LSM-KV 项目报告

杜心敏 521021910952

2023 年 5 月 27 日

1 背景介绍

LSM-KV 全称 Log-Structured Merge Key-Value Tree[1]。是一个基于磁盘的存储和管理键值对数据结构，主要用于提高高概率插入和删除的性能。其通用的结构如图 1，本项目中未用到 Immutable Memtable，暂未实现 WAL Log。

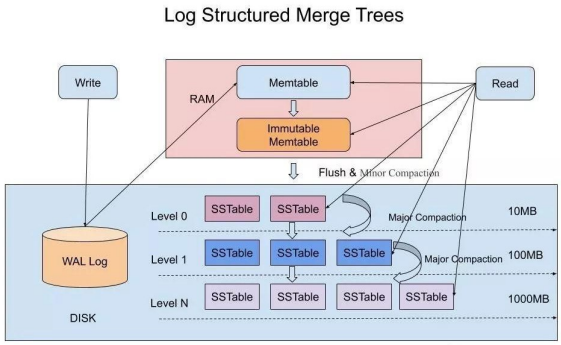


图 1: 主要结构示意

1.1 动机与应用

LSM 树的主要解决了磁盘存储系统在写入性能和查询性能之间的矛盾。采用顺序写入和批量合并，提高写入操作的性能，通过索引保证较好的查询性能。

磁盘存储系统使用随机写入的方式来更新数据，每次写入操作需定位，磁盘寻道和旋转延迟成为写入性能的瓶颈；另一方面，采用顺序读取的方式处理查询操作，在数据量很大时，延迟显著增加。

LSM 树将数据缓存在内存中，定期批量写入磁盘，以实现高效的写入性能。同时，将数据分布在不同层次的磁盘文件中，使用索引定位，来提高查询性能。在保证高写入性能的前提下，

提供较好的查询性能，适用于大规模写入的应用场景。如日志记录、分布式数据库系统、搜索引擎等领域。

1.2 背景知识

五分钟规则（Five Minute Rule），是一个经验法则，指页面的访问频率高于每分钟一次时，增加内存缓冲空间可以节省磁盘 I/O 的开销，从而降低磁盘访问的频率。

跳表（Skip List），一种有序链表的数据结构，通过在每一层链表中建立索引节点，使得查找时可以跳过一部分节点，提高搜索效率。LSM 树中内存部分使用跳表结构。

布隆过滤器（Bloom Filter），一种高效的概率数据结构，利用多个哈希函数和一个位数组来表示集合中的元素，快速判断一个元素是否属于一个集合。在 LSM 树中，布隆过滤器用于减少不必要的磁盘访问。

磁盘层级（Disk Level），LSM 树采用多层磁盘结构来存储数据。每个磁盘层级包含若干文件，较新的数据会存储在较高的层级（最新的存在内存）。通过 compaction 操作 (合并多个文件 SSTable) 来清除冗余的记录，避免 SSTable 数量的膨胀。

2 性能测试

2.1 预期结果

延迟测试预期结果：

1. Get 操作延迟。预期 Get 操作的延迟相对较低，因为 LSMkv 树的查询通常可以在较低的时间复杂度内完成。

2. Put 操作延迟。预期 Put 操作的延迟相对较高，因为数据的写入需要经过多个层级的写入操作，包括内存和磁盘的写入过程。可能会触发 compaction 操作，需要将文件合并，重新拆分。

3. Delete 操作延迟。预期 Delete 操作的延迟与 Put 操作类似，因为删除操作实质上是插入删除标记。

吞吐测试预期结果：

1. Get 操作吞吐。预期 Get 操作的吞吐量较高，因为 LSMkv 树的查询通常可以并行处理，充分利用系统资源。

2. Put 操作吞吐：预期 Put 操作的吞吐量较低，因为写入操作涉及到多个层级的数据结构更新和磁盘写入，性能受限于磁盘的写入速度。

3. Delete 操作吞吐：预期 Delete 操作的吞吐量与 Put 操作类似，因为删除操作也需要进行相应的写入操作。

2.2 测试集说明

测试用例中 key 的范围是 $0 < key < testSize$，value 的长度范围 $0 < value.size() < testSize$。

$testSize$ 取三种规模，分别为 512，1024×2，1024×32。

2.3 常规分析

表 1: 延迟测试（单位：纳秒）

Test	**Operation**	**Time (ns)**	**Time per Operation (ns)**
512	Put	747000	1458
	Get	153400	299
	Delete	285900	558
2048	Put	17091300	8345
	Get	83036800	40545
	Delete	88019000	42978
1024 * 32	Put	32183963700	982176
	Get	1863248900	56861
	Delete	2900707100	88522

