LSM-KV 项目报告

卢天宇 519370910127

2022 年 5 月 18 日

1 背景介绍

LSM Tree (Log-structured Merge Tree) 是一种可以高性能执行大量写操作的数据结构。它于 1996 年，在 Patrick O'Neil 等人的一篇论文中被提出。现在，这种数据结构已经广泛应用于数据存储中。Google 的 LevelDB 和 Facebook 的 RocksDB 都以 LSM Tree 为核心数据结构。

2 数据结构和算法概括

在实现本项目的过程中，我们大量使用了 C++ 标准模板类 STL 中的数据结构，例如 std::vector,std::list,std::pair 等等。根据他们的特性，我们在不同的情境下使用二者。例如，对于 SSTable 的头部缓存，我们利用

$$\text{std::vector}\langle\text{std::list}\langle\text{Buffer}\rangle\rangle$$

的数据结构来进行存储。这是因为，缓存的层级一般只增不减，而且只会在末端插入或删除一层，所以为了达到最大的随机访问效率，我们在最外层使用 vector。而对于每一层中的若干 SSTable 的缓存 Buffer，我们使用 list 进行存储，这是因为我们默认把一层中的 Buffer 按照时间戳排序，然后，在新写入 Buffer 的时候，是在末尾写入的，而删除本层溢出的 Buffer 时，则需要在头部删除，这种同时需要对两端进行操作的情况，使用以链表为实现基础的 list 更合适。

3 测试

3.1 性能测试

在本节中，对 LSM-KV 系统进行性能测试，分为两个实验，一个测试顺序操作，一个模拟实际情况的随机操作。

对于第一个实验，具体测试方法如下。

首先，初始化系统，内外存中的数据以及缓存数据均为空。然后顺序执行如下三个操作：

1. 顺序插入 n 个键值对，索引值从 0 至 n-1 递增，第 i 个索引的数据为长度为 i+1 的字符串。
2. 顺序查找 n 个索引值，索引值从 0 至 n-1 递增。
3. 顺序删除 n 个键值对，索引值从 0 至 n-1 递增。

进行三组平行实验，每组实验的 n 值分别设置为 5,120、10,240、20,480。

对于第二个实验，具体测试方法如下。

1. 在 min_key 和 max_key 中随机插入 n 个键值对，数据的长度在 min_length 与 max_length 之间随机取值。
2. 在 min_key 和 max_key 中随机搜索 n 个索引值。
3. 在 min_key 和 max_key 中随机删除 n 个键值对。

进行三组平行实验，每组实验的 n 值分别设置为 5,120、10,240/20,480。min_key 均设置为,0，max_key 设置为 n 的 2 倍。min_length 均设置为 1，max_length 设置为 n 的 2 倍。

3.1.1 预期结果

实验一

对于存储空间，一方面，在进行插入操作的时候，实际插入的数据量是 $O(n^2)$ 的，所以被分成的 sst 文件也是 $O(n^2)$ 的。另一方面，当 sst 存储的最深层数是 k 时，由于每层能容纳的 sst 文件是线性于当前层序数的，于是总共能容纳的 sst 文件数量是 $O(k^2)$ 的。于是，最深层数是 $O(n)$ 的。

对于执行时间。

1. 顺序插入数据。按上述实验设计进行合并操作时，并不会有索引值交错，或多相同索引值合并的可能。当第 0 层写入第 3 个文件时，会触发合并操作，将 3 个文件一起写入第一层。若第一层原本是满的，则同时也需要将时间戳较小的 3 个文件删除并写入下层。以此类推，对于每一个已经达到文件数量上限的层级，都需要进行 3 次写入和删除操作。而触发的总合并次数是正比于 sst 文件数量的，于是总合并次数是 $O(n^2)$ 的，一次合并的文件读写次数是 $O(\text{最深层数}) = O(n)$ 的，所以总读写次数是 $O(n^3)$ 的，于是相应的总时间也应与 n 的值呈现 3 次方的增长规律。

