键值存储实验报告

焦景辉 计97 2019013253

功能设计与实现

我们这次需要实现一个基于 SSD 的多线程的键值存储，我们知道键值存储最经典的实现就是基于 LSM Tree 的 LevelDB，同时又能从多个方面进行优化，比如进行键值分离等。考虑到本次作业的工作量的要求，我基本按照 LevelDB 的结构和算法进行了本次的实验。

在存储的层级上，包括存放在内存中的 memtable 和 immtable，其中 memtable 是可写的，而 immtable 是不可写的，在 SSD 存储的是 sstable，每个 sstable 的大小限制大约是 2MB。在 LevelDB 的实现中，memtable 内部的数据结构是跳表，其实我觉得这是个很奇怪的选择，因为从性能上的各个指标来看，跳表都不是最优的选择，在互联网上可以找到的唯一选择跳表的原因是实现比较简单。所以在我的实现中，memtable 内部使用的数据结构就是 `std::map`。由于 `std::map` 并不是线程安全的，我们使用读写锁很显然会限制并行度，因此我进行了 sharding 处理，对于 key 进行 hash，随机分配到 16 个 `std::map` 当中。这样我们的 Memtable 就可以最大支持 16 个线程进行并行的读写。

我对于 sstable 的实现是非常基础的，考虑到实现的复杂性，我并没有采用 LevelDB 当中 sstable 的前缀压缩以及没有引入 Bloom Filter。也就是说我的 sstable 可以分为三个部分，第一部分是按需存放的 KV 数据，然后是每 4K 设置的一个 Block Index，存储当前这个 Block 的第一个 Key，最后一个部分存储每个 Block Key 的 offset。在运行的时候我们会把最后一个部分读区到内存当中，方便我们快速确定数据在那个 Block 当中。参考 LevelDB 的实现，第一层最多包括 4 个 sstable，每个 sstable 的范围是可能相交的，接下来第 i 层每层可以包括 10^i 个 sstable，同一层里都是不相交的，因此我们就能快速的确定应该查询哪一个 sstable。

对于持久化和崩溃一致性的处理，我引入了 WAL 来解决这个问题。每次对 memtable 的写入都会提前写入到一个 log 当中，当 memtable 变为 immtable 后，开辟一个新的 log 文件用于存储新的 memtable 的 log 数据。我们会有一个后台线程不断的将新的 immtable 存储为 sstable，一旦操作完成，就可以清除掉 immtable 对应的 log 文件了。当我们的引擎启动时，如果观察到有 memtable 或者 immtable 对应的 log 文件，就会先将这些操作 replay 一遍，保证数据的持久化。

LevelDB 使用了 MVCC 来实现事务之间的隔离，这里可以达到的是 Snapshot Isolation，显然无法达到本次实验要求的 Linearizable。但是很显然我们这次的实现每次操作最多只有一次写，Snapshot Isolation 在这个限制下基本就可以认为是 Linearizable 了。因此这里我也采用 MVCC 来实现，每次操作都会有一个 LSN，我们在查询时忽略掉所有 LSN 大于当前查询 LSN 的记录即可。

同时，为了比较好的性能表现，我们显然不应该让后台的压缩进程堵塞前台的读写，而如果我们简单的按照上面的实现，就必然要求在压缩时对所有 sstable 的访问上锁，否则就可能导致某个线程正在读某个 sstable 时由于压缩导致这个 sstable 被删除。因此对于整个 SSD 的存储我实现了一个简单的版本控制，使用 `std::shared_ptr` 索引每个 sstable，而每个 sstable 被划分在多个 `Version` 里，每次的压缩操作都会创建一个新的 `Version`。任何前台的操作，包括插入、删除、访问，都会先获取一个 `Version` 的 `shared_ptr`，保证这个 `Version` 下的所有 sstable 不会被删除。

对于附加部分，我实现了 `visit` 和 `snapshot` 这两个功能，并且可以通过所有的附加测试。我的 visit 的实现比较暴力，就是获得所有范围涉及的 sstable 的迭代器，然后进行一个归并排序返回结果。对于 snapshot，由于我们在 `Version` 的实现上本身就一个 Snapshot 的性质，因此对于所有的 snapshot 请求，我会把当前的 immtable 和 memtable 全部变为 sstable 存储到一个 Version 中，然后直接返回这个 Version 作为 snapshot。

如何保证崩溃一致性

通过 WAL 实现。

并发控制与多线程安全

memtable 使用 RWLock 来保证一个 shard 里最多只有一个线程在写；WAL 使用 `O_APPEND` 模式，保证多线程插入的原子性；sstable 只读，无需并发控制。

实现结果与性能表现

我的实现可以通过所有的基本测例和附加测例，也就是我实现了 `Engine::snapshot` 和 `Engine::visit` 这两个接口，对于 `Engine::gc` 我没有实现，直接返回了 `kSucc`。

