TiDB 底层存储结构 LSM 树原理介绍

京东云开发者 2023-01-11 15,922 阅读9分钟

智能总结 复制 重新生成

文章介绍了分布式关系型数据库 TiDB 底层存储结构 LSM 树的原理，包括其提出背景、算法思路、组成部分、Compact 策略、插入修改删除和查找方法，还与 B+树做了比较，总结了 LSM 树特点及适用场景，如写操作吞吐量高、读操作吞吐量较高的场景。

关联问题: LSM树怎样选策略 LSM树有何优势 B+树怎与它不同

基于该文章内容继续向AI提问

作者：京东物流 刘家存

随着数据量的增大，传统关系型数据库越来越不能满足对于海量数据存储的需求。对于分布式关系型数据库，我们了解其底层存储结构是非常重要的。本文将介绍下分布式关系型数据库 TiDB 所采用的底层存储结构 LSM 树的原理。

1 LSM 树介绍

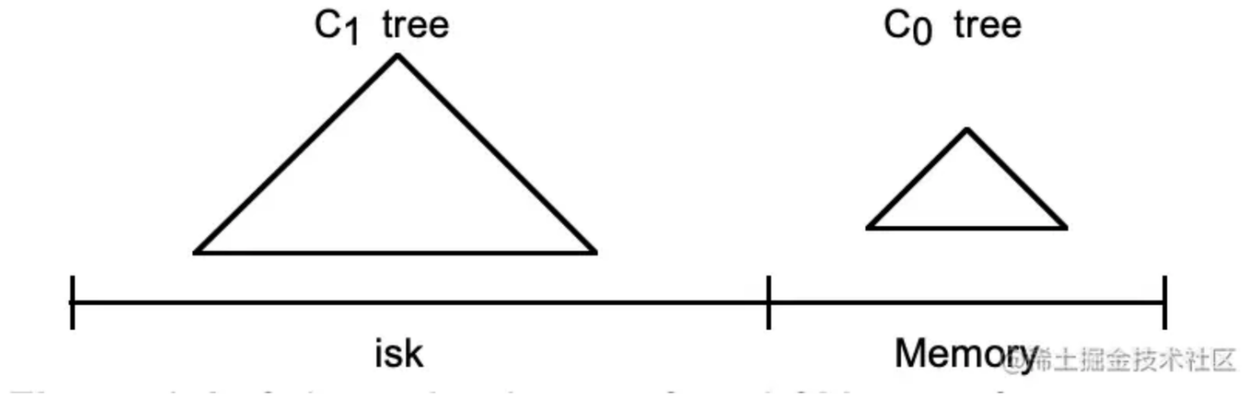
LSM 树（Log-Structured-Merge-Tree) 日志结构合并树由 Patrick O'Neil 等人在论文《The Log-Structured Merge Tree》(www.cs.umb.edu/~poneil/lsm...

LSM 树的核心特点是利用顺序写来提高写性能，代价就是会稍微降低读性能（读放大），写入量增大（写放大）和占用空间增大（空间放大）。

LSM 树主要被用于 NoSql 数据库中，如 HBase、RocksDB、LevelDB 等，知名的分布式关系型数据库 TiDB 的 kv 存储引擎 TiKV 底层存储就是用的上面所说的 RocksDB，也就是用的 LSM 树。

2 LSM 树算法大概思路

LSM 树由两个或多个树状的结构组成。
这一节我们以两个树状的结构构成的简单的双层 LSM 树举例，来简单说下 LSM 树大概思路，让大家对 LSM 树实现有个整体的认识。



原论文中的图

2.1 数据结构

京东云开发者 LV.7
技术运营 @京东科技信息技术...
作者榜No.1 优秀作者
1.8k 文章 3.0m 阅读 19k 粉丝
关注 私信

目录 收起

双层 LSM 树有一个较小的层，该层完全驻留在内存中，作为 C0 树（或 C0 层），以及驻留在磁盘上的较大层，称为 C1 树。
尽管 C1 层驻留在磁盘上，但 C1 中经常引用的节点将保留在内存缓冲区中，因此C1经常引用的节点也可以被视为内存驻留节点。

