Project 1: LSM Tree 键值存储系统

1. 系统介绍

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，由 Patrick O’ Neil 等人在一片论文中提出。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。

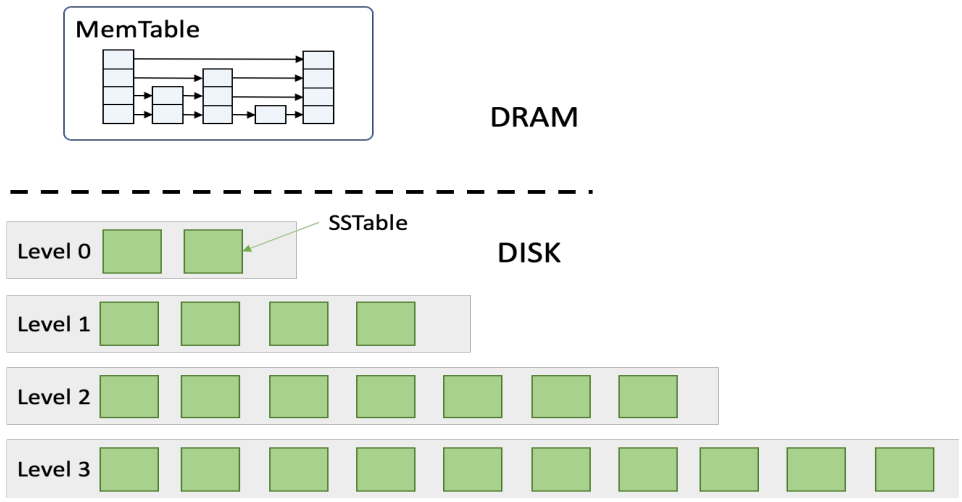
本项目将基于 LSM Tree 开发一个简化的键值存储系统。该键值存储系统将支持以下基本操作。

- PUT(K, V)设置键 K 的值为 V。
- GET(K)读取键 K 的值。
- DELETE(K)删除键 K 及其值。

其中 K 是 64 位有符号整数，V 为字符串。

2. 基本结构

LSM Tree 键值存储系统分为内存存储和硬盘存储两部分，采用不同的存储方式（如下图）。



内存存储结构被称为 MemTable，其通过跳表或平衡二叉树（此 project 中统一使用跳表）等数据结构保存键值对。相关数据结构已经在课程中进行过讲解，此处不再赘述。

硬盘存储采用分层存储的方式进行存储，每一层中包括多个文件，每个文件被称为 SSTable（Sorted Strings Table），用于有序地存储多个键值对（key-value pairs）。SSTable 在磁盘结构上分为数据区和索引区两部分。每个 SSTable 文件的前部分为数据区，用于存储有序的键值对数据（sorted key-value pairs）。文件的后端部分为索引区，用来存储索引数据。每个文件的索引数据可以帮助快速的对键值对进行查找和定位。索引数据包括文件中所有键（key）以及键所对应的键值对存放位置（即其相对于文件开头的字节数偏移量（offset））。

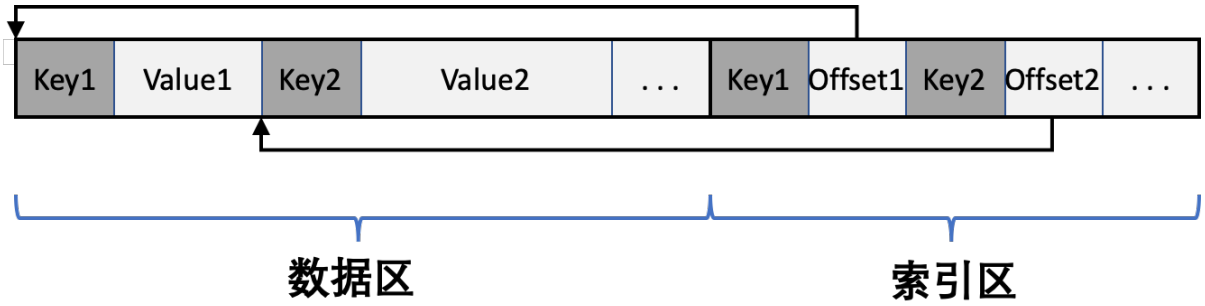
当我们要在某个 SSTable 中查找键为 K 的键值对时，最简单的方法是把该 SSTable 索引中的所有键与 K 逐一进行比较，时间复杂度是线性的。又考虑到 SSTable 索引中的键是有顺序的，我们可以通过二分查找在对数时