表 2: 吞吐量测试

操作	操作数量	总运行时间 (秒)	吞吐量
Put	512	0.000747	686.89
Get	512	0.0001534	3340.31
Delete	512	0.0002859	1789.47
Put	2048	0.0170913	119.66
Get	2048	0.0830368	24662.92
Delete	2048	0.088019	23271.52
Put	32768	32.1839637	1017.73
Get	32768	1.8632489	17577.72
Delete	32768	2.9007071	11303.32

根据数据显示，Put 操作的耗时是最长的，原因是 put 极可能产生 compaction，以及需要定时写入内存。Delete 操作本质和 put 类似，但是由于 value 大小固定，比较小，产生 compaction 的频率比较低，所以比 put 的效率高。Get 操作耗时最短，由于查询不产生 compaction，不需要写入内存，而且查询的时候利用了 Bloom Fliter，以及索引的二分查找，减少了内存访问次数。其相应的吞吐量如下。

2.3.1 索引缓存与 Bloom Filter 的效果测试

对比了下面三种情况 GET 操作的平均时延

1. CASE0: 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. CASE1: 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. CASE2: 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

表 3: 索引缓存与 Bloom Filter 的效果测试

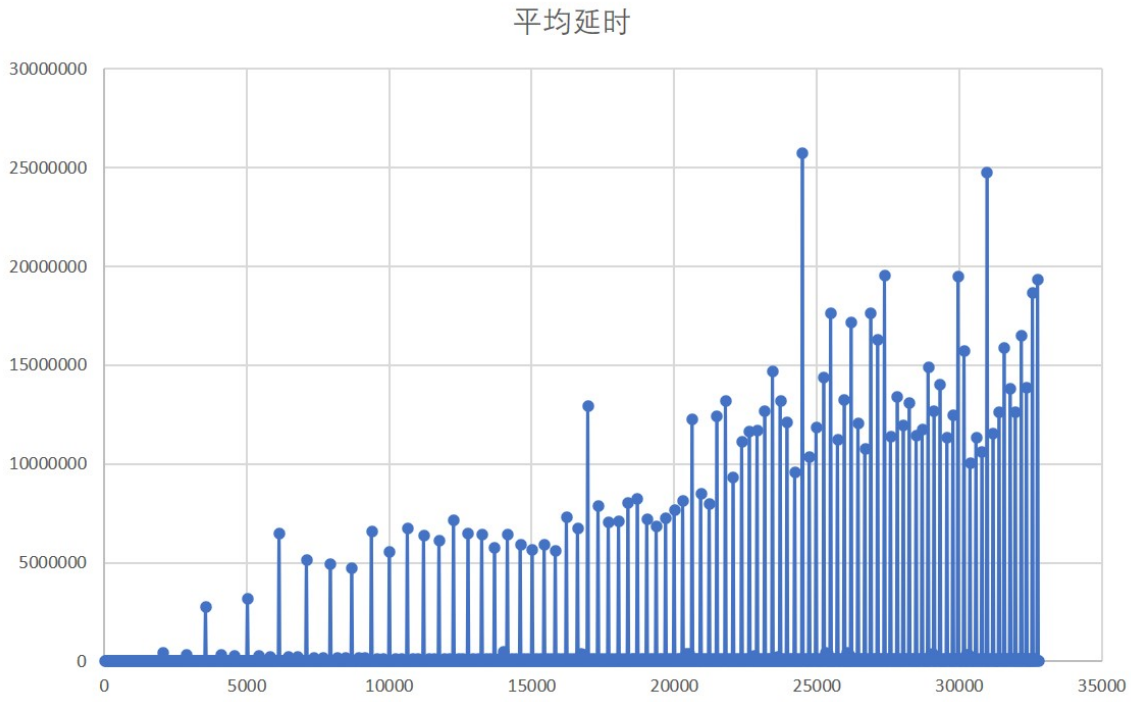
测试案例	查询次数	总时间 (纳秒)	平均时间 (纳秒)
CASE0	2048	143235200	69939
	8192	367972200	44918
CASE1	2048	46056000	22488
	8192	179843000	21953
CASE2	2048	42848300	20922
	8192	170806300	20850

