

Project: 基于 LSM Tree 键值存储

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。在此基础上，我们设计的近似最近邻查找系统，旨在利用 LSM Tree 的高效读写特性，为大规模数据集提供快速且准确的相似性搜索功能。

在本项目的第一阶段中，你需要基于 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。
- DEL(K) 删除键 K 及其值。

其中 **K** 是 64 位有符号整数，**V** 为字符串。

接下来，我们基于 LSM Tree 作为底层持久化存储的工具，来构建我们的近似最近邻查找模块。

ANN (近似最近邻搜索) 是一种在高维数据空间中快速查找与目标点“近似最近”邻居的技术。其核心目标是在 精度与效率之间平衡，通过允许少量误差换取搜索速度的指数级提升，适用于大规模数据场景（如图像检索、推荐系统）。与精确最近邻 (Exact NN) 相比，ANN 显著降低计算复杂度，成为现代机器学习和大数据的基石技术之一。

2. 实现 LSM Tree

2.1 LSM Tree 基本结构

在 Project 的第一阶段中，你将基于我们提供的代码，补全一个 LSM Tree 的键值存储系统。

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）。

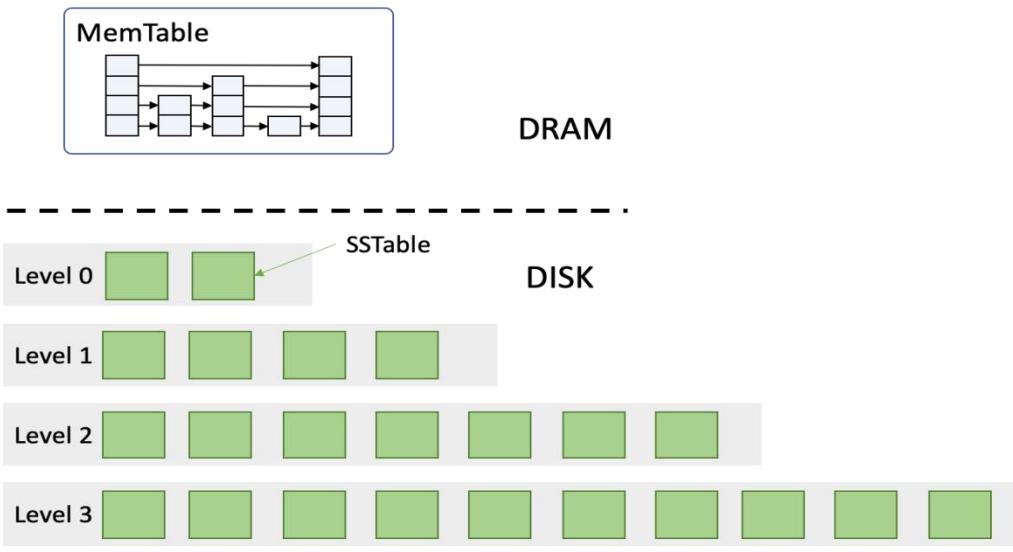


图 1 LSM-Tree 键值存储系统结构

内存存储结构被称为 MemTable，其通过跳表或平衡二叉树（该项目中统一使用跳表）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。

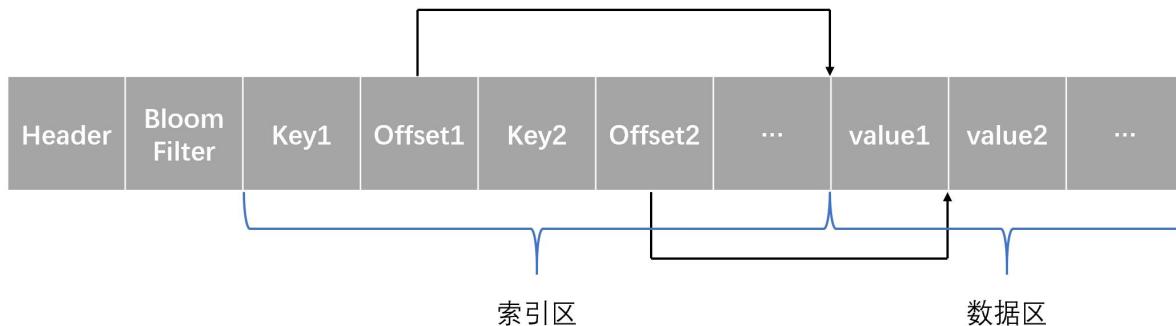


图 2 SSTable 结构

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)，该项目中一个 SSTable 的大小为 2 MB。如图 2 所示，每个 SSTable 文件的结构分为四个部分。

- 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（有符号 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 10 KB（10240 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。
- 3) 索引区，用来存储有序的索引数据，包含所有的键及对应的值在文件中的 offset（无符号 32 位整型）。
- 4) 数据区，用于存储数据（不包含对应的 key）。

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。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 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，我们可以 将 SSTable 中除了数据区外的其他部分缓存在内存中。之所以能够将其缓存在内存中，得益于以下两点：

1. 除了数据区之外，SSTable 中其余区的大小比较小，不包括值 value，因此缓存在内存中不会占用过多的内存。
2. SSTable 是只读的，一旦创建不可改变。因此，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。