2. 顺序查询数据。在第一步的基础上，索引值越小的数据，存储在外存中的层级越深，需要的时间也越多。由于我们的查询保证都能查到有效的索引值，所以每次查询都会有外存读取。具体而言，单次查询的时间复杂度，与所处的层级深度是 $O(n)$ 的，而查询次数是 $O(n)$ 的，故总查询时间应该是 $O(n^2)$ 的。

3. 顺序删除数据。LSM 的删除功能是在 GET 和 PUT 已经实现的基础上实现的。先使用 GET 尝试查找欲删除的索引是否存在，如果存在，则插入 ~DELETED~ 标记。在本实验设计中，GET 操作总是会返回成功，于是 PUT 操作总会得到执行。但是，插入的新数据在合并至最后一层时都会被抹除，于是数据总层数基本不会继续增长。另一方面，删除操作插入的字符串是固定长度的，长度很短，需要累积很多次删除操作才会触发一次写外存，而触发合并操作则更少。综上，总操作时间，应为 $O(n^2)$。

实验二

与实验一相仿，仅仅将数据的产生变成了随机的方式。产生重复索引值的概率较小，而大多数情况下，产生的索引值和数据长度都是不同的。则应具有与实验一相似的时间复杂度。

在不同的的 n 值下，我们用 n 的值除以平均时延，可以得到单位时间内，系统能够进行的某种操作的数量，我们将测试结果作为对数据结构吞吐量的估计值。

3.1.2 常规分析

1. 对于三种操作的时延，我们使用实验一进行说明，测试结果如表 1（时间单位：秒）。

表 1: 在不同数据量的情况下，三种操作的时延变化

n	PUT	GET	DEL
5,120	0.398	0.366	0.422
10,240	1.566	1.029	1.038
20,480	9.962	3.552	4.951

观察 PUT 操作的时间复杂度，发现其随 n 的增长呈现略小于 n^3 的趋势。原因可能是因为数据量较小时，n，n^2 项的影响占比较大。

同理，观察 GET 操作的时间复杂度，发现其随 n 的增长呈现略小于 n^2 的趋势。原因同理。

对于 DEL 操作，我们观察到从 5,120 增长到 10,240 时，时间并没有增长四倍而是两倍。这是因为，在数据量过小的情况下，外存总层级数量只有一至两层，当 0 层数据触发合并操作时，所有删除标记立即被合并并删除，而不会出被保留在第一层，使得合并入下一层的文件是更早的 sst 文件的情况。

2. 对于三种操作的吞吐，我们使用实验二进行说明，测试结果如表 2（时间单位：次/每秒）。

表 2: 在不同数据量的情况下，三种操作的吞吐量

n	PUT	GET	DEL
5,120	4,368	22,654	29,941
10,240	1,525	13,350	15,778
20,480	398	4,406	4,566

我们可以看到，随着操作次数的增多，三种操作的吞吐量都在下降。于是我们可以得出结论，在某一时刻，LSM 系统的操作吞吐量，与当时

系统中的总数据量呈负相关。已有的数据量越大，单次操作的期望效率就越低。

3.1.3 索引缓存与 Bloom Filter 的效果测试

在该项目中，我们使用了索引缓存和 Bloom Filter 来提升 GET 操作的效率，下面将通过对比实验的方式，展示索引缓存和 Bloom Filter 对性能的影响。实验数据采用 3.1中的实验一。

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，编号为第一组。
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，编号为第二组。
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，编号为第三组。

三组的吞吐量在设定的实验条件下如表 3所示。

表 3: 三组策略下，GET 操作随数据量的吞吐量变化

n	第一组	第二组	第三组
5,120	3244	12704	14104
10,240	990	7037	7052
20,240	251	5380	5624

从上表中，可以看出，如果不对索引值进行缓存，对性能的影响是非常严重的，因为这样会增加很多不必要的外存操作。相比之下，如果不利用 Bloom Filter 对数据进行过滤，效率只是稍有下降。

