Project: LSM KV 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，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。2016 年，Lanyue Lu 等人发表的论文 Wisckey 针对 LSM Tree 长期被人诟病的写放大（Write Amplification）问题提出了相应优化：键值分离。在过去几年中，工业界中也陆续涌现出键值分离的 LSM Tree 存储引擎，例如 TerarkDB、Titan 等。

在本项目中，你需要基于 LSM Tree 以及键值分离技术开发一个键值存储系统。该键值存储系统将支持下列基本操作，并支持持久化、垃圾回收等存储系统特性（其中 Key 是 64 位无符号整数，Value 为字符串）。

- PUT(Key, Value) 设置键 Key 的值为 Value。
- GET(Key) 读取键 Key 的值。
- DEL(Key) 删除键 Key 及其值。
- SCAN(Key1, Key2) 读取键 Key 在[Key1, Key2]区间内的键值对。

提示：为了减少代码频繁改动，请同学们仔细、完整阅读整个文档，确保能够理解系统各个组件功能与设计细节后再开始开发。开发过程中注意代码质量，才能提高系统迭代、debug、测试乃至报告撰写的效率。

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（如图 1 所示）。硬盘存储中的键值被分离存储，因此硬盘存储又分为 SSTable 和 vLog 两部分。

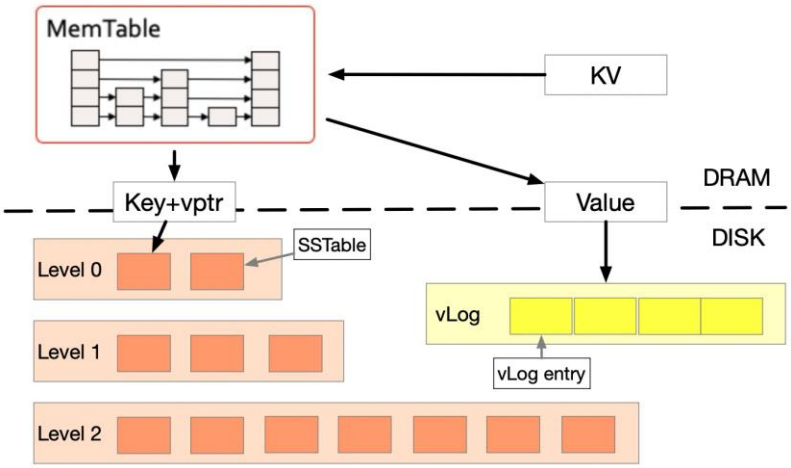

图 1 LSM-Tree 键值存储系统结构

2.1. 内存存储

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

2.2. 硬盘存储 —— SSTable

硬盘存储包括存储键（以下使用 Key 指代）的 SSTable 和值（以下使用 Value 指代）的 vLog。

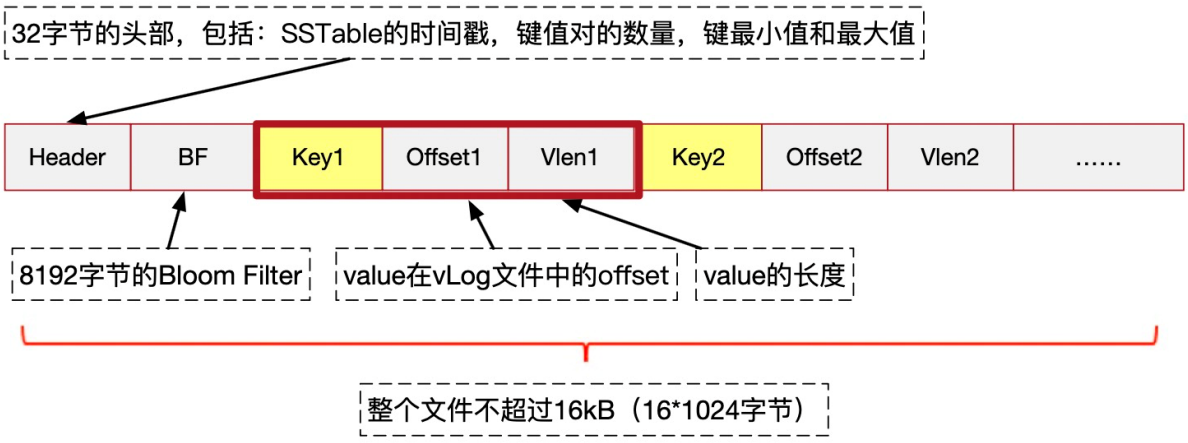

图 2 SSTable 结构

Key 采用分层的方式进行存储，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 Key，以及其对应的 Value 在 vLog 中的位置与长度。**本项目中一个 SSTable 的大小不超过 16 kB，但应尽量接近**。如图 2 所示，每个 SSTable 文件的结构分为三个部分。

1）Header 用于存放元数据，按顺序分别为该 **SSTable 的时间戳（无符号 64 位整型），SSTable 中键值对的数量（无符号 64 位整型），键最小值和最大值（无符号 64 位整型），共占用 32 B**。

