Dmdedup: Off-line Deduplication

Amit Khandelwal and Anshul Anshul
Stony Brook University

Draft of 2015/05/07 16:29

Abstract

Storage deduplication technologies are increasingly being deployed to reduce cost and increase space-efficiency in corporate data centers. Data deduplication can be implemented at the file system or block layer. In-line deduplication helps in the scenarios where the I/O operations should be minimal and disk-space is a constraint. Off-line deduplication is used when the requirement is to have fast read and write accesses. We extend the deduplication platform, DmDedup, to support off-line deduplication in this work. Off-line deduplication operates at the block level and is implemented in a modular way so that it can be used over the existing file systems and applications. A flexible backend API to support this is provided. There are two ways of triggering the off-line deduplication: by issuing user message-command and by kernel worker thread. Three different modes for performing off-line deduplication are implemented: full, incremental and bitmap. The system is implemented and evaluated on a disk Copy-On-Write B-tree backend.

1 Introduction

In the present era of technology, everyone's day-today life is becoming more engrossed and involved around computers and its applications. Every computer application has storage needs and thus data storage demands are increasing exponentially [4]. However, the decreasing price of the hardware is not able to compensate total storage costs. The concept of deduplication [5] (which removes duplicate data from storage media) helps in reducing the total amount of physical storage requirement. Deduplication has often been applied to backup database systems which contain duplicate data in huge amount. They represent the majority of enterprise data. In recent times, primary data storage requirement has grown substantially and hence the research to explore the primary storage deduplication has also grown [7, 13].

To facilitate research in primary-storage deduplication, a flexible and fully operational primary-storage deduplication system, Dmdedup, is developed and implemented in the Linux kernel by the File System Lab (FSL) at Stony Brook University. Deduplication can be implemented at the *application* layer, *file system* layer or the *block level* [2]. Dmdedup is designed as a stackable Linux kernel block device that operates at the same layer as software RAID and the Logical Volume Manager (LVM) [9, 13].

Data deduplication can be done in two ways; *in-line* as data is being written, or *off-line* after it has been written. In particular, in-line deduplication reduces the size of data while data is being written to the persistent storage whereas offline deduplication writes the data first and then reduces it by removing redundancy. Both methods have advantages and disadvantages and are used as per the requirement. In in-line deduplication, data is only passed and processed once and this processed data is instantaneously available for post storage tasks. However, it decreases the overall write throughput [11]. On the other hand, off-line deduplication backs up data faster, but requires more disk-space initially because full data is temporarily stored on physical media to speedup the writing process [10].

This project is aimed at performing off-line deduplication. As per the initial design, we create LBN→PBN mapping of all the writes (in aligned data chunks of configurable power-of-two size) in-line. Then, during the process of deduplication we create <hash value, PBN> entry for all the Physical blocks as present in each LBN→PBN mapping, check for duplicates, modify the <LBN, PBN> entry for duplicates and free-up the corresponding

PBN so that it can be reused. All these mappings are stored in the metadata device which is a separate disk. The deduplication process is triggered manually by issuing a Dmdedup command which in turn calls an API. The API is separately created to implement offline deduplication.

Dmdedup is designed in such a way that different metadata backend models (data structures for maintaining hash indexes, mappings, and reference counters) can be integrated through simple yet expressive API. There are three different backends implemented: an in-RAM hash table, an on-disk hash table, and a persistent Copy-on-Write B-tree [13, 14]. In this paper, we performed our experiments on Copy-on-Write B-Tree backend and include our detailed experimental results.

2 Design

In this section, we discuss the Dmdedup's off-line design mode, the device-mapper framework and present off-line Dmdedup's architecture and system components.

2.1 Classification

Deduplication needs to first identify the unit of data to work upon to find if the unit already exists on the disk. Deduplication can be implemented at the application, file system or block level. It means that it can be either file-aware (knows about file boundaries and works on files) or can work at block levels.

