ComboTree 设计与实现

张丘洋

2020 年 12 月 12 日

1 概览

根据 ComboTree 的设计，将数据结构分为 CLevel、BLevel 和 ALevel。其中 BLevel::Entry 中的键值对缓存使用 KVBuffer，CLevel::Node 使用 SortBuffer 来保存数据。

下面介绍的一致性都是系统的崩溃一致性，只能保证在整个系统崩溃后通过重启来恢复，若某个线程在中途崩溃了则无法修复，因为该机制没有运行时检查。

2 KVBuffer

KVBuffer 使用**前缀压缩**保存若干键值对数据。其数据结构定义为：

```
struct KVBuffer {
  union {
    uint16_t meta;
    struct {
      uint16_t prefix_bytes : 4;  // 共同前缀字节数
      uint16_t suffix_bytes : 4;  // 后缀字节数，与前缀之和为 8
      uint16_t entries      : 4;  // 目前保存的键值对数
      uint16_t max_entries  : 4;  // 最大可保存的键值对数
    };
  };
  uint8_t buf[112];
};
```

KVBuffer 包含了 2 个字节的元数据 meta，以及 112 个字节的 buf 来保存键值对。键值对的保存是**追加**的。

当后缀字节数为 1，一个 KVBuffer 能够保存 12 个键值对；当后缀字节数为 2 时，能够保存 11 个键值对。

由于使用了前缀压缩，键的大小是不固定的，但是值的大小固定为 8 个字节，为了使值在保存时能够 8 字节**对齐**，以及使键聚集在一起，让**所有键都位于同一个 Cache Line 中**，buf 从左到右顺序保存键，从右到左顺序保存值，如下图所示。

key(0)	key(1)	···	key(n)	···	value(n)	···	value(1)	value(0)

2.1 崩溃一致性

2.1.1 插入

插入操作首先将值和键追加到相应的位置，刷写一次 Cache Line，之后更新 8 字节元数据使其生效。元数据小于 8 个字节所以是原子操作。

2.1.2 更新

更新操作只需要更新 8 个字节的值，是原子操作。

2.1.3 删除

由于 B 层数据结构比较紧凑，现在没有设置删除标记位。

目前删除操作的执行步骤为：

1. 查找该键对应的位置
2. 若键处于最后一个位置，则直接原子性更新元数据即可。若不是，执行下一步
3. 将处于最后一个位置的键拷贝到当前位置。若该步骤完成后系统崩溃，则在下一次启动时可以通过扫描重复的键判断出此步骤出错
4. 将处于最后一个位置的值拷贝到当前的位置。此步骤完成后出错与上一步相同
5. 原子性更新元数据，这样就把最后一个位置的键值对移动到了被删除的地方

3 SortBuffer

SortBuffer 是 C 层每个节点保存数据的数据结构。SortBuffer 整体上与 KVBuffer 类似，不同之处是有额外的 6 个字节来保存排序索引。

SortBuffer 的定义如下所示,能够看到与 SortBuffer 相比多了一个 sorted_index。

```
struct SortBuffer {
  union {
    uint64_t header;
    struct {
      union {
        uint16_t meta;
        struct {
          uint16_t prefix_bytes : 4;  // LSB
          uint16_t suffix_bytes : 4;
          uint16_t entries      : 4;
          uint16_t max_entries  : 4;  // MSB
        };
      };
      uint8_t sorted_index[6];
    };
  };
  uint8_t buf[112];
};
```

`sorted_index` 虽然是个 `uint8_t` 数组，但是作为 `uint4_t[12]` 使用的，也就是 12 个 4 比特数组。之前说过每个缓存中最多能保存 12 个数据，而 4 个比特可以表示 0–11。这个比特数组中第 n 个数据表示从小到大第 n 个键在 `buf` 里的下标。

`sorted_index` 和其他的元数据一起组成了 8 个字节，所以同时修改他们是原子的。

`sorted_index` 数组除了用于保存排序数据之外，还用于**垃圾回收**，由于该数据包含了 12 个数据，在保存时，数组从前往后是目前有效数据的排序数组，从后往前是空闲位置的下标。

