2018
7
13
03:49 PM
433次查看

key / value 数据库的选型

分类：数据库　标签：无

一直以来在我的观念中，key/value 数据库就三种选项：

- 内存可存放：Redis
- 单机磁盘可存放：RocksDB
- 超过 TB 级：Cassandra、HBase……

然而在实际项目中使用 RocksDB 时，才发现了一堆问题，折腾许久才搞定。

先介绍下我使用 RocksDB 的背景。
这个项目有很多 key/value 数据（约 100 GB）需要使用，使用时基本是只读的，偶尔更新时才会批量导入，且可以忍受短暂的停机导入。我一想 TiKV 和 Pika 等很多 key/value 数据库都选用了 RocksDB，应该是比较靠谱的，于是就选它了。

接着就发现这东西的编译依赖有点多。我的项目是用 Go 写的，而这个玩意需要安装一堆 C 库，并且不能交叉编译到其他平台。不但不能在 Mac OS X 上编译 Linux 的版本，甚至 Ubuntu 16.04 上编译的都不能在 CentOS 7 上运行（后者的 GCC 版本较低，动态链接库版本也低）。因为懒得降级 GCC，最后我选择用 docker 来编译了。

而且我发现数据量小时还挺快，但是数据量大了就越来越慢。平时插入 10 万条数据（约 50 MB）大概 0.2 ~ 0.5 秒，但一段时间后就会持续遇到需要几秒甚至几十秒的情况。
于是我又开始寻找其他的替代品，诸如 LMDB、BadgerDB 和 TerarkDB 等。在这个过程中我开始了解到它们实现的原理，也算有不少收获。

传统的关系型数据库大多是使用 B+ 树，这种数据结构可以很快地进行顺序读写，也能以 O(log(N)) 的时间复杂度来进行随机读，但不适合随机写（会导致 B+ 树重新调整平衡，造成写放大）。
而 LevelDB 引入了 LSM 树，就是为了解决 B+ 树随机写性能低的问题，它把随机写以跳跃表的形式保留在内存中（memtable），积累到足够的大小就不再改写它了，并将其写入到磁盘（L0 SST file），这样就只有顺序写了。因为 memtable 和 L0 中的数据可能会重复，而且 key 很分散，所以搜索时需要遍历它们。如果没找到的话，还要向下层查找（关于层数下文会解释），不过 L1 之后的 SST file 都是有序分段的，因此可以用二分查找来找到 key 所在的数据文件，再在这个文件中用二分查找来找到这个 key。为了降低搜索的代价，RocksDB 还使用了 Bloom filter 来判断数据是否在某个文件中（有误判，但能显著减少需要搜索的文件数）。由此可见，LSM 树对写入做了优化，但降低了随机读的性能，顺序读则和 B+ 树差不多。
此外，L0 的数据可能会有很多过期数据（被更新或删除过），因此需要在达到阈值后进行合并（compact），去掉这些重复和无效的数据，并写入 L1。而 L1 也可能会有过期的数据，也需要被合并写入 L2……这就相当于数据要多次写入不同的文件中，也就造成了写放大。而合并不重叠的数据文件是很快的，因此顺序写还是要比随机写快，但合并可以在其他线程中执行，在不会持续随机写入大量数据的情况下，基本能保持 O(1) 的写入。

事实上，我遇到的 RocksDB 变慢的问题就是 compact 引起的，默认配置下它只使用一个线程来 compact，如果 compact 跟不上写入的速度，RocksDB 就会降低写入速度，甚至停止写入。考虑到我的电脑有 4 个核，于是我把线程数改成了 4：

```
bbto := gorocksdb.NewDefaultBlockBasedTableOptions()
opts := gorocksdb.NewDefaultOptions()
opts.SetBlockBasedTableFactory(bbto)
opts.SetCreateIfMissing(true)
opts.OptimizeLevelStyleCompaction(1 << 30)
opts.IncreaseParallelism(4)
opts.SetMaxBackgroundCompactions(4)
env := gorocksdb.NewDefaultEnv()
env.SetBackgroundThreads(4)
opts.SetEnv(env)
db, err := gorocksdb.OpenDb(opts, "db")
```

修改后就发现插入时间基本稳定在 0.2 ~ 0.5 秒之间了，CPU 占用率超过了 400%，磁盘写入速度超过了 800 MB/s……
另外，RocksDB 还提供了 `db.GetProperty("rocksdb.stats")` 这个方法来查看状态，需要关注的数据主要有 W-Amp（写入放大倍数）和 Stalls（因为 compact 而降速）。