2）Bloom Filter 用来快速判断 SSTable 中是否存在某一个键值，**本项目要求 Bloom Filter 的大小为 8 kB （8192 字节），hash 函数使用给定的 Murmur3，将 hash 得到的 128-bit 结果分为四个无符号 32 位整型使用**。超出 Bloom Filter 长度的结果要进行取余。

3）<Key, Offset, Vlen>元组，用来存储有序的索引数据，**包含所有的 Key、Key 对应的 Value 在 vLog 文件中的偏移量 offset (无符号 64 位整数)以及值的长度（无符号 32 位整数）。**

当我们要在某个 SSTable 中查找 Key 对应的键值对时，最简单的方法是把该 SSTable 索引中的所有键与 Key 逐一进行比较，时间复杂度是线性的。但考虑到 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 Key 的查找，并通过 offset 快速从 vLog 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 的 Bloom Filter 和索引是很耗时的操作。为了避免多次磁盘的读取操作，**需要将 SSTable 缓存在内存中**，具体缓存方法和策略可自行设计。如果不缓存，可能无法在规定时间内完成测试。

硬盘存储每一层的文件数量上限不同，层级越高，上限越高。每一层的文件数量上限是预设的，本项目中 Level n 层的文件数量为 2^{n+1} （即 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，例如在 Level 0 层中，两个文件包含的 key 的范围可以分别是 0~100 和 1~101，而在其他层中则需要确保任意两个不同文件的键值区间不相交。

SSTable 以 “.sst” 作为拓展名，所有文件存放在数据目录中（数据目录作为构造函数参数给出），Level 0 层的文件应保证在数据目录中的 “level-0” 目录下，Level 1 层的文件应保存在数据目录中的 “level-1” 目录下，以此类推。具体的文件名不做要求，你可以通过扫描该目录中所有

的文件进行初始化。

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 Key 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，又考虑到每个 SSTable 中的数据是同时被写成文件的，因此其实只需在 SSTable 的 Header 中记录当前 SSTable 生成的时间戳即可。为了简化设计，在本项目中，你需要使用 SSTable 的生成序号表示时间戳：在键值存储系统被**初始化/reset** 之后，第一个生成的 SSTable 的时间戳为 1，第二个生成的为 2，以此类推。**注意，键值存储系统在启动时，如果没有进行初始化操作，你可能需要想办法找到此前所生成的最后一个时间戳，才能用于下一个 SSTable 的生成。**