File system level deduplication [12] works at the file system level and hence compare files for duplicacy. Deduplication in the file system benefits many applications. Since the files can be large, it can adversely affect both deduplication ratio and throughput. The deduplication algorithm can be applied on full file or sub-file levels. Full file level duplicates easily can be eliminated by calculating single checksum of the complete file data and comparing it against existing checksums of the already-backed-up files. Its simple and fast, but the extent of deduplication is less, as this process does not address the problem of duplicate content found inside different files or data-sets. So, the two files might vary by just 10% and still exist on the disk as separate entities. There are three approaches for deduplication at the file system level: modifying an existing file system such as Ext3 [2]; (2) implementing a new deduplicating file system from scratch in kernel; or (3) creating a stackable deduplicating file system either in kernel. Each approach has flaws associated with them. Modifying existing file system may lead to instability; implementing a new file system from scratch is a laborious task and developing deduplication system in kernel is a difficult task.

Block level deduplication of data works on either fixed size of data or variable size of data [3, 6]. Unlike file system specific solutions, this can be used beneath any block based file system. The process of breaking up byte stream of data is popularly known as chunking and the blocks of data is called as *chunk* [8]. Block level deduplication allows to bypass a file systems limitations and can be used to design own block allocation policies. It is easier to implement and provides higher deduplication throughput. However, the deduplication ratio suffers if consecutive bytes stream of data is marginally different. Like any system, block level deduplication also has certain disadvantages. Variable length chunking mentioned above is difficult to implement. Also, important information about the file system context is lost. Extra data structures are required to be implemented to support block level data deduplication.

In-line vs Off-line deduplication. In in-line deduplication, the duplicates are eliminated as they appear. The deduplication process is triggered when data write request is sent to the disk. In other words, once the back-end storage device receives the data, the deduplication process will start the data content comparison and deletion tasks at the same time.

In off-line deduplication, the duplicates are stored on disk and eliminated later. Off-line deduplication means that the deletion of redundant data starts after the data is completely written to the disk. It might be triggered using a specific criteria or scheduled periodically by some heuristic.

Efficiency and Applicability. Each of the above techniques has its share of advantages and disadvantages. The former saves the bandwidth by avoiding repetitive reads and writes on the storage device. It is also used in the system which lacks idle time periods. However, it may be argued that computation of hash and other lookups may lead to increase in time for each write and thus negatively impacting the performance of system. The latter tend to wait

for system idle time to deduplicate previously written data. Since no operations are introduced within the write path; write latency is not affected, but reads remain fragmented [12]. Off-line deduplication is useful when the system requirement is to have fast read and write disk operations. It backs up data faster but negatively impacts as more disk-space is required temporarily until the deduplication is performed on the newly written data. Therefore, the database models where high quality and expensive hardwares are not used, but faster access operations are required, off-line deduplication is of utmost usage.

2.2 Device Mapper

The Device Mapper *(DM)* is Linux kernel's framework for mapping physical block devices onto higher-level *virtual block devices*. It can be thought of as a layer between file system and the disk, and can add features like *RAID*, *block caching* and *encryption*. It supports stackable block devices. A new block device can be created by building a *DM target* and registering it with the Operating System. Then, a user with root privileges can create corresponding target *instances*. Now, these instance acts as a regular block device for the upper layers (file system and applications). We chose DM framework for the implementation of deduplication due to its high performance and adaptability. DM operates in the kernel which further improves its performance.

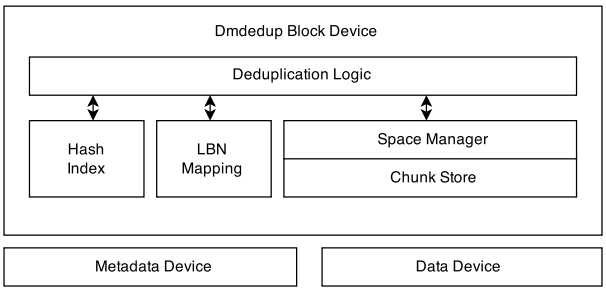

Figure 1: Dmdedup Architecture [13]

2.3 Dmdedup Components

Figure 1 portrays Dmdedup's building blocks and its typical setup. As every DM, Dmdedup is a stackable block device that operates on the top of physical devices such as disk drives, SDDs and RAIDs. Due to this versatility, Dmdedup provides high configurability which is of great importance in research and production domain.

Dmdedup uses two block devices to operate: a *data device* for storing the actual user data and a *metadata device* for storing the deduplication metadata. It can also be used when only one storage device is present in the system by simply creating two partitions on the device. Any combination of different type of storage medium can be used as a metadata and a data device. However, using fast and expensive media for metadata can provide great performance enhancement. Metadata device acts as a performance-critical factor as every I/O operation requires metadata access prior to actual I/O.

