LSM Tree 实验报告

周嘉豪 519021911217

5 月 25 日 2021 年

1 背景介绍

LSM(Log-Structured Merge) Tree 是一种可以高性能执行大量写操作的数据结构。它最初于 1996 年被 O'Neil 等人的一篇论文中提出，如今已广泛用于数据储存中。Google 的 LevelDB 和 Facebook 的 RocksDB 都以 LSM Tree 为核心数据结构。LSM Tree 用一种阶梯式数据储存和检索的模式来实现高性能的大批量写操作。其在内存中维护一个高效的缓存区，当缓存区填满时才写入内存，这使得我们能快速操作 hot data，提升了读写性能。其次，LSM Tree 在内存中缓存了索引和 Bloom Filter, 这极大的提升了读取磁盘数据的速度。LSM Tree 的每一层在逻辑上都是相互分离的，便于我们在此结构上做出更多的优化。

2 挑战

在本次 LSM Tree Project 的完成中，最有挑战的部分是 compaction 的实现，在仔细理清思路以后我顺利的完成了这一部分。但是有一个 bug 困扰了我整整一天，最后在董明凯学长的帮助下得以顺利解决。它具体是：我复用了一个 fstream 的变量，但是我并不清楚在 close 这个变量以后他标记 EOF 等的状态位都不会被 reset，所以再次 open 的时候会导致很奇怪的行为，明明 SSTable 文件存在，并且 key 也是合法的，但是却读不到东西。最后我在 close 以后再 clear 了一下才得以解决这个 bug。

还有一个也是关于文件读写的 bug，在 windows 下如果不用 *ios :: binary* 的方式打开文件的，会导致写入.sst 文件的时候写入实际结果与预期结果不符。

值得一说的是我这次在编译环境上也碰到了一个较为棘手的问题。在 windows + minGW 环境下，CLion 在 debug 的时候不支持 STL 库的显示。这让我在 debug 的时候非常难受，最后换成 windows + WSL 就好用多了，不过在这种环境下程序运行的速度没有 minGW 快。

最后，由于这个 Project 是由 memTable 和 SSTable 两个块组成的，我在写的时候不断在低耦合度和更高性能之间摇摆，导致写得有些痛苦。但我后面还是坚定了想法，听从 Donald Knuth 的名言 *premature optimization is the root of all evil*，先写了一个低耦合的版本出来通过了测试 (当然时间性能很差)，再用 valgrind 进行了性能瓶颈分析，做了一些优化以后得到了较为满意的时间性能。

3 测试

此部分主要阐述 LSM Tree 的性能测试。

3.1 性能测试

3.1.1 预期结果

1. **常规分析:** Get、Put、Delete 操作的延迟应该随着数据量的增加而增大，吞吐量应该随着数据量的增大而减少。这是由于 SSTable 的增多以及 compaction 的频率变高而导致的。Get、Delete 操作的吞吐量应较为接近，且 Delete 略小于 Get。因为每次 Delete 时都需调用 Get，然后往 memTable 里面插入删除记录，而往 memTable 里插入删除记录需要的开销较小，故 Delete 吞吐量应该与 Get 接近。Get、Delete 操作的吞吐量随着数据量的增大下降应该较为明显，因为每次 GET 或 DELETE 都要遍历所有 SSTable。而数据量的增大对 PUT 操作的吞吐量影响应该较小，因为每次 PUT 都只需要往 memTable 插入，数据量的增大只会增大 compaction 的开销，所以总体来说数据量对 PUT 操作的影响应该远小于 GET 与 PUT。

2. **索引缓存与 Bloom Filter 效果测试:** 在这种情况下延迟应该是
$$Index + BloomFilter > Index > None$$
因为 Bloom Filter 可以帮我们迅速过滤不存在于 SSTable 中的 key，

而 SSTable 中缓存的索引信息能让我们通过二分法来较为快速得到对应的 offset，而不用去内存中一个一个读取。

3. **Compaction 的影响:** 从纵向比较的角度来看，随着时间的增长，磁盘中 SSTable 数量和层数的增多，这导致每次插入时 compaction 所花费的代价越来越大，故吞吐量总体随时间增长应该呈下降趋势。

   从横向比较的角度来看，随着每一次 PUT 操作插入值大小的增长，吞吐量应该呈下降的趋势。这是因为插入值越大，memTable 转成 SSTable 存入磁盘中的频率就越高，compaction 的频率也会越高，故吞吐量降低了。