2.2 写入

写入时，首先将记录行写入顺序日志文件 WAL 中，然后再将此记录行的索引项插入到内存驻留的 C0 树中，然后通过异步任务及时迁移到磁盘上的 C1 树中。

2.3 读取

任何搜索索引项将首先在 C0 中查找，在 C0 中未找到，然后再在 C1 中查找。
如果存在崩溃恢复，还需要读取恢复崩溃前未从磁盘中取出的索引项。

2.4 Compact 过程

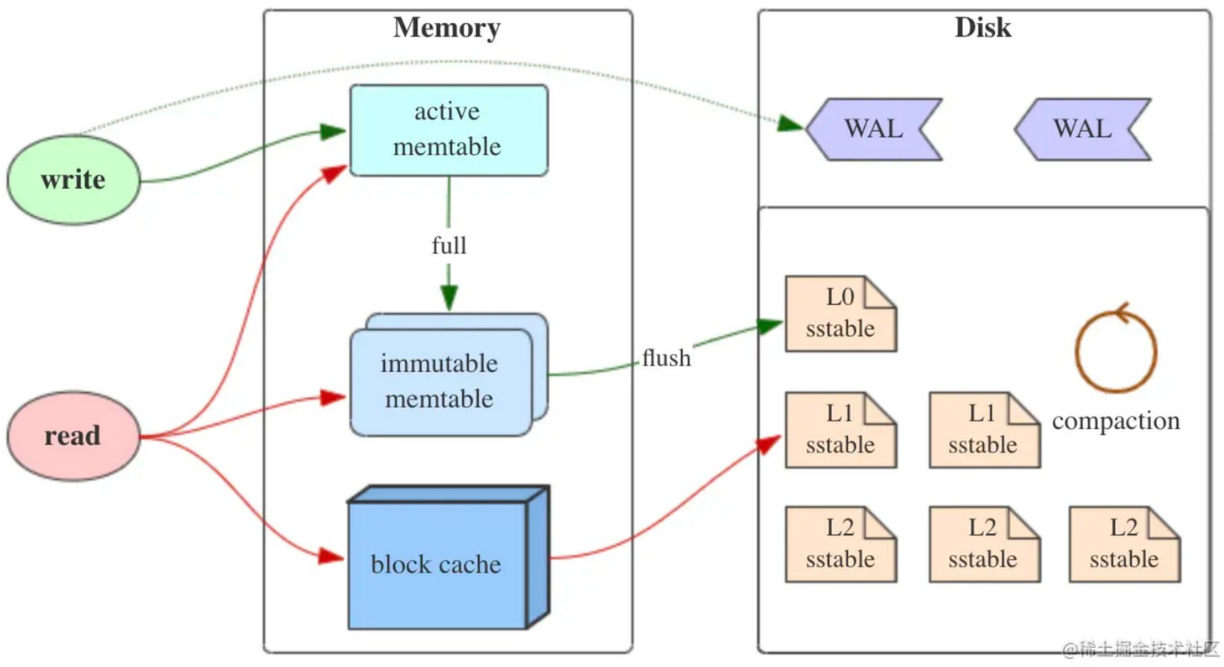
将索引条目插入驻留在内存中的 C0 树的操作没有 I/O 成本，然而，与磁盘相比，容纳 C0 组件的内存容量成本较高，这对其大小施加了限制。达到一定大小后，我们就需要将数据迁移到下一层。
我们需要一种有效的方法将记录项迁移到驻留在成本较低的磁盘介质上的 C1 树中。为了实现这一点，当插入达到或接近每一层分配的最大值的阈值大小，将进行一个滚动合并（Compact）过程，用于从 C0 树中删除一些连续的记录项，并将其合并到 C1 中。
Compact 目前有两种策略，size-tiered 策略，leveled策略，我们将在下面的内容里详细介绍这两种策略。