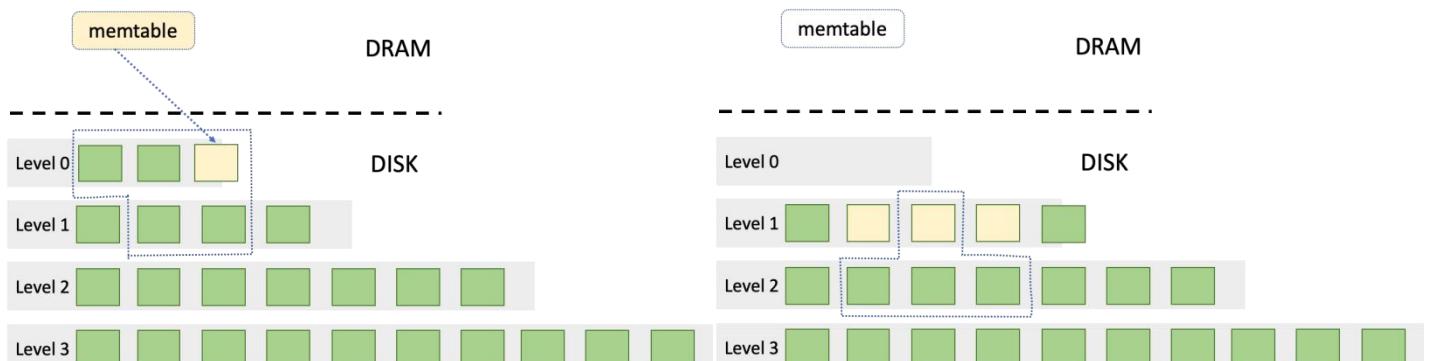
硬盘存储每一层的文件数量上限不同，层级越高，上限越高。每一层之间的文件数量上限是预设的，该项目中 Level n 层的文件数量上限为 2^{n+1} （即 Level 0 是 2，Level 1 是 4，Level 2 是 8，……）；除了 Level 0 之外，每一层中各个文件的键值区间不相交，例如在 Level 0 层中，两个文件包含的 key 的范围可以分别是 0~100 和 1~101，而在其他层中则需要确保任意两个不同文件的键值区间不相交。

SSTable 以 “.sst” 作为拓展名，所以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。因此一个键 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，如果键值存储系统在启动时，默认不会进行初始化操作，其需要先读取现有的 SSTable，将数据区之外的其他部分载入到内存中，找到最后一个 SSTable 对应的时间戳。

2.2 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 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，随后将其直接写入到硬盘的 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 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 2 MB 分成一个新的 SSTable 文件（**最后一个 SSTable 可以不足 2 MB**），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）。

注意

- 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

2.3 基本操作的实现

● PUT(K,V)

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，因此如果其中存在键 K 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 中数据大小超出 MemTable 限制（注意这里指的是 MemTable 转换为 SSTable 之后大小超过 2 MB），则暂不进行插入或覆盖，而是首先将 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。在这些操作完毕之后，再进行键 K 的插入。

● 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，则先从内存里逐层查看缓存的每一个 SSTable，先用 Bloom Filter 中判断 K 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 offset，之后从硬盘中读取对应的文件并根据 offset 取出 value。如果找遍了所有的层都没有这个 K 值，则说明该 K 不存在。

● DEL(K)

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，我们在 MemTable 中查找键 K，如果搜索到记录，则将该记录删除。但这还不够，因为虽然 MemTable 中不存在键 K，但其可能保存在某个 SSTable 中。因此，不管在 MemTable 中是否找到并删除了键 K，我们都要在 MemTable 中再插入一条记录，表示键 K 被删除了。我们称此特殊的记录为“**删除标记**”，其键为 K，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“删除标记”进行特殊处理。**唯一一个例外，是在最后一层中合并时，所有的“删除标记”应被删除。**

● 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。因此，其启动后应读取到上次系统运行所记录的 SSTable 数据。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）删除，清除内存中 MemTable 和缓存等数据，将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 中的所有数据以 SSTable 形式写回（类似于 MemTable 满了时的操作）。**

2.4 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。

2.4.1 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以及程序根目录下的 README.md 。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。

2.4.2 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：

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）。

测试报告模板同 lab1 learned-index，请按照模板中的要求进行书写。

2.5 阶段 1 代码内容及注意事项

1. 实现合并操作，具体过程参考第 3 节 合并操作。需要注意的是，**只有 Level 0 层需要把所有的文件都合并到下一层，其他层只需要按照 SSTable 时间戳从小到大(相等则按键区间从小到大)，将超出上限的文件合并到下一层即可。**

2. 实现 fetchString，从硬盘文件上读取 string 内容。
3. 参考 skiplist.h，实现 skiplist.cpp，实现其中的跳表功能。你也可以适当的对 skiplist.h 修改，以通过测试为标准。

更多细节要求请参考代码目录下的 README 文件。

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），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. 实验报告，按照给定的 LaTeX 模板生成 PDF 文件，不超过 1000 字，具体内容参考实验报告模板及 2.4.2 性能测试。

将源代码目录（命名为“LSM-KV”）使用 zip 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 zip 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。