3.1.2 常规分析

在这一部分我们用不同的数据量测试 Get、Put、Delete 操作的吞吐量和延迟。

1. Get、Put、Delete 操作的吞吐量如下表所示:

吞吐量 数据量 / 操作	1024 * 16	1024 * 32	1024 * 64
GET	14855	7149	3613
PUT	9966	7407	6862
DELETE	13567	7045	3536

上述数据是多次测量的平均值。

2. Get、Put、Delete 操作的延迟如下表所示:

延迟 ($\times 10^{-5}$) 数据量 / 操作	1024 * 16	1024 * 32	1024 * 64
GET	6.73	14.00	27.68
PUT	10.06	13.73	14.57
DELETE	7.38	14.19	28.28

上述数据是多次测量的平均值。

这一部分的测试结果与我们的预期符合得较好。

3.1.3 索引缓存与 Bloom Filter 的效果测试

这一部分我们分三种情况来对比 Get 操作得平均时延，通过对比体现索引缓存和 Bloom Filter 的效果。

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。

延迟 ($\times 10^{-5}$) 数据量 / case	1024 * 16	1024 * 32	1024 * 64
None	19.85	40.24	78.97
Index	1.62	3.20	6.15
Index + Bloom Filter	1.54	2.98	5.67

上述数据是多次测量的平均值。

我们可以看到只有 Index 与 Index + Bloom Filter 的情况延迟相差其实不是特别大，这是因为我在插入数据时是按顺序插入的，再加上每层的每个 SSTable 之间（除去第 0 层）key 的范围都是不交叉的，所以我们可以直接通过 *minKey* 和 *maxKey* 来判断大部分 SSTable 中是否可能含有这个 key，而只有少数 SSTable 中 key 的范围正好覆盖了我们要找的 key，这时候如果我们要找的 key 不存在于此 SSTable 中，有 Bloom Filter 的情况就有可能可以直接判定它不存在，继续找下一个 SSTable。换句话来说，在我们这个测试用例下，Bloom Filter 能够起作用的情况是较为严苛的，故这两种情况的延迟相差不是特别大。

而 None 与剩余两种情况延迟差距明显，与我们的预期符合的较好。

3.1.4 Compaction 的影响

连续往 LSM Tree 中插入数据，统计每秒钟处理的 PUT 吞吐量，绘制如下折线图。图例中的数字代表一次 PUT 插入值的大小。

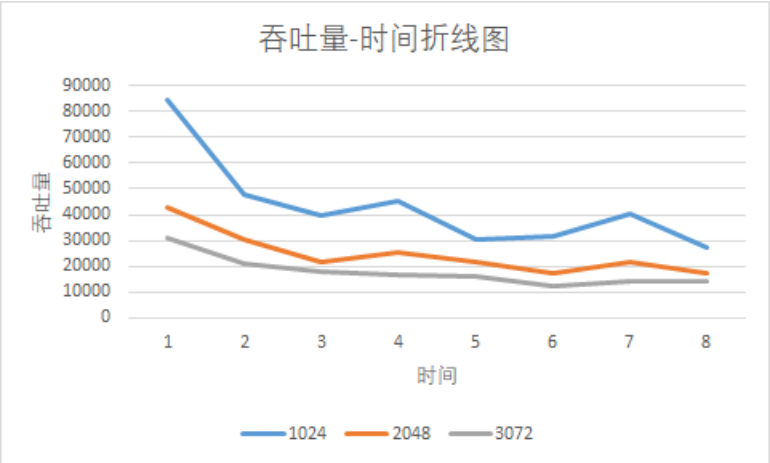

图 1: 吞吐量-时间折线图

纵向角度与我们的预测有一些出入，我们可以观察到局部有稍许上升的趋势，这可能时是因为之前的 compaction 将 SSTable 合并到了较深的层，导致这一段时间的 PUT 操作的 compaction 都在较浅的层中进行，引发 compaction 的频率降低了。

