Titan 简介与实战

在读过分布式存储层 TiKV 的工作原理的相关介绍后，大家对 TiKV 如何将数据可靠的存储在大量服务器组成的集群中有所了解。我们还了解到 RocksDB 是运行在每一个 TiKV 实例存储数据的关键组件。RocksDB 作为一款非常优秀的单机存储引擎，在诸多方面都有着不俗的表现。但其 LSM-tree 的实现原理决定了，RocksDB 存在着数十倍的写入放大效应。在写入量大的应用场景中，这种放大效可能会触发 IO 带宽和 CPU 计算能力的瓶颈影响在线写入性能。除了写入性能之外，写入放大还会放大 SSD 盘的写入磨损，影响 SSD 盘的寿命。

Titan 是 PingCAP 研发的基于 RocksDB 的高性能单机 key-value 存储引擎，通过把大 value 同 key 分离存储的方式减少写入 LSM-tree 中的数据量级。在 value 较大的场景下显著缓解写入放大效应，降低 RocksDB 后台 compaction 对 IO 带宽和 CPU 的占用，同时提高 SSD 寿命的效果。

8.1 Titan 原理介绍

Titan 存储引擎的主要设计灵感来源于 USENIX FAST 2016 上发表的一篇论文 WiscKey。WiscKey 提出了一种高度基于 SSD 优化的设计，利用 SSD 高效的随机读写性能，通过将 value 分离出 LSM-tree 的方法来达到降低写放大的目的。

在存在大 value 的典型应用场景中，Titan 在写、更新和点读等场景下性能都优于 RocksDB。同原生 RocksDB 相比，Titan 通过一定程度上牺牲硬盘空间和范围查询的性能为代价，来取得更高的写入性能。随着 SSD 单位存储

