LSM-KV 项目报告

王雨杭 522021910183

2024 年 5 月 22 日

1 背景介绍

LSM-KV 是一个基于日志结构合并树（Log-Structured Merge-tree，简称 LSM tree）的键值存储系统。LSM tree 是一种磁盘存储数据结构，用于处理大量随机写入的情况。它通过将随机写入转化为顺序写入，从而提高写入性能。LSM-KV 项目的主要目标是实现一个高性能、可扩展的键值存储系统。它主要用于处理大数据应用，如搜索引擎、数据库和文件系统等，这些应用需要处理大量的数据写入和查询。同时，LSM tree 也是 RocksDB、LevelDB、HBase 以及 Prometheus 等知名开源项目底层的存储引擎的数据结构，具有很高的学术和工业应用价值。

LSM-KV 的主要特点包括高性能（通过使用 LSM tree，LSM-KV 可以提供高性能的数据写入和查询）、可扩展性（LSM-KV 设计为可以在多台机器上分布式运行，从而处理大规模的数据）、支持多种数据类型（LSM-KV 可以存储键值或复杂的数据类型，如列表、集合和哈希表）等，这些特点使 LSM-KV 成为处理大数据应用的理想选择。

对比 B+ 树等持久化数据结构，LSM 树具备一些专门为记录日志等磁盘写入密集型应用优化的特性，它的主要优势有写入效率高（LSM tree 通过缓存写入操作并批量进行，可以显著减少磁盘 I/O 操作，从而提高写入效率）、高效的磁盘空间利用（LSM tree 通过合并和压缩操作，可以有效地利用磁盘空间，减少数据冗余）、高并发性能好等，但在读取效率上存在一定的劣势。

在本项目之中，实现了基本的键值类型为 int, string 的键值存储 LSM 树，支持 GET、PUT、DELETE 和 SCAN 操作，并实现了键值分离存储、支持 GC、持久性和缓存、修改布隆过滤器大小等配置的特性。

2 测试

我的测试实现如下：在使用 cache 和 Bloom Filter，使用 Cache 不使用 Bloom Filter，两者都不使用三种情况下以及都使用，改变 BloomFilter 大小为 1024 * i(i 从 1 到 15 变化) 字节几种情况，数据（value）大小分别为 [10, 100, 1000, 10000] 字节，开始 GET，POST，DEL，SCAN

前已有分别为 [1000, 3000, 5000] 组随机数据的情况下进行测试，每次测试随机 GET 10000 次取平均，PUT 1000 次取平均，DEL 1000 次取平均，SCAN 100 次（范围随机）取平均测量耗时和吞吐量。

2.1 性能测试

2.1.1 预期结果

基于 LSM tree 的特性，应该表现出：

1. GET 请求较慢，在使用缓存的情况下，会稍快
2. PUT 和 DEL 请求较快，且由于 DEL 请求最坏只需要 PUT 一个小字符串，DEL 应该比 PUT 更快
3. SCAN 请求非常慢，因为要遍历每一层，与 GET，PUT，DEL 可能有数量级差距
4. 在触发合并时，PUT 请求会显著变慢

2.1.2 常规分析

1. 对于 Get、Put、Delete、Scan 四种操作，在 value 大小为 value_size，预先插入键值对个数为 prebuilt_data_num 时，单次操作耗时如下表

value_size	prebuilt_data_num	Get	Put	Del	Scan
10	1000	1	4	<1	24
100	1000	2	4	<1	26
1000	1000	9	7	<1	23
10000	1000	99	57	<1	39
10	3000	5	11	<1	868
100	3000	4	11	<1	656
1000	3000	19	15	<1	2684
10000	3000	140	59	<1	18429
10	5000	6	15	<1	1095
100	5000	8	13	<1	1243
1000	5000	21	17	<1	2804
10000	5000	147	62	<1	21554

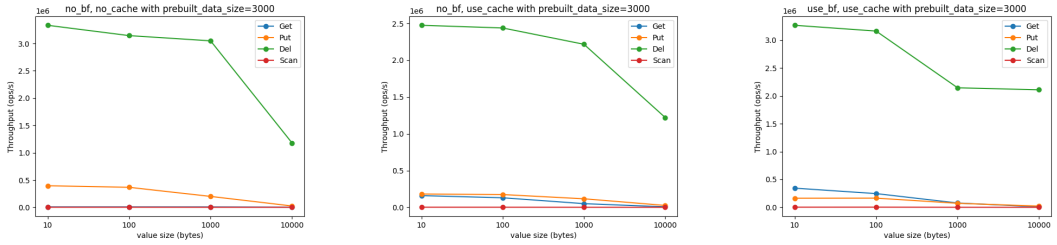
表 1: GET,PUT,DELETE,SCAN 时延表（微秒）

2. 在 value 大小为 10、100、1000、10000 字节，在 LSM 之中预插入 1000、3000、5000 个数据，使用缓存和布隆过滤器，布隆过滤器大小为 8KB 的情况下，四种操作的吞吐如下表（key 为 8 个字节，单位 ops/s）：