2.3. 硬盘存储——vLog

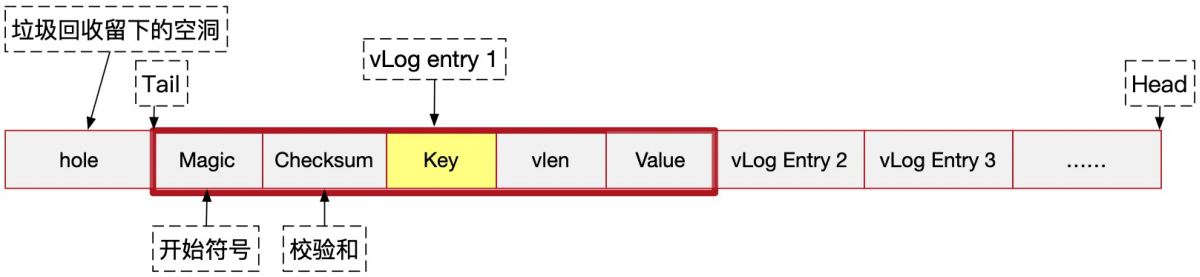

图 3 vLog 文件结构

vLog 用于存储键值对的值，vLog 文件（文件名作为构造函数参数给出）的结构如图 3 所示，其中 head 为新数据 append 时的起始位置，tail 为空洞之后第一个有效数据的起始位置，文件由连续的 vLog entry 组成，每个 entry 包括五个字段：

1) Magic (1 Byte)：作为每个 entry 的开始符号，可自己定义，如 0xff。由于系统没有持久化 vLog 文件 head 和 tail 的位置，Magic 被用来查找 vLog 数据开始的位置。
2) Checksum (2 Byte)：通过对由 key, vlen, value 拼接而成的二进制序列计算 crc16 得到（crc16 函数在 utils.h 中给出）。Magic 和 checksum 二者在读取数据时共同用于检查数据是否被完整写入，这有利于在系统

初始化时确定 tail 的位置（详见第 4 节 reset 操作）。

3) Key (8 Byte)：键，为了方便垃圾回收，vLog 同时保存了 Key，具体原因请见第 5 节垃圾回收。

4) vlen (4 Byte)：值的长度。

5) Value：值。

由于对键值存在修改或删除操作，因此每间隔一段时间系统就需要对 vLog 文件进行垃圾回收。简单来说会扫描 vLog 尾部 (tail) 一段 vLog entry，检查其是否过期（被修改或删除），将没有过期的 entry 重新插入到 LSM Tree 中，并使用文件系统的 fallocate 系统调用将这块回收的区域打洞。垃圾回收的具体流程将在第 5 节详细介绍，在此先不赘述。

3. 合并操作 (Compaction)

当内存中 MemTable 数据达到阈值（即转换成 SSTable 后大小超过 16kB）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，随后将其直接写入到硬盘的 Level 0 层中，生成 SSTable 之后 MemTable 清空。若 Level 0 层中的文件数量超过限制，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 SSTable 与 Level 1 层中的部分 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 16 kB 分成一个新的 SSTable 文件（最后一个 SSTable 可以不足 16 kB），写入到 Level 1 中。

3. 若产生的文件数超出 Level 1 层限定的数目，则从 Level 1 的 SSTable 中，优先选择时间戳最小的若干个文件（时间戳相等选择键最小的文件），使得文件数满足层数要求，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）。

