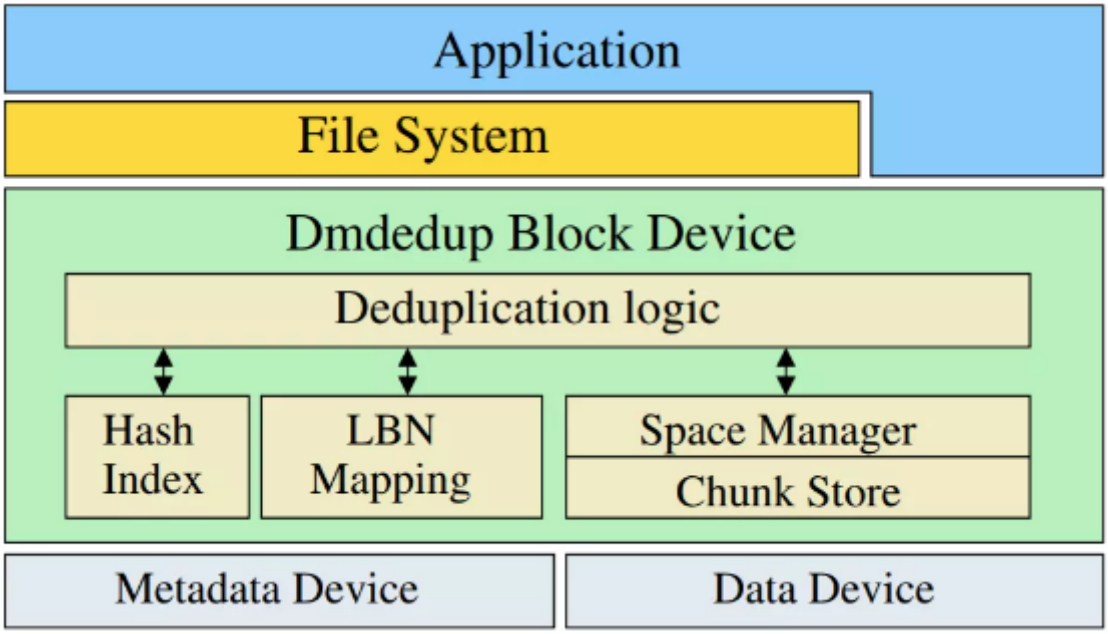
Device-mapper Dm-dedup详解

1. Basic

de-dedup在github上的代码： https://github.com/dmdedup/dmdedup4.13

设计文档（论文）：http://www.fsl.cs.stonybrook.edu/docs/ols-dmdedup/dmdedup-ols14.pdf

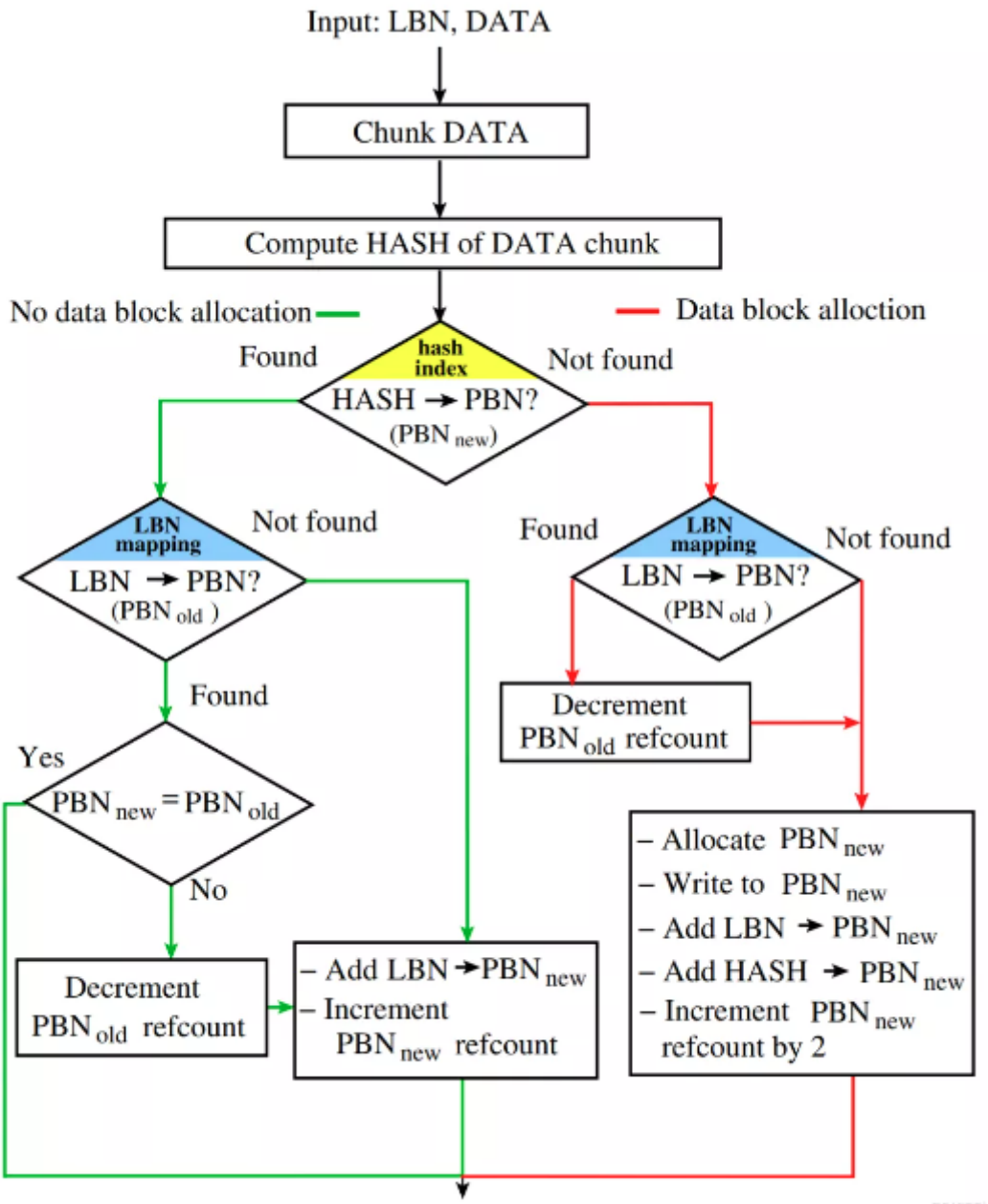
2. Overview



Dm-dedup的设计思想主要分为三个逻辑部分：

- Hash index：这一部分实际上与我们目前的研究目标无关，这里主要是包括不同种类的hash算法，比如SHA1、SHA2、SHA256、SHA512等，寻求hash length与较小碰撞概率的trade-off。
- LBN Mapping：这指的是逻辑块号映射，此处被用来描述逻辑块到物理块的映射关系，共包括两种<**分别是Hash-PBN以及LBN-PBN**>。
- Space Manager：空间管理器。

3. Processing



从上图可知，逻辑处理大致可分为三个步骤：

1. Chunk Data：指的是将输入的文件切换，或者将小文件汇集之后再切块。<需要注意的是，切块算法也是该领域的一个研究方向，不过对于我们的领域中，一般采用平均切块算法。该算法的特点是计算相对简单>
2. 计算每一个chunk的指纹，并且查询这个hash是否存在。如果存在，即代表已经存了这个数据，如果没有，则需要去sapce manager申请chunk大小的block来防止这个request并且计算和保存它的hash值。<**在此处，Hash-PBN的映射来表明数据是否已经存在**>
3. 接下来还需要讨论LBN和PBN的情况，这是考虑到系统可能存在更新或者一些其他的操作，一共被讨论成以下四种情况：
    - **No hash && no LBA**：一般是产生了新的文件和新的内容
    - **No hash && LBA**：数据发生了更新
    - **Hash && no LBA**：体现重删程序价值
    - **Hash && LBA**：一般来说就是在一个系统下的批量文件同步/覆盖的，这种情况副本A和副本B的引用不变，逻辑映射关系改变。

4. Code Analysis

4.1 chunk data

```
while (ci.sector_count && !error) {
            error = __split_and_process_non_flush(&ci);
            if (current->bio_list && ci.sector_count && !error) {

                struct bio *b = bio_split(bio, bio_sectors(bio) -
ci.sector_count,
                                  GFP_NOIO, &md->queue->bio_split);
                ci.io->orig_bio = b;
                bio_chain(b, bio);
                ret = generic_make_request(bio);
                break;
            }
        }
```