空间价格的降低，和 SSD 设备随机 IO 能力的提升，Titan 的这种设计取舍在未来会产生更加积极的作用。

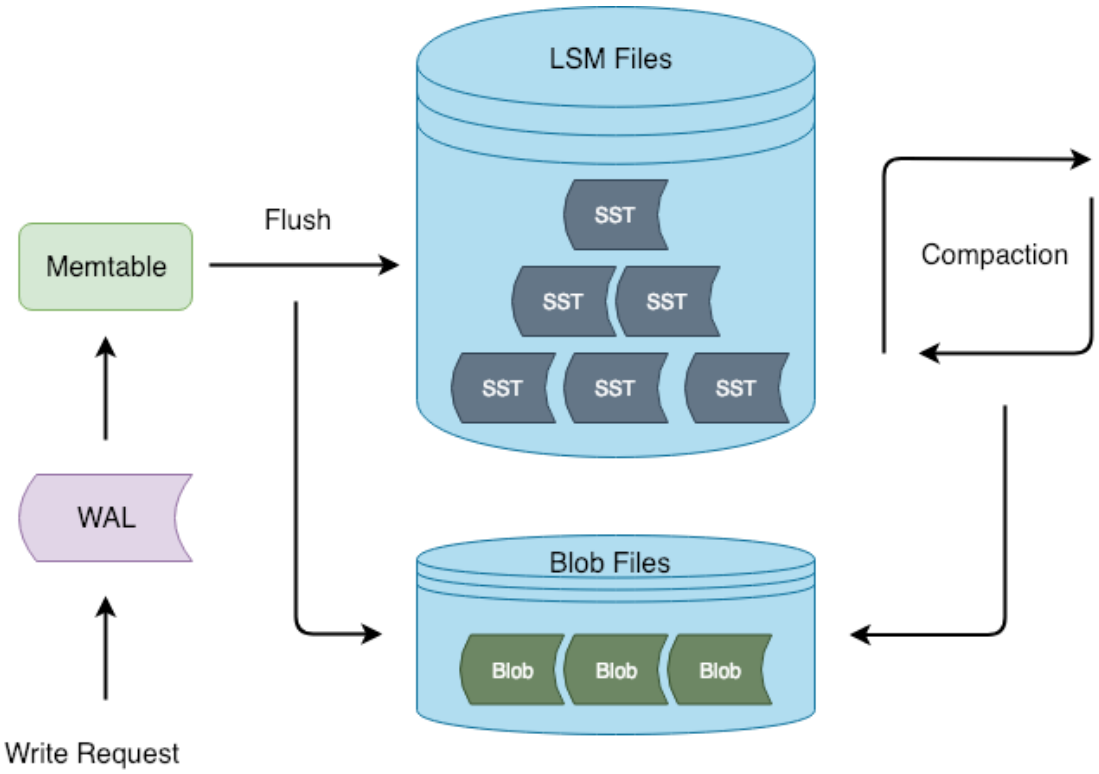
8.1.1 设计目标

作为一个成熟且拥有庞大用户群体的项目，TiKV 在方方面面都需要考虑用户现有系统的平滑过渡。TiKV 使用 RocksDB 作为其底层的存储引擎，因此作为 TiKV 的子项目，Titan 首要的设计目标便是兼容 RocksDB。为用户提供现有基于 RocksDB 的 TiKV 平滑地升级到基于 Titan 的 TiKV。基于这些考虑，我们为 Titan 设定了下面的四个目标：

- 将大 value 从 LSM-tree 中分离出来单独存储，降低 value 部分的写放大。
- 现有 RocksDB 实例无需长期停机转换数据，可在线升级 Titan
- 100% 兼容 TiKV 使用的全部 RocksDB 特性
- 尽量减少对 RocksDB 的侵入性改动，提升 Titan 同未来新版本 RocksDB 的兼容性。

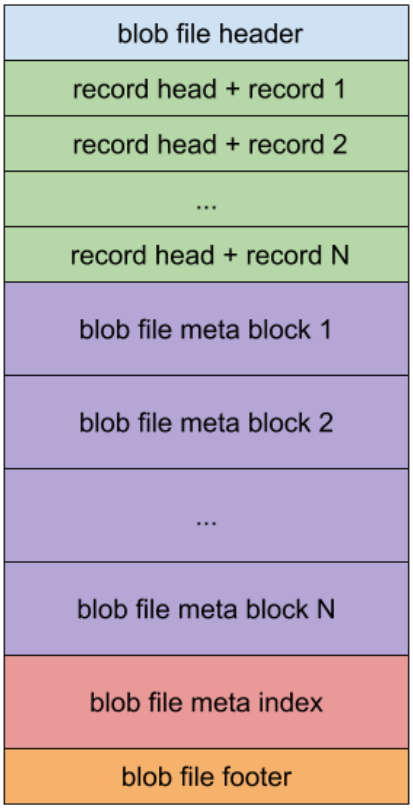
8.1.2 架构与实现

Titan 维持 RocksDB 的写入流程不变，在 Flush 和 Compaction 时刻将大 value 从 LSM-tree 中进行分离并存储到 BlobFile 中。同 RocksDB 相比，Titan 增加了 BlobFile，TitanTableBuilder 和 Garbage Collection（GC）等组件，下面我们将会对这些组件逐一介绍。



8.1.2.1 BlobFile

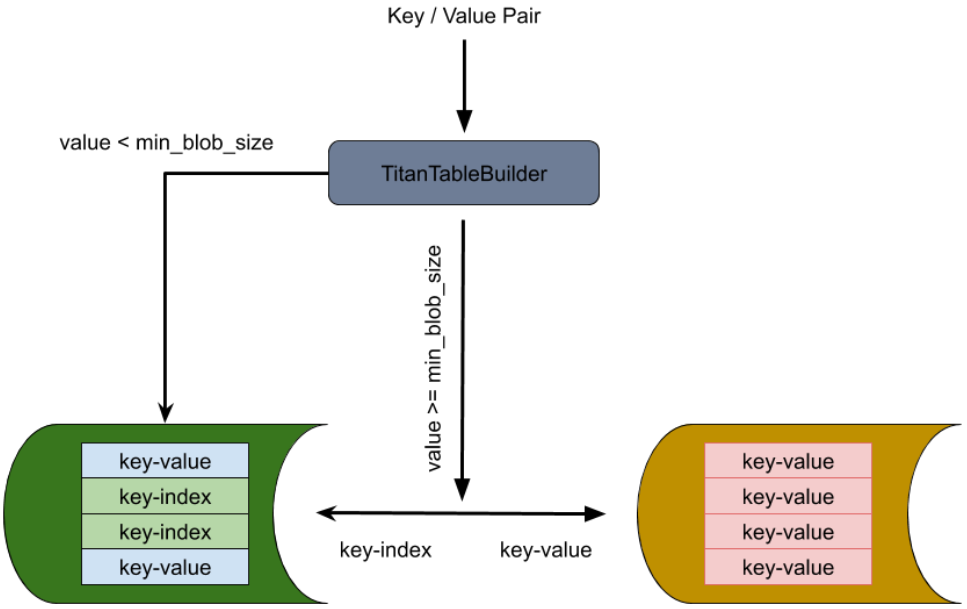
BlobFile 是存放 LSM-tree 中分离得到的 KV 记录的文件，它由 header、record 、meta block、meta index 和 footer 组成。其中每个 record 用于存放一个 key-value 对；meta block 用于在未来保存用户自定义数据；而 meta index 则用于检索 meta block。



