Project-LSM-KV 项目汇报报告

李昱翰 520021910279

2022 年 4 月 12 日

目录

1 背景介绍

本次的 project LSM-KV 主要是实现一种利用磁盘以及缓存共同作用来实现大量数据存储的数据结构。LSM-KV 结构与数据结构课程中所学习的 B 树 (B+ 树) 结构以及作用类似。LSM-KV 结构的主要组成部分为：Memtable，SSTable 以及缓存索引区。其中，MemTable 主要是起到了一个过渡的作用，是用于暂时存储数据的一种数据结构；缓存区主要适用于存储每个 SSTable 中所有的 key 以及对应 data 的偏移索引信息；SSTable 是本次 project 的核心结构，主要用于存储大量的数据。对于本次 project 而言，主要是实现了 PUT DELETE GET 以及 Compaction 等基本操作来实现基本的 LSM-KV 结构。

2 数据结构和算法概括

本次 project 使用的主要的数据结构为：

1. **MemTable**: 主要使用跳表来实现对于 MemTable 的插入、删除、查找以及遍历等基本操作。
2. **缓存区**: 主要使用自定义的 SSTable 类、Bloom Filter 类以及 KVStore 类来实现缓存区中对 SSTable 的时间戳、键值对数量及范围以及每个 data 对应的 key 以及位置偏移量等信息的存储。同时使用 vector 数据结构来实现每层文件对应的缓存区的存储，并使用一个 vector 来存储所有的层级信息。(类似于 hash 中的散列表法。)
3. **SSTable**: 主要是利用 ifstream 中提供的一些基本读写功能，按照 project 的要求，将每个 SST 缓存区中对应的信息以及数据按照给定的格式写入对应名称的 SSTable 文件中去。

使用的主要算法为:

1. **Compaction**: 使用多路归并排序算法对符合条件的文件中的 key 进行归并操作，并生成新的 SST 文件，之后生成对应的缓存区以对应层级位置的 SST 文件。
2. **SCAN 操作**: 仍然是通过多路归并的方式对所有层级的所有 SST 文件中的最终仍未被删除的 key 进行排序，之后按照递增的顺序将位于目标键值范围内的 keys 以及对应 data 生成的 pairs 放入目标 list 中。

3. **PUT 操作**: 主要是基于跳表的插入算法实现的。同时在每次插入之后检查是否到达 sst 文件大小阈值来决定是否将当前 MemTable 清空并生成对应的 SST 文件。

4. **GET 操作**: 首先是使用跳表的查找算法来在 MemTable 中进行查找，若不存在则去 SSTable 中查找。

5. **DEL 操作**:DEL 操作本质上就是 PUT 操作与 GET 操作的结合。

3 测试

3.1 性能测试

3.1.1 预期结果

1. **常规测试 1**: 对于本测试我的预期结果是: 随着单个 data 的数据量变大，在总的操组数不改变的情况下，涉及到 compaction 以及 SST 文件的读写操作的可能性更大，故应有随着 data 的数据量增长而时间逐渐变长的结果, 从结果来看是符合预期的; 而对于三个操作的具体时间而言，DEL 的操作由于要同时进行 PUT 以及 GET 操作，故其平均时延应为最长，这点在实验过程中也得到了验证，而 PUT 操作由于需要进行大小检查以及 sst 文件写入等操作的反复执行，可能会导致其平均时延应小于 GET 操作。

2. **常规测试 2**: 与常规测试 1 的分析类似。但与之相反的，随着数据量的增大，平均时延的增长，每个操作的平均吞吐量应该下降。

3. **常规试验 1 与 2 出现的偏差分析**: 通过之后的测试结果，可以知道数据结果总体上是符合预期的，但是当 data 的数据量变得较大时，性能却出现了很大的下滑，超出了我的预期。虽然对于磁盘的访问性能确实远远小于对于内存的直接访问性能，但是我个人认为出现较大的下滑还与电脑磁盘存储大小 (我的电脑的 C 盘容量有些不足) 有关。

4. **缓存存储与 Bloom Filter 使用效果测试**: 在本测试中，方式 a 由于要进行大量不可避免的直接的磁盘访问，故其时间性能应该最差并且应远远弱于其他两种，而对于方式 b 与方式 c，实现了 Bloom Filter 的版本 c 由于在一些情况下会利用 Bloom Filter 进行查找的简化操作，