一些注意事项：

1) 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。

2) 在合并时，如果遇到相同键 K 的多条记录，通过比较时间戳来决定

键 K 的最新值，时间戳大的记录被保留。

3) 完成一次合并操作之后需要更新涉及到的 SSTable 在内存中的缓存信息。

4) 多个 SSTable 合并时，生成的 SSTable 时间戳为原 SSTable 中最大的时间戳，因此生成的多个 SSTable 时间戳是可以相同的。

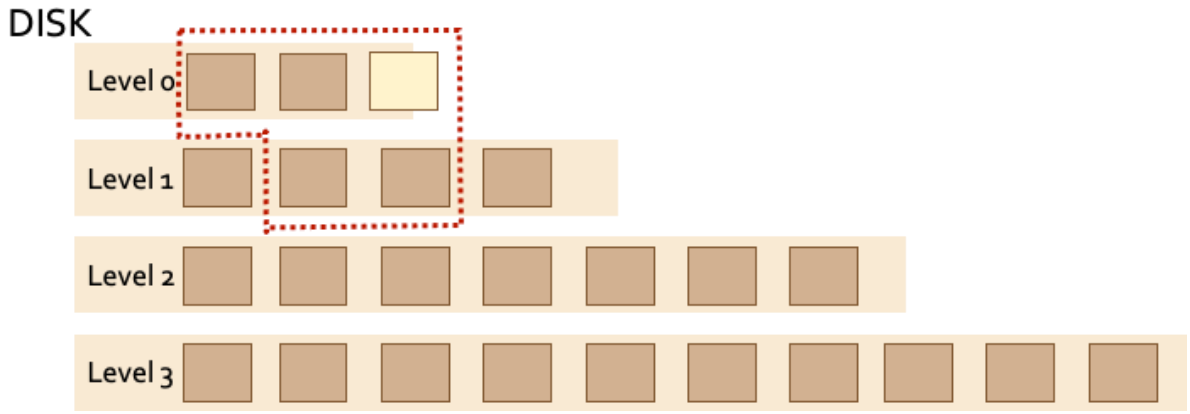

图 1 Level 0 到 Level 1 的合并，Level 0 的所有 SSTable 均被选取

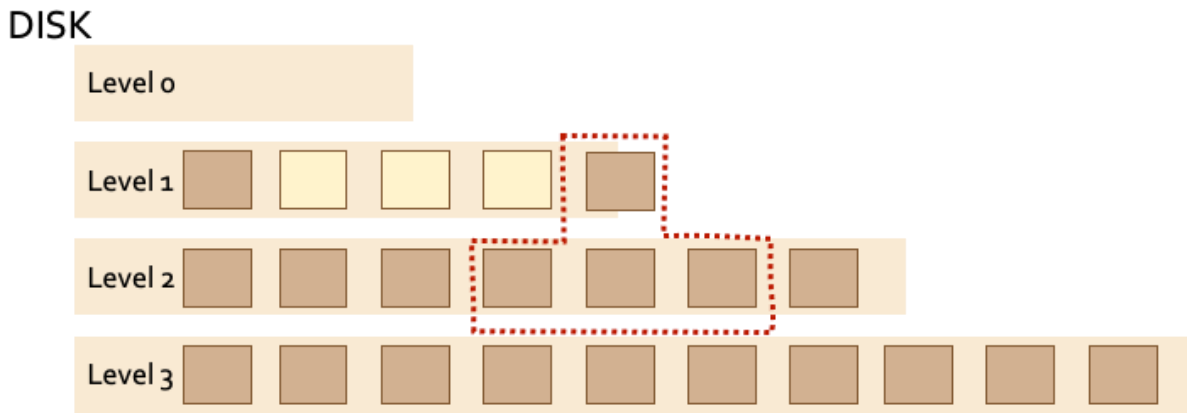

图 2 Level 1 到 Level 2 的合并，Level 1 多余的 SSTable 被选取

4. 基本操作的实现

- PUT(Key, Value)

  对于 PUT 操作，首先尝试在 MemTable 中进行插入。由于 MemTable 使用跳表维护，因此**如果其中存在 Key 的记录，则在 MemTable 中进行覆盖（即替换）而非插入**。同时，如果在插入或覆盖之后，MemTable 的大小超出限制**（注意这里指的是 MemTable 转**

换为 SSTable 之后大小超过 16kB，请注意不要计入值 value 的大小），则暂不进行插入或覆盖，而是首先将 MemTable 中的数据转换成 vLog entry 写入 vLog 中，并依次得到每个 entry 在 vLog 文件中的 offset。接着，依据得到的 offset 生成相应的 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。在这些操作完毕之后，再进行 Key 的插入。

- GET(Key)

  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 Key，则先从内存里逐层查看缓存的每一个 SSTable，先用 Bloom Filter 中判断 Key 是否在当前 SSTable 中，如果可能存在则用二分查找在索引中找到对应的<key, offset, vlen>元组。之后即可从 vLog 中根据 offset 及 vlen 取出 value。如果找遍了所有的层都没有这个 Key 值，则说明该 Key 不存在。

