LSM-KV 项目报告模板

XXX 522XXXXXXXX

2024 年 5 月 21 日

1 背景介绍

LSM-KV 是基于 LSM-Tree 的数据结构实现的, 能进行键值对简单的持久化存储和查询的小型 KVStore。LSM-Tree（Log-Structured Merge-Tree）不同于一般意义上的树状数据结构（B-Tree, BST），LSM-Tree 是一种的数据存储结构。通过 SST 文件分层存储以及后台文件 Merge 等操作，大幅优化了数据的写入性能，被广泛应用与如 LevelDB、RocksDB 等 NoSQL 数据库中。本次项目实验中的使用的 LSM-Tree 不同于常见的 LSM-Tree，而是一种键值 KV 分离的对 SSD 更加友好的基于 LSM-Tree 的存储引擎设计。它通过 KV 分离降低了 LSM-TREE 令人诟病的写放大。但键值分离在降低写放大的同时，也不可避免引入了一系列新问题，例如删除 Value 的回收、Get 操作额外引入的随机读。本报告中集中在对键值分离的 LSM-Tree 的四项基本操作 (Get、Put、Del、Scan) 进行测试和分析。

2 测试

此部分主要是展现你实现的项目的测试，主要分为下面几个子部分。测试部分应当是文字加上测试数据的图片展示。

2.1 性能测试

2.1.1 预期结果

LSM-Tree 的四项基本操作 Get、Put、Del、Scan 操作性能主要受数据大小、数据量以及索引影响。此外，部分操作性能可能受后台 Compaction 影响，以及 BloomFilter 性能测试。

1. 数据大小对性能的影响

   LSM-Tree 基本操作性能随数据大小增大而减少。能够通过数据大小对性能产生影响的主要是 Value 的大小，由于项目中键的类型为 uint64，编码长度为固定的 8bytes。而 Value 是可变长的，故操作性能受 Value 的大小影响。

数据大小对性能影响通过多方面作用造成的。首先是直接的影响。随着键值增大，在插入 Memtable 拷贝的耗时增加。而且过大的 value 大小可能大于 Cache Line 大小，造成 Cache Miss，减慢了操作性能。其次，在生成 SST 文件时，大的 Value 构建 ValueLog 文件格式的成本增大，生成 crc16 的操作耗时也随 value 大小增大, 进一步降低了操作性能。而且随 Value size 增大而增大的 valuelog 文件，也进一步减慢了磁盘读取和追加写的性能。

2. 数据量对性能的影响

对于四项基本操作来说，数据量增大会降低操作性能。除了数据量的增大导致的类似数据大小增大的一系列对操作减慢的影响外。还有随着数据量增大，SST 文件数量增多，发生 Compaction 操作次数增多，频繁的触发 Compaction 可能导致后台线程不能够及时处理，被迫让主线程等待，拖慢了操作性能。

3. 索引程度对性能的影响

索引的程度越高性能越高。在本次实验条件下，进仅对数据的 key 进行了缓存，而没有做 Value 的缓存，因此一定程度上 Get 操作较慢。在不考虑 Value 的缓存下，Key 的缓存主要是对 SST 的索引或者 Bloom Filter 进行缓存。理论上，两者都会对性能产生明显影响。

但实验结果显示在两者之中 SST 索引的缓存对性能影响较大, 而 Bloom Filter 对性能影响较小。原因在本次实验条件下，只有对索引全部缓存或者全部不缓存两种选择，所有的 BloomFilter 操作并不能减少读 SST 文件的时间 (因为不用读)，而 Get 的开销主要体现在读取 SST 文件中，这一点在实验数据中明显体现，具体表现为无索引信息下 GET 操作的延迟是正常情况的 500 倍。所以在实验中 BloomFilter 主要减少了对索引进行二分查找的次数，而 SST 索引中键对数为 408，平均操作次数和操作时间非常短，而 BloomFilter 检测可能会引入一定操作时间，所有几乎对 GET 方法没有任何影响，而实验数据中的性能略微差距很有可能是实验中的环境噪音。

4. 其他

Compaction 的影响

在本次实验实现了后台线程处理 Write 和 Compaction。分析数据可以发现，开启后台线程后，Put 的吞吐量的平均值较不开启时高，且 Compaction 时 Put 操作下降的次数和幅度减小。虽然后台线程的开启，一定程度上降低了主线程的处理能力，但平均吞吐量较未开启相比增加了近 2000op/s。这一数据较低的原因是，本次实验条件下 SST 文件键值对数为 408，数量太小，插入速度高，Memtable 满的速度远高于后台 Write 和 Compaction 的速度，所以体现出来并行 Write 和 Compaction 对性能影响不大。