As per the availability of resources, we are using a 8GB HDD for storing the metadata information and a 25GB HDD for storing data. In off-line deduplication, data is directly written to data device with the creation of a <LBN, PBN> entry for every data chunk [8]. The main components of off-line Dmdedup are:

- off-line deduplication logic that deduplicates data based on metadata mappings.
- a *mapping* of Logical Block Numbers (LBNs) to Physical Block Numbers(PBNs), which is created during write operation in the metadata backend.
- a *mapping* of hash index to physical location of data (PBN), which is created during the deduplication process.
- space manager that tracks reference counts and reclaims unreferenced data.
- an API to trigger deduplication process which is based on user action.

2.4 Deduplication Step

Data Structures Used. The system maintains three data structures; LBN→PBN map, HASH→PBN map to store metadata information about the data in disk and space manager reference count structure to maintain the reference count of each PBN existing in the system.

The LBN→PBN map is used to store the LBN to PBN mapping. This is used to check a valid entry corresponding to the given LBN and to find the PBN associated with a given LBN. The second data structure, HASH→PBN stores the PBN corresponding to a given hash value. This hash value is generated from the data stored in that physical block.

Reference count for any PBN is increased with every entry of PBN in LBN→PBN map as well as in HASH→PBN map. So, every valid PBN has two refcounts. After running dedup on the disk if any PBN's refcount is one, it means there is an entry for this PBN in HASH→PBN map but none in LBN→PBN map. This corresponds to unreferenced block which should be reclaimed. All data structures are created during the intial setup phase of the deduplication.

The main off-line deduplication logic views both LBN→PBN and HASH→PBN mapping as an abstract key-value stores.

Metadata updates. Several cases must be handled for metadata updates. During the deduplication process, we traverse the whole disk by checking all LBNs. For each LBN, it is checked whether the target LBN exists in the LBN→PBN mapping or not. The lookup on the LBN→PBN returns the PBN$_{curr}$ associated with the current LBN. We then read the data corresponding to the PBN$_{curr}$ found from the LBN→PBN map. A hash value is generated for this data. This hash value is looked up into the HASH→PBN map. Two cases arise in this scenario. If the hash value doesn't exists in the HASH→PBN map, it means that we haven't previously seen such data. In this case, we make a new entry into the HASH→PBN map. If the hash value exists in the HASH→PBN map, then it means that there exists a block with the similar data. This condition may also lead to two cases. If the hash value is mapped to a different PBN$_{new}$ as compared to the PBN$_{curr}$ found in the LBN→PBN map, then we can deduplicate the data as both points to the same data. In that case, the reference count of the PBN$_{curr}$ is decremented and the reference count of the PBN$_{new}$ is incremented. An entry corresponding to the current LBN is also updated in the LBN→PBN map by changing it to the PBN$_{new}$ from the PBN$_{curr}$. In the other case when PBN$_{curr}$ and PBN$_{new}$ is same, it means that we have already deduplicated the data for this block. We need not to do anything in this case.

The flowchart in the Figure 2 shows the deduplication process. We first find a valid <LBN, PBN> mapping. Then, we read the data block corresponding to PBN synchronously and computes it hash. If the hash value is not found, then we go to the left side and insert a <hash value, PBN> entry into HASH→PBN map. In the right side, we check whether the PBN from the HASH→PBN map is similar to the PBN found from the LBN→PBN or not. Updates are made as defined above.

We also need to make sure that the space manager is aware of the above steps for the correct behavior. Decrementing the reference count of the PBN$_{curr}$ ensures that freed PBN can be reused later.