- DEL(Key)

  由于我们不能修改 SSTable 中的内容，我们需要一种特殊的方式处理键值对的删除操作。首先，我们查找键 Key 。如果未查找到记录，则不需要进行删除操作，返回 false；若搜索到记录，则在 MemTable 中再插入一条记录，表示键 Key 被删除了，并返回 true。我们称此特殊的记录为“删除标记”，其键为 Key，值为特殊字符串“~DELETED~”（测试中不会出现以此为值的正常记录）。当读操作读到了一个“删除标记”时，说明该 Key 已被删除。

  在使用 MemTable 生成 SSTable 以及 vLog 时，只需在 SSTable 中将该 Key 对应的 vlen 设为 0 而不需要写 vLog，在 GET 操作时，即可通过 vlen 是否为 0 来判断是否为删除操作。你也可以通过其他方式来处理删除，但需要保证操作的正确性。在执行合并操作时，根据时间戳将相同键的多个记录进行合并，通常不需要对 vlen 为 0 的记录作特殊处理。唯一一个例外，是在最后一层中合并时，所有 vlen 为 0 的记录应被丢弃。

- SCAN(K1, K2)

SCAN 操作需要返回一个 std::list<K, V>，其中按递增顺序存放了所有键在 K1 和 K2 （左右均包含）的键值对。SCAN 有多种实现方法，可以扫描 MemTable 和所有 SSTable 中的数据，也可以通过多个指针进行堆排序。此处具体实现不做要求。但是不能通过 for (i = K1; i <= K2; ++i) 的方式实现，因为测试中 K2- K1 可能非常大。

- RESET()

  本项目中所实现的键值存储系统**在启动时，需检查现有的数据目录中各层 SSTable 文件，并在内存中构建相应的缓存；同时还需要恢复 tail 和 head 的正确值，具体来说**：head 的值就是当前文件的大小；tail 则需要首先定位到文件空洞后的第一个 Magic，接着进行 crc 校验，如果校验通过则该 Magic 的位置即为 tail 的值，否则寻找下一个 Magic 并进行校验，直到校验通过，则对应的 Magic 位置即为 tail 的正确值。因此，其启动后应读取到上次系统运行所记录的 SSTable 数据及 vLog 文件。在调用 reset() 函数时，其应将所有层的 SSTable 文件（以及表示层的目录）、vLog 文件删除，清除内存中 MemTable 和缓存等数据，将 tail 和 head 的值置 0，使键值存储系统恢复到空的状态。同时，**系统在正常关闭时（可以实现在析构函数里面），应将 MemTable 中的所有数据写入 SSTable 和 vLog（类似于 MemTable 满了时的操作）**。

5. 垃圾回收（Garbage Collection）

在没有使用键值分离优化时，垃圾回收在合并 SSTable 时即可自动完成，然而当引入了键值分离之后，value 被单独存在 vLog 中，在合并操作时不会删除，因此需要额外的垃圾回收机制回收过期的（被删除或修改）的 vLog entry。

在本项目中，你需要实现在代码框架中预定义接口的 GC 函数，测试函数会在特定的时间调用该函数。GC 函数的主要包含以下参数：

1) chunk_size：本轮 GC 扫描的 vLog 大小（严格不小于，例如 chunk_size = 1024 Byte，而 vLog 头部的 10 个 entry 加起来大小为 1023 Byte，则需要再多扫描一个 entry）。