Bloom Filter 大小配置的影响

随 Bloom Filter 大小的减小，PUT 操作时延应该逐渐减少。原因同 Compaction 的影响，Bloom Filter 减小增大了键值对数，减少了 SST 文件数量，进而也减少 Compation 的次数，故性能提升。而 GET 操作时延则体现为先减少再增大，原因是当 Bloom 过大时，每个 SST 文件中键数过少，SST 文件多，查询需要访问的缓存多，进行 Bloom 过滤次数多，二分查找次数多，常数项较大，因此时延最高。而随着 Bloom Filter 减小，键数多大大增加，远远超过 Bloom Filter 的大小，在此时 Bloom Filter 形同虚设，只会多一次 Bloom Filter 过滤操作，而不会产生实质过滤效果，相当于未开启 Bloom Filter，因此时延增大。

2.1.2 常规分析

1. 包括 Get、Put、Delete、Scan 操作的延迟，你需要测出不同数据大小时的操作延迟，为了测试的合理性，你应当对每个数据大小测量然后计算出平均延迟

   数据分析: 实验数据见图 1，延迟的数据单位为 μs，其中 scan 操作的延时单位为 ms。原因是 scan 操作过慢，较其他操作小 3 个数量级之多，故为了数据展示，将其单位更换为 ms。可以看到数据与前述分析相符。

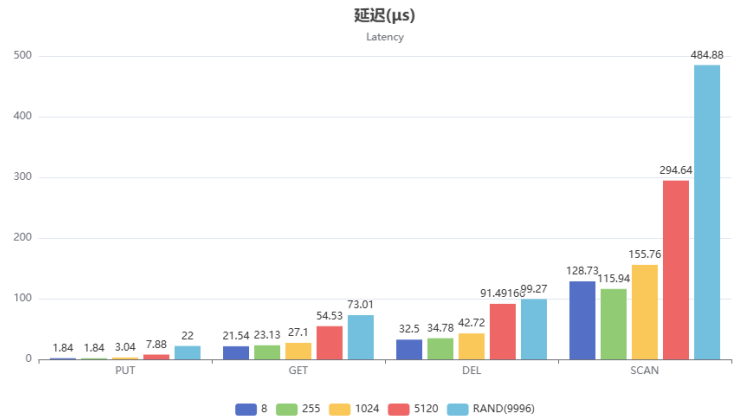


图 1: 不同数据大小时的操作延迟

2. 包括 Get、Put、Delete、Scan 操作的吞吐，意思是指系统单位时间内能够相应的请求的次数，显然，在展示你测试的时候你需要指明 Key、Value 的大小（注意是数据的大小，并不是具体的值）

   数据分析: 实验数据见图 2，吞吐量的数据单位为 10k op/s，其中 scan 操作吞吐量的数据单位为 op/s。原因是 scan 操作过慢，较其他操作小 3 个数量级之多，故为了数据展示，将其单位更换为 op/s。可以看到数据与前述分析相符。

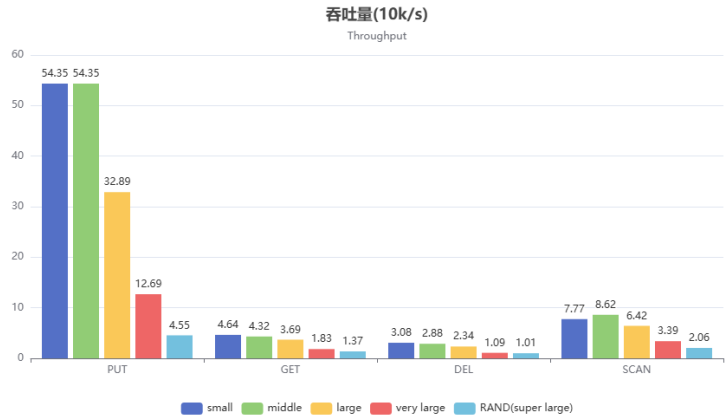


图 2: 不同数据大小时的操作吞吐量

2.1.3 索引缓存与 Bloom Filter 的效果测试

需要对比下面三种情况 GET 操作的平均时延

1. 内存中没有缓存 SSTable 的任何信息，从磁盘中访问 SSTable 的索引，在找到 offset 之后读取数据
2. 内存中只缓存了 SSTable 的索引信息，通过二分查找从 SSTable 的索引中找到 offset，并在磁盘中读取对应的值
3. 内存中缓存 SSTable 的 Bloom Filter 和索引，先通过 Bloom Filter 判断一个键值是否可能在一个 SSTable 中，如果存在再利用二分查找，否则直接查看下一个 SSTable 的索引