2.5 崩溃恢复

在 C0 树中的项迁移到驻留在磁盘上的C1树之前，存在一定的延迟（延迟），为了保证机器崩溃后C0树中的数据不丢失，在生成每个新的历史记录行时，首先将用于恢复此插入的日志记录写入以常规方式创建的顺序日志文件 WAL 中，然后再写入 C0 中。

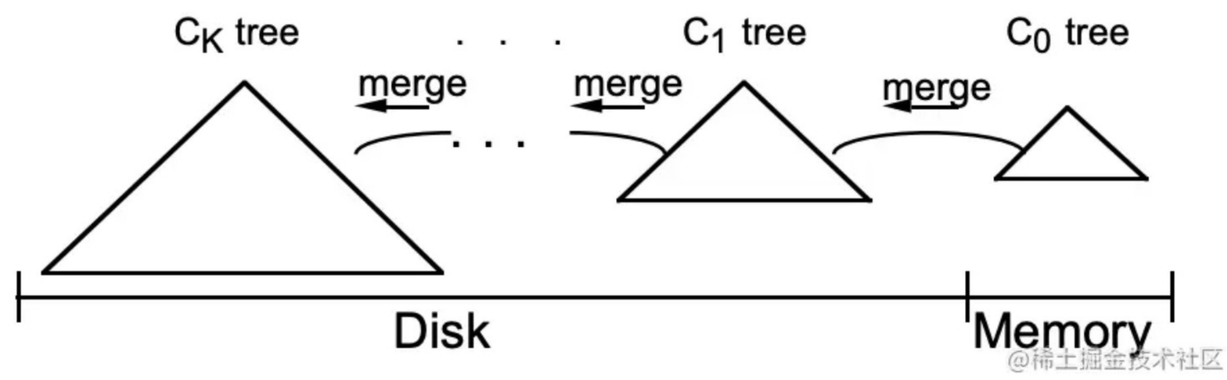
3 LSM 树的组成

LSM树有三个重要组成部分，MemTable，Immutable MemTable，SSTable(Sorted String Table)，如下图。



这张经典图片来自 Flink PMC 的 Stefan Richter 在Flink Forward 2018演讲的PPT

这几个组成部分分别对应 LSM 树的不同层次，不同层级间数据转移见下图。这节就是介绍 LSM 树抽象的不同层的树状数据结构的某个具体实现方式。



3.1 MemTable

MemTable 是在内存中的数据结构，用于保存最近更新的数据，会按照 Key 有序地组织这些数据。LSM 树对于具体如何组织有序地组织数据并没有明确的数据结构定义，例如你可以任意选择红黑树、跳表等数据结构来保证内存中 key 的有序。

3.2 Immutable MemTable

为了使内存数据持久化到磁盘时不阻塞数据的更新操作，在 MemTable 变为 SSTable 中间加了一个 Immutable MemTable。
当 MemTable 达到一定大小后，会转化成 Immutable MemTable，并加入到 Immutable MemTable 队列尾部，然后会有任务从 Immutable MemTable 队列头部取出 Immutable MemTable 并持久化磁盘里。

3.3 SSTable(Sorted String Table)

有序键值对集合，是 LSM 树组在磁盘中的数据结构。
其文件结构基本思路就是先划分为数据块(类似于 mysql 中的页)，然后再为数据块建立索引，索引项放在文件末尾，并用布隆过滤器优化查找。