GC 的流程包括以下几步：

1) 逐个扫描 vLog 头部 GCSize 大小内的 vLog entry，使用该 vLog entry 的 Key 在 LSM Tree 中查找最新的记录，比较其 offset 是否指向该 vLog entry。
2) 如果是，表明该 vLog entry 仍然记录的是最新数据，则将该 vLog entry 重新插入到 MemTable 中。
3) 如果不是，表明该 vLog entry 记录的是过期的数据，不做处理。
4) **扫描完成后，不论此时 MemTable 是否可以容纳更多的数据（即继续插入新数据后仍然可以使得转化成的 SSTable 的大小满足要求），只要 MemTable 中含有数据，就需要主动将 MemTable 写入 SStable 和 vLog（类似于 MemTable 满了时的操作）。扫描到的未过期的数据会在这一步重新 append 到 vLog 头部。**
5) 使用 de_alloc_file() 帮助函数对扫描过的 vLog 文件区域打洞（详见第 7 节介绍）。

6. 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。

6.1. 正确性测试

此次项目提供一个正确性测试程序，其中包括以下测试：

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。

以上每个测试均要求在 20 分钟内运行完毕。

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。

6.2. 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，并产生测试图表和报告。测试内容应包括：

1）常规分析：测试 PUT、GET、和 DEL 三种操作的吞吐量（每秒执行操作个数）和平均时延（每一个操作完成需要的时间）。
2）索引缓存与 Bloom Filter 的效果：对比以下三种设计中，GET 操作的平均时延
   a）内存中不缓存 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。
   b）内存中只缓存了 SSTable 的索引信息（<Key，offset，vlen> 元组），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。
   c）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。
3）Compaction 的影响：不断插入数据的情况下，统计插入操作的时延的变化情况。测试需要表现出 Compaction 对时延的影响，即当某次插入操作触发 Compaction 之后该插入的 latency 应该会明显上升。
4）Bloom Filter 大小的影响：Bloom Filter 过大会使得一个 SSTable 中索引数据较少，进而导致 SSTable 合并操作频繁；Bloom Filter 过小又会导致其 false positive 的几率过大，辅助查找的效果不好。你可以尝试不同大小的 Bloom Filter 并保持 SSTable 大小不变，比较系统 Get、Put 操作性能的差异并思考原因。

测试报告模板：https://latex.sjtu.edu.cn/read/kkknnrhbkxyw ，请按照模板中的要求进行书写，**篇幅和字数不作为评分考量（字数不要超过 2000 字）**。

7. 文件空洞与部分 Util 函数说明

1. 文件空洞是什么?

fallocate 函数可以用于释放文件中的部分空间。被释放的空间称为“空洞”区域。**空洞不会实际占用磁盘空间**，在被读取时会返回全零（注意读取文件的时候，读取的是字节，因此读到的全是 0x00）。在成功调用 fallocate 之后，虽然文件的实际大小发生了改变，**但通过 tellg() 等方式得到的文件大小不会发生改变**。

2. 如何打洞?

结合使用 FALLOC_FL_PUNCH_HOLE 和 FALLOC_FL_KEEP_SIZE 标志即可在给定的 offset 处打下长度为 len 的洞。具体细节可查阅 https://www.man7.org/linux/man-pages/man2/fallocate.2.html 。本项目已经给出了相关辅助函数，你可以直接使用。

- uint16_t crc16(const std::vector<unsigned char> &data)：返回二进制序列 data 的校验和。
- int de_alloc_file(const std::string& path , off_t offset, off_t len)：对 path 文件从 offset 开始长度为 len 的区域打洞。
- off_t seek_data_block(const std::string& path)：返回自 path 文件开头起第一个数据（即 vLog 尾部第一个非洞的字节）的偏移量。**注意这个偏移量可能会比实际数据所处的偏移量小，你可能需要继续往后读取直到找到你自定义的 vLog entry 的开始符号，并在确认 crc16 校验通过之后，才能设置 tail 的值（详见第 4 节 reset 操作）**。

8. 作业内容及注意事项

作业内容总结:

1. 利用跳表实现 MemTable，**其大小上限为 16kB (指转换为 SSTable 之后大小不超过 16kB，不包括 Value 的大小)**。
2. 在内存存储层次实现基本操作 (PUT, GET, DELETE, SCAN)。
3. 实现硬盘的分层存储，当 MemTable 达到上限之后，以 SSTable 和 vLog

