File Integrity Checker for Dmdedup

Bharath Kumar Reddy Vangoor and Nikhil Mohan
Stony Brook University

Draft of 2015/04/13 20:21

Abstract

In this project we will implement a user-level file system type integrity checker and fixer for Dmdedup, a storage deduplication platform that operates at the block layer. Dmdedup includes a collection of data and metadata stored in on-disk data structures and there is a chance of data inconsistencies because of operator errors or hardware failures. So, implementing a user-level tool which scans the on-disk data structures and verifies their integrity is a way to detect data inconsistencies and fix them. Some of the file system checkers that were implemented earlier usually check only metadata, such as journal log but never checks the data itself, but the tool *dmdedup_check* which we are planning to implement for Dmdedup will scan the data and check against the metadata for better results. Moreover, many file system checkers (*fsck*) are implemented at the file systems level there by restricting them to run tools as file system dependent but the tool which we are coming up will be run at the block layer thereby making it file system independent. We will implement the file system integrity checker on two backend APIs that are developed in Dmdedup: a persistent Copy-on-Write B-tree and an on-disk hash table. Our initial focus is on Copy-on-Write B-tree as it is going in to mainline at this point.

1 Introduction

With the continuous increase of storage data [1], a file system integrity checks should always be performed in order to detect and fix any data inconsistencies. If a data inconsistency is discovered, corrective action must be taken. In General, file system checkers run automatically at boot time before the file system gets mounted or manually invoked depending upon the usability. Most of the checkers works directly on on-disk data structures stored on disk which are internal to the file systems and their implementations vary. Commonly used file system checkers provide a command-line interface to the users to invoke the checker and provide a mechanism to pass the arguments to the checker in order to by-pass some functionalities.

In our case, we develop a user-level integrity checker tool *dmdedup_check* for the Dmdedup [8] which scans the on-disk data structures and verifies their integrity. The first stage of tool will scan the metadata device and will report any data inconsistencies it finds. In the second stage we will enhance the tool to fix the errors that can be fixed and we name it as *dmdedup_repair*. The tools will take various options at command-line which include superblockcheckonly, help, version, ignorefatalerrors. We are planning to develop this in user space where it takes the data device and metadata device as arguments and checks the integrity among both.

2 Background

The file system checker that we discuss in the next sections will check the integrity of Dmdedup. In this section we will discuss Dmdedup, primary storage deduplication platform that operates at the block layer. Dmdedup requires two block devices to operate: one each for data and metadata. Data device store the actual user information; metadata device track the deduplication metadata (hash maps). Dmdedup provides a block interface reads and writes with specific sizes and offsets. With the help of this information every write to a Dmdedup instance is checked against previous data. If a duplicate is detected, the corresponding metadata is updated and no data is written. Conversely, a write of new content is passed to the data device and tracked in the metadata. The metadata contains two mappings LBN (Logical Block Number) to PBN (Physical Block Number) and HASH to PBN. There are other user space tools for checking on disk metadata that are being stored in thin-provisioning which are *thin_check*,*cache_check* [2–4] as part of their checker implementation, the tool only checks for the consistency of the B-tree's and they are not verifying (cross checking) the data (mappings) that has been storing in metadata with the data on physical disk. Following are the checks that have been implemented [5] in *thin_check*,*cache_check* for the metadata (mappings) that they are storing using persistent B-tree's :

1. Check whether number of entries in a B-tree node are divisible by 3.
2. Check whether number of entries in a B-tree node are greater than maximum entries of B-tree node or less than minimum entries of B-tree node (underflow check of B-tree).
3. Check whether the keys that are present in a node

are in sorting order or not.

4. Parent key value of a B-tree node should be less than or equal to the minimum key value of the child B-tree node
5. The last key value of the previous leaf node should be less than the first key value of the present leaf in a B-tree leaf node.

The above checks that are implemented checks only the B-tree orientation and doesn't cross check the values (key-value maps) that are stored against the physical device. As part of Dmdedup we store LBN-PBN, Hash-PBN, reference counts in metadata, so the *dmdedup_check* will cross verify the data in metadata device with physical device in contrast to the above checkers. The details are discussed in the next section.

