key-value存储引擎实验报告

张晨，计71，2017011307

1 通用版存储引擎-engine1

1.1 基本假设

内存不足以存下所有的key，也不足以存下所有的value，但可以存下所有的key的哈希值及其在文件系统中的位置。

1.2 原理

1.2.1 文件系统

存储格式

1. $*.dab$ 中存每一条key-value的完整信息，格式为(key的长度，value的长度，key的内容，value的内容)。
2. $*.log$ 中存对数据库的所有写操作的摘要，格式为(时间戳，key的长度，完整操作在$*.dab$中的位置，key的哈希值，key的内容)。“完整操作在$*.dab$中的位置”指文件编号和在文件中的偏移量，下同。
3. 000.mtd (metadata)中存已用时间戳的最大值。

$*.dab$,$*.log$**实现细节** bench有至多64个线程同时运行，因此同时打开64个文件，若某个线程需要向文件中写，则分配一个空闲的文件。空闲的文件采用队列进行维护。若在某次写入后文件到达一定大小，则关闭之并打开一个新文件。

时间戳的细节 时间戳记录的是已经进行的写操作的个数，使用mmap写到文件中。为减少IO量，每1000次写操作更新一次，在引擎重启时直接将时间戳+1000以防重复。

1.3 内存中的内容

多线程哈希表 维护从key的哈希值到(key在$*.dab$中的位置，key的出现次数)的映射。对于读操作，直接利用字符串的哈希值从此表中找到表项在$*.dab$文件中的存储位置，然后进行读取，校验key值是否相同，若相同则返回对应的value，若不相同则报错（哈希值是64位的，实际出现哈希冲突的概率极小，若追求更高的可靠性则可考虑使用更大的哈希值）。

热数据表 输入数据符合zipfan分布，因此考虑将常用数据的key-value值都存在内存里。为判断数据是否应该放在内存中，需要维护一个表项按出现次数排序的有序表。我选择使用插入排序进行维护，每次操作只会将某个表项的出现次数加一，因此将该表项前移很少的位置即可满足出现次数的有序性。有序表仅维护需要存在内存中的数。此外，数据分布是不变的，使用最开始的一段时间的统计结果即可较为准确的得到最常用的key是哪些，因此当某个表项的出现次数大于256之后便不再在表中进行更新。使用std::map记录每个数据是否在这个有序表中。

1.4 Cache Consistency的保证

1.4.1 操作流程

借鉴write ahead log的思想，通过仔细设计各更新操作的顺序来保证cache consistency，大原则是先更新文件，再更新内存，具体如下

1. 在$.dab$文件中写入该条目的完整信息

2. 在.*log*中添加此条目
3. 更新哈希表
4. 更新热数据表

1.4.2 文件内容的恢复

通过∗.*log*文件中的内容即可将哈希表（除出现次数）完整恢复，哈希表中的“出现次数”、热数据表不进行恢复。

1.5 测试结果

测试结果如图1所示，主要有如下发现：

- 由于IO量过大，此程序基本不具备扩展性。
- bench的数据规模随线程数的增加而增加。在线程数较大时，因为数据规模过大，每秒的操作数反而降低。
- zipfun分布与均匀分布的效率相似，这是因为程序的主要瓶颈在写操作上，读操作的具体实现方式对运行时间影响有限。

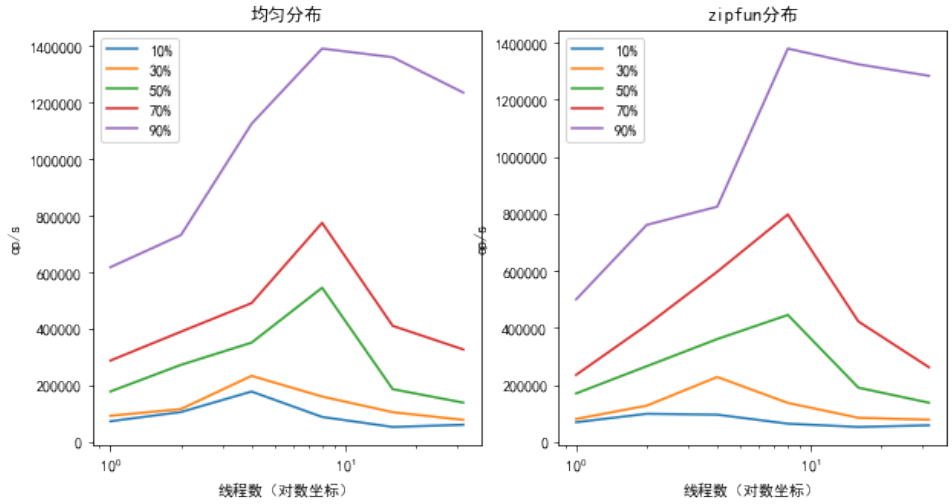

图 1: 通用版存储引擎测试结果

1.6 总结

巨大的写操作量使得这个程序的优化潜力很小，因而我放弃对其进行更细致的优化，而是进行了另外的尝试。

2 面向bench的存储引擎-engine2

2.1 bench的特性

1. key长度固定为8，可以被转化为64位大端法表示的无符号整数，且整数的取值范围为0至8e5。
2. value长度固定为4KB，内容随机生成，但重复率极高。
3. 没有删除操作。
4. 开始会进行较为均匀的写操作。

