ComboTree 实现

张丘洋

2020 年 12 月 7 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` 中的缓存与 `CLevel::Node` 使用相同的数据结构 `KVBuffer` 来保存数据。

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**排序**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**，`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)

3 CLevel

```
// 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 `KVBuffer` 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 BLevel

5 ALevel

6 NVM 管理