的形式写入硬盘中的 Level 0 层，SSTable 文件结构需参考本文之前的描述。

4. 在内存中缓存 SSTable，当执行 GET 操作的时候利用缓存来减少对磁盘的访问从而加快查询速度。
5. 实现垃圾回收函数，在垃圾回收后不影响系统的正确性、持久性（即此前在 SSTable 中 的 Key/Value 不会因为垃圾回收而丢失）。
6. 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，需要测试和分析的内容见实验报告模板。

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. 创建文件夹、删除文件夹、删除文件、获取目录中的文件名等操作请使用 utils.h 中提供的基本接口，不要使用绝对路径。
2. 编译时推荐使用 c++20 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. 实验报告，按照给定的 LaTeX 模板生成 PDF 文件，具体内容参考实验报告模板及 6.2 性能测试。

将源代码目录（命名为 “LSM-KV”）使用 7z 格式打包压缩后在 Canvas 平台上提交。注意在压缩前应清除测试时使用的 sst 文件，整个 7z 文件大小不应超过 5 MB。实验报告需要在 Canvas 中的单独的一个作业中提交。

9. 可选项

这里的功能并非作业必需的要求，有兴趣的同学可选做以下功能。

1. 目前键值系统中 MemTable 持久化为 SSTable 及 vLog 是同步完成的，这会大大增加某一个 Put/Del 操作的时延。为了屏蔽这段时延，可以维护两个 MemTable，当一个 MemTable 写满后，令其在后台并行地进行持

久化，而由另一个 MemTable 响应用户请求。

10. 此前学期的一些 Q&A

【问题】Bloom Filter 长度不足 32 位无符号整型
【回答】取余

【问题】sstable 文件路径和命名
【回答】 在给的 pdf 文档里有要求

【问题】memtable 生成 sstable 之后是否清空?
【回答】是的

【问题】插入或者更新会导致文件超过 16kB 能不能先做完操作在写到 sstable 中
【回答】不能，严格不大于 16kB

【问题】MemTable 用跳表实现，那 SSTable 里面的索引区是不是就是一个 MemTable?
【回答】不是。是 SSTable 自己的格式

【问题】MEMTable 转化为 SSTable 的过程是要全都遍历一次生成一个 bloom filter 吗?
【回答】是的

【问题】多个 SSTable 合并的时候，时间戳怎么确定?
【回答】多个 SSTable 合并的时候，生成的 SSTable 时间戳为原 SSTable 中最大的时间戳。因此生成的多个 SSTable 的时间戳是可以相同的，再因此文档里面“时间戳相等选择键最小的文件”是没问题的

【问题】多阶段提交中前几个阶段是否一定要完成 PPT 上指定的功能?
【回答】不是。 PPT 上只是给出了推荐顺序，项目各功能实际完成顺序和时间可以自行制定。

【问题】project 后续会给用于测试的文件吗?
【回答】目前用于测试的文件只有给出的 correctness 和 persistence ，可以先保证通过这些测试内容，评分会基于此做参数改动来防止针对性编程。如果有新增额外测试文件，会通过微信群和 canvas 公告的方式通知大家。

【问题】DEL 操作是在 MemTable 中用“~DELETED~”标记覆盖掉原来的值还是新建键值对?
【回答】如果在 MemTable 中查找到了 key 就直接覆盖返回，没有查找到就去 SSTable 查找。如果存在在某个 SSTable 中，就在 MemTable 中插入一条新的键值对。

【问题】请问在第一阶段实现了 MemTable 后，怎么利用 correctness 和 persistence 进行测试呢?
【回答】只实现 MemTable 部分，应该执行 correctness 是可以通过所有 Simple Test 的。Large Test，GC Test 和 persistence 都用到了硬盘存储，这部分实现可以按照阶段要求去完成，在每一阶段完成时都应该是一个可以通过测试的版本，通过逐渐迭代添加完整功能。