表 4: 查询操作的性能统计

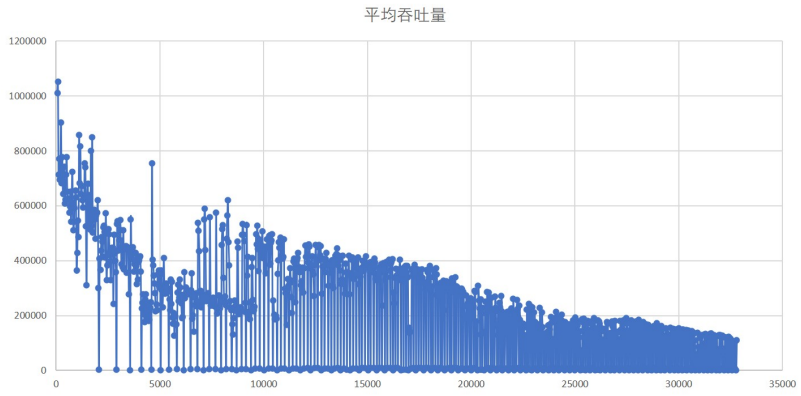
可以看出内存中缓存 Header 信息对性能提升极大，因为磁盘访问和内存访问的速度差距明显。测试集中有一半数据是未查找到的，BloomFliter 可以迅速返回结果，所以 CASE2 的性能是最高的，但是由于内存中二分查找也十分快速，所以相比而言提升不算很明显。如果对于很多次搜索都 miss 的情况，bloom 过滤器可以有很大帮助。

2.3.2 Compaction 的影响

不断插入数据的情况下，（每 30 个为一组）统计平均延迟、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。



从平均延时可以看出，每次遇到 compaction，延时有很明显的阶跃，并且随着数量增大，compaction 递归深度增加，compaction 所需要的时间明显增加，相应的吞吐量计算如下。



2.3.3 Level 文件数配置的影响

Level 配置的影响：每一层的最大文件数目是可以配置的变量，由于最大层数对 compaction 的影响最大，所以对比了下面三种情况 PUT 操作的平均时延

1. CASE0:level0 容量很大，之后依次加 1
2. CASE1:level0 容量较小，之后依次为上一层的两倍
3. CASE2:level0 容量较小，之后依次为上一层的三倍
4. CASE2:level0 容量较小，之后依次为上一层的四倍

表 5: Put 操作的性能统计

Case	操作数量	平均延时
CASE0	512	1375
	2048	7164
	32768	1360736
CASE1	512	1408
	2048	7655
	32768	842141
CASE2	512	1411
	2048	7192
	32768	592887
CASE3	512	1369
	2048	7288
	32768	433353

可以看到，level 最大容量增长速度越大，对大测试的性能提升效果越明显，这是因为文件数越多，每次 compaction 的代价越大，层最大容量增长速率提高，可以减少 compaction 的递归深度。

3 结论

本次 project 实现了 put,delete,get 功能（增删改查）, 由于时间关系没有实现 scan 功能。对于 scan 功能，由于每个 SSTable 是内部有序的，所以 scan 功能可以依赖于头文件的最大最小值，以及索引的二分查找功能来提高效率，并使用归并排序，把找到的数据块合并成一个有序数列。

比较了每种操作的延时和吞吐量，以及 Bloom Fliter，二分查找，内存索引区等技巧对性能提升的帮助。实验结果基本符合预期。可以看出 compaction 操作是程序的性能瓶颈，对于该操作，程序中已经实现的优化有：使用归并排序合并 SSTable，提高层最大容量的增长速率减少 compaction 递归深度。还可以实行的优化是：使合并操作并行化。

4 致谢

1. 感谢 Github 用户 leiyx 的 github 开源代码 My-LSM-KVStore。其中实现了代码的并行化，以及使用 Cache 进行优化。https://github.com/leiyx/My_LSM_KVStore
2. 感谢知乎用户 Pele 的博客讲解：LSM 树详解。对于本项目的数据结构的解释非常到位。https://zhuanlan.zhihu.com/p/181498475
3. 感谢 Github Copilot 提供的代码帮助，其有较好的代码补全功能，提升了编程效率。

5 其他和建议

LSM-KV 项目让我对一些数据结构的理解还是很有帮助的。完成一个比较大的项目对整体代码框架的构建，Debug 的能力，以及编程规范都有很大的锻炼。

参考文献

[1] Patrick O'Neil, Edward Cheng, Dieter Gawlick, and Elizabeth O'Neil. *The Log-Structured Merge-Tree (LSM-Tree).* San Val, 1995.