3.1.4 Compaction 的影响

在本节中，我们对 LSM 数据结构进行持续的插入操作，并在程序运行过程之中，每隔 10 秒钟记录一次该段时间内的平均吞吐量。在本节中，为

了消除数据大小对触发合并操作频率的影响，我们使用固定长度的字符串，并按索引值顺序进行插入，数据个数 n 取 65,536，字符串长度取 20,480。

实验结果如图 1所示

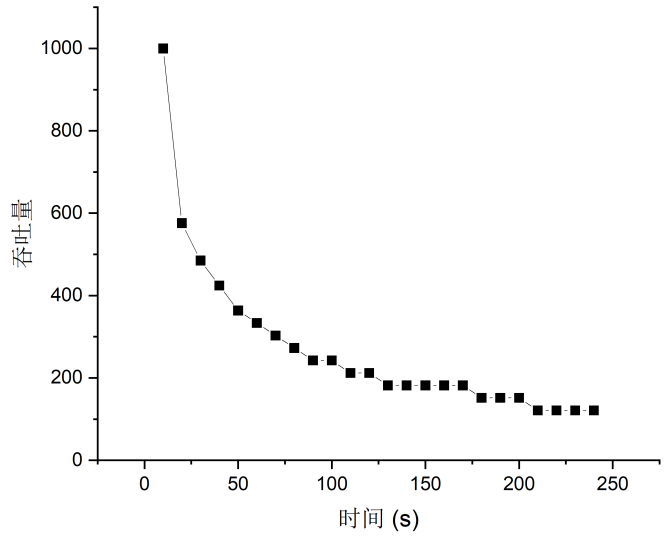

图 1: 持续插入过程中，PUT 操作吞吐量随时间变化

整个插入过程共花费约 260 秒的时间。从图中我们可以明显看到，吞吐量随着时间呈现反比例下降趋势。这是符合预期的，因为在插入过程中，外存中的层级数越来越多，而每次合并操作都要合并到最后一层，导致平均吞吐量下降。

3.1.5 对比实验

上面的几节，我们专注于外存的性能，而对于内存而言，我们再本项目中采用的跳表，可能并不是最优的选择，还可以选择 std::map 等键值存储数据结构。在本节中，我们就对两者进行性能上的比较。实验数据采取与3.1中的实验一相同的方案。

测试结果如表 4所示（单位：秒）。

从两表的对比中，我们看到其实两者的性能差异并不大，因为在平均意义上来说，两种数据结构的时间复杂度是一样的。由此合理推测，由于主要

表 4: 两种数据结构作为 MemTable 的性能对比

n	PUT	GET	DEL
5,120	0.398	0.366	0.422
10,240	1.566	1.029	1.038
20,480	9.962	3.552	4.951

跳表

n	PUT	GET	DEL
5,120	0.480	0.331	0.336
10,240	1.615	0.899	0.926
20,480	9.839	3.982	4.126

std::map

的时间花费在外存操作上，故所有具有对数级别的查询，插入，删除性能的数据结构，理论上应该都会得到类似的性能表现。

4 结论

LSM-KV 键值存储系统是一种快速高效的键值管理系统。在本实验报告中，我们对该数据结构的内存、外存两部分分别进行了不同方面的测试。我们得出以下几个结论。

1. 随着系统中已存储数据量的增大，LSM 的性能逐渐降低，单次操作需要的平均时间更长，单种操作的平均吞吐量也会下降。

2. 在内存中对所有 SSTable 中的索引信息和 Bloom Filter 信息进行缓存是有必要的，能够显著提高相同条件下，LSM 的平均吞吐量和综合性能。

3. 内存数据结构采用 std::map 或跳表实现，效率差异不大。

5 致谢

经过一两个月的努力，我终于把 LSM-KV 数据结构实现了，并且进行了初步的优化。在此过程中，有许多同学给了我很多帮助，包括杨景凯，冯逸飞，董琛等等。在此，我表示对以上同学的感谢。

6 其他和建议

我觉得这个项目最大的坑就在于，我开始使用了好多 vector，但是后来发现 list 和 vector 在默写情况下就是不能互相替代，比如如果有在中间插入的需求，那么 vector 虽然也可以实现，但是不是一个合理的选择，所以必须要谨慎选择数据结构。

另外,我起初不敢使用 vector⟨pair⟨uint64_t, list⟨pair⟨uint64_t, string⟩⟩⟩⟩这种超长的复杂数据结构，后来被证明是必要的。。