4 LSM 树的 Compact 策略

当某层数据量大小达到我们预设的阈值后，我们就会通过 Compact 策略将其转化到下一层。

在介绍 Compact 策略前，我们先想想如果让我们自己设计 Compact 策略，对于以下几个问题，我们该如何选择。

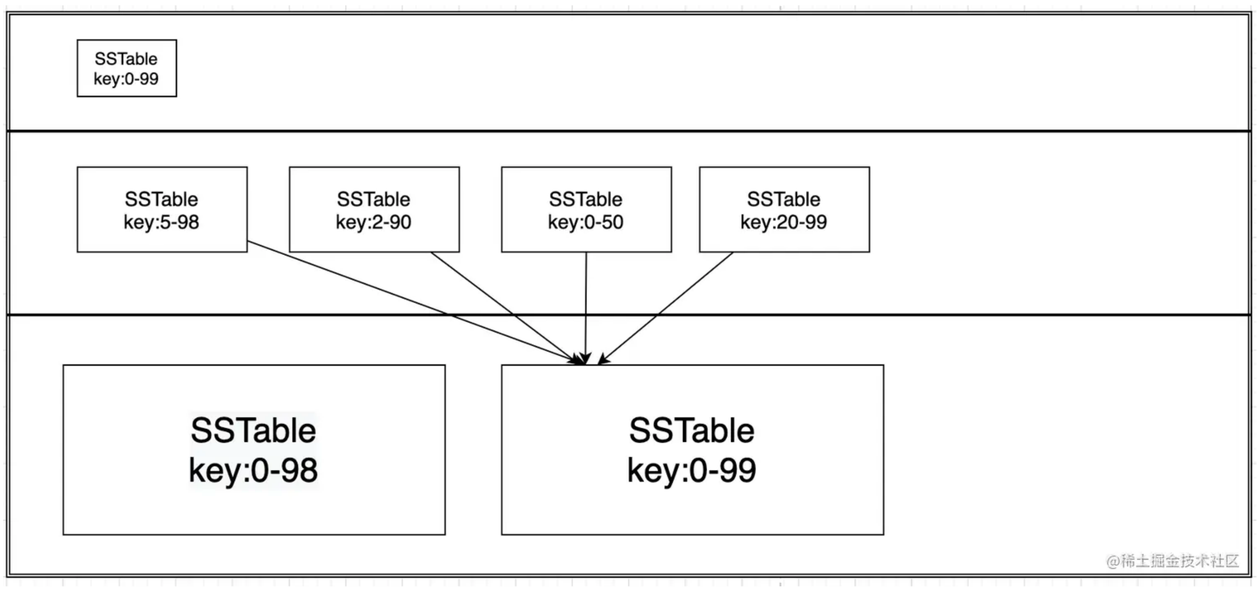
1. 对于某一层的树，我们用单个文件还是多个文件进行实现?
2. 如果是多个文件，那同一层 SSTable 的 key 范围是有序还是重合？有序方便读，重合方便写。
3. 每层 SSTable 的大小以及不同层之间文件大小是否相等。
4. 每层 SSTable 的数量。如果同一层 key 范围是重合的，则数量越多，读的效率越低。

不同的选择会造成不同的读写策略，基于以上 3 个问题，又带来了 3 个概念：

1. 读放大：读取数据时实际读取的数据量大于真正的数据量。例如在 LSM 树中可能需要在所有层次的树中查看当前 key 是否存在。
2. 写放大：写入数据时实际写入的数据量大于真正的数据量。例如在 LSM 树中写入时可能触发Compact 操作，导致实际写入的数据量远大于数据的大小。
3. 空间放大：数据实际占用的磁盘空间比数据的真正大小更多。LSM 树中同一 key 在不同层次里或者同一层次的不同 SSTable 里可能会重复。

不同的策略实际就是围绕这三个概念之间做出权衡和取舍，我们主要介绍两种基本策略：size-tiered 策略和 leveled 策略，这两个策略对于以上 3 个概念做了不同的取舍。