横向角度与我们的预测较为符合。

为了更加直观的观察 compaction 的影响，下面给出连续插入 10000 个数据得出的延迟折线图。

图 2: 延迟-插入次数折线图

我们可以观察到图中有若干个峰值，这些峰值即代表了此次操作发生了compaction，可以看到发生了 compaction 的 PUT 操作花费的时间大大增加了。

4 结论

4.1 收获

在做 LSM Tree Project 的过程中，我最大的收获是初步体会到了用于大批量读写操作的数据结构所具备的一些特征，例如不仅要保证正确性，也要保证持久性，要权衡时间性能与空间性能，协调查找与更新的吞吐量等等。我也对数据结构的分层设计有了更加深刻的理解，进一步体会了低耦合度和不要过早做出优化的重要性。再者，我 debug 的能力也有所增强，学会了慢慢缩小 bug 范围，打条件断点，分块测试等技能。

4.2 建议

1. 为了更好的体会分层设计的优势，可以设置一些优化的任务。例如优化跳表的实现，扩展 memTable 的个数等等。

2. 在这个 Project 中，其实如何写测试程序也是很重要的，可以有意的介绍如何写出一个测试用例合理并且对程序员友好的测试程序。

3. 还有其实 TA 给的测试用例测试不是很全面，比如在持久性测试里面，只测了 get 操作，并没有测 delete 和 put 操作。还有一个是在持久性测试里面好像也没有测试一些错误的情况，如缺失掉某一层、在 restore 过程中碰到有一层由于上一次意外终止而导致 compaction 未完成等特殊情况。

