KV Store实验报告

郑凯文

2021 年 5 月 5 日

1 简介

本项目是一个简易的基于NVM的KV存储引擎，支持`Read`、`Write`、`Range`和`Snapshot`接口。项目由非易失日志和易失索引两部分组成，通过读写锁和写入顺序的设计保证并发安全性、线性一致性和崩溃一致性。我实现了AVL树、B+树两种不同的索引数据结构，对其进行了比较测试，并添加了`Range`和`Snapshot`的正确性测试、性能测试。项目在较为全面的测试下保持了全面而高效的性能。

在`utils.h`文件中指定了使用的平衡树类型以及B+树阶数等参数，可进行修改。

2 功能实现与优化

2.1 基本思路

模拟NVM的方式是使用`open`+`mmap`打开指定文件，并直接对内存进行读写。在使用真实NVM时，还应刷新物理Cache，但我缺乏相关接口的知识因此没有实现。由于只能打开一个NVM文件，使得诸如将日志分散在多个文件的操作较为困难，因此我的实现中只有一份日志。与此相对，key的索引采用内存中的平衡树进行，我根据key的第一个字符将键值哈希到256个桶中，给每个桶分配一棵树，以此提高多线程并行性。全局为日志分配一个读写锁，再为256棵树各分配一个读写锁。

在程序正常退出时，会将所有树内部的数据追加到文件尾部，下次打开时可直接读取从而节省时间。而若程序中途崩溃，文件中便只有日志，程序将读取日志并从零构建索引树。

日志头中记录当前序列号和当前分配的文件大小。对于每个Write指令，都会被分配唯一的序列号。日志头之后便是各条日志，格式为有效位、key长度、value长度、key、value。设置有效位的原因是减小临界区，在每次Write时，全局先上写锁，接着分配一个序列号、分配好文件中的一段空间。这时便可以解开锁，之后进行较为耗时的复制key、value的操作，复制完毕后将有效位设为1。这样既可以保证崩溃一致性，又减小了冲突。在日志写入完毕后，才对相应索引树加写锁，并进行树的插入操作。之后解除树的锁。

在Read时，对相应树上读锁并进行搜索，得到value在日志中的偏移。再解除树的锁，直接读日志得到value。由于日志是追加式的，这时可以不上读锁。

在Range时，依次对每棵树进行上读锁-查询-解锁的过程，这样不同桶中的树互不冲突。

Snapshot的实现我采取了不太正式的方法，目的是向时间和空间复杂度妥协。在索引树的每个节点上，我都挂了一个vector，记录着这个key所有插入过的数据及其序列号。Snapshot的接口为`Snapshot *GetSnapshot()`，可以获取当前状态的快照，同时在Read和Range接口中可以传入`Snapshot*`指针作为参数，空指针表示当前状态。实际上，`Snapshot`类型的内部只是一个序列号，由于索引树节点上的vector中的序列号一定是递增的，可以利用`Snapshot`的序列号在其中进行二分查找，这样就能获取某一历史版本。若采用哈希表而非索引树，可以采用可持久化数据结构，而索引树很难这样做。我的方案在指定某一快照时进行Read时间复杂度为$O(\log n + \log k)$，其中n为插入过的key的个数，k为索引树节点上的历史版本数，增加的复杂度与可持久化数据结构持平，但在普通的Read、Write时完全没有增加复杂度。同时在空间上，历史版本的个数、日志大小均为$O(m)$，m为进行过的Write的次数，这样没有增加整体的空间复杂度。一种读快照更快的方法是，每个快照内部将索引树复制一份，这样在极端情况下会使空间爆炸（我在bench中测试了这种情况），这其实类似于Backup而非Snapshot。同时在我的实现中，Snapshot在程序退出后不再保持，而Backup应该是将日志、索引树全部备份一份，离线后仍然有效。

在进行Range时，对于AVL树我采用递归进行中序遍历，而B+树可直接以$O(\log n)$复杂度找到头尾叶子，利用叶子之间组成的链表结构顺序遍历。设结果数为p，那么Snapshot会使Range访问叶子时的总复杂度由$O(p)$变为$O(p \log k)$。

2.2 线性一致性

在Write返回时，记录必然写入了日志、写入了索引树，且写入过程中有写锁的保护。这样，其余的读操作均能从索引树中查到Write写入的key，从而在日志中找到value，保证了线性一致性。

2.3 崩溃一致性

Write的过程为

1. 上全局读锁
2. 分配序列号、空间
3. 解除全局读锁
4. 复制键值
5. 有效位置1
6. 上索引树写锁
7. 索引树插入

8. 解除索引树写锁

若在第5步之前崩溃，要么尚未更改全局序列号，要么有效位为0，此条记录均未写入日志，再次打开时不会被读取。若在5~8步崩溃，记录已被写入日志，由于程序非正常退出，索引树数据不会被写入文件，索引树插入情况无关紧要。这样，再次打开时会被读取。