value_size	prebuilt_data_num	Get	Put	Del	Scan
10	1000	5.22e+05	2.48e+05	2.74e+06	4.01e+04
100	1000	3.38e+05	2.26e+05	2.61e+06	3.71e+04
1000	1000	1.02e+05	1.29e+05	2.61e+06	4.18e+04
10000	1000	1.01e+04	1.74e+04	1.45e+06	2.53e+04
10	3000	1.69e+05	9.04e+04	2.46e+06	1.15e+03
100	3000	2.29e+05	8.38e+04	2.34e+06	1.52e+03
1000	3000	5.11e+04	6.28e+04	2.40e+06	3.73e+02
10000	3000	7.10e+03	1.68e+04	1.62e+06	5.43e+01
10	5000	1.54e+05	6.59e+04	2.38e+06	9.13e+02
100	5000	1.23e+05	7.16e+04	2.45e+06	8.04e+02
1000	5000	4.74e+04	5.63e+04	2.16e+06	3.57e+02
10000	5000	6.78e+03	1.61e+04	1.62e+06	4.64e+01

表 2: GET,PUT,DELETE,SCAN 吞吐量表 (ops/s)

典型的四种操作性能对比图如下 1：



(a) (b) (c)

图 1: 性能对比图

分析两表和图 1可以得出以下结论：

1. 四种操作的速度从大到小为：DEL>GET>PUT>SCAN
2. 有缓存的情况下，GET 和 PUT 的速度差别不大；但没有缓存的情况下，GET 明显慢于 PUT。

3. DEL 速度非常快。这是因为在我的实现之中对 DEL 进行了优化，DEL 只在内存之中查找，最坏也只需要 PUT 一个短的 value

4. value 较大时，GET 操作退化的速度比 PUT 快，在 value_size 为 1000 个字节数量级时，PUT 的速度就已经超越了 GET。这是因为 GET 需要多次 query 来确认 key 对应的 value 有效，而在存储的键值对都较长时，相关开销（例如，文件定位、取到验证用的 magic 和校验和）的开销远远超过了内存操作的开销，成为了开销的主体。这也验证了 LSM 结构在写上的优越性和读上效率不高的问题。

5. SCAN 操作也如预期的非常慢，与其他三种操作有数个数量级的差距。这是由于 SCAN 需要在 sstable 之中对每一层都进行查找归并，同时在 vlog 里面查找多个值。

2.1.3 索引缓存与 Bloom Filter 的效果测试

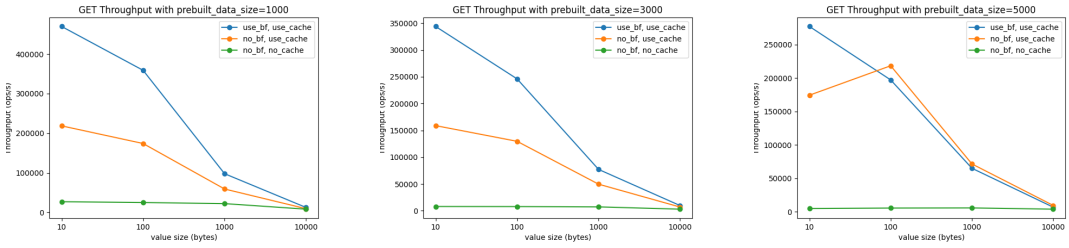
对比下面三种情况 GET 操作的平均时延

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

GET 操作平均时延对比如下图 2.1.3。可以看到，在 value 大小较小时，内存操作是影响 GET



(a) (b) (c)

平均时延的核心，因而缓存和布隆过滤器均有极为明显的加速效果。而当 value 变大到 10000 字节量级时，影响 GET 的平均时延的关键点转移到了磁盘操作，不同缓存策略下时延趋于相同。而对于布隆过滤器，在数据量较大时，由于它会使得每个 SSTable 容纳的键值对数量变少，LSM 树更深，优化程度有所下降，需要权衡。