RocksDB 有 3 种 compact 的方式：leveled、universal 和 FIFO。
Leveled 是从 LevelDB 继承过来的传统形式，也就是当一层的数据文件超过该层的阈值时，就往它的下层来 compact。L0 之间因为可能有重复的数据，因此需要全合并后写入 L1。而 L1 之后的数据文件不会有重复的 key，因此在 key 范围不重合的情况下，可以并发地向下合并。RocksDB 默认有 L0 ~ L6 这 7 层，L1 容量是 256 MB（建议把 L0 和 L1 大小设为一样，可以减小写入放大），之后每层都是上一层容量的 10 倍。很显然，层数越高就表示写入放大倍数越高。
那么可不可以不分这么多层，以减小写入放大倍数呢？Universal 这种风格就是尽量只用 L0，并将新的 SST 不断合并到老的 SST，因此数据文件的大小是不等的。但单个 SST 也是有上限的，不然内存扛不住，二分查找也会变慢，于是达到上限时，就往 L6 写，而 L0 以外的层不会有重叠的 key 范围，所以合并时只需要简单地拼接就行了。如果 L6 也不够用，就继续往 L5、L4 等层写入。这种策略增大了每层能容纳的大小，并且因为先写 L6，而 L6 是容量最大的，数据量较小时就不需要用到 L5 等其他层了，也就减少了层数，对应着也就降低了写入放大倍数。但是因为 L0 的上限变大了，单个 SST 的上限也变大了，所以读性能可能会稍微下降（部分情况下因为层数和 SST 少，读取速度可能更快）。此外，L0 变大也会影响打开数据库的耗时，因为需要读取到内存中。
FIFO 严格来说不算是合并策略，它的做法是所有的数据都放在 L0，当数据量达到上限时，就把最老的 SST 删掉。它还能搭配 TTL 使用，也就是优先把过期时间较早的数据删掉。这种策略一般只用于缓存，但是对于不超过内存容量的缓存，我更倾向于放 Redis 里。
TiKV 和 Pika 都选择了 leveled 风格，也是 RocksDB 的默认值，应该是适合大部分情况的。但如果需要更高的写入性能，并且总数据容量不大（例如少于 100 GB），可以选择 universal。

其实 RocksDB 还有挺多可以调优的参数，但是都需要做测试，在 SSD 和 HDD 上表现也可能不一样，这里我只列几点：
在我的电脑上（用 SSD），允许 MMAP 读取会稍微拖慢读取速度，允许 MMAP 写入可以稍微加快写入速度。设置合理的 block cache 可以加快读取速度，而填充读缓存则可以加快频繁访问的 key 读取。关闭 WAL 会极大地加快写入速度（时间约减少 1/3），因为需要写入的数据量少了一半，对于不是实时写入的场景（例如批量导入）推荐关闭。

RocksDB 还提供了一个 Column Family 的功能，设计上就和 MySQL 的分表差不多，就是人为地将数据分散到多个 Column Families 中（例如按 key 的首字节或 hash 来分库），使多个 Column Families 可以并发读写。相对于手动分到多个 db 而言，利用 Column Family 可以原子性地操作多个 Column Families 中的数据，并且能保持它们在一个事务中的一致性。

RocksDB 就讲到这里，接下来看看其他的选项。
我最先看中的是 LMDB，因为很多评测都说它比 RocksDB 更快，性能波动更小。它的原理是用 MMAP 将数据文件映射到内存中，也就避免了写入时的系统调用（实际上 RocksDB 将数据合并后一次性的顺序写也没有多少开销），但是一页（4 KB）只能存放 2 条数据，而且不会进行块压缩，所以比较费空间；数据结构选的是 B+ 树，因此读性能是很好的。
直觉上我觉得 B+ 树的随机写入会很慢，实际测试确实如此，并且随着数据量的增大，写入速度基本是指数级下降的，于是果断放弃了。

接着就找到了 BadgerDB，它的原理和 LevelDB 差不多，但是又做了个重要的优化：将 key 和 value 分开存放。因为 key 的空间占用会小很多，所以更容易放入内存中，能加快查询速度。而在合并时，合并 key 的开销很小（只是修改 value 的索引地址），合并 value 也只是删掉老的 value 即可，甚至不需要和 key 的合并同步进行，定期清理下就行了。而且因为 key 单独存放，所以遍历 key 和测试 key 是否存在也会快很多。不过如果 value 长度很小，那么分开存放反而增加了一次随机读，这是要结合实际项目来考虑的。
我测试发现随机读确实挺快，大概是 RocksDB 的 100 倍；但随机写没有太大优势，和 universal 的 RocksDB 差不多，而后者可以关闭 WAL 以大幅提高写入速度。
虽然空间占用比 RocksDB 要高一些（大概 10%），但是打开数据库的速度却要快几倍，也许是只需要加载 key 的原因。
而在内存占用方面，BadgerDB 比 RocksDB 更吃内存些，并且随着数据量的增长，占用的内存也越来越多，如果物理内存不够的话，就会使用 SWAP，并导致写入速度变慢。
在我的 SSD 上测试时发现 LoadToRAM 和 MemoryMap 相对于 FileIO 没有太大的优势，但是内存占用会多很多，所以我将默认的配置各降了一级：

```
opts.TableLoadingMode = options.MemoryMap
opts.ValueLogLoadingMode = options.FileIO
```

对于内存足够的场景而言，BadgerDB 确实是一个很好的 RocksDB 替代品，因为大部分情况下是读多写少的。而它的缺点也很明显，只有 Go 语言的版本，没有其他语言的 binding。

最后一个吸引我眼球的是 TerarkDB。它在 RocksDB 的基础上进行了改进，将所有 key 进行了可检索压缩，这个压缩算法能在不解压的情况下进行搜索，而 value 则进行了可定点访问的压缩，可以直接定位并解压需要的部分。这种实现使得缓存能更高效地使用，也能利用上 SSD 的随机读较快的优点，相当于把很多需要读磁盘的操作在内存中搞定了。另外，全局压缩比 RocksDB 使用的块压缩的压缩率更高，所以需要写入的数据会减少，也会改善写入速度。而在合并时，它选择采用 universal 的风格以减少写入放大。
它最大的缺点就是核心代码没有开源，压缩算法也是专利，需要购买商业版来使用，所以我就不测试了。

综上，对于几十 GB ~ 几百 GB 的 key / value 数据而言，如果只使用 Go 来开发的话，BadgerDB 在很多情况下是很好的选择，否则也只剩 RocksDB 了。