下面报告我的性能表现，在我的 MacBook Pro 上测试，16GB 内存，CPU 为 M1 Pro，使用 Clang++ 15.0.7 进行编译。

对于单线程，使用默认参数，测试三次的结果分别是：

```
[summary] ops: 482321, ops(thread): 482321, avg_lat: 2073 ns
[summary] ops: 155405, ops(thread): 155405, avg_lat: 6434 ns
[summary] ops: 173340, ops(thread): 173340, avg_lat: 5769 ns
```

对于多线程，使用默认参数，测试三次的结果分别是：

```
[summary] ops: 430014, ops(thread): 53751.8, avg_lat: 2325 ns
[summary] ops: 427784, ops(thread): 53473, avg_lat: 2337 ns
[summary] ops: 168573, ops(thread): 21071.6, avg_lat: 5932 ns
```

可以看到，不论是多线程还是单线程，最好的 ops 都是在 4.5×10^5 左右，最差在 1.5×10^5 左右。这里存在比较大的波动的主要原因是，在运行时如果存在 memtable 与 immtable 都满的情况下，默认并不会阻塞等待后台压缩的完成，而是直接写入 memtable，也就是接受可能的 memtable 运行过程中的过量写入。如果后台压缩进行的较慢，就会使得对 KV 的操作约等于直接在内存中读写，也就相当于机器负载的上限。而如果后台压缩完成的较快，那么就还需要访问 sstable 等结构，就回到了平常的速度。

但是其实这样并不影响我们的正确性，因为还有 WAL 来保证崩溃一致性。这里整个系统受到的限制其实相当于写入 SSD WAL 的速度限制。

问题与限制

我的实现当然还是存在一些问题的，首先是在后台压缩比较慢的情况下，我的系统本质上只做了写 WAL 以及 in-memory 的查询，如果插入过快，那么就会造成 log 过大，从而使得后续的压缩过程变得更加不稳定。也就是说我的实现稳定性比较差，无法保证一个比较均匀的访问时延。

另一个问题就是 MVCC 会导致所有的历史数据都存储在数据库中，我们需要有效的垃圾回收机制来保证数据库的空间不会过量。考虑到我们实际 benchmark 的数据量都比较小，以及实现起来比较复杂，因此我这里就没有实现。同时，由于我们并不支持 batch 化的读写，也就是实际上没有可用的事务的概念，难以作为存储引擎等被实际使用。

还有就是 sstable 的实现可以更加的精细化，可以引入 bloom filter，以及前缀压缩、键值分离等。这里实现起来比较复杂，就被我略过了。

思考题

如何保证和验证存储引擎的 Crash Consistency

我们的 Crash Consistency 主要是通过 WAL 来保证的，因为 memtable 本身整个是 in-memory 的，丢失后需要整个重建，因此恢复只需要简单 Replay WAL 即可。同时操作系统保证了写入 WAL 的原子性，而 sync 只有在 flush 完成后才会返回，因此无论是进程、OS、电源任何一个出现故障，我们的崩溃一致性都可以保证。

SSD 与 HDD 在读写的不同方式上有什么差别

- 使用 `read, write, fsync` 是直接通过操作系统直接操作存储，在吞吐、IO 利用率等指标上主要取决于存储系统实现者的调度水平。
- `mmap` 相当于是让操作系统来管理存储，我们不直接与存储系统进行交互，实现起来会比较简单。但是显然 OS 并不能很完美的应对所有的 workload，使用这种方式的指标大概率不如直接交互好。
- 异步 IO 框架，主要是可以异步的与存储系统进行交互，相对可以比较方便的实现一个比较好的存储系统。

SSD 与 HDD 相比，SSD 的读写效率更高，同时支持多线程的读写。在多个线程下可以达到与顺序读写差不多的性能。而 LSM Tree 是直接针对顺序读写优化的，WiscKey 就是一个针对 SSD 进行优化了的键值存储系统，可以达到很高的吞吐效率。

对比 WAL 与 CoW

- 如果 value 是存储在一页内的，那么这个操作是原子的，如果跨页存储，那么就不是原子的。
- 我们可以记录当前在 SSD 中的系统的 LSN，在启动时对比 WAL 的 LSN 与存储系统的 LSN，Replay 没有执行的 Log 即可。在修改的过程中，可能存在 1. Log 与 Update 均未写入 2. Log 写入 Update 未写入 3. Log 与 Update 均写入，第一种在重启后可以认为该操作未执行，后两种都可以认为已执行。
- CoW 应该执行的步骤包括 1. 分配新的页 2. 复制旧页的内容 3. 在新页上更改 4. 更新页表，其中 4 操作是原子的，在 4 操作执行之前崩溃，恢复时都是之前的状态，可以通过垃圾回收机制来回收之前分配的页，而 4 操作崩溃之后，由于更新已写入页表，因此不会丢失。