3 Design

In this section we present the design for the Dmdedup file system type integrity checker *dmdedup_check* that checks data device blocks with the metadata mapping. The Dmdedup initially computes HASH of data chunks of fixed size using MD5 hash function and stores this HASH to PBN mapping in metadata, it also stores LBN to PBN mapping in the metadata and also the reference counts for the PBN. There are possibilities for the metadata and the data on the device to go inconsistent over the time due to hardware failures and various other reasons, so the *dmdedup_check* tool will check the integrity of metadata and the data present. Please see Figure 1 for design flow and following are the design steps:

1. *dmdedup_check* will be invoked from the command line with arguments data device, metadata device, options like superblockcheckonly, help, version.
2. *dmdedup_check* will scan the metadata device from the start i.e (super block) and check for the superblock consistency by comparing with *magic number*.
3. If superblock consistency check passes then tool will go ahead with LBN-PBN, HASH-PBN mapping, reference count consistencies check otherwise will fail and report the super block inconsistency.
4. From super block, we get the root node of LBN-PBN map and using B-tree walk API [6] parse the LBN-PBN values and store them in a array like data structure.
5. Using the values from array, for each LBN-PBN we read the actual data from physical data device using PBN value and compute the HASH. And using this HASH we lookup in the HASH-PBN map (from metadata) and get the corresponding PBN′ (from metadata).
6. If both PBN's (PBN from LBN to PBN map and PBN′ from HASH lookup) match then the data is consistent otherwise there is inconsistency in either LBN-PBN mapping or HASH-PBN and report.
7. For reference count consistency check, we calculate ref counts for each PBN from the array data structure (some thing like finding duplicates in a array) and then if any Hash for the PBN found we increment the ref count of PBN. Once we get the ref count for PBN we compare it against the ref count of PBN′ in the metadata and check if they are same or not and report.

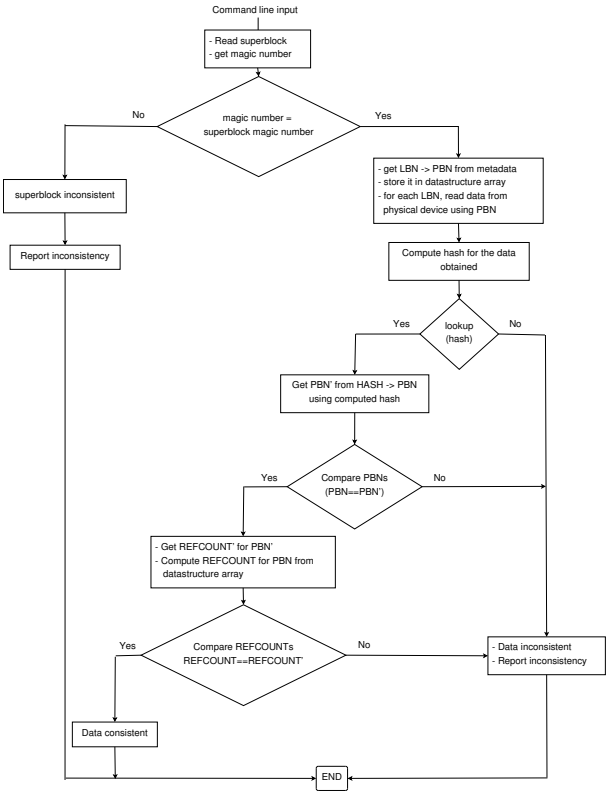

Figure 1: Dmdedup checker design

We will implement the above described design on the Copy-on-Write B-tree metadata backend. COW B-tree is a persistent data structure that is offered by the device mapper persistent library. The library [6] offers the following files which are used by the Dmdedup to implement the COW B-tree backend:

1. dm-block-manager: Provides access to the data on disk in fixed sized-blocks. There is a read/write locking interface to prevent concurrent accesses.
2. dm-transaction-manager: Provides access to blocks and enforces copy-on-write semantics.
3. dm-space-maps: On-disk data structures that keep track of reference counts of blocks. Also acts as the allocator of new blocks.
4. dm-btree: B-tree data structure to store the values which can have arbitrary size.

Some of the functions [5, 7] that will help in insertion,deletion and lookup into the metadata tables are as follows :

1. dm-btree-lookup: tries to find a key that matches exactly
2. dm-btree-insert: insert the key-value pair
3. dm-btree-remove: Removes the key if it is present
4. dm-btree value_type: contains size in bytes of each value and the function increment ref count ,decrement ref count and to check if both values are equal

Superblock Check. Initially in Dmdedup setup we are initializing superblock (refer to Figure 2) magic number of metadata device as 2126509509, therefore when we check for super block consistency we compare the magic number of metadata device with the above number and if both matches then super block is consistent. We also check for the block number of super block as it should be 0 always other wise we report saying superblock is inconsistent.