4.1 size-tiered 策略



4.1.1 算法

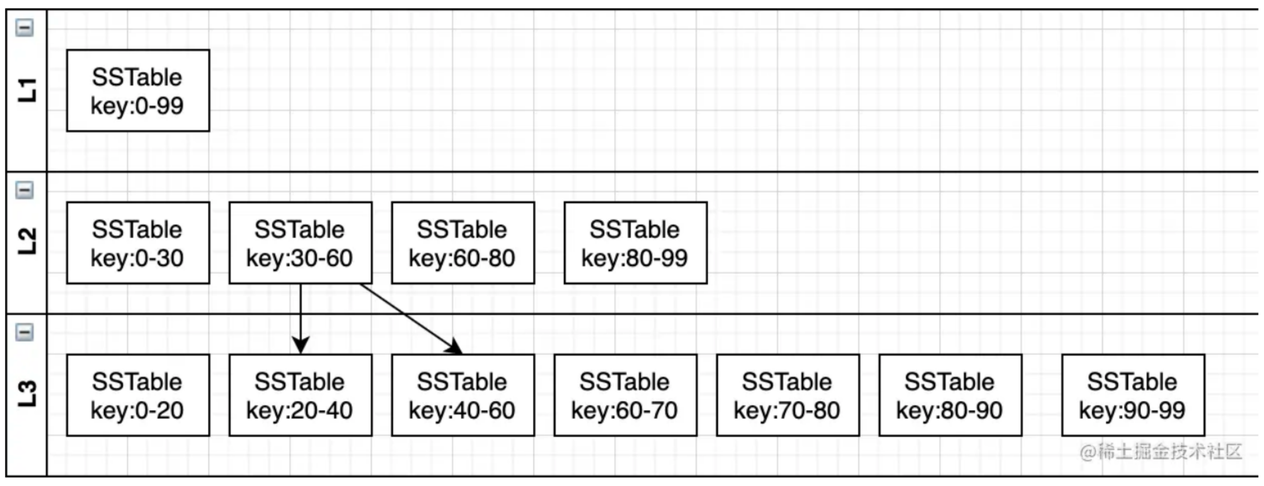
1. size-tiered 策略每层 SSTable 的大小相近。
2. 当每一层 SSTable 的数量达到 N 后，则触发 Compact 操作合并这些 SSTable，并将合并后的结果写入到一个更大的 SStable。
3. 新的更大的 SStable 将直接放到下一层 SStable 的队尾。所以同一层不同 SStable key 范围重合，查找时要从后向前扫描，且最坏情况下可能会扫描同一层所有 SStable ，这增大了读放大的问题(之所以说增大，是因为 LSM 树不同层之间也有读放大问题)。

4.1.2 总结

由此可以看出 size-tiered 策略几个特点：

1. 每层 SSTable 的数量相近。
2. 当层数达到一定数量时，最底层的单个 SSTable 的大小会变得非常大。
3. 不但不同层之间，哪怕同一层不同 SSTable 之间，key 也可能会出现重复。空间放大比较严重。只有当该层的 SSTable 执行 compact 操作才会消除这些 key 的冗余记录。
4. 读操作时，需要同时读取同一层所有 SSTable ，读放大严重。

4.2 leveled 策略



4.2.1 算法

1. leveled 策略和 size-tiered 策略不同的是，它限制 SSTable 文件的大小，每一层不同 SSTable 文件 key 范围不重叠且后面的最小 key 大于前一个文件的最大 key
2. 当每一层 SSTable 的总大小达到阈值 N 后，则触发 Compact 操作。
3. 首先会随机选择一个 SSTable 合并到下层，由于下一层 key 是全局有序的，这就要求 leveled 策略 Compact 操作时需要当前 SSTable 和下一层里和当前 SSTable key 存在范围重叠的所有 SSTable 进行合并。最坏情况下可能下一层所有 SSTable 都参与合并，这就增大了写放大问题(之所以说增大，是因为 LSM 树不同层之间 Compact 也有写放大问题)。

4.2.2 总结

由此可以看出 leveled 策略几个特点：

1. 不会出现非常大的 SSTable 文件。
2. 每一层不同 SSTable 文件 key 范围不重叠。相对于 size-tiered 策略读放大更小。
3. Compact 操作时，需要同时和下一层 SSTable 一起合并，写放大严重。

5 LSM 树的插入、修改、删除

从 LSM 树的名字，Log-Structured-Merge-Tree 日志结构合并树中我们大概就能知道 LSM 树的插入、修改、删除的方法了——顺序追加而非修改(对磁盘操作而言)。

