LSM-KV 项目报告

张祺骥 521021910739

2023 年 5 月 23 日

1 背景介绍

LSM-Tree 是一种基于磁盘的键值存储结构，其主要思想是将数据存储在多个有序的文件中，这些文件被称为 SSTable，每个 SSTable 都有一个索引文件，用于加速查找。LSM-Tree 的主要优势是写入速度快，但是读取速度较慢，因此在读多写少的场景下表现较好。LSM-Tree 的主要应用场景是键值存储系统，比如 LevelDB、RocksDB 等。

本次项目的目标是实现一个基于 LSM-Tree 的键值存储系统，即 LSM-KV。需要支持的操作包括 PUT、GET、DELETE，以及 Compaction。

2 设计

对于整个项目的设计我花了很长的时间来思考和完善。也经过了几次的修改和返工。

2.1 磁盘

辅助数据结构的设计如下：

```
struct Header
{
    uint64_t timeStamp;
    uint64_t keyValueNum;
    uint64_t minKey;
    uint64_t maxKey;
    ...
};
struct IndexData
{
    uint64_t key;
    uint32_t offset;
    ...
};
```

```
struct IndexArea
{
    std::vector<IndexData> indexDataList;
    ...
};
struct KVNode
{
    uint64_t key;
    std::string value;
    ...
};
```

SSTable 的设计如下：

```
class SSTable
{
public:
    std::string fileName;
    Header *header = nullptr;
    BloomFilter *filter = nullptr; // 10240字节
    IndexArea *indexArea = nullptr;
    char *dataArea = nullptr;

    uint32_t dataSize = 0;

    // 用于合并SSTable的时候用（因为如果是char来存储数据的话，没有办法merge）
    std::vector<KVNode> KVPairs;

public:
    // 读取所有char中的信息，生成键值对的vector，加速查找、用于compaction
    void formKVVector();
    // merge两张表。此处默认第一张表时间戳大于第二张表、第一张表层级低于第二张表（第0层为最低级）
    static SSTable* mergeTwoTables(SSTable *&table1, SSTable *&table2);
    // 把merge完以后巨大的SSTable切开
    std::vector<SSTableCache *> splitAndSave(std::string routine, int counter);
    // 切出来一个单独的SSTable，返回这个SSTable对应的SSTableCache
    SSTableCache * cutOutOneSSTable(int fileTag, std::string routine, int & currentSize);
    // 这个是留给外面的类调用的
    static void mergeTables(std::vector<SSTable*> &tableList);
    ...
private:
    static void mergeRecursively(std::vector<SSTable*> &tableList);
};
```

2.2 内存

内存我使用了之前作业的时候编写的跳表。此处不加赘述。

2.3 缓存

```
class SSTableCache
{
public:
    Header *header;
    BloomFilter *bloomFilter;
    IndexArea *indexArea;
    std::string fileRoutine;

    // 由于可能存在时间戳相同的文件, 因此需要用index将其区分
    uint64_t timeStampIndex = 0;
    ...
};
```

3 测试

3.1 测试环境

1. 机型：DELL DESKTOP-FK33F9A
2. 处理器：11th Gen Intel(R) Core(TM) i7-1165G7 @ 2.80GHz 1.69 GHz
3. 机带 RAM：16.0 GB (15.7 GB 可用)
4. 系统类型：64 位操作系统, 基于 x64 的处理器

3.2 性能测试

3.2.1 预期结果

1. find in memory：由于 memory 中使用 skip list 存储，因此查找的时间复杂度是 O(logn)，因此预期结果是随着数据量的增加，查找时间的增加是不明显的。
2. find in disk：现在缓存中查找；缓存中由于缓存了 bloom filter，因此判否的时间复杂度是线性的；若使用二分查找在 index area 里面查找，时间复杂度为 O(logn)。
3. put(without merge)：不发生归并的情况下，相当于在 skip list 中插入数据；因此时间复杂度为 O(logn)。
4. put(with merge)：根据实际情况而定，较难估计。

3.2.2 常规分析

执行了四组测试。四组测试中，插入的键值对数目均为 20000 个。四组测试的 value 大小分别为 512，1024，2048，4096 字节。延迟量数据如表**??**所示。

表 1: 不同数据大小 PUT, GET, DEL 操作平均延迟

数据大小/bytes	**put 延迟/us**	**get 延迟/us**	**del 延迟/us**
512	67.5161	3054.96	3083.28
1024	90.1122	2337.58	2316.6
2048	49.0046	1601.63	1512.99
4096	64.0194	1386.95	1394.3

lsm 的特点就是写入速度快，读取速度较慢。因此上述数据大体上是符合预期的。

get 和 del 的时间复杂度是相近的，这是符合预期的。因为在我实现的算法中，删除前进行了一次全局搜索（若没有搜索到，实际上就可以直接不插入代表删除的 tag）。而 put 的延迟相对于 get 是很低的；因此 del 的延迟基本和 get 相当。

选取 4096bytes 的情况下，吞吐量数据如表**??**所示。

表 2: 数据量 4096 bytes 时三种操作的吞吐

数据量/bytes	**put/s^{-1}**	**get/s^{-1}**	**del/s^{-1}**
4096	15620.3	721.007	717.206