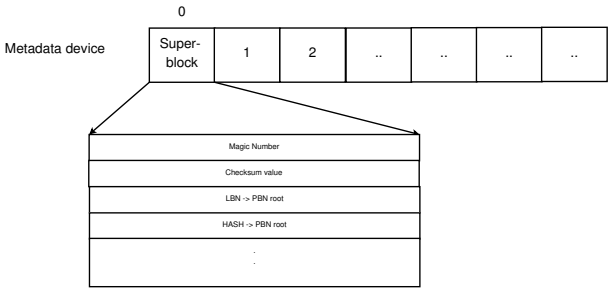

Figure 2: Super Block Lay out.

Mapping Checks. The superblock contains the LBN-PBN root node(refer to figure 2), HASH to PBN root node, with the help of the root nodes we will store the LBN to PBN values (refer to figure 3 for LBN to PBN map) in Array data structure and using this information we scan each LBN to PBN value and read data from physical device using the PBN value and compute the HASH. Using this HASH value we lookup in the HASH to PBN mapping table in metadata and if we get any PBN′ value then we compare both PBN's and if they are same then data is consistent otherwise we report saying data is inconsistent.

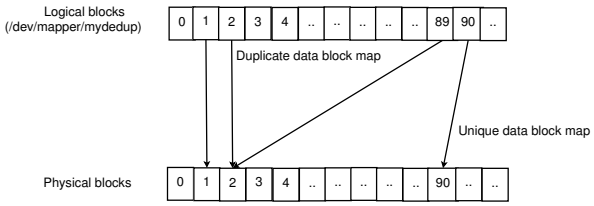

Figure 3: Block Layer layout.

Reference Count Checks. For reference count consistency check, we calculate ref counts for each PBN from the array data structure (some thing like finding duplicates in a array) and then if any Hash for the PBN found we increment the ref count of PBN. Once we get the ref count for PBN we compare it against the ref count of PBN′ in the metadata and check if they are same or not and report.

4 Implementation

As part of our implementation we have started with the super block verification, and we have added a new message command for the dmsetup which is "scanner", we added the following code snippet in "dm-dedup-target.c" under message function :

The following code parses the input command-line arguments and compares it with "scanner" string and if it matches it invokes the superblock check and after that read the LBN to PBN mappings that are stored as part of metadata and also their corresponding reference counters.

```
1
2         } else if (!strcasecmp(argv[0],
              "scanner")) {
3                 totalLBlocks =
                      dc->lblocks;
4                 printk(KERN_ALERT
                      "Scanner function
                      called\n");
5                 r = dc->mdops->
6                 check_super_block(dc->bmd);
7                 printk(KERN_ALERT
                      "check_super_block
                      function returned
                      with value : %d\n",r);
8                 printk(KERN_ALERT
                      "Scanning LBN to
                      PBN\n");
9                 printk(KERN_ALERT "Number
                      of LBNs :
                      %llu\n",totalLBlocks);
10                for(lbn =
                      0;lbn<=totalLBlocks;lbn++
                      ){
11  r = dc->kvs_lbn_pbn->
12     kvs_lookup(dc->kvs_lbn_pbn,
13     (void *)&lbn,
14     sizeof(lbn), (void *)&lbnpbn_value,
           &vsize);
15     if (r == 1){
16  printk(KERN_ALERT "Found Pbn : %llu for
       Lbn : %llu\n",lbnpbn_value.pbn,lbn);
17  printk(KERN_ALERT "Reference count of Pbn
       : %llu is : %d\n",
18  lbnpbn_value.pbn,
19  dc->mdops->get_refcount(dc->bmd,
20  lbnpbn_value.pbn));
21                    }
22                }
```

```
23          }
```