这里简单略过，据我分析，应该就是简单将chunk切割成等大的4K块。然后进而通过一个队列来讲请求发送到各个CPU中去。

```
static void process_bio(struct dedup_config *dc, struct bio *bio)
{
    int r;

    if (bio->bi_opf & (REQ_PREFLUSH | REQ_FUA) && !bio_sectors(bio)) {
        r = dc->mdops->flush_meta(dc->bmd);
        if (r == 0)
            dc->writes_after_flush = 0;
        do_io_remap_device(dc, bio);
        return;
    }

    switch (bio_data_dir(bio)) {
    case READ:
        r = handle_read(dc, bio);
        break;
    case WRITE:
        r = handle_write(dc, bio);
    }

    if (r < 0) {
        bio->bi_error = r;
        bio_endio(bio);
    }
}
```

继而解析每个请求，在此处被分为handle_read() 和handle_write()。

4.2 Space Manager

首先是两个inde表的创建

```
dc->kvs_hash_pbn = dc->mdops->kvs_create_sparse(md,
crypto_key_size,sizeof(struct hash_pbn_value),dc->pblocks, unformatted);

dc->kvs_lbn_pbn = dc->mdops->kvs_create_linear(md, 8,sizeof(struct
lbn_pbn_value), dc->lblocks, unformatted);
```

比较有意思的是，在创建kvs的时候，可以通过linear和hash index两种方式，我猜测linear应该是对于LBN和PBM，hash index则是服务于Hash-PBN。

```
static struct kvstore *kvs_create_linear_inram(struct metadata *md,u32 ksize,
u32 vsize,u32 kmax, bool unformatted)
{
        struct kvstore_inram *kvs;
        u64 kvstore_size, tmp;


        kvs = kmalloc(sizeof(*kvs), GFP_NOIO);
        if (!kvs)
            return ERR_PTR(-ENOMEM);

        kvstore_size = (kmax + 1) * vsize;
        kvs->store = vmalloc(kvstore_size);
        /*确定kvs->store的大小，这里的思想很简单，
        就是64 bit的lbn寻址到一个ksize的pbn上面，一般的pbn也是64 bit*/
        /*kmax是 逻辑设备的大小，这个map-table的含义就是lbn-pbn的映射关系*/

        tmp = kvstore_size;
        (void)do_div(tmp, (1024 * 1024));

        memset(kvs->store, EMPTY_ENTRY, kvstore_size);

        kvs->ckvs.vsize = vsize;
        kvs->ckvs.ksize = ksize;
        kvs->kmax = kmax;

        kvs->ckvs.kvs_insert = kvs_insert_linear_inram;   /*插入api*/
        kvs->ckvs.kvs_lookup = kvs_lookup_linear_inram; /*查找api*/
        kvs->ckvs.kvs_delete = kvs_delete_linear_inram; /*删除api*/
        kvs->ckvs.kvs_iterate = kvs_iterate_linear_inram; /*迭代api*/
        md->kvs_linear = kvs;

        return &(kvs->ckvs);
}
```

还有一个需要注意的点在于，**因为内存大小的有限，因此，这两个index的结构应该在disk和memory都存在。**因此，*kvs_create_linear_inram是通过linear的方式来创建inram的**LBN-PBN**，其index内部支持<**Insert, Delete, Lookup, Iterate**>

```
static int kvs_insert_sparse_inram(struct kvstore *kvs, void *key,s32 ksize,
void *value, s32 vsize)
{

    u64 idxhead = *((uint64_t *)key); /*无论key是128bit还是64bit,我们取64bit出来*/
    u32 entry_size, head, tail;
    char *ptr;
    struct kvstore_inram *kvinram = NULL;

    kvinram = container_of(kvs, struct kvstore_inram, ckvs);

    entry_size = kvs->vsize + kvs->ksize;  /*确立每个单位entry_size，一般是：64bit +
128 bit*/
```

```
    head = do_div(idxhead, kvinram->kmax);
    /*这一步算出具体key在散列的位置，是把idxhead取余为head，
    hash出来的key相同的概率之前算过了是非常低，除以kmax取余，
    就是给key在找位置，这个位置head也一部分key的散列属性*/

    tail = head;

    /*这个循环很逗，虽然我的调试里没有显示出它起了作用，
    但是我们前面说head虽然具有一定的散列属性，但它在这里并不具有唯一性，
    因为key本身非常大128bit，他能代表一个pbn的内容的唯一性，
    取余出的head却是一个小值，他不能唯一代表pbn，
    虽然你可以认为head:150702代表idxhead:0x24100420901436，
    并且在这个head位置保存它，但它却没有一唯一性，
    那么如果产生了取余后的head所在位置的*ptr已经存在了，
    就需要给这个key另找一个head来保存它，代码这里的做法是向后取，
    找到一个NULL或者被deleted掉的*ptr，在这个位置把key记录下来*/
    do {
        ptr = kvinram->store + entry_size * head;

    if (is_empty(ptr, entry_size) || is_deleted(ptr, entry_size)) {
        memcpy(ptr, key, kvs->ksize);
        memcpy(ptr + kvs->ksize, value, kvs->vsize);
        return 0;
    }

    head = next_head(head, kvinram->kmax);

    } while (head != tail);

    return -ENOSPC;
}
```