为了充分利用 prefetch 机制提高顺序扫描数据时的性能，BlobFile 中的 key-value 是按照 key 的顺序有序存放的。除了从 LSM-tree 中分离出的 value 之外，blob record 中还保存了一份 key 的数据。在这份额外存储的 key 的帮助下，Titan 用较小的写放大收获了 GC 时快速查询 key 最新状态的能力。GC 则会利用 key 的更新信息来确定 value 是否已经过期可以被回收。考虑到 Titan 中存储的 value 大小偏大，将其压缩则可以获得较为显著的空间收益。BlobFile 可以选择 Snappy、LZ4 或 Zstd 在单个记录级别对数据进行压缩，目前 Titan 默认使用的压缩算法是 LZ4。

8.1.2.2 TitanTableBuilder

RocksDB 提供了 TableBuilder 机制供用户自定义的 table 实现。Titan 则利用了这个能力实现了 TitanTableBuilder，在不对 RocksDB 构建 table 流程做侵入型改动的前提下，实现了将大 value 从 LSM-tree 中分离的功能。



在 RocksDB 将数据写入 SST 时，TitanTableBuilder 根据 value 大小决定是否需要将 value 分离到外部 BlobFile 中。如果 value 大小小于 Titan 设定的大 value 阈值，数据会直接写入到 RocksDB 的 SST 中；反之，value 则会持久化到 BlobFile 中，相应的位置检索信息将会替代 value 被写入 RocksDB 的 SST 文件中用于在读取时定位 value 的实际位置。同样利用 RocksDB 的 TableBuilder 机制，我们可以在 RocksDB 做 Compaction 的时候将分离到 BlobFile 中的 value 重新写入到 SST 文件中完成从 Titan 到 RocksDB 的降级。

*熟悉另一个 KV 分离存储的 LSM-tree 实现 Badger 的读者可能想问为什么 Titan 没有选择选择将直接用 VLog 的方式保存在 WAL 中，从而避免一次

额外的写入放大开销。假设我们将 LSM-tree 的 max level 和放大因子分别设定为 5 和 10，则 LSM-tree 的总写入放大大概为 1 + 1 + 10 + 10 + 10 + 10 = 42。其中由 BlobFile 引入的写入放大同 LSM-tree 的整体写入放大相比仅为 1 : 42，可以忽略不计。并且维持 WAL 也可以避免对 RocksDB 的侵入性改动，这也是 Titan 的重要设计目标之一。

8.1.2.3 Garbage Collection

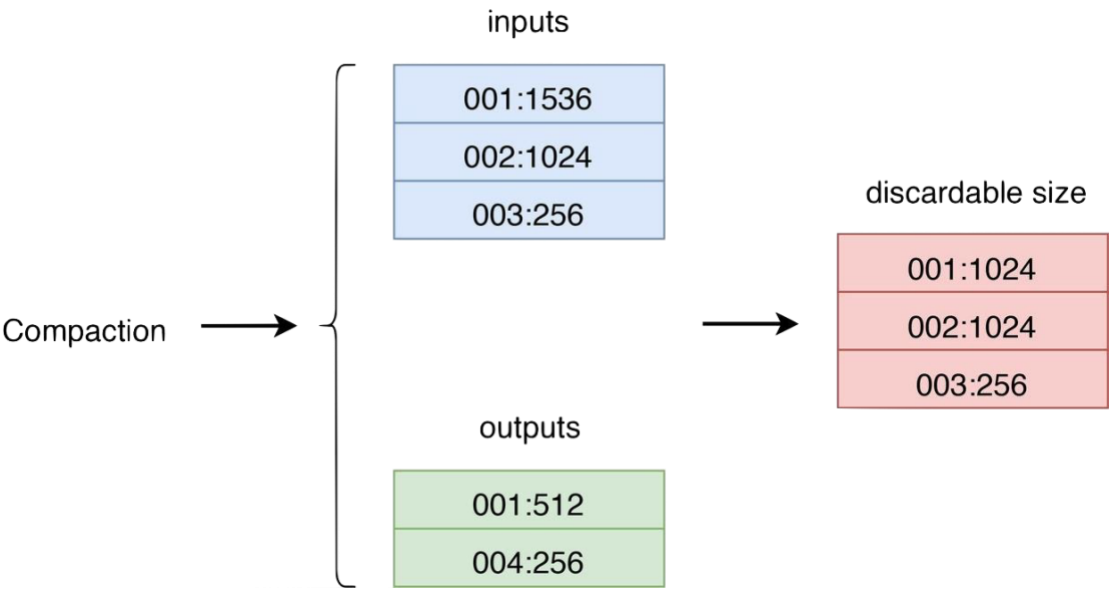
RocksDB 在 LSM-tree Compaction 时对已删除数据进行空间回收。同样 Titan 也具备 Garbage Collection (GC) 组件用于已删除数据的空间回收。在 Titan 中存在两种不同的 GC 方式分别应对不同的适用场景。下面我们将分别介绍「传统 GC」和「Level-Merge GC」的工作原理。