4.3 运行截图

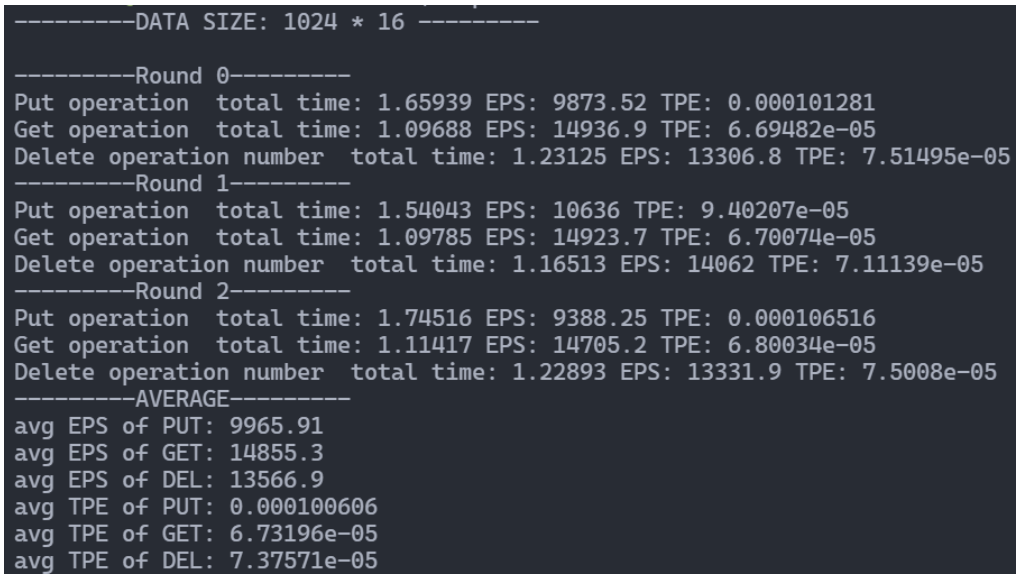

图 3: 数据量为 1024×16

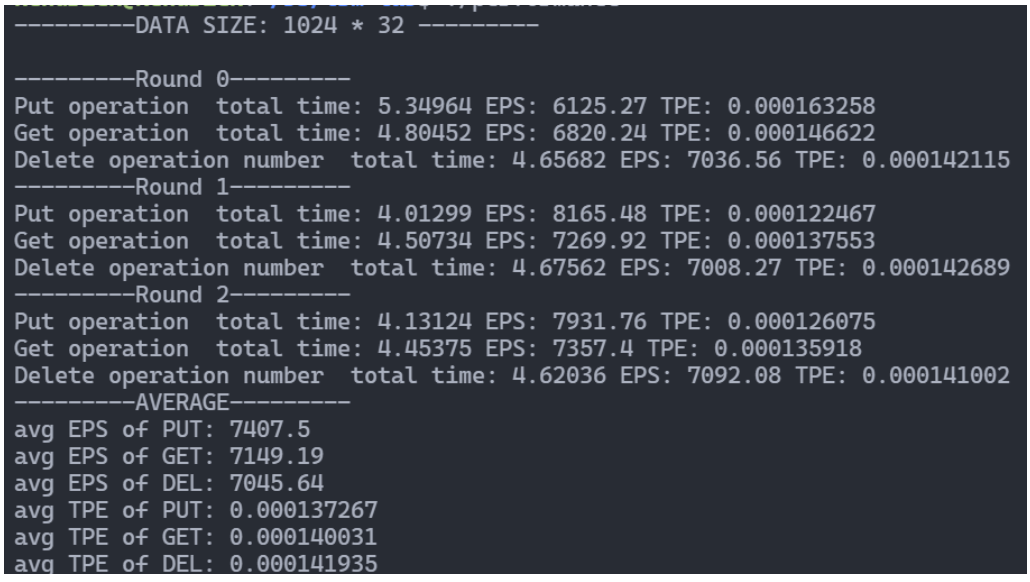

图 4: 数据量为 1024×32

```
---------DATA SIZE: 1024 * 64 ---------

---------Round 0---------
Put operation  total time: 9.61565 EPS: 6815.56 TPE: 0.000146723
Get operation  total time: 17.9478 EPS: 3651.47 TPE: 0.000273862
Delete operation number  total time: 18.4943 EPS: 3543.58 TPE: 0.000282201
---------Round 1---------
Put operation  total time: 9.52158 EPS: 6882.89 TPE: 0.000145288
Get operation  total time: 18.2211 EPS: 3596.71 TPE: 0.000278032
Delete operation number  total time: 18.5513 EPS: 3532.69 TPE: 0.00028307
---------Round 2---------
Put operation  total time: 9.51336 EPS: 6888.84 TPE: 0.000145162
Get operation  total time: 18.255 EPS: 3590.04 TPE: 0.000278549
Delete operation number  total time: 18.5524 EPS: 3532.49 TPE: 0.000283087
---------AVERAGE---------
avg EPS of PUT: 6862.43
avg EPS of GET: 3612.74
avg EPS of DEL: 3536.25
avg TPE of PUT: 0.000145724
avg TPE of GET: 0.000276814
avg TPE of DEL: 0.000282786
```

图 5: 数据量为 1024×64

```
---------DATA SIZE: 1024 * 16 ---------

---------Round 0---------
Get operation  total time: 3.23969 TPE: 0.000197735
---------Round 1---------
Get operation  total time: 3.27932 TPE: 0.000200154
---------Round 2---------
Get operation  total time: 3.27748 TPE: 0.000200042
---------Round 3---------
Get operation  total time: 3.27693 TPE: 0.000200008
---------Round 4---------
Get operation  total time: 3.35147 TPE: 0.000204557
---------Round 5---------
Get operation  total time: 3.22525 TPE: 0.000196854
---------Round 6---------
Get operation  total time: 3.22268 TPE: 0.000196697
---------Round 7---------
Get operation  total time: 3.19622 TPE: 0.000195082
---------Round 8---------
Get operation  total time: 3.2452 TPE: 0.000198071
---------Round 9---------
Get operation  total time: 3.20597 TPE: 0.000195677
avg TPE of GET: 0.000198488
```