Write正常返回时，记录已经被写入日志。我们认为使用真实NVM时，写入mmap映射的内存便写入了NVM（忽略我没有刷新物理Cache），这样保证了记录被持久化。

3 测试结果

3.1 正确性测试

我在原有的测例基础上，添加了`range_test`和`snapshot_test`。`range_test`中，进行了数千次Range查询，且测试了`lower`、`upper`为空字符串时的边界情况，保证遍历的数目、顺序均正确。`snapshot_test`中创建了10个快照并对同一个key的10次插入验证其正确性。

对于AVL树和B+树，我都通过了所有测例（其中线性一致性使用knossos进行了检查），输出这里就不放了。

当然这里还有许多情况没有实现和覆盖，比如析构时崩溃，要保证完全的正确性是很复杂的，我只实现了主要的部分。

3.2 性能测试

我对B+树的阶数进行了尝试，在阶数为16时，B+树在各个bench中都优于AVL。在磁盘中，B+树的高效与page size有关，而内存中则与CPU的cache line有关，在NVM中可能需要不同的阶数。下面的测试结果都是基于B+树。

我添加了Range和Snapshot的bench，前者在插入了2000条记录后进行了20W次Range查询，后者插入10W条记录后，对同一个key插入10W次，创建10W个快照（每插入一次创建一个快照），并使用这10W个快照各Read了一次。

测试结果如下（由于有波动，取一个近似的值）：

表 1: read/write bench

吞吐率 读比例/分布 / 线程数	99/0	99/1	90/0	90/1	50/0	50/1
1	1140W	650W	720W	520W	350W	365W
2	1800W	1000W	870W	760W	290W	310W
4	2350W	1700W	800W	680W	245W	275W
8	3700W	2300W	800W	630W	215W	245W

表 2: range bench

线程数	吞吐率
1	1.01W
2	1.75W
4	3.34W
8	5.56W

表 3: snapshot bench

线程数	吞吐率
1	175W

其中Range的bench中吞吐率为1s内Range查询的次数，Snapshot的bench中吞吐率为1s内快照读的次数，均没有和写操作混杂。

对于read/write bench，可以看出读比例的减小会使吞吐率急剧减小，这是由于读操作的成本远远低于写操作，读操作不会对日志上锁，且不会发生索引树的旋转、重构。在读比例较大时，适当增多线程数会使得吞吐率上升，但线程数过多时反而不如单线程，存在一个峰值，这是锁、哈希桶的利用率和锁的开销的trade off。峰值随读比例减小而减小，当读比例下降到50%时，单线程吞吐率最高，这应该是因为写操作的频繁使得全局锁、树锁处于长期占用状态，线程冲突严重。

对于range bench，多线程能有效提高吞吐率，这是由于在一棵树中遍历时是比较耗时的，此时其它树都处于空闲状态，多线程可以增加桶的利用率。

对于snapshot bench，在一个树节点上有10W条历史记录时，快照读的效率大约是普通读效率的1/20，这是可以接受的。

4 思考题

1. 如何保证和验证 Key Value 存储引擎的 Crash Consistency？考虑如下 Crash 情况: a. KV 崩溃(进程崩溃) b. 操作系统崩溃 c. 机器掉电

   对于KV崩溃，可以直接使用操作系统提供的功能如kill来模拟。此时保证崩溃一致性比较容易，可以直接使用‘write()‘，或‘mmap()‘后写内存。即使进程崩溃，内存中有脏页，操作系统仍可以辅助完成后续写回磁盘等操作。

   对于操作系统崩溃，可以使用虚拟机模拟。在本实验中，认为写入映射的内存便写入了NVM，而对于磁盘的写入，‘write()‘或‘mmap()‘并不保证调用/写内存后数据信号便被发送到磁盘，可能仍在内存中，若操作系统崩溃便会丢失脏页。此时可以采取Direct IO的方式，不经过内存，直接写入磁盘，并配合日志，检测写入到了哪一条数据。

   对于机器掉电，可以使用真机模拟，或许有专用的软件可以帮助模拟（涉及了磁盘等设备掉电，比较复杂）。此时使用纯软件方法很难满足要求，应使用特殊的硬件，如写入并持久化后返回一个成功信号，此时Write才能返回。若放宽限制，不要求Write返回必写入成功，只要求要么写入要么没写入，那么就可以采取日志或事务机制等。

5 小组分工

我独自完成了所有工作。