故其性能理论上是应该优于没有实现 Bloom Filter 的方式 b。但是由于 SST 文件中数据存储的有序性，以及随着 sst 的序号增大单个 sst 文件存储键值对的数量减少，也确定了是否实现了 Bloom Filter 对时间性能不会产生过大的影响。

5. **效果测试偏差分析**: 通过之后的实现效果测试可以发现，整体的结果是符合之前的理论分析的，但是对于方式 b 与方式 c，仍然存在部分数据点相邻较近或出现顺序颠倒的情况，分析原因如下:

   (a) 可以通过 Bloom Filter 进行直接过滤的文件在所有文件中只占有了一个较小的比例，所以只在方式 b 的基础上额外实现了 Bloom Filter 并不会产生过大的影响，而只会在平均意义上提升一定的性能。甚至有可能在部分没有被直接过滤掉的文件中进行查找时由于进行了 Bloom Filter 的查找而花费了额外的时间而使得时间变长。

   (b) 对于每种实现方式的测试值进行了 5 次，测试次数较少也可能会导致产生一定的偏差。

6. **Compaction 对性能的影响的测试**:Compaction 功能是本次 project 的一个核心功能。其通过降磁盘中的文件进行分层存储，从而提高了空间的利用效率。但是由于 compaction 本质上还是对磁盘的操作，故在 sst 文件达到 compaction 的条件时，对应的当次的 put 的效率会明显下降，时间应该明显提升。亦即会对 PUT 的平均时延产生较大的影响。

7. **对比实验测试**:std::map 的实现是基于**红黑树**这种数据结构的。而红黑树本质上是一种特殊的二叉搜索树，故其 insert 操作应该具有 O(logN) 的时间复杂度。而对于跳表实现的 insert 操作，具有 O(log N) 的时间复杂度，但是还要考虑跳表可能存在的上下层之间的重复移动的消耗的时间，故对于两者插入性能而言，std::map 版本应该略由于跳表实现的版本。对于查找操作，跳表实现的版本有着平均 O(logN), 最坏情况下 O(N) 的时间复杂度，而 map 实现的版本平均时间复杂度为 O(logN), 故两者在查找操作时也应该有相近的时间性能。最后，对于 DEL 操作，由于在本次 project 中，DEL 操作实际上是 PUT 与 GET 操作的结合，故 std::map 的 DEL 操作的性能也应优于跳表版本。

8. **对比测试偏差分析**:

   (a) 对于 PUT 操作，跳表版本的性能并不是总弱于 map 版本的，这可能是由于每次测试时相同数据随机出来的最大层数并不完全相同，从而产生偏差，另外，还可以发现跳表版本的 PUT 操作的平均时延曲线分布不很均匀，这可能也是由于随机高度发生变化导致的，同时实验的次数较少也可能会导致此类偏差。

   (b) 对于 GET 操作，可以发现，大部分数据点是符合理论分析的，但是仍有部分数据点产生了些许偏差，这可能是由于不同实验随机的产生最大高度不同以及较少次数的实验中产生的偶然误差导致的。

3.1.2 常规分析

1. **常规测试 1**: 对 PUT GET DEL 基本操作的平均时延进行测试：

   在本测试中，主要对 data 大小为 **256 bytes,1024 bytes,10240 bytes,20480 bytes** 的数据分别进行了 2000 次连续的 PUT,GET,DEL 操作, 其测试结果如下图 1。

数据量：2000个				数据量：2000个			
数据大小：256bytes	PUT	GET	DEL	数据大小：1024bytes	PUT	GET	DEL
TotalTime(s)	0.6173	0.4344	0.8769	TotalTime(s)	0.7857	0.5791	1.1213
平均时延	0.0003087	0.000172	0.0004385	平均时延	0.0003929	0.0002896	0.0005607
数据量：2000个				数据量：2000个			
数据大小：10240bytes	PUT	GET	DEL	数据大小：20480bytes	PUT	GET	DEL
TotalTime(s)	13.4025	8.7909	12.2477	TotalTime(s)	21.8345	10.7711	13.4961
平均时延	0.006701	0.004395	0.006124	平均时延	0.01092	0.005386	0.006748

图 1: 结果图-1

之后，在上述结果的基础上，计算不同 data 长度下的**总体平均时延**，得到如下图 2所示结果。

2. **常规测试 2**: 对 PUT GET DEL 基本操作的吞吐量进行测试:

   在本测试中, 使用与常规测试 1 相同的测试集,**(256 bytes/1024 bytes/10240 bytes/20480 bytes)**, 同时数据操作次数仍然为 2000 次，且 key 的

(数据量：2000个)	PUT	GET	DEL
总体平均时延	0.004581	0.002561	0.003468

图 2: 常规测试 1 最终结果

取值为从 0 开始的连续整数值。经过测试，可以得到如下图 3所示的结果。

key:0--1999(连续)				key:0--1999(连续)			
数据大小:256 bytes	PUT	GET	DEL	数据大小:1024 bytes	PUT	GET	DEL
TotalTime(s)	0.6173	0.4344	0.8769	TotalTime(s)	0.7857	0.5791	1.1213
吞吐量	3239.92	4604.05	2280.76	吞吐量	2545.51	3453.63	1648.81
key:0--1999(连续)				key:0--1999(连续)			
数据大小:10240 bytes	PUT	GET	DEL	数据大小:20480 bytes	PUT	GET	DEL
TotalTime(s)	13.4025	8.7909	12.2477	TotalTime(s)	21.8345	10.7711	13.4961
吞吐量	149.23	227.51	163.29	吞吐量	91.59	185.68	148.19

图 3: 结果图-2

之后，根据上述结果，可以计算得到不同 data 长度下的**总体平均吞吐量**, 结果如下图 4所示。

(数据量：2000个)	PUT	GET	DEL
总体平均吞吐量	1506.56	2117.72	1060.26

图 4: 常规测试 2 最终结果

3.1.3 索引缓存与 Bloom Filter 的效果测试

本测试主要是通过对不同的 get 的实现方式的性能的比较，来判断使用缓存存储以及使用 Bloom Filter 对 KV 系统查找性能的影响。本测试我是首先在 KV 系统中插入了 4000 个键值对，并且每个 data 的大小均为 10240 bytes，之后利用不同的 get 方式进行多次测量并取得平均值，最为最终结果。本测试具体结果表格如下图 5所示，利用 Origin 绘制的折线图如

下图 6所示。

注: 在图中的方式 a,b,c 分别对应如下几种 get 的实现策略。

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. 满足该 Project 所有要求的 get 实现方式 (即同时使用了缓存存储以及 Bloom Filter 的方式)。

测试次数	1	2	3	4	5
get方式 a)	45.9951	45.7648	44.7045	44.8046	44.1458
get方式 b)	16.6611	16.6589	15.5718	16.2222	15.8967
get方式 c)	15.7044	14.9683	14.1724	15.2959	14.3986

图 5: get 不同实现方式测试结果表

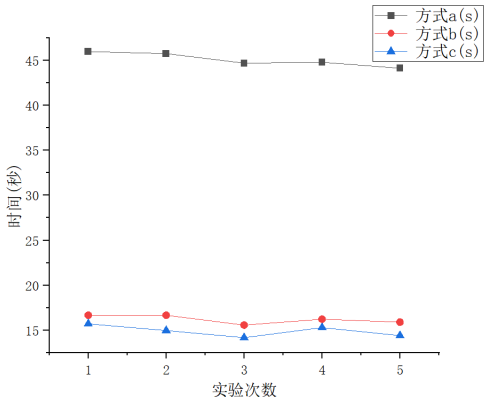

图 6: get 不同实现方式测试结果图

3.1.4 Compaction 的影响

本测试要求体现 compaction 操作对于 PUT 的平均时延 (吞吐量) 的影响。我采用的策略是，首先在 KV 系统中插入 1000 个键值对，每个键值对

对应的 data 大小为 **1024 * 512 bytes，从而尽可能的提高 compaction 发生的频率**。之后，在每次 PUT 操作结束之后，输出当前的总的运行时间，**同时在发生了 compaction 的部分输出对应的 key 取值来更好地区分 compaction 的发生**。之后**计算每次 PUT 的用时作为当次 PUT 的时延**，最后取出记录的数据中的 60 组数据绘制成折线图，如下图 7所示。