数据分析: 实验数据见图 3，延迟的数据单位为 μs，可以看到数据与前述分析相符。

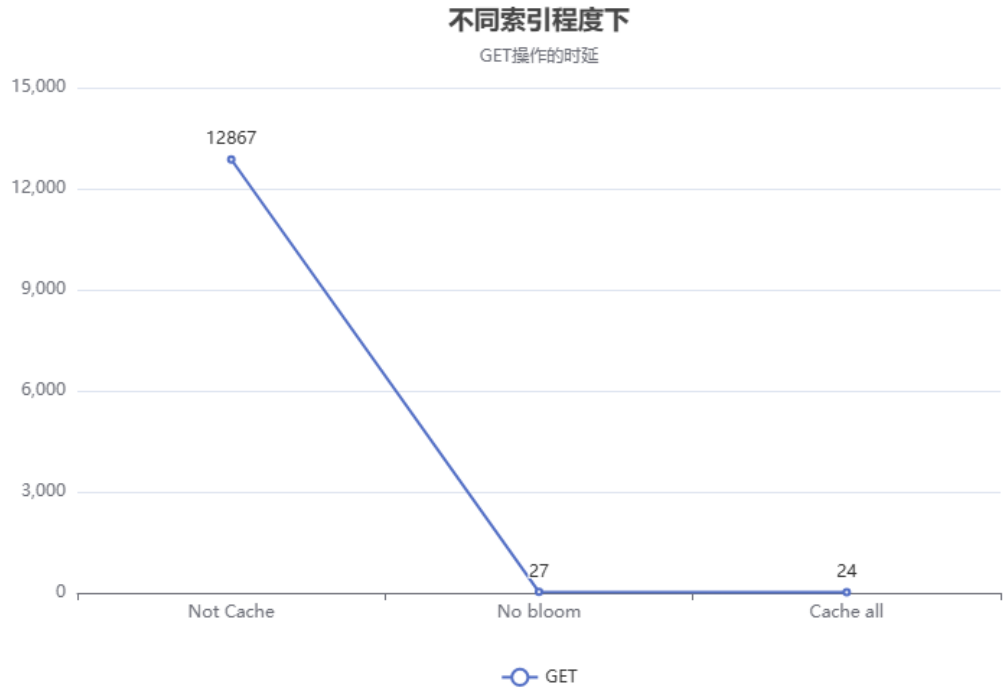
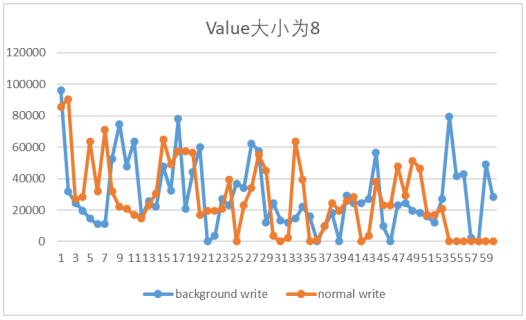


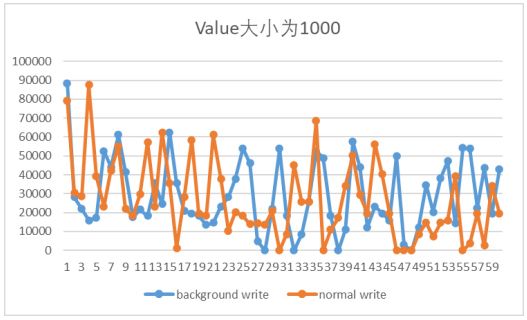
图 3: 索引缓存与 Bloom Filter 的效果测试

2.1.4 Compaction 的影响

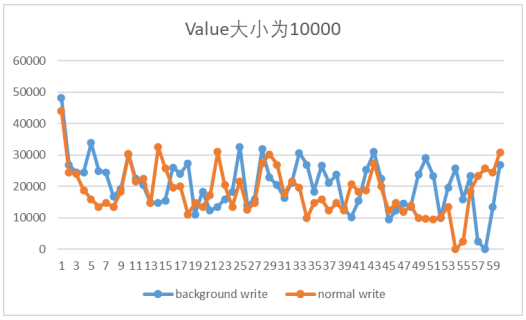
不断插入数据的情况下，统计每秒钟处理的 PUT 请求个数（即吞吐量），并绘制其随时间变化的折线图，测试需要表现出 compaction 对吞吐量的影响。**数据分析**: 实验数据见图 4，图标纵轴为吞吐 (ops)，横轴为时间轴 (s), 可以看到数据与前述分析相符，后台线程的开启提高了吞吐量。