间内完成键 K 的查找，并通过 offset 快速从文件的相应位置读取出键值对。

SSTable 是保存在磁盘中的，而磁盘的读写速度比内存要慢几个数量级。因此在查找时，去磁盘读取 SSTable 索引是很耗时的操作。为了避免多次磁盘的读取操作，我们可以将 SSTable 中的索引部分缓存在内存中。之所以能够将其缓存在内存中，得益于以下两点：

1. SSTable 的索引部分的大小比较小，只包括了键和偏移量 offset，而不包括值 value。因此从将 SSTable 缓存在内存中不会占用过多的内存。

2. SSTable 是只读的。因此，SSTable 的索引一旦生成，不会有后续的修改，将其缓存在内存中，其内容与 SSTable 文件中的索引内容始终保持一致，不会产生不一致的情况。



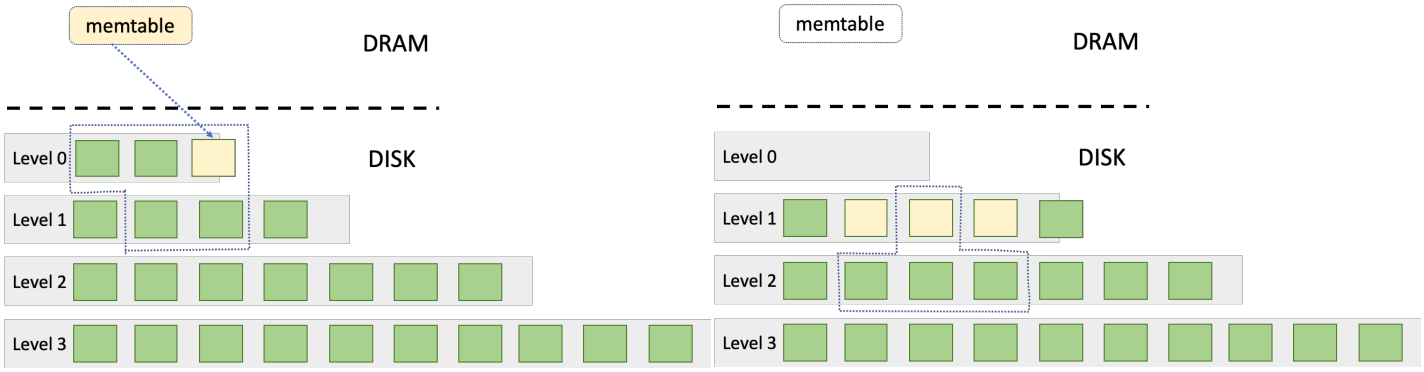
硬盘存储每一层的文件数量不同，层级越高，数量越多。每一层之间的文件数量比例是预设的（例如 Level 0 是 2，Level 1 是 4，Level 2 是 8，……）。除了 Level 0 之外，每一层中各个文件的键值区间不相交。

需要注意的是，SSTable 文件一旦生成，是不可变的。因此在进行修改或者删除操作时，只能在系统中新增一条相同键的记录，表示对应的修改和

删除操作。因此一个键 K 在系统中可能对应多条记录，为区分它们的先后，可以为每个条目增加一个**时间戳**。

3. 合并操作 (Compaction)

当内存中 MemTable 数据达到阈值（即转换成 SSTable 后数据和索引部分超过 2MB）时，要将 MemTable 中的数据写入硬盘。在写入时，首先将 MemTable 中的数据转换成 SSTable 的形式，随后将其写入到硬盘的 Level 0 层中。若 Level 0 层中的文件数量超出限制，则开始进行合并操作。对于 Level 0 层的合并操作来说，需要将所有的 Level 0 层中的所有 SSTable 与 Level 1 层的中部分 SSTable 进行合并，随后将产生的新 SSTable 文件写入到 Level 1 层中。



具体方法如下：

1. 先统计 Level 0 层中所有 SSTable 所覆盖的键的区间。然后在 Level 1 层中找到与此区间有交集的所有 SSTable 文件。

2. 使用归并排序，将上述所有涉及到的 SSTable 进行合并，并将结果每 2MB 分成一个新的 SSTable 文件（包括索引和数据部分，最后一个 SSTable 可以不足 2MB），写入到 Level 1 中。合并时要注意同样的键值应该保留时间戳最新的记录。

3. 若产生的文件数超出 Level 1 层限定的数目，则继续从该层中选择 SSTable 文件，以同样的方法继续向下一层合并（若没有下一层，则新建一层），直至文件数满足层数要求。

注意，从 Level 1 层往下的合并开始，仅需将超出的文件往下一层进行合并即可，无需合并该层所有文件。

在合并时，如果遇到相同键 K 的多条记录，可通过比较时间戳来决定键 K 的最新值。

4. 基本操作的实现

- PUT(K,V)