图 6: None, 数据量为 1024×16

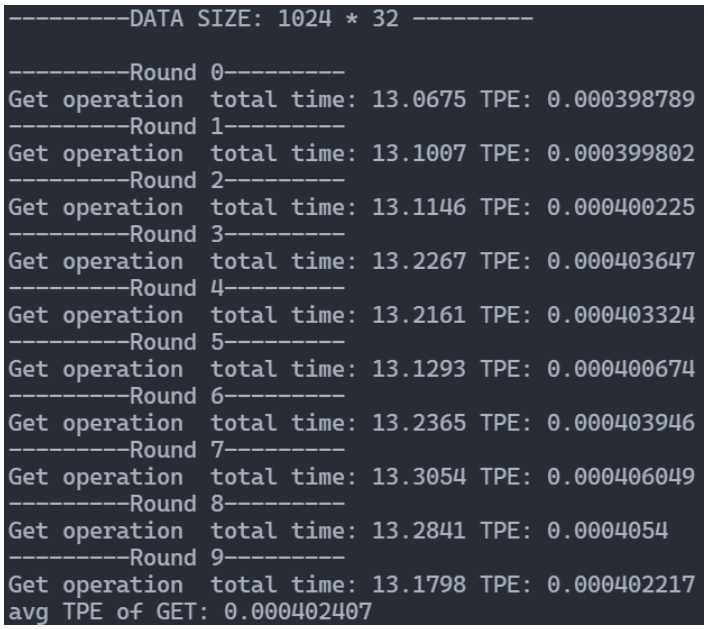

图 7: None, 数据量为 1024×32，

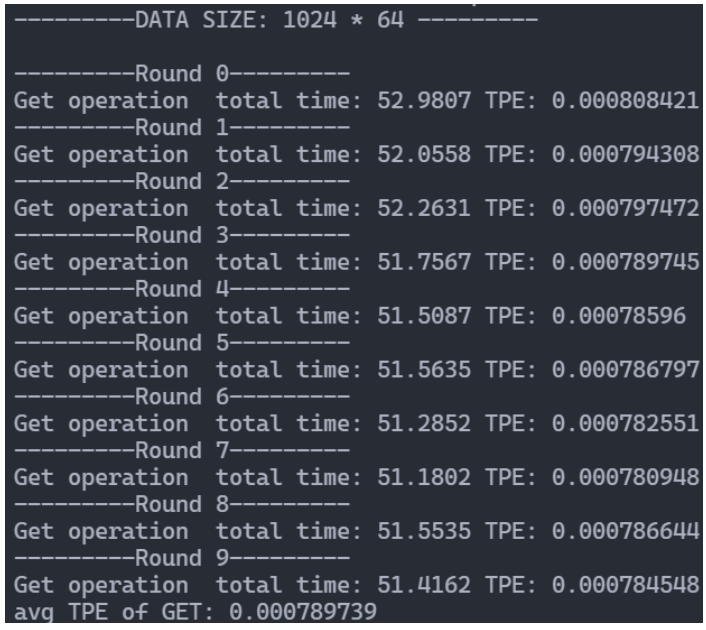

图 8: None, 数据量为 1024×64

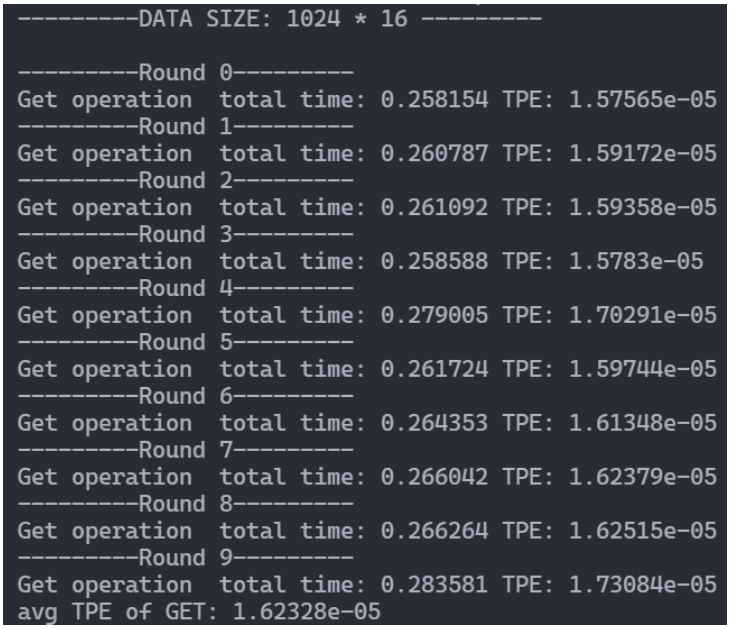

图 9: Index, 数据量为 1024×16

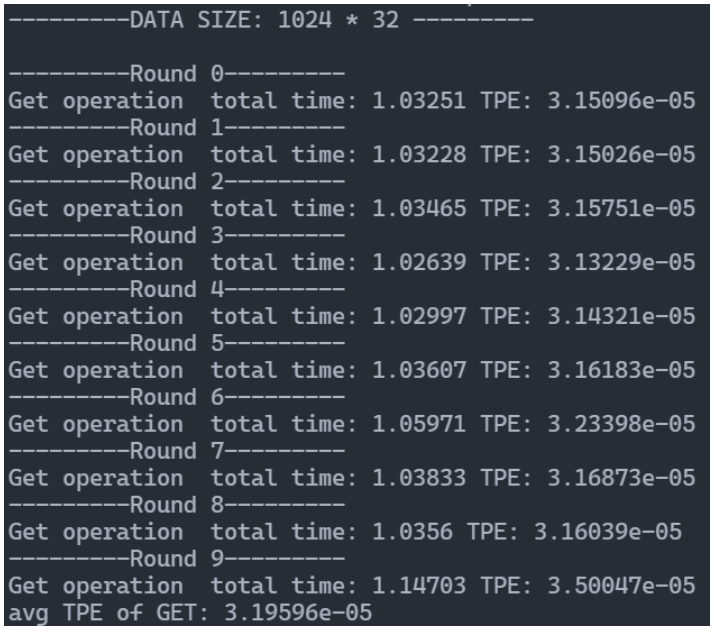

