Aerospike源码分析

一、实现原理

wblock是aerospike进行数据写入的单位,默认情况是1M. 当存储介质为SSD时,这个值推荐设置为128k(这个值是SSD每次I/O的最小单位).

每个device(以下如无特殊说明,device都表示SSD) 被划分为多个wblock. 分配后device的逻辑结构如下:

```
1. <device header> <wblock1> <wblock2> <wblock3>...<wblockN>
```

wblock对应的代码结构如下:

```
1. typedef struct {
2.     cf_atomic32         rc;
3.     uint32_t            wblock_id;
4.     uint32_t            pos;
5.     uint8_t             *buf;
6. } ssd_write_buf;
7. typedef struct ssd_wblock_state_s {
8.     //对ssd_write_buf并发操作时的锁
9.     pthread_mutex_t     LOCK;
10.    //已用空间
11.    cf_atomic32         inuse_sz;
12.    //当buffer没有flush时,这个引用存在,可以当做cache使用
13.    ssd_write_buf       *swb;
14. } ssd_wblock_state;
```

每个record记录会占用n(n为整数)个rblock,每个rblock的大小为128 bytes.
例如,在如下实例的wblock结构中, rblock1–rblock3 三个连续的rblock构成一个record的实际存储位置.

```
1. <page header> <rblock1> <rblock2> <rblock3> .... <rblock N>
```

而record对应的key,也被称作索引,会被aerospike保留在内存中,用于快速定位record所在的位置.
默认情况下aerospike并不保存key的实际值,只是将key通过RIPEMD160算法生成一个20byte定长的hash值保存起来.

key所处的索引对应的结构体如下:

```
typedef struct as_index_s {
    .....
    //key 经过 RIPEMD160 后的 hash 值
    cf_digest keyd;  //20 byte
    // 过期时间
    uint32_t void_time: 30;
    //上次更新时间
    uint64_t last_update_time: 40;
    //数据所在的位置
    uint64_t rblock_id: 34;
    uint64_t n_rblocks: 14;
    uint64_t file_id: 6;
    .....
}
typedef struct as_index_s as_record;
```

上面只是列出了as_index_s的重要的字段. aerospike为了提高读取效率,把as_index_s的大小被固定为64byte,这刚好为cpu的cache line大小.在索引常驻内存的情况下,可以高效利用cpu缓存.

数据写入过程
record会先被写入每个device对应的ssd_write_buf, 当buffer满加入到device维护的swb_write_q队列. 后台会有线程从这个队列中读取数据持久化到device上.
详细的过程代码描述如下:

```
//获取要写入的大小,用于判断当前wblock是否够用和分配空间
uint32_t write_size = ssd_write_calculate_size(record);
//涉及到空间分配,加锁防止并发
pthread_mutex_lock(&ssd->write_lock);
ssd_write_buf *swb = ssd->current_swb;
if (write_size > ssd->write_block_size - swb->pos) {  //swb的剩余位置不够
    //重新生成一个
    swb = swb_get(ssd);
    ssd->current_swb = swb;
}
//分配空间
swb->pos += write_size;
//这里通过引用计数来决定是否flush这个swb
cf_atomic32_incr(&swb->n_writers);
//在swb中分配完空间就解锁
pthread_mutex_unlock(&ssd->write_lock);
//依次写入每一个bin
for (n_bins_written = 0; n_bins_written < rd->n_bins; n_bins_written++) {
    //aerospike中, 每个Bin Type的操作都对应着一个particle类型. 例如: integer类型对应着particle_integer
```

```
    uint32_t particle_flat_size = as_bin_particle_to_flat(bin, buf);
}
cf_atomic32_decr(&swb->n_writers);  //writer成功
```

读取数据过程

读取过程中,并不是每次从device获取. 看之前写入的过程,aerospike维护了一个内存队列来保存wblock,如果wblock的数据在内存中存在(flush到磁盘后会放入到一个队列中,并不是立即销毁),可以直接从内存中获取. 可以理解时一个LRU缓存.

详细的读取过程如下:

```
    //从内存中的索引as_record获取wblock信息
    uint32_t wblock = RBLOCK_ID_TO_WBLOCK_ID(ssd, r->rblock_id);
    //wblock是否在内存中
    swb_check_and_reserve(&ssd->alloc_table->wblock_state[wblock], &swb);
    if (swb) {//缓存中存在
        cf_atomic32_incr(&ns->n_reads_from_cache);
        ...
    }else{ //从SSD中读取
        cf_atomic32_incr(&ns->n_reads_from_device);
        //计算位置
        uint64_t read_offset = BYTES_DOWN_TO_IO_MIN(ssd, record_offset);
        //定位
        lseek(fd, (off_t)read_offset, SEEK_SET)
        //读取
        ssize_t rv = read(fd, read_buf, read_size);
    }
```

数据更新过程

aerospike在进行record更新时并不直接更新,而是采用copy-on-write的方式, 先读出记录,然后在合并写入到新的wblock中.

这样一来,原来所在的wblock关于这块record的空间就被浪费掉了. aerospike是怎么解决这个问题呢?

原来在每次更新时,都会对应减少原来的wblock实际占用的空间. 如果wblock实际有效的使用空间比例小于配置的某个阈值时,就会启动defrag(碎片整理)程序,将原有wblock的空间释放出来.

整个defrag过程代码描述如下:

```
if (resulting_inuse_sz < ssd->ns->defrag_lwm_size) {//小于阈值
    //设置wblock为defrag的状态
    ssd->alloc_table->wblock_state[wblock_id].state = WBLOCK_STATE_DEFRAG;
    //将wblock加入到defrag_wblock_q,等待被defrag线程处理
    cf_queue_push(ssd->defrag_wblock_q, &wblock_id);
}
```

```
7. //defrag线程
8. void* run_defrag(void *pv_data) {
9.     while (true) {
10.         uint32_t q_min = ssd->ns->storage_defrag_queue_min; //进行defrag的队列大小的阈值
11.         if (cf_queue_sz(ssd->defrag_wblock_q) > q_min) {
12.             cf_queue_pop(ssd->defrag_wblock_q, &wblock_id,CF_QUEUE_NOWAIT)
13.             //整理移动碎片
14.             ssd_defrag_wblock(ssd, wblock_id, read_buf);
15.             //放回free_wblock_q复用
16.             cf_queue_push(ssd->free_wblock_q, &wblock_id);
```

最后我们来看一下device的结构:

```
1. typedef struct drv_ssd_s{
2.     //用于处理write操作产生的数据写入
3.     ssd_write_buf   *current_swb;
4.     //用于处理defrag操作产生的数据写入
5.     ssd_write_buf   *defrag_swb;
6.     //当前可用的wblock
7.     cf_queue        *free_wblock_q;
8.     //需要defrag的wblock
9.     cf_queue        *defrag_wblock_q;
10.     //将要被flush到设备的queue
11.     cf_queue        *swb_write_q;
12.     // post_write_q 中的数据释放后,都会到这里. 但是这里只是一块数据的内存区域
13.     cf_queue        *swb_free_q;        // pointers to swbs free and waiting
14.     //swb_write_q中的swbflush到磁盘后,会缓存到这个swb队列,这个队列的大小是有限制`storage_post_write_queue`
15.     cf_queue        *post_write_q;
```