  对于 PUT 操作，首先在 MemTable 中进行插入，（此处有如何维护 MemTable 大小的问题）。若在插入后，MemTable 中数据大小超出 MemTable 限制，则将 MemTable 中的数据转换成 SSTable 保存在 Level 0 层中，若 Level 0 层的文件数量超出限制，则开始进行合并操作。

  合并操作可按照前文所述具体方法。

- GET(K)

  对于 GET 操作，首先从 MemTable 中进行查找，当查找到键 K 所对应的记录之后结束。若 MemTable 中不存在键 K，则从 Level 0 开始逐层进行查找，直到找到键值对记录，或找遍了所有层。

在查找每个 SSTable 时，可直接使用内存中缓存的索引，若要查询的键 K 在 SSTable 的键区间内，则使用二分法进行查找。查到记录后需要从磁盘文件中获取键值对内容。

- DELETE(K)

  为了提升性能，我们的系统使用延迟（Lazy）的方法会处理 DELETE 操作。当进行 DELETE 时，我们插入一条新的记录，表示键 K 被删除。在执行合并操作时，根据时间戳将相同键 K 的多个记录进行合并。

5. 性能测试和瓶颈分析

在这一部分中，你将对实现的键值存储系统进行评测。

5.1. 正确性测试

此次项目提供一个正确性测试程序，其中包括一下测试：

1）基本测试将涵盖基本的功能测试，其数据规模不大，在内存中即可保存，因而只实现内存部分即可完成本测试。

2）复杂测试将按照一定规则产生大量测试请求。其将测试磁盘数据结构，以及各个功能在面对大规模数据时的正确性。

3）持久性测试将对磁盘中保存的数据进行验证。测试脚本将多次造成测试程序意外终止，此后重新访问键值存储，检查其保存的键值对数据的正确性。

提供的基本测试和复杂测试在文件（correctness.cc）中，持久性测试的代码在文件（persistence.cc）中，使用说明请见相关程序。注意，在最终评分时的测试程序会有所变化（如修改参数，随机随机生成请求等），请不要针对所提供测试程序中的测试用例进行设计和实现。

5.2. 性能测试

在性能测试中，你需要通过编写测试程序，对键值存储系统进行测试，并产生测试图标和报告。测试内容应包括：

1）时延：在正常情况下（无 compaction 操作）PUT、GET、DELETE 操作所需要的平均时间的柱状图；

2）吞吐量：不断插入数据的情况下，每秒钟处理的 PUT 请求个数（即吞吐量）随时间变化的折线图。（注：由于涉及到 compaction 操作，你的测试需要表现出 compaction 对吞吐量的影响）

测试报告应至少包含一下部分：

1. 测试环境；

2. 测试参数和方法；

3. 测试结果（图表）；

4. 对测试结果的分析。需先对整体结果进行描述，随后对结果中细节以及异常现象进行描述并分析原因。

6. 作业要求

作业的内容包括：

1. 利用跳表实现 LSM Tree 系统的内存存储。
2. 在内存存储层次实现基本操作(PUT, GET, DELETE)。
3. 实现硬盘的分层存储，在内存超过一定阈值后将数据写入硬盘。
4. 实现合并操作。
5. 保证程序的正确性（可通过提供的测试程序，但最终测试会增加测试），并进行性能测试和分析。

为了大家能够顺利完成作业，建议大家先完成内存部分，并进行简单测试，最后再完成剩余任务。

注意事项：

1. 请保证可以在不同平台进行编译，不要使用平台相关代码（比如 windows.h），不要使用绝对路径。
2. 编译时请使用 c++17 标准。

提交材料：

1. 项目源代码（不包括可执行程序），注意，在最终测试时我们会使用不同的测试代码。
2. 设计文档，其中包括设计和测试两部分。设计部分，包括对整个系统设计的理解，以及自己实现的独特的地方；测试部分至少应包括前述性能测试和分析。

将源代码和设计文档使用 zip 格式打包压缩后，重命名为“学号-姓名.zip”提交。整个文件大小不应超过 5MB。

7. 可选项

这里的功能并非作业必需的要求，有兴趣的同学可选以下功能。

1. 在从 SSTable 中搜索键 K 时，先利用 Bloom Filter 快速判断其是否肯定不在此 SSTable 中，从而加快搜索速度。

2. 实现 SCAN(K1, K2)，返回迭代器以获取键值在 K1~K2 之间的所有键值对。

3. 增加 write-ahead-log，即在对 MemTable 进行写操作之前，先把操作的信息（K, V, 操作的顺序）记录在 log 中，再将 log 写入硬盘（利用 fsync）。在系统发生崩溃后重启，利用提前写入磁盘的 log 信息，在内存中恢复出崩溃之前的状态。