KV 存储引擎实验报告

李晨昊 2017011466

2020 年 3 月 24 日

目录

1 简介

实现一个简单的 KV 存储引擎，支持写入键值对，搜索键和的范围搜索功能，并且保证了写入的崩溃一致性和证线性一致性。

我把几乎全部的代码都实现在了`engine_race.h`中，这是我的个人习惯，没有什么特殊的意义。

我修改了`Makefile`，将`-std=c++11`修改成了`-std=c++17`，因为我用到了其中的一些新特性。希望助教评测时用的编译器足够新。

2 实现思路

2.1 基本数据结构

我使用内存中的std::map来存储和检索键值对，持久存储的部分则没有复杂的数据结构，可以简单理解成记录下来了所有插入的键值对的序列。持久存储的每一项逻辑上可以看成这样的数据结构：

```
struct alignas(4096) Data {
  u32 key_sz;
  u8 key[key_sz];
  u32 val_sz;
  u8 val[val_sz];
};
```

读取和写入硬盘的时候都是按照这样的格式。

2.2 写入的线性一致性

也就是要保证存储引擎的线程安全。我使用pthread_rwlock_t来实现写入的线性一致性，在Read操作的时候使用读锁，在Write操作的时候使用写锁。具体实现上我还做了一些优化，包括：

- 尽可能最小化临界区。Write操作的时候只保护了对内存中的数据结构的修改，没有保护write调用，这是因为write本身保证原子性，即只要写入了一定数目的内容，则保证这些内容不会和别的write写入的内容交叉。不过如果write没能一次性写入所有的内容，则这样的原子性对我们的应用就没有意义了，不应该继续调用write来写入后续的内容了，我选择直接返回kIncomplete。
- 将key的第一个字节用作它的散列值，把不同的键分散到 256 个槽中。这个散列函数当然不算一个好的散列函数，但是还是能有效减少冲突，而且保持了字符串间的顺序，不会影响Range的实现。

  在Range操作的时候，先锁住所有相关的槽，在完成访问后再解锁它们。由于这里锁的获取是有序的，根据死锁产生的必要条件，可以保证不会产生死锁。

2.3 写入的崩溃一致性

我使用 Linux 的 Direct IO 来实现写入的崩溃一致性：

```
f.fd = open(buf.c_str(), O_RDWR | O_CREAT | O_APPEND | O_DIRECT, 0666);
```

后续直接对f.fd进行 IO 操作即可。不过需要注意的是 Direct IO 模式下要求每次写入的长度都按页对齐，所以写入键值对的时候，需要先把必要的信息写入一个按页对齐的缓冲区，再把缓冲区中的内容写入文件：

```
u32 tot_sz = (8 + key_sz + val_sz + PAGE_SIZE_M1) & ~PAGE_SIZE_M1;
u8 *buf = (u8 *)((usize)(BUF + PAGE_SIZE_M1) & ~PAGE_SIZE_M1);
*(u32 *)buf = key_sz;
memcpy(buf + 4, key.data(), key_sz);
*(u32 *)(buf + 4 + key_sz) = val_sz;
memcpy(buf + 8 + key_sz, val.data(), val_sz);
if (write(f.fd, buf, tot_sz) != tot_sz) {
  // if IO cannot be performed in one `write` call, there will be
   synchronization problems
  return kIncomplete;
}
```

这样每次最少写入一个页的内容。这可能会导致一定的空间浪费，不过也没有什么更简便的做法了。

这个缓冲区BUF是一个thread_local变量，所以对它的操作不需要锁的保护。我尝试过用alignas关键字来标记BUF，让它按页对齐，不过似乎没有效果，所以现在采用手动计算对齐的方式。

我尝试过使用mmap + msync来达成写入一致性，在完成对mmap内存的写入后再调用msync来同步对应的页。经测试这样实现的性能非常差，无论是预先使用ftruncate来预留文件长度，还是每次写入前先调用ftruncate来调整文件长度，每次msync同步一个页都会耗时约7ms ~10ms，与现在的实现有数量级的差异。