2.1.4 Compaction 的影响

不断插入数据的情况下，PUT 吞吐量随时间变化如图 2：可以发现，大趋势是 put 的吞吐

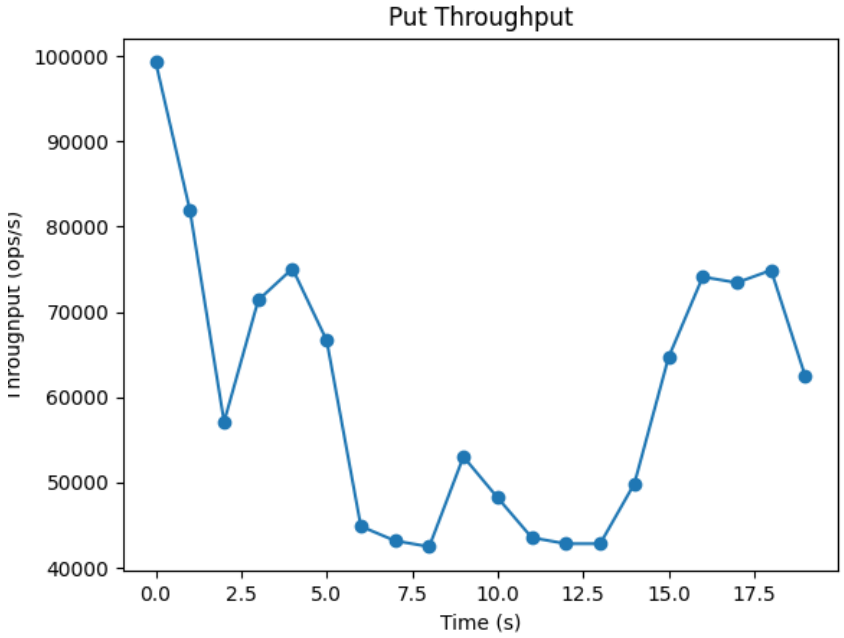


图 2: PUT 吞吐量随时间变化图

量随时间增加而下降，但是在以秒计数的时间间隔之中，数值抖动明显，并且不断出现了局部吞吐量随时间增加而上升的情况。原因为：初始时数据较少，合并开销小，吞吐量非常高；而随着时间的增加，合并可能涉及的层数和 SSTable 个数也越来越多，合并的平均耗时增加。而抖动的原因是，新建一层的大合并（或者任意层数为 n）的合并出现所需要的新增键值对个数随指数增长，在一秒的时间间隔内，已经无法保证合并的均匀性，在产生大合并的时间段内，吞吐量就低；而大合并结束后，吞吐量高，出现抖动。

2.1.5 Bloom Filter 大小配置的影响

理论上，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。

但在我的实验之中，Bloom Filter 不同大小对 PUT 带来的效果不太明显，在不同大小 Bloom Filter 上运行 20 秒 PUT，value_size 取 100，吞吐量随时间变化的测试结果如图 3。

可以看到，曲线之间区别较小，可见在本次实验之中，PUT 的性能决定点为 vLog 的写入部分，而 BloomFilter 大小的间接影响虽然不小，但相较 vLog 写入并不是最关键的。而对于

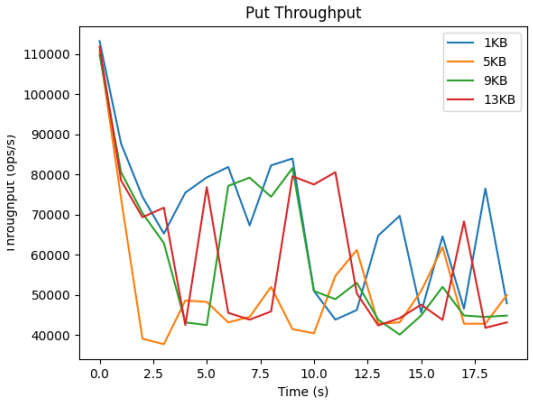


图 3: 不同 BloomFilter 大小下的 PUT

GET 操作，我在不同 value 大小和预置数据大小上进行了测试，见图 4。

可见在布隆过滤器较小时，实测结果和预测有一定出入，具体表现在布隆过滤器大小对性能的影响较小、抖动明显，且在布隆过滤器较小时，反而有更高的效率。这可能是由于我的测试集本身较小（最大 5000 个数据），使得布隆过滤器在 LSM 层数上的负面影响更大。但基本的，当布隆过滤器变得较大时，GET 的性能较差，这是符合预期的。

3 结论

在这个 Project 之中，我实现了一个具备较高鲁棒性和可用性的 LSM 键值存储系统，同时通过多种方法实现了较为完备的正确性和性能分析，性能表现符合预期，让我对这个 project 和对应的 LSM 树具备了更深刻的理解。