1. LSM 树的插入、修改、删除都是在 L0 层的树里插入、修改、删除一条记录，并记录记录项的时间戳，由于只需要取最新的内容即可，所以不需要操作后面层次的树。
2. 历史的插入、修改、删除的记录会在每次 Compact 操作时被后面的记录覆盖。

6 LSM 树的查找

1. 由于后面的操作会覆盖前面的操作，所以查找只需从 L0 层往下查，直到查到某个 key 的记录就可以了，之前的记录不需要再查了。
2. 对于 size-tiered 策略，同一层 SSTable 需要从后向前遍历，直到找到符合的索引项。
3. 在查找过程中也会使用其他一些手段进行优化，例如增加缓存、布隆过滤器等。

7 LSM 树和 B+ 树的比较

1. 不考虑写日志等操作，插入、修改、删除一条记录 B+ 树需要先找到数据位置，可能需要多次磁盘 IO；LSM 树不需要磁盘 IO，单次插入耗时短，所以其写入的最大吞吐量是高于 B+ 树的。
2. LSM 树后面的 Compact 操作也会操作这条数据几次，总的写入量是大于 B+ 树的，但可以通过将 Compact 操作放到业务低峰时来降低这个劣势的影响。
3. 查找时， LSM 树需要遍历所有层次的树，查找效率上要低于 B+ 树，但 LSM 树写入时节省的磁盘资源占用，可以一定程度上弥补读效率上的差距。

8 总结

LSM 树特点：顺序写入、Compact 操作、读、写和空间放大。
LSM 树适用场景：对于写操作吞吐量要求很高、读操作吞吐量要就较高的场景，目前主要在 NoSql 数据库中用的比较多。

标签： 数据库 大数据 TiDB

评论 0

登录 / 注册 即可发布评论!

暂无评论数据

为你推荐

Sorry！Hbase的LSM Tree就是可以为所欲为！
王知无 | 4年前 | 1.2k | 2 | 2 | 大数据

浅谈存储系统：LSM 树设计原理
labuladong | 2年前 | 1.7k | 4 | 评论 | 面试

分享：数据库存储与索引技术（二） 分布式数据库基石——LSM树
OceanBase数据库 | 1年前 | 1.5k | 5 | 评论 | 数据库

从零实现LevelDB 1. 日志结构合并树和LevelDB介绍
李沐阳_ | 1月前 | 65 | 点赞 | 评论 | 数据库

LSM树揭秘：NoSQL存储系统的核心
写bug写bug | 6月前 | 117 | 3 | 评论 | 后端 | LevelDB | HBase

浅谈数据库存储结构
Ddupg | 1年前 | 243 | 点赞 | 评论 | 数据库

TDSQL | DB·洞见回顾|基于LSM-Tree存储的数据库性能改进
腾讯云数据库 | 3年前 | 286 | 2 | 评论

OB有问必答 | LSM Tree的技术原理是什么？OceanBase的存储引擎为什么基于LSM Tree?
OceanBase数据库 | 4年前 | 1.3k | 2 | 评论 | 数据库

数据库 - MySQL、PG、MongoDB 索引
寒沧 | 3年前 | 1.5k | 1 | 评论 | MySQL

[译]理解 LSM 树：一种适用于频繁写入的数据库的结构
双峰插云 | 4年前 | 1.3k | 5 | 评论 | 掘金翻译计划

磁盘IO系列（四）：B树和RUM猜想
数据智能老司机 | 5月前 | 1.1k | 3 | 评论 | 数据结构 | 数据库 | 架构

一些有趣的B+树优化实验
腾讯云数据库 | 2年前 | 1.2k | 3 | 评论

MySQL存储引擎及索引简介
京东云开发者 | 1月前 | 80 | 点赞 | 评论 | 数据库

架构师日记-从数据库发展历程到数据结构设计探析 | 京东云技术团队
京东云开发者 | 1年前 | 885 | 3 | 评论 | 数据库 | 数据结构 | 架构

一文让你对mysql索引底层实现明明白白
京东云开发者 | 7天前 | 321 | 6 | 评论 | 数据库