3.1 崩溃一致性

3.1.1 插入

插入数据时，首先从 `sorted_index` 的最后取出一个空闲的位置，将键值对写入对应的位置中，并刷写一次 Cache Line，之后原子更新 8 个字节的头部数据即可。

3.1.2 更新

更新操作只需要更新 8 个字节的值，是原子操作。

3.1.3 删除

删除只需要修改 8 个字节的头部数据即可（修改目前的有效键值数量以及 sorted_index）。

4 CLevel

```cpp
// sizeof(Node) == 128
struct __attribute__((aligned(64))) Node {
  enum class Type : uint8_t {
    INVALID,
    INDEX,
    LEAF,
  };

  Type type;
  uint8_t __padding[7]; // used to align data
  union {
    // uint48_t pointer
    uint8_t next[6];    // used when type == LEAF. LSB == 1 means NULL
    uint8_t first_child[6]; // used when type == INDEX
  };
  union {
    // contains 2 bytes meta
    KVBuffer leaf_buf;  // used when type == LEAF, value size is 8
    KVBuffer index_buf; // used when type == INDEX, value size is 6
  };
};

// sizeof(CLevel) == 6
class CLevel {
  ......
 private:
  uint8_t root_[6]; // uint48_t pointer, LSB == 1 means NULL
};
```

CLevel 采用的是 **B+ 树**,每个节点大小为 **128 字节,两个 Cache 行**,与 `BLevel::Entry` 的大小相同。节点使用 **`SortBuffer`** 来保存键值对和子节点。节点**没有保存父指针**，通过函数的递归来传递父指针，这样避免了扩展时父指针的修改，也减少了父指针的存储。

C 层使用 `SortBuffer` 的原因是：在扩展时，需要按顺序依次扩展，若 C 层无序，则扩展时每个 C 层节点都必须重新排序，导致扩展时间较长。另外若采用 `KVBfuffer`，在 C 层分裂删除旧节点中后半部分数据时会比较费时。

C 层的中间节点在保存子节点**指针时使用 6 个字节来保存**，也就是 `KVBuffer` 键值对中的值大小为 6，从而有更大的扇出。

无论是子节点指针还是根节点指针都通过将指针的**最低比特位置 1** 来表示 NULL，这样避免了多字节的拷贝，并且 *Optimizing Systems for Byte-Addressable NVM by Reducing Bit Flipping* 论文中提到这种方式可以减少比特翻转从而提高 NVM 寿命。

整个 C 层 `KVBuffer` 的**前缀压缩字节数是一样的**，前缀字节数和共同前缀可以由对应的 `BLevel::Entry` 来确定。

C 层无锁，由 `BLevel::Entry` 对应的锁来提供并发访问控制。

4.1 崩溃一致性

4.1.1 插入

若插入时不涉及到分裂，则一致性由 `SortBuffer` 提供。下面主要叙述节点分裂的步骤：

1. 创建新的节点，将旧节点后半部分数据拷贝到新的节点，更新新节点的 `next` 指针，持久化新节点。此时对旧节点没有任何的修改，若此步骤后崩溃没有任何问题。注意，分裂会在插入之后进行，也就是说**先插入后分裂**而不是先分裂后插入。
2. 将新的节点插入到父节点中。插入的一致性由 `SortBuffer` 保证。若此步骤后崩溃，在重启扫描时会发现该问题进行修复。
3. 修改旧节点的 `next` 指针。此步骤后崩溃与上一步骤一样。
4. 修改旧节点的元数据信息，从而删除后半部分数据。此步骤后崩溃，则下一步父节点可能需要分裂但却没有执行，在重启扫描时可以修复该问题。
5. 函数返回到父节点，父节点判断当前节点是否满，若满则该节点从第 1 步开始执行分裂，若不满，则分裂完成。这里父节点需要判断是否分裂是因为需要**先插入后分裂**，否则第 2 步执行时若父节点满则会出现问题。此步骤后崩溃可能父节点分裂没有执行，重启扫描可以修复。