4 致谢

首先，我要感谢 2022 级软件工程专业的同学们，他们在我的学习和研究中提供了很多帮助和建议。特别是廖承凡同学，他在文档的理解交流过程中给予了我巨大的帮助。

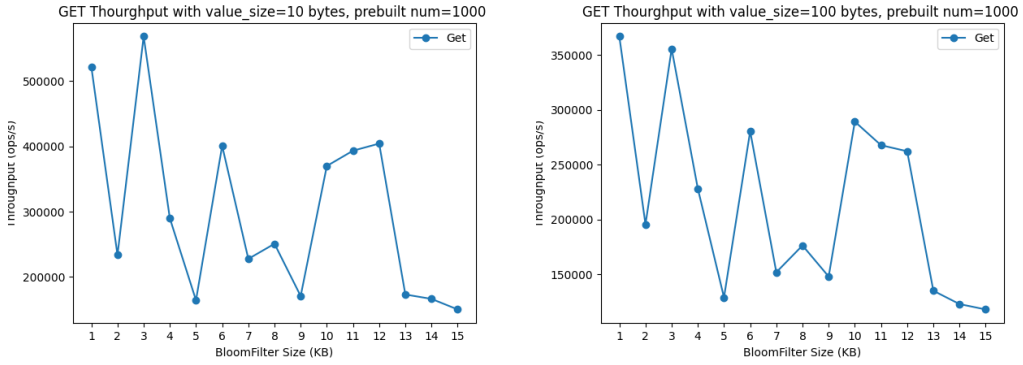
其次，我要感谢张迟学长的 mini-lsm 项目[1]，他对 lsm 的深入理解和剖析让我对如何改进这个项目有了许多新的想法。我还要感谢 google 的 gtest 项目[2]，在这个框架的支持下，项目的测试和维护变得轻松不少。

感谢知乎的文章[3] ，让我对 LSM 树的实现有了更加清晰的认识；感谢 GeeksforGeeks 的

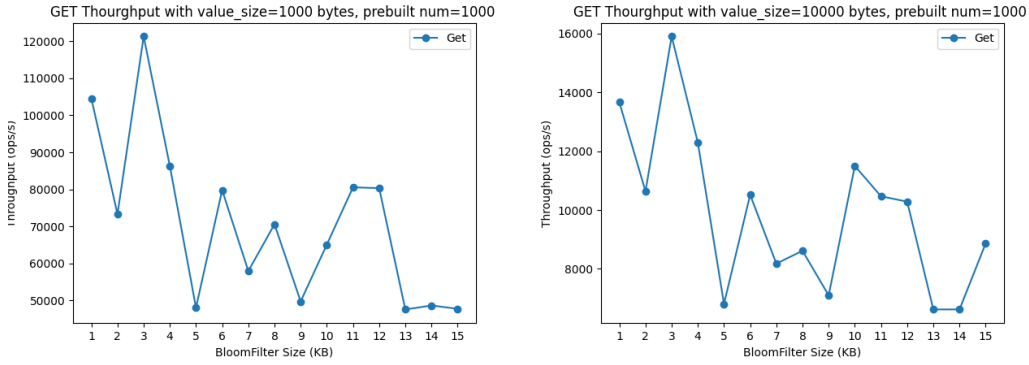
[1]https://github.com/skyzh/mini-lsm

[2]https://github.com/google/googletest

[3]https://zhuanlan.zhihu.com/p/415799237



(a)



(b)

图 4: 不同 BloomFilter 大小下的 GET

一系列算法博客，让我对布隆过滤器和跳表都有了完整和清晰的认识；感谢 Effective modern cpp、cppcon 和 cppreference，在这些书籍、视频和网站的帮助下，我了解了诸多项目中需要用的现代 c++ 的语法知识。

5 其他和建议

1. 困难与挑战：主要在于对 modern cpp 语法的不了解和 c++ 文件操作带来的 debug 困难，这个 project 是一次很好的 TDD（测试驱动开发）的实践，在不断地写单元测试、跑单元测试和回归测试的过程之中，我对每个子模块和整体的项目的正确性和架构如何解耦产生了许多新的认知。如何对持久化操作测试是有挑战的。

2. bug: 印象最深的有两个：一个是传了 vector 的引用进一个函数，但在函数内部同时修改

vector 和使用 vector 的迭代器/引用——造成了重定位带来的迭代器失效问题；还有一个是在布隆过滤器的函数的实现里面捕获了一个悬垂指针。

3. 吐槽：最后我的测试代码写得比实现代码都多了，感觉可以再给多一些测试框架，这是其一；助教的测试用例用”sssssss...” 根本无法 debug，这是其二；这个试验报告要测试的东西也太多了，我 python 脚本和 c++ 测试文件又写了四五百行, 这是其三。