构建kvs_hash_pbn的方式与linear不太一致，其组织结构更加复杂。不过与kvs_lbn_pbn，其同样在两个组件内支持并提供一部分增删改查API。

4.3 IO写流程

① 处理的场景为no hash && no lbn，指代产出了新的文件和新的内容。

- 1- 计算hash，获得hash_pbn
- 2- 通过hash查找对应的hash_pbn_value，没有找到
- 3- 寻找bil是否具有lbn，也没找到
- 4- 执行handle_write_no_hash
- 处理流程为接来下就是需要把所需要的新的hash index和LBN都产生出来，并且记录。

```
static  int __handle_no_lbn_pbn(struct dedup_config *dc, struct bio *bio,
uint64_t lbn, u8 *hash)
```

② 处理的场景为no hash && has lbn，一般指代chunk IO得到了更新

- 1- 获得hash
- 2- 没有找到pbn
- 3- 找到了lbn
- 4- 到达此处理函数
- 处理流程：只要应用层修改的数据小于4k，比如我们的文档和代码工作。出现这种情况程序需要去将曾经的LBN-oldPBN给替换成新的LBN-newPBN，并且将hash index的refcount -1代表，我

们少了一次映射在这个已经存在的hash-PBN上，这时这个hash-PBN很可能成为了孤儿，没有LBN与之map，但是程序并不着急释放掉它，更希望它能够被情况③所利用。

```
static int __handle_has_lbn_pbn(struct dedup_config *dc,struct bio *bio,
uint64_t lbn, u8 *hash,u64 pbn_old)
```

③ 处理的场景是hash && no lbn，指代可以重复删除的数据

- 1- 获得hash
- 2- 找pbn找到
- 3- 找lbn没找到
- 4- 到达此执行函数
- 处理的流程是在这种情况下只需要添加一条LBN-PBN的映射关系，并将PBN的引用+1，
- 即可以节省一个BLOCK的空间。这就是最能够体现重删程序价值的地方，节省了实际空间。

```
static int __handle_no_lbn_pbn_with_hash(struct dedup_config *dc,struct bio
*bio, uint64_t lbn,
u64 pbn_this,struct lbn_pbn_value lbnpbn_value)
```

④ 该处理场景为hash && lba，但是不一定代表数据没有改变。 因为hash-PBN的存在，仅仅是某个物理块和request的内容一样，但是不代表曾经的LBA里面存 在内存也是这个内容，所以还需要判断曾经的LBA里面是否也是这个内容，那么需要比对一下LBN-PBN ?= hash-PBN的PBN-number）， 如果不一样就把PBN-old refcount -1和把PBN-new refcount+1，并且更新LBN-PBN。通常这种情况是：在一个系统下的批量文件同步/覆盖的， 这种情况副本A和副本B的引用不变，逻辑映射关系改变。

```
static int __handle_has_lbn_pbn_with_hash(struct dedup_config *dc,struct bio
*bio, uint64_t lbn,
u64 pbn_this,struct lbn_pbn_value lbnpbn_value)
```

5. 后续工作

原定的计划是先调试SmartSSD继而build Dmdedup && FileBench，但是后续出现了一些不可抗力，因此，并未功成。下周，打算build以下并测试以下Dm-dedup的基本功能以及相关TPS、Latency等指标。