图 10: Index, 数据量为 1024×32，

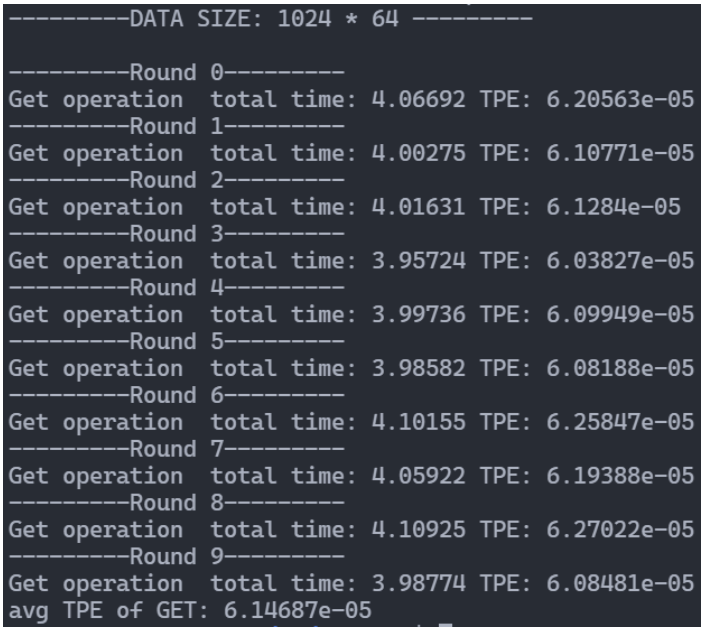

图 11: Index, 数据量为 1024×64

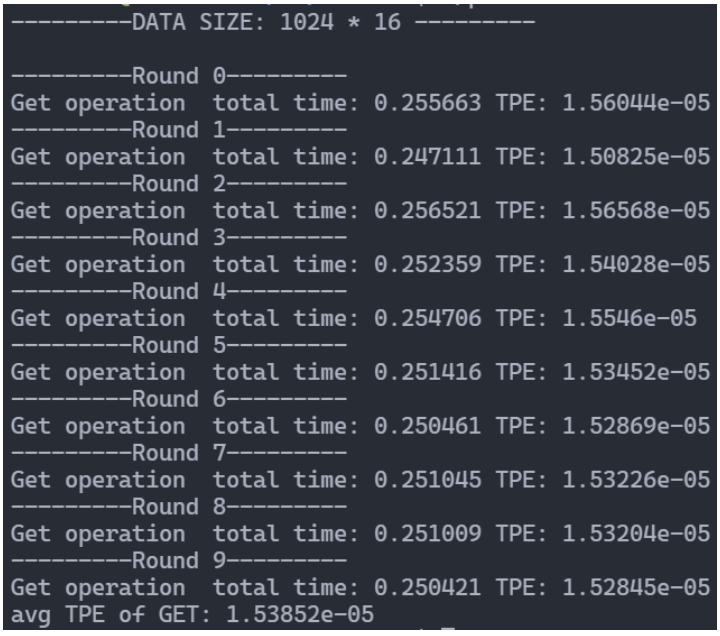

图 12: Index + Bloom Filter, 数据量为 1024×16

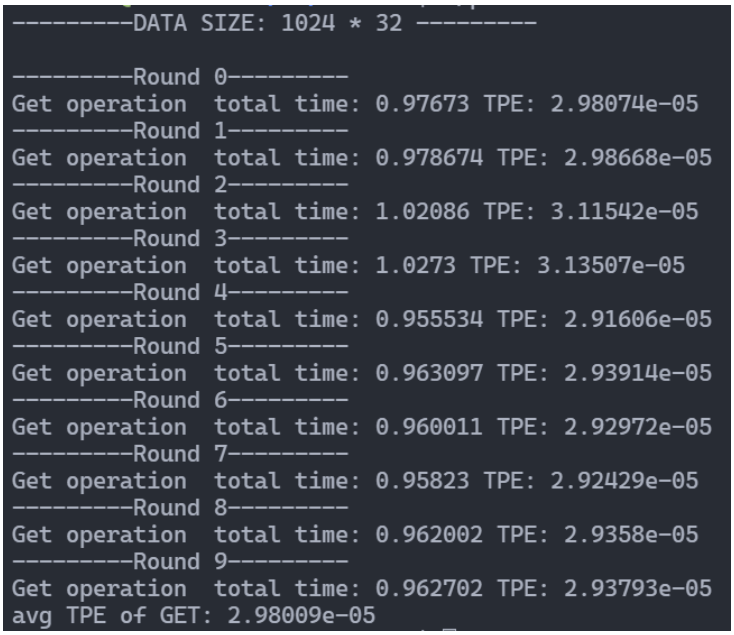

图 13: Index + Bloom Filter, 数据量为 1024×32，