1. 传统 GC

通过定期整合重写删除率满足设定阈值条件的 Blob 文件，我们可以回收已删除数据所占用的存储空间。这种 GC 的原理是非常直观容易理解的，我们只需要考虑何时进行 GC 以及挑选哪些文件进行 GC 这两个问题。在 GC 目标文件被选择好后我们只需对相应的文件进行一次重写只保留有效数据即可完成存储空间的回收工作。

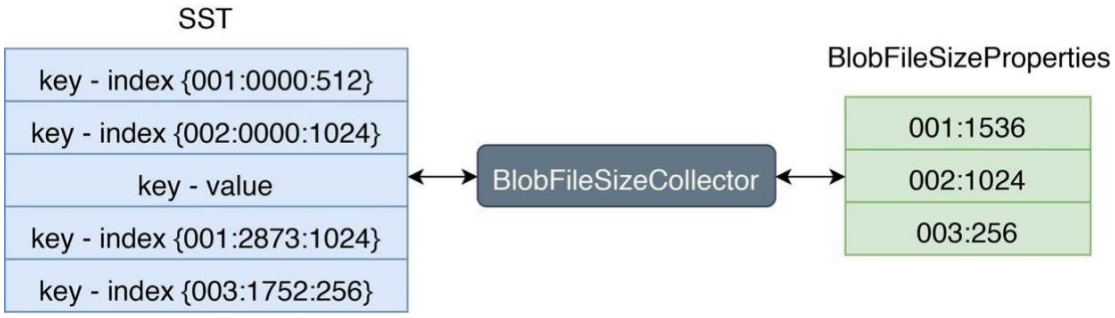
首先 Titan 需要决定何时开始进行 GC 操作，显然选择同 RocksDB 一样在 compaction 时丢弃旧版本数据回收空间是最适当的。每当 RocksDB 完成一轮 compaction 并删除了部分过期数据后，在 BlobFile 中所对应的 value 数据也就随之失效。因此 Titan 选择监听 RocksDB 的 compaction 事件来触发 GC 检查，通过搜集比对 compaction 中输出和产出 SST 文件对应的

BlobFile 的统计信息（BlobFileSizeProperties）来跟踪对应 BlobFile 的可回收空间大小。



图中 inputs 代表所有参与本轮 compaction 的 SST 文件计算的得到的 BlobFileSizeProperties 统计信息（BlobFile ID：有效数据大小），outputs 则代表新生成的 SST 文件对应的统计信息。通过计算这两组统计信息的变化，我们可以得出每个 BlobFile 可被丢弃的数据大小，图中 discardable size 的第一列是文件 ID 第二列则是对应文件可被回收数据的大小。

接下来我们来关注 BlobFileSizeProperties 统计信息是如何计算得到的。我们知道原生 RocksDB 提供了 TablePropertiesCollector 机制来计算每一个 SST 文件的属性数据。Titan 通过这个扩展功能自定义了 BlobFileSizeCollector 用于计算 SST 中被保存在 BlobFile 中数据的统计信息。



BlobFileSizeCollector 的工作原理非常直观，通过解析 SST 中 KV 分离类型数据的索引信息，它可以得到当前 SST 文件引用了多少个 BlobFile 中的数据以及这些数据的实际大小。