4.1.2 更新

更新一致性由 `SortBuffer` 提供。

4.1.3 删除

为了实现简单，删除目前**不会合并节点**，只会在子节点的 SortBuffer 中删除对应的数据，一致性由 SortBuffer 提供。

5 BLevel

B 层的主要数据结构定义如下所示：

```
// sizeof(Entry) == 128
struct __attribute__((aligned(64))) Entry {
  uint64_t entry_key;     // 8 bytes
  CLevel clevel;          // 6 bytes
  KVBuffer<48+64,8> buf;  // 114 bytes
};

struct BRange {
  uint64_t start_key;
  uint32_t logical_entry_start;
  uint32_t physical_entry_start;
  uint32_t entries;
};

class BLevel {
  ......
 private:
  ......
  Entry* __attribute__((aligned(64))) entries_;
  size_t nr_entries_;
  std::atomic<size_t> size_;

  // 锁粒度为 Entry
  std::shared_mutex* lock_;

  // B 层分为 EXPAND_THREADS 个 range，这里多分配一个 range 减少编程难度
  BRange ranges_[EXPAND_THREADS+1];

  // 每个 range 分为多个小区间，记录每个小区间中的键值对数
  uint64_t interval_size_;
  uint64_t intervals_[EXPAND_THREADS];
```

```
32    std::atomic<size_t>* size_per_interval_[EXPAND_THREADS];
33
34    // 记录扩展时每个区间已经扩展的 entry 数目以及最大的键
35    static std::atomic<uint64_t> expanded_max_key_[EXPAND_THREADS];
36    static std::atomic<uint64_t> expanded_entries_[EXPAND_THREADS];
37  };
```

B 层是由 `BLevel::Entry` 组成的数组。每个 `Entry` 的大小为 **128 字节，两个 Cache 行**，与 `CLevel::Node` 的大小相同。节点使用 **`KVBuffer`** 来保存键值对和子节点。

在插入数据时，会先插入到 Entry 的 `KVBuffer` 缓存中，如果发现缓存满了，会先把 `KVBuffer` 刷入到对应的 C 层再进行插入。

`BLevel::Entry` 中的 `buf` 和 `CLevel::Node` 的 `buf` 大小相同，在 B 层 Entry **第一次将数据刷写到 C 层时，会直接复制 `buf`**，而不需要一个一个地将数据插入到 C 层（由于 C 层节点有排序数组所以还需要获得排序数组）。

5.1 B 层的大区间 `BRange` 和小区间 `interval`

为了能够实现多线程扩展，将 B 层分为多个 `BRange`，`BRange` 内部是连续的，`BRange` 之间不一定连续。`BRange` 的个数等于扩展线程的个数，扩展时每个线程扩展一个 `BRange`。

为了尽量让每个线程扩展的数据量相同,将每个 `BRange` 再细分为多个 `interval`，`interval` 目前最大为 128。每个 `interval` 会记录该区间内的键值对数，在开始扩展时，会将所有 `BRange` 的 `interval` 合在一起，并将其根据键值数尽量等分，分给不同的扩展线程。也就是说，在扩展时，每个扩展线程对应一个新的 `BRange`，但并不对应一个旧的 `BRange`。

5.2 崩溃一致性

5.2.1 插入、更新和删除

由 `KVBuffer` 和 C 层提供一致性。

5.2.2 扩展

B 层扩展时，会持久化每个 range 中当前已经扩展的最大键值（目前的实现没有持久化），旧的 B 层只会在扩展完成后才会被删除。

6 ALevel

A 层与原论文一致。A 层不需要修改，整个位于内存中，重启时重新生成即可。

7 NVM 管理

使用 PMDK 中的 `libpmem` 来管理 NVM。

B 层和 C 层保存在 NVM DAX 文件系统的两个文件中，在扩展时会创建新的文件，扩展完成后删除旧的文件。