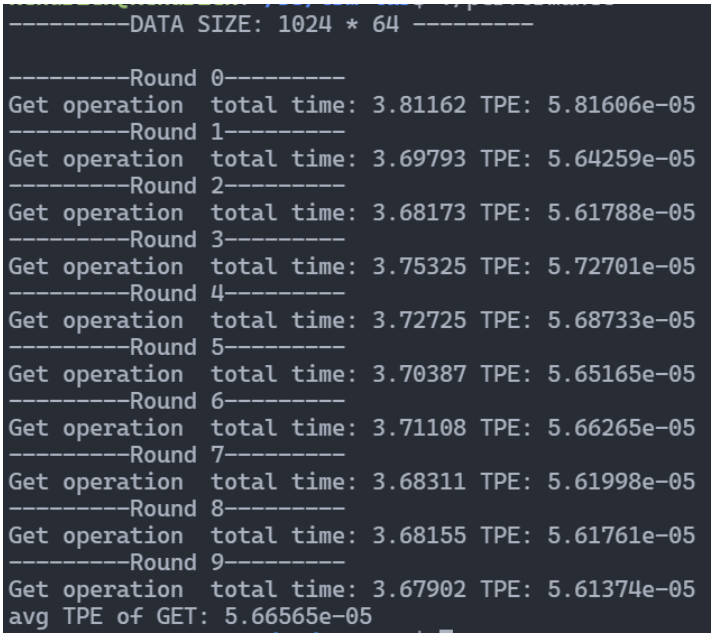

图 14: Index + Bloom Filter 数据量为 1024×64

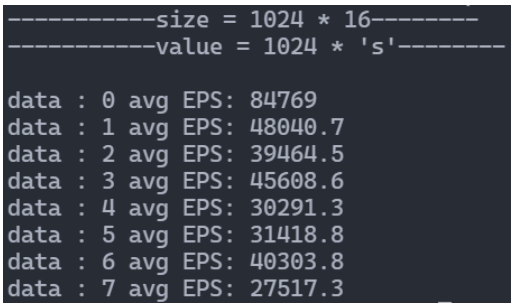

图 15: 值大小为 1024

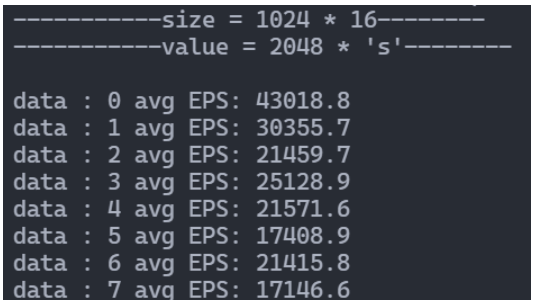

图 16: 值大小为 2048

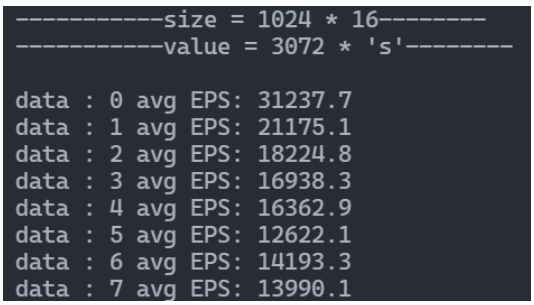

图 17: 值大小为 3072