(a) Value size is 8



(b) Value size is 1000



(c) Value size is 10000

图 4: Campaction 的影响

2.1.5 Bloom Filter 大小配置的影响

Bloom Filter 大小的影响：Bloom Filter 过大会使得一个 SSTable 中索引数据较少，进而导致 SSTable 合并操作频繁；Bloom Filter 过小又会导致其 false positive 的几率过大，辅助查找的效果不好。你可以尝试不同大小的 Bloom Filter 并保持 SSTable 大小不变，比较系统 Get、Put 操作性能的差异并思考原因。**数据分析**: 实验数据见图 5, 延迟的数据单位为 μs，分析可见前述分析。

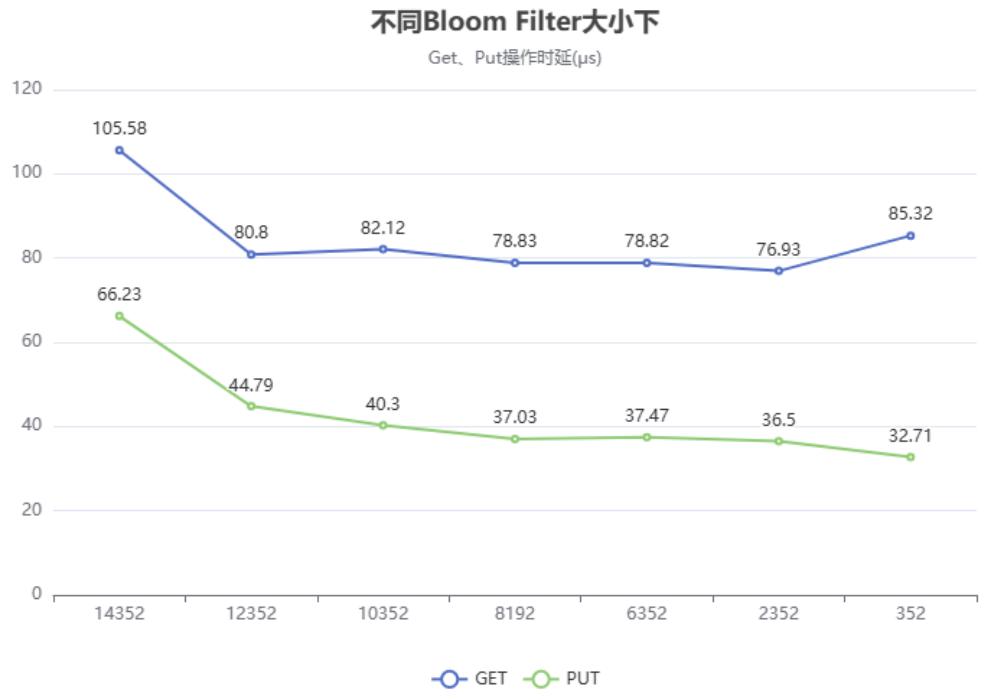


图 5: Bloom Filter 大小配置的影响

3 结论

通过这次 Project，对 Modern C++ 有了初步的接触，了解了 Modern C++ 的实践。初步了解了并发编程，也实际使用了 ICS 课程中关于缓存的知识，可以看到可观的性能提升。最后实验的吞吐量比预期高很多，可以看到 LSM-Tree 对 KVStore 性能提升。

4 致谢

感谢开源项目 LevelDB[1] 的源码，能够清晰而全面了解到 LSM-Tree 构建的过程，在阅读源码的过程中也学到了许多 C++ 代码在工程上的实践。非常感谢 Clion(除了它的 ssh)，它的 Debug 功能帮我排除了极多的错误，若是单单只通过 gdb 进行 debug，恐怕现在还在忍受 Segmentaionfault 的痛苦。

5 其他和建议

在本次 LSM-KV Project 中遇到了最大困难是在 De 完 GC 的 bug 之前，先将 Merge 以及 Compaction 的操作实现，导致 Bug 膨胀，大大加大了 Debug 的负担。其中印象最深的 Bug 是 Compaction 时，上层文件和下层文件在时间戳相同时，可能仍有相同的键值。

此外也深刻意识到“过早优化是万恶之源”的道理所在。在实现过程中，过早优化很有可能可能不会对最终性能产生正面影响，但绝对会加大 DEBUG 的难度。

参考文献

[1] Jeff Dean and Sanjay Ghemawat. Leveldb. `https://github.com/google/leveldb`, 2011. Accessed: 2024-05-22.