2.2 key、value的编号

key 虽然key可直接转化为一个小于8e5的整数，但是由于key是大端法储存，转化得到的整数的大小关系与key的大小关系不同。8e5小于2^{24}，因此key的后五个字符的ASCII值一定是0，key的前三位可以方便的转化为小于2^{24}且与key的大小关系相同的整数。（该整数下称keyID)

value 将value按第一次出现的顺序依次编号(下称valueID)，利用哈希表维护编号的过程。通常情况下，会选择在哈希表中使用value的哈希值作为索引，但是bench中的value是随机生成的，因此将value的前若干位整合成一个64位无符号整数与遍历整个字符串求一个哈希值的效果几乎是一致的。为提高速度，我选择了前者，这个优化使得我的Read操作做到了几乎没有常数的O(1)。

2.3 文件系统

1. data.txt,下称数据库，存所有出现过的value值，格式为(长度，内容，#)。“#”标志着条目的结束。尽管bench中长度固定，理论上可以不存每个value的长度，但为了顺利通过test，我选择将它们存了下来。
2. log.txt,下称log，存所有的写操作，格式为(keyID, valueID, #)，“#”标志着条目的结束。

2.4 内存中的内容

多线程哈希表 用以维护出现过的value及其对应的valueID

普通数组1 用以维护每个valueID对应的value

普通数组2 用以维护每个keyID对应的valueID，Read时直接从这里返回。

分块有序表 将keyID的取值范围(0-2^24)按每256个值一块进行分块，块内使用插入排序有序地维护已经存在的表项。Range Search时遍历与所查找区间有交的所有块，并选择合适的进行返回。为保证Range Search时访问的是同一时刻的数组，需要使用锁保证Range Search和Write不同时进行。

2.5 Cache Consistency的保证

2.5.1 操作流程

1. 若value出现过，转到5
2. 向数据库中写入value，根据写入顺序得到valueID
3. 向普通数组1的位置valueID写入value
4. 向多线程哈希表中写入(value, valueID)
5. 向log中写入(keyID, valueID)
6. 向普通数组2中写入(keyID, valueID)
7. 向分块有序表中写入(keyID, valueID)

2.5.2 数据恢复

通过读数据库和log，可恢复内存中的所有内容。

2.6 性能优化

采用64线程，读写比1:1，zipfun分布进行调优。

- 通用版存储引擎——120493op/s
- 本存储引擎——314118op/s
- 将所写内容统一复制到一个buffer里，再将buffer写入文件——880960op/s

- 将文件打开命令由“O_RDWR|O_APPEND”改为只有“O_RDWR”——951354op/s
- 在打开文件后将文件truncate到足够大——$1.115107 * 10^6$op/s
- 使用mmap写log——$1.3259871 * 10^7$op/s

至此，无法使用profiler找到运行时间超过10%的函数，故继续优化的意义十分有限。

2.7 性能分析

本存储引擎优于通用版 因为IO量大大减小。

复制到buffer后写入文件 一方面，若不复制到buffer，则为了保证内容写到一起，需要给写入操作加锁，使得操作系统无法对写操作进行自动优化；另一方面，这样也减少了write操作的调用次数。

删除O_APPEND 所有文件操作不涉及指针的回退，因此删除后依然是对的。删除此选项使得写的时候系统不用获取当前文件大小并移动指针。

将文件truncate到足够大 这省去了由于文件增大而造成的磁盘内数据移动。

mmap mmap对程序有接近10倍的优化，这是因为log的写操作数量极多，但是每次的长度仅有固定的9，故并行程度较直接写要好得多。

2.8 测试结果

测试结果如图2所示，主要有如下发现：

- 程序扩展性较好
- zipfun分布的运行效率较低，这是因为那部分经常出现的数据的出现次数过多，各个进程同时修改/访问这些数据造成进程间的等待。

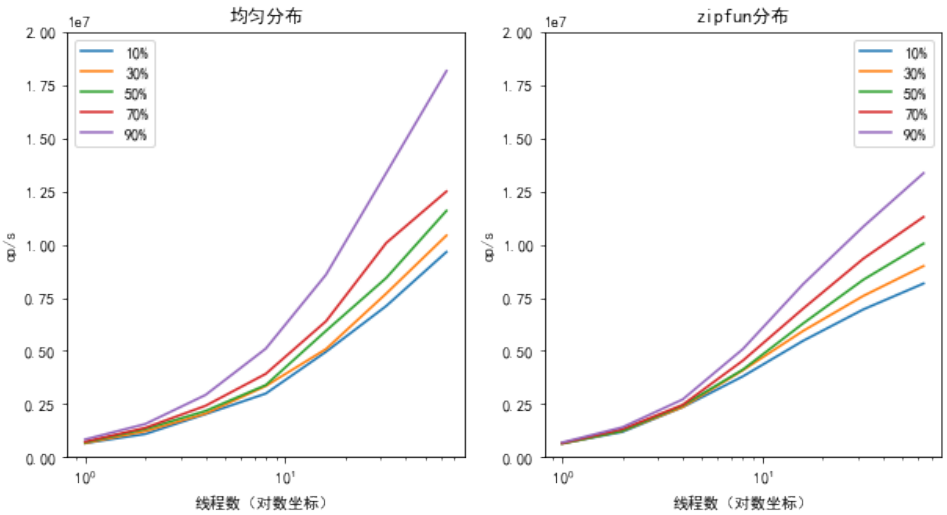

图 2: 面向bench的存储引擎测试结果

2.9 其他

实现了在key的格式与bench相同的情况下的range search，单元测试见test/range_test.cc