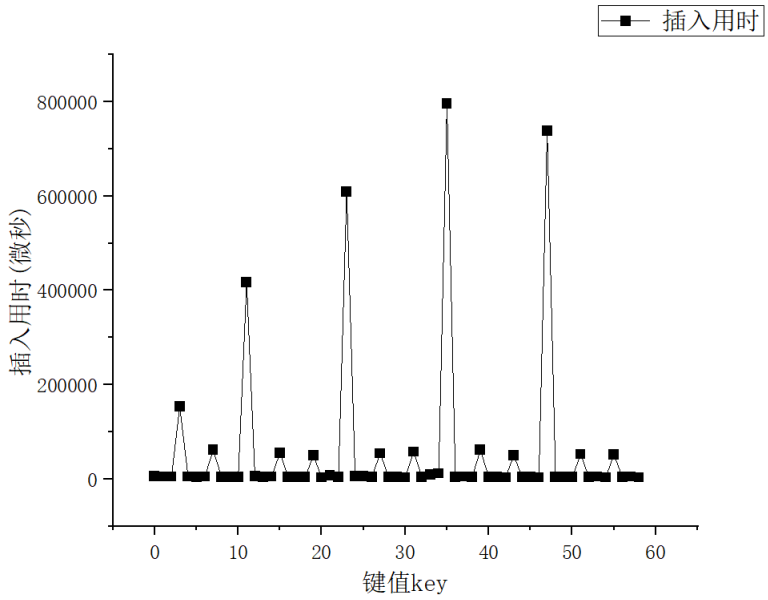

图 7: Compaction 对性能影响的测试结果折线图

3.1.5 对比实验

本测试要求在利用 std::map 来实现另外版本的 MemTable，并与原始的跳表版本进行性能比较，并分析适用场合。在本次测试中，我首先是**在 KV 系统中插入了 3000 个键值对，key 的取值为 0-2999 的连续整数，每个 data 的大小为 1024*200 bytes。之后依次进行查找以及删除操作，并分别记录各个操作所用的时间，并对每种实现方式测试了 5 次，最终分别计算每个操作的平均时延的平均值**。最终测试得出的结果表格如下图 8所示。根据数据绘制的折线图如下图 9所示。

平均时延(秒/次)	跳表实现			std::map实现		
对比实验	PUT	GET	DEL	PUT	GET	DEL
1	0.07393	0.01274	0.02039	0.07294	0.01677	0.01829
2	0.07246	0.01272	0.01929	0.06607	0.01391	0.01506
3	0.07004	0.01285	0.01916	0.06961	0.01317	0.01432
4	0.08753	0.01238	0.02011	0.06457	0.01264	0.01424
5	0.07549	0.01261	0.01857	0.06499	0.01274	0.01412
平均时延-平均值	**0.07589**	**0.01266**	**0.019504**	**0.067636**	**0.013846**	**0.015206**

图 8: 对比测试结果表

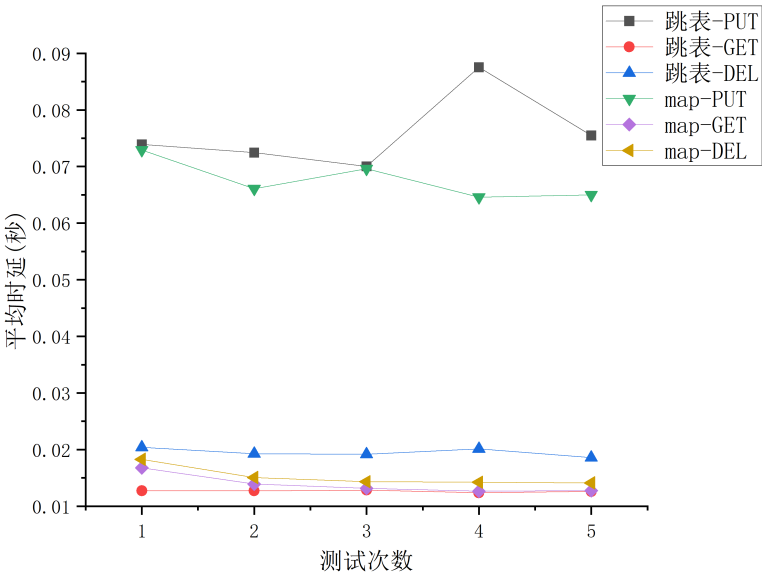

图 9: 对比测试结果折线图

4 结论

1. **常规测试 1**: 从上述常规测试 1 结果可以得出，DEL 平均时延最长，PUT 次之，GET 的平均时延最短，这与之前的理论分析相吻合，而 DEL 操作的平均时延明显小于 PUT+GET，这可能是由于 DEL 操作只写入了" DELETED " 字符串，长度较小，因而可能使得平均写入时间在数据量较大时明显小于 PUT 的时间，故可能导致其平均时延较