We added a new function to check the super block under Dmdedup COW B-tree functions. And also added the Super block magic number initialisation. we defined the superblock magic number to 2126509509, and assigning the same to the super block while writing initial superblock, changed the "metadata_ops" structure to include the new super block check function. Following is the code for respective changes in file "dm-dedup-cbt.c":

```
1  #define SUPERBLOCK_MAGIC 2126509509
2
3  static int
       write_initial_superblock(struct
       metadata *md)
4  {
5  #previous code
6  disk_super = dm_block_data(sblock);
7  disk_super->magic =
       cpu_to_le64(SUPERBLOCK_MAGIC);
8  #previous code
9  }
10
11 struct metadata_ops metadata_ops_cowbtree
       = {
12 #previous functions
13 .check_super_block =
       check_super_block_cowbtree,
14 #previous functions
15 }
16
17 static int
       check_super_block_cowbtree(struct
       metadata *md)
18 {
19     int r;
20     struct dm_block *sblock;
21     struct metadata_superblock
           *disk_super;
22     r = dm_bm_read_lock(md->meta_bm,
           METADATA_SUPERBLOCK_LOCATION,
23                 NULL, &sblock);
24     if (r)
25         return r;
26     disk_super = dm_block_data(sblock);
27
28     printk(KERN_ALERT "Magic Number
           obtained from SBlock : %llu\n",
29 le64_to_cpu(disk_super->magic));
30 if (le64_to_cpu(disk_super->magic) !=
       SUPERBLOCK_MAGIC) {
31 DMERR("superblock_check failed: magic
       obtained %llu: wanted
       %llu",le64_to_cpu(disk_super->magic)
32 ,(unsigned long long) SUPERBLOCK_MAGIC);
33         return 1;
34     }
35     dm_bm_unlock(sblock);
36     return r;
37 }
```

5 Results

We have implemented a new option "scanner" for Dmdedup message function but we will change it to user-level code. Following are the results:

Commands for setting up Dmdedup:

1. dd if=/dev/zero of=/dev/sdb1 bs=4096 count=1 oflag=direct
2. modprobe dm-dedup
3. echo "0 'blockdev –getsize /dev/sdc1' dedup /dev/sdb1 /dev/sdc1 4096 md5 cowbtree 1000" — dmsetup create mydedup

metadata device used : /dev/sdb1
data device device used : /dev/sdc1

Result 1. output of Scanner to check the super block consistency:

dmsetup message /dev/mapper/mydedup 0 scanner
OutPut(from dmesg) :
[66577.082648] Scanner function called
[66577.083603] Magic Number obtained from SBlock : 2126509509
[66577.083959] check_super_block function returned with value : 0

Result 2. Writing unique data and then running scanner to get the LBN to PBN mappings and reference counts.

Command to write 5 blocks (4096 bytes each) of random unique data :
dd if=/dev/urandom of=/dev/mapper/mydedup count=5 bs=4096 oflag=direct
5+0 records in
5+0 records out
20480 bytes (20 kB) copied, 0.043482 s, 471 kB/s
dmsetup message /dev/mapper/mydedup 0 scanner
OutPut(from dmesg) :
[67051.415766] Scanner function called
[67051.425122] Magic Number obtained from SBlock : 2126509509
[67051.425468] check_super_block function returned with value : 0
[67051.425811] Scanning LBN to PBN
[67051.426146] Number of LBNs : 6553344
[67051.426479] Found Pbn : 0 for Lbn : 0
[67051.426812] Reference count of Pbn : 0 is : 2
[67051.427143] Found Pbn : 1 for Lbn : 1
[67051.427474] Reference count of Pbn : 1 is : 2
[67051.427807] Found Pbn : 2 for Lbn : 2
[67051.428220] Reference count of Pbn : 2 is : 2

[67051.428555] Found Pbn : 3 for Lbn : 3
[67051.428892] Reference count of Pbn : 3 is : 2
[67051.429228] Found Pbn : 4 for Lbn : 4
[67051.429560] Reference count of Pbn : 4 is : 2

LBN to PBN map		
LBN Value	PBN Value	Reference Count
0	0	2
1	1	2
2	2	2
3	3	2
4	4	2

Result 3. Writing duplicate data and then running scanner to get the LBN to PBN mappings and reference counts.
Command to write 2 blocks (4096 bytes each) of data :
dd if=temp.txt of=/dev/mapper/mydedup count=2 bs=4096 oflag=direct
1+1 records in
1+1 records out
4810 bytes (4.8 kB) copied, 0.0196349 s, 245 kB/s
Command to write same 2 blocks of data but to different blocks using seek argument:
dd if=temp.txt of=/dev/mapper/mydedup seek=2 count=2 bs=4096 oflag=direct
1+1 records in
1+1 records out
4810 bytes (4.8 kB) copied, 0.006265 s, 768 kB/s
dmsetup message /dev/mapper/mydedup 0 scanner
[68747.124361] Scanner function called
[68747.133981] Magic Number obtained from SBlock : 2126509509
[68747.134310] check_super_block function returned with value : 0
[68747.134656] Scanning LBN to PBN
[68747.134981] Number of LBNs : 6553344
[68747.135361] Found Pbn : 0 for Lbn : 0
[68747.135685] Reference count of Pbn : 0 is : 3
[68747.136013] Found Pbn : 1 for Lbn : 1
[68747.136329] Reference count of Pbn : 1 is : 3
[68747.136640] Found Pbn : 0 for Lbn : 2
[68747.136945] Reference count of Pbn : 0 is : 3
[68747.137253] Found Pbn : 1 for Lbn : 3
[68747.137560] Reference count of Pbn : 1 is : 3

