。

另外，机械硬盘用户，可以把另一个选项 setDisableDataSync 设置为 true，这样可以使用操作系统提供的 Buffer 来提升性能。

Set Num Levels

ColumnFamilyOptions 的 setNumLevels 方法，可以设置总层级数。这个参数设置大一些没关系，最多就是高层次没有数据而已。

默认值是 7，表示最多有 7 层。如果数据实在太多，可以设置为更大的值。

总结

RocksDB 提供了丰富的设置项，而 Flink 在此基础上封装为了自己的参数（大多可以通过配置项来设置，个别需要修改 StateBackend 初始化的代码），并将部分 RocksDB 的统计指标展示给用户，方便定位性能问题。RocksDB 官方提供了性能优化指南 [5]，也可以根据这些来进行参数调优。

同样的，1.10 新版的 Flink 提供了全自动的 RocksDB 内存管理（Managed）[6]，以大大简化容器环境下内存的最大用量设置，避免因为超过了允许使用的最大内存量而被系统 KILL 或发生 OOM.

对于 RocksDB 基础概念不熟悉的同学，也可以参考这个中文版的 RocksDB 笔记 [7]，它对 RocksDB 的基础概念讲解的比较清晰。

对于学术界和业界，RocksDB 也有不少的调优论文，例如 Keren Ouaknine 等人提出的优化 Redis-on-Flash 的 RocksDB 性能的文章 [8]，较科学地评测了该场景下的最优参数，并给出保持稳定吞吐量和时延，减少停顿的建议。此外，还有关注区块链场景下，对 RocksDB 读性能进行调优和参数分析的论文 [9]，文中评估了 Bloom Filter 的位数对内存占用、性能的影响，根据测试结论，多方均建议保持默认的 10 位不变。还有三星研究所的 Fine-tuning RocksDB for NVMe SSD 报告，也说明了不同场景下测得的最优参数。

由此看来，RocksDB 参数虽然繁杂，但是了解了原理和测试方法以后，用户可以针对具体的场景进行有效的调整。经过我们的调研，对默认参数进行优化后，读性能有将近 800% 的提升，而写性能也有不同程度的改善，因此 RocksDB 调优是非常值得进行的。

希望本文能够对大家的 RocksDB 使用有所启发，如有疏漏和不足之处，欢迎指出：）

参考阅读

[1] RocksDB 官方网站. https://rocksdb.org/

[2] Redis as State Backend. https://issues.apache.org/jira/browse/FLINK-3035

[3] RocksDB 基本概念. https://github.com/facebook/rocksdb/wiki/RocksDB-Basics

[4] Flink 配置参数列表. https://ci.apache.org/projects/flink/flink-docs-stable/ops/config.html

[5] RockDB Tuning Guide. https://github.com/facebook/rocksdb/wiki/RocksDB-Tuning-Guide

[6] Advanced RocksDB State Backend Options. https://ci.apache.org/projects/flink/flink-docs-stable/ops/config.html#advanced-rocksdb-state-backends-options

[7] RocksDB 笔记. http://alexstocks.github.io/html/rocksdb.html

[8] Ouaknine, K., Agra, O., & Guz, Z. (2017, May). Optimization of Rocksdb for Redis on Flash. In Proceedings of the International Conference on Compute and Data Analysis (pp. 155-161).

[9] Kim, H., Park, J. H., Jung, S. H., & Lee, S. W. Optimizing RocksDB for Better Read Throughput in Blockchain Systems. In 2019 23rd International Computer Science and Engineering Conference (ICSEC) (pp. 305-309). IEEE.