小。

2. **常规测试 2**: 从上述常规测试 2 结果可以得出，DEL 吞吐量最小，PUT 次之，GET 最大。PUT GET DEL 的平均吞吐量都是符合之前的理论分析的。

3. **常规测试小结**: 通过上述两个测试，我完成了对于 LSM-KV 程序的基本操作程序的性能测试。虽然结果趋势较为符合预期，但是从另外一方面来看，当单个 data 的数据量出现明显变化时 (如从 1024 bytes 变为 10240/20480 bytes)，三个基本操作的平均吞吐量以平均运行时延均出现了明显的下降/上升，**这同时展示了当程序对于磁盘的访问量增加时对于程序运行的时间性能的明显影响**。

4. **缓存存储以及 Bloom Filter 过滤效果测试**: 通过上述效果测试中得出的折线图可以看出，方式 a,b,c 整体而言曲线趋势相对平稳，其中，方式 a 由于大量的磁盘访问而导致性能最差，并且远远低于方式 b 与方式 c; 而方式 b 与方式 c 两种实现方式之间的时间性能差距较小，总体而言平均有约 1 秒的性能时间差。通过本实验可以发现，**利用缓存存储而不是直接大量访问磁盘信息会极为明显的改善时间性能，同时添加 Bloom Filter 也会在一定程度上起到时间性能方面的优化作用。**

5. **Compaction 对 PUT 操作性能影响测试**: 通过性能影响测试的折线图可以知道，在没有发生 compaction 的 PUT 操作中，其每次操作的时延较为平均，均为 1000 微秒左右；而发生了 compaction 的 PUT 操作的平均时延明显变长，最大达到了 8000 微秒。通过这个数据可以得出结论：**Compaction 操作由于进行了对待合并文件的重复磁盘读写以及重新创建新文件等操作，使得时间性能明显下降，单次操作平均时延明显提高。**但是，compaction 操作虽然会使得时间性能有所下滑，但是其通过对磁盘文件进行分层以及合并，也使得 KV 系统对磁盘空间的利用效率提高。

6. **对比试验测试**: 通过上述对比测试的最终平均值结果以及折线图可以看出，map 实现版本的 DEL 性能明显优于跳表版本的 DEL 性能；而 map 版本与跳表版本的 PT 操作的性能存在一定的浮动，但是从平均值角度来看 map 版本的 PUT 操作性能要优于跳表版本的；对于 GET 操作，两种实现方式的结果仍然存在一定的浮动，但从平均值的角度

来看，应该为跳表版本的 GET 操作平均性能要优于 map 版本的。由此可以得出，**对于要进行大量连续数据的插入以及删除的场合，使用 map 的性能要较优于跳表；而需要进行大量数据的查找的场合，使用跳表来实现要好于 map 实现**。

5 致谢

在此我要向在本次 project 完成过程中向我提供帮助的所有个人、集体以及博客表示由衷的感谢:

1. 首先向向我提供了极大帮助的曾同学表示感谢，在他的帮助下，我成功解决了内存泄漏问题；并在他的协助下对原始代码版本的磁盘读写操作进行了优化，很好地提升了程序的运行性能；同时他也帮助我认识到了自己在代码实现方面的种种不足，希望之后可以做出改进。
2. 还要向向我提供了思路的知乎专栏–《码洞》表示感谢，通过阅读此专栏的文章，我了解到了磁盘多路归并的基本原理并将此原理运用到了本次 project 的核心功能–Compaction 操作的实现中去。
3. 也要向 CSDN 博客表示感谢，通过阅读其中的文章，我很好的解决了在进行二进制文件的 read 以及 write 操作时遇到的问题以及在使用 std::map 进行一些基本操作时遇到的问题。

6 其他和建议

遇到的 BUG:

1. Bad Alloc 问题，最终发现是由于过多临时的动态指针在使用之后没有及时释放导致的。
2. Compaction 的文件归并之后结果出错的问题，最终发现是由于在计数当前跳表的大小时，没有在清空跳表之后正确的重新归零。
3. read/write 的问题。尤其是在进行字符串相关读写时，会出现无法正确读出结果的问题。
4. std::map 的使用问题。主要是 std::map 如何遍历，以及如何排序的问题。(最终发现 map 可以自动按照 key 进行升序排序).