LBN to PBN map		
LBN Value	PBN Value	Reference Count
0	0	3
1	1	3
2	0	3
3	1	3

6 Evaluation Plan

In this section we discuss about how we are going to conclude that our design goals are met. We will create unit test cases that will test the consistency of superblock as well as mappings that are present in the metadata. Our unit test cases include writing huge amount of unique as well as duplicate data to the device and then running *dmdedup_check* on metadata device to check the consistency. Test cases include both positive and negative, details are as follows:

- Positive test cases, the test cases for which the *dmdedup_check* will return success that means the metadata device is consistent. We write these kind of test cases to pass sanity testing.
- Negative test cases, the test cases for which the *dmdedup_check* will return failure that means the metadata device is inconsistent. We write these kind of test cases to find any regressions. This kind of test cases are obtained by tweaking the metadata device from outside before running *dmdedup_check*...

Currently there is a device-mapper-test-suite in which the tests are divided up into suites, which are specific to a particular target. Many device mapper targets like thin-provisioning, cache are using this test suite. If tests for Dmdedup are added then new tests for *dmdedup_check* can be added. We will benchmark our system using time it is taking to scan the metadata device when the device is full.

7 Future Work

In this section we discuss the work that is to be done to complete the *dmdedup_check* and also possible enhancements for the tool. From the LBN to PBN map that we got from metadata we store it in a array data structure, and for each PBN value we get the data from physical device and compute its HASH. We lookup for this HASH in HASH to PBN map in metadata and compare the PBN values. And we also check the consistencies for reference counts. If we find any inconsistencies we report. A possible enhancement for the tool is to repair/fix the data inconsistencies which are reported by developing a new tool *dmdedup_repair*.

8 Acknowledgments

We thank Sonam Mandal, Jason Sun for helping us out in technical difficulties, spending time on long discussions during initial days explaining the project and during the design phase. We are thankful for having them as our mentors. We are very thankful to Prof. Erez Zadok for giving us the opportunity to work on this Project. We thank developers of tools *thin_check*, *cache_check* which are integrity checkers for other device mapper targets whose implementation gave us the initial design ideas.

References

[1] R. E. Bohn and J. E. Short. How much information? *2009 report on american consumers.* Available at http://hmi.ucsd.edu/pdf/HMI_2009_ConsumerReport_Dec9_2009.pdf.

[2] Heinz Mauelshagen Joe Thornber and Mike Snitzer. Documentation for cache. Available at https://git.kernel.org/cgit/linux/kernel/git/torvalds/linux.git/tree/Documentation/device-mapper/cache.txt.

[3] Heinz Mauelshagen Joe Thornber and Mike Snitzer. Documentation for cache policies. Available at https://git.kernel.org/cgit/linux/kernel/git/torvalds/linux.git/tree/Documentation/device-mapper/cache-policies.txt.

[4] Heinz Mauelshagen Joe Thornber and Mike Snitzer. Documentation for thin provisioning. Available at https://git.kernel.org/cgit/linux/kernel/git/torvalds/linux.git/tree/Documentation/device-mapper/thin-provisioning.txt.

[5] Heinz Mauelshagen Joe Thornber and Mike Snitzer. Thin provisioning tools code base. Available at https://github.com/jthornber/thin-provisioning-tools.

[6] Mike Snitzer Joe Thornber, Heinz Mauelshagen. Documentation for persistent data structures. Available at https://git.kernel.org/cgit/linux/kernel/git/torvalds/linux.git/tree/Documentation/device-mapper/persistent-data.txt.

[7] Vasily Tarasov. Dmdedup code base. Available at http://git.fsl.cs.sunysb.edu/?p=linux-dmdedup.git.

[8] Geoff Kuenning Sonam Mandal Karthikeyani Palanisami Philip Shilane Sagar Trehan Erez Zadok Vasily Tarasov, Deepak Jain. Dmdedup: Device mapper target for data deduplication. *2014 Ottawa Linux Symposium (OLS14).*