3.2.3 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。

可以发现，很明显无缓存时，延迟是非常巨大的，比有缓存时大了一个数量级左右。单纯缓存 index 时提升性能的效果是最好的；缓存 bloom filter+index 时性能稍逊色于 index；有可能是自己实现的 bloom filter 本身调用耗时相较于 std::vector 中查找实际上耗时更久。

表 3: 不同缓存策略下 GET 平均时延

数据量/byte	无缓存/ms	索引/us	索引与布隆过滤器/us
512	6727.12	2184.19	3054.96
1024	7950.77	1453.56	2337.58
2048	14760.9	1414.13	1601.63
4096	34822.8	1393.77	1386.95
average	16065.39	1611.412	2095.28

3.2.4 Level 配置的影响

本节中我探究了每一层最大文件数目对于系统 put 操作的时延的影响。对于 leveling 和 tiering 的配置，和默认配置相同。

测试了下面几种情况：

1. 每层文件数目为公比为 2 的等比数列；第 0 层为 2 个，第 1 层为 4 个，依此类推。第 n 层为 2^{n+1}。
2. 每层文件数目为公比为 3 的等比数列；第 0 层为 3 个，第 1 层为 9 个，依此类推。第 n 层为 3^{n+1}。
3. 每层文件数目为公比为 5 的等比数列；第 0 层为 3 个，第 1 层为 9 个，依此类推。第 n 层为 5^{n+1}。
4. 每层文件数目为公比为 7 的等比数列；第 0 层为 3 个，第 1 层为 9 个，依此类推。第 n 层为 7^{n+1}。

插入数据为：100000 个 kv-pairs，每个 value 的大小为 2048bytes。

测试得到的数据如表**??**所示。

表 4: 不同 level 配置对 latency 的影响

等比数列公比	put latency/us
2	125.405
3	129.969
5	138.564
7	180.868

可以看到，等比数列公比越大，性能越差。原因如下：

（1）较低层文件数目较多时（认为 level0 为最低级），由于 level0 在 compaction 时需要包括所有文件，因此当 level0 文件数目较多时，会带来 compaction 时巨大的开销增长。

（2）较低层文件数目较多也意味着每次查找目标键值对的时候需要遍历更多的文件，因此时间开销会更大。

指导建议：

（1）level 较低的层级（认为 level0 为最低级），文件数目配置通常应该较小。较小的文件数目可以提高读取性能，因为查找目标键值时需要遍历较少的文件。较小的文件数目还可以减少合并操作的开销。

（2）层级较高的时候，文件数目配置通常可以比较大。较大的文件数目可以提高存储空间利用率，因为较大的文件可以更好地利用磁盘块的大小。但是，较大的文件数目可能会降低读取性能。而注意到，层级较高意味着这些数据存在的时间已经较长，被访问的概率相比于低层级的数据是低得多的，因此我们把较高层级中文件数目的最大值设置的比较大也是合理的。

4 结论

整个 project 我还是颇花了一番功夫去实现。我认为最困难的部分是先去设计出来整个架构。但是设计的时候也是不可能面面俱到的，在实际 coding 的时候我也一直遇到各种问题，需要见招拆招去解决。

project 能通过两个测试，正确性得到保证。

5 其他

这里我想记录我遇到的一些困难。

5.1 换了又换的编辑器/IDE

以前一直使用的是 clion，这次发现是 makefile，印象中 clion 好像对 makefile 不太友善（后来发现其实是支持的，就是近几年更新以后的事情），就换了 vscode。这也是我第一次用 vsc 来写 cpp。总之花了很久上手 vscode。

后来我突然意识到，其实也可以用 cmake 啊！我自己写一个不就完了！于是果断换回熟悉的 clion。比起 vscode 里面土鳖地 cout，clion 里面断点 debug 可爽了。

但是后来突然间发现 clion 里面 cmake 能跑过的测试，换 makefile 又不能过了；询问了助教以后才发现，clion 里面默认 cmake 是用 debug 模式来编译的，debug 下栈的布局和非 debug 有不一致，有些内存越界可能在 debug 下没有破坏重要的数据，Debug 下没有问题不代表 bug free。

于是心态直接崩了（只不过是从头再来!）。后来又换回了 makefile 来写，在 clion 里面直接用命令行运行了。

前前后后，在工具上面我真的折腾了很久。后来为了检测 memory leaking，就把项目挂到 github，在 ubuntu22.04 虚拟机里面直接从 github 上面把项目整个拉下来，再用 valgrind 查。

虽然折腾，但是学到了很多东西！很感激助教对我各式各样的啰里啰唆的问题都能够不厌其烦地解答。

5.2 bug 记录

这里记录我碰到的几个让我 de 了很久的 bug：

1. timeStamp-index。对于每个 SSTable，我以时间戳命名；我已经预料到后面切割 SSTable 的时候会出现重名的 SSTable，所以我提前加了-index 作为后缀来区分；但我没意识到后缀是不能从 0 开始的，因为下一层有可能也会出现同名的文件，导致缓存里面出现异常，或者写文件的时候直接覆写导致数据丢失；这导致出现了很诡异的查找不到的情况（此 bug 耗时两天 debug）。

2. 布隆过滤器的 bug：在 char 和 bool 的转换上。我原本以为可以直接用 01 代表 true、false 来直接转换。我被坑了好长时间。cpp 里面对于这个东西的要求还是很严格的。（此 bug 耗时约一天）