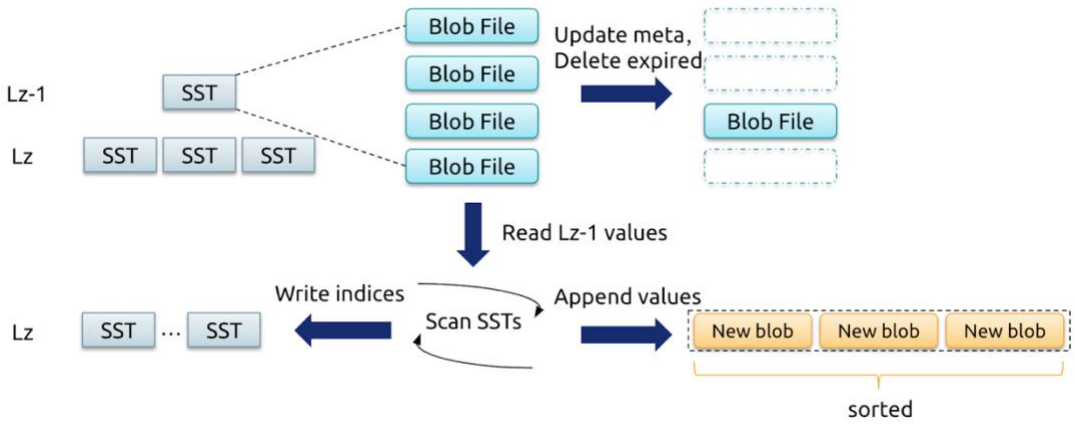
为了更加有效的计算 GC 候选目标 BlobFile，Titan 在内存中为每一个 BlobFile 维护了一份 discardable size 的统计信息。在 RocksDB 每次 compaction 完成后，可以将每一个 BlobFile 文件新增的可回收数据大小累加到内存中对应的 discardable size 上。在重启后，这份统计信息可以从 BlobFile 数据和 SST 文件属性数据重新计算出来。考虑到我们可以容忍一定程度的空间放大（数据暂不回收）来缓解写入放大效应，只有当 BlobFile 可丢弃的数据占全部数据的比例超过一定阈值后才会这个 BlobFile 进行 GC。在 GC 开始时我们只需要从满足删除比例阈值的候选 BlobFile 中选择出 discardable size 最大的若干个进行 GC。

对于筛选出待 GC 的文件集合，Titan 会依次遍历 BlobFile 中每一个 record，使用 record key 到 LSM-tree 中查询 blob index 是否存在或发生过更新。丢弃所有已删除或存在新版本的 blob record，剩余的有效数据则会被重写到新的 BlobFile 中。同时这些存储在新 BlobFile 中数据的 blob index 也会同时被回写到 LSM-tree 中供后续读取。需要注意的是，为了避免删除影响当前活跃的 snapshot 读取，在 GC 完成后旧文件并不能立刻删除。

Titan 需要确保没有 snapshot 会访问到旧文件后才可以安全的删除对应的 BlobFile。而这个保障可以通过记录 GC 完成后 RocksDB 最新的 sequence number，并等待所有活跃 snapshot 的 sequence number 超过记录值而达成。

1. Level-Merge

传统 GC 的实现直观可靠，但大家可能也发现了在传统 GC 过程中还需要伴随着对 LSM-tree 的大量 record key 查询以及 record index 更新操作。GC 时对 LSM-tree 的大量操作不可避免的会对在线的读取和写入产生压力和负面的影响。针对这一问题 Titan 在近期加入了全新的 Level-Merge 策略，它的核心思想是在 LSM-tree compaction 时将 SST 文件相对应的 BlobFile 进行归并重写并生成新的 BlobFile。 重写的过程不但避免了 GC 引起的额外 LSM-tree 读写开销，而且通过不断的合并重写过程降低了 BlobFile 之间相互重叠的的比例，使得提升了数据物理存储有序性进而提高 Scan 的性能。



Level-Merge 在 RocksDB 对 level z-1 和 level z 的 SST 进行 compaction 时，对所有 KV 对进行一次有序读写，这时就可以对这些 SST 中所使用的 BlobFile 的 value 有序写到新的 BlobFile 中，并在生成新的 SST 时直接保存对应记录的新 blob index。由于 compaction 中被删除的 key 不会被写入到新 BlobFile 中，在整个重新操作完成的同时也就相当于完成了相应 BlobFile 的 GC 操作。考虑到 LSM-tree 中 99% 的数据都落在最后两层，为了避免分层 Level-Merge 时带来的写放大问题，Titan 仅对 LSM-tree 中最后两层数据对应的 BlobFile 进行 Level-Merge。