3 测试结果

3.1 正确性测试

提供的所有测试程序都能正确通过，对于Range操作我也自己编写了一些测试 (在test/range_test.cpp中)，也能顺利通过。一组输出如下：

```
====================== single thread test ============================
open engine_path: ./data/test-20274702559820
```

```
====================== single thread test pass :) ======================
--------------------------------------
====================== multi thread test ============================
open engine_path: ./data/test-20279021176544
====================== multi thread test pass :) ====================
--------------------------------------
====================== crash test ============================
open engine_path: ./data/test-20710228443442
====================== crash test pass :) ====================
--------------------------------------
====================== range test ============================
open engine_path: ./data/test-20713103413864
====================== range test pass :) ====================
```

3.2 性能测试

我在自己的笔记本 (Intel(R) Core(TM) i5-6300U) 上使用不同参数进行了测试，Throughput(op/s) 结果如下。因为这个笔记本的性能相当低下，所以如果使用其它平台来测试的话，结果也许会有显著的提升。

线程数 \ 读比例，分布	99, 0	99, 1	90, 0	90, 1	70, 0	70, 1	50, 0	50, 1
1	332227	376357	76308	79737	27431	25282	13962	15567
2	556985	576568	78242	81228	27784	20520	11431	13936
3	582371	659195	82718	80461	21689	18849	10387	13631
4	463965	510715	59205	49560	14246	15430	10666	8828
5	527341	508689	56476	46661	14674	13257	10121	10275
6	505485	552604	56899	56646	13838	18941	9364	9227

观察第一列可以看出，即使不涉及任何线程冲突，读比例的减低也将显著降低性能，这是因为写操作的开销远远大于读操作，前者只需要操作内存中的数据结构，而后者需要进行硬盘操作。

在线程数不多时，随着线程数的增多，在读比例相对高的情景下性能提升更多，这有可能是因为线程冲突和写锁的开销导致的，也有可能是机器的硬盘的写性能已经达到瓶颈导致的。随着线程数的继续增加，性能会大致稳定到一个值，可以看出读比例越低，稳定时的线

程数也越少（然而具体在线程数为多少时稳定，显然受 CPU 和硬盘性能的影响，这个数据仅供参考）。

此外，当数据分布为 1 的时候性能有一定的提升（虽然不大），这说明输入数据的分布也会影响到存储引擎的性能。在读比例较高的时候提升较大，这可能是因为在我的视线中，输入数据的分布对内存的访问模式的影响较大，而对硬盘的访问模式的影响较小。

4 思考问题

如何保证和验证 Key Value 存储引擎的 Crash Consistency? 考虑如下 Crash 情况：

1. KV 崩溃 (进程崩溃)

   为了测试这种情况，可以使用杀死进程等纯软件手段来使进程崩溃，然后测试数据正确性。

   为了在这种情况下保证 Crash Consistency，只需要使用正常的`write`，或者写`mmap`内存，因为即使进程崩溃时内核中有脏的缓存，操作系统依旧会完成后续的处理。注意`libc`提供的`fwrite`仍然不能满足要求，因为`libc`的缓存在用户内存，操作系统不一定会完成后续的处理。

2. 操作系统崩溃

   为了测试这种情况，可以使用虚拟机，如 QEMU 等模拟断电或者操作系统崩溃，然后测试数据正确性。

   为了在这种情况下保证 Crash Consistency，可以使用 Direct IO，或者`write` + `fsync`，或者写`mmap`内存 +`msync`。总之是需要让操作系统发出写硬盘的指令并成功完成后，`Write`才能返回。

3. 机器掉电

   为了测试这种情况，可以拿真机来运行并断电，然后测试数据正确性。但是这样实际上很难精确控制断电的时机，可能需要相当数量的测试。

   为了在这种情况下保证 Crash Consistency，普通的操作系统提供的软件接口不太可能能够满足要求，因为即使硬盘控制器返回了成功写入的信息，也有可能只是写入了硬盘的缓存，而没有真正写入持久存储。一些可能（部分）满足要求的方法包括：使用特定设备的特殊的，保证返回成功信息时即保证成功写入持久存储的指令；或者采用校验位之类的策略，排除没有成功写入的数据（这不总能保证`Write`返回时写入必定成功，但是至少可以避免错误的数据引发更大的错误）。