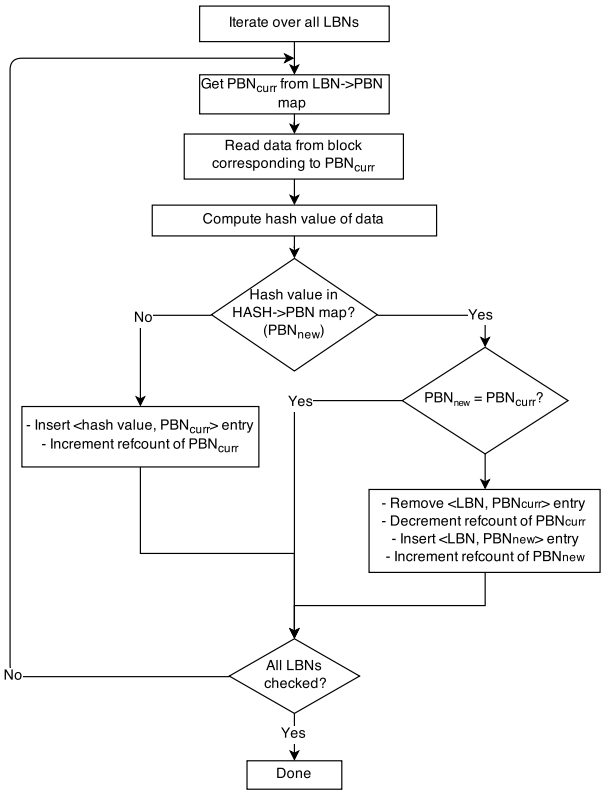

Figure 2: Offline Deduplication Process

The off-line deduplication functionality is implemented as an API similar to the write request implemented by V. Tarasov et. al [13]. This API is called by user action to perform the data deduplication.

2.5 Write Request Handling

In off-line deduplication, data write request is processed in the following steps:

- Every data write request first splits the data into aligned chunks with configurable size of power-of-two and generate smaller sub-requests. Each sub-request handle write operation for one data chunk.
- For each sub-request, a new <LBN, PBN>

mapping entry is created and inserted into LBN→PBN map in the metadata backend. PBN is chosen from a pool of free PBNs (physical disk blocks) available with respect to disk.
- Then, chunk's data is written to the data device.
- At the end, when all write sub-requests are processed, Dmdedup reports completion of the write operation.

2.6 Read Request Handling

Every incoming read request is processed in the following steps:

- Every read is split into aligned data chunks and queued for processing by worker threads.
- Then the chunk's LBN value is checked for existence in the LBN→PBN map in the metadata device.
- If such a mapping exists, data is read from the physical location using the PBN value.
- The read sub-requests (LBNs) for which no mapping exist are filled with zeroes. When all read requests are processed, Dmdedup reports completion of read operation. As such, there is no difference in implementation of read request handling with respect to the in-line deduplication.

Garbage collection. When the same LBNs are used to write some data (overwrites), Dmdedup does not reclaim PBNs (physical blocks) immediately nor does it remove the corresponding entries from the HASH→PBN map. However, reclaiming of the unreferenced device data blocks and PBNs must be eventually done. We implemented an offline garbage collector partially that iterates over all the <hash value, PBN> entries in the HASH→PBN map and reclaims those that are not referenced. The full working implementation is yet to be implemented.

2.7 Deduplication Variations

Incremental Vs Full. As off-line deduplication is CPU-intensive and time consuming task, it can efficiently work when CPU is idle and there is less user activity. But there is an effective alternative which doesn't require the mentioned constraints for better performance of deduplication . If we divide a hard disk into lets say x number of sub-partitions and run off-line deduplication on one partition at a time, it will be done fast (within short span of time) and without much hindrance to user interaction with the system. We implemented an *incremental* deduplication [1] mode which works on the same principle. We divide the full range of disk LBNs into 10 (can be configured to any number) sub-ranges and trigger Dmdedup on a single sub-range at a time.

Moreover, we implemented an interactive command-line API which allows user to choose between 'incremental' and 'full' deduplication mode. In case of full deduplication mode, the entire disk is scanned for data redundancy and deduplication is performed accordingly.

Bitmap. In a typical offline deduplication system, one has to scan the LBNs starting from zero to the full range. For each LBN, we need to check whether the <LBN, PBN> entry exists in the LBN→PBN map or not. If such a mapping exists, then we perform the expensive read operation to the corresponding PBN on the disk. A hash value of the data read is generated to check the existence in the HASH→PBN map. This step is only necessary if we have written new data to the disk after the last off-line deduplication process. In case of LBNs that has a valid mapping in the LBN→PBN map but already deduplicated in the earlier iteration, this leads to the wastage of effort in terms of read operations on the disk and the lookups in the LBN→PBN and HASH→PBN map which are expensive. This wastage of effort can be avoided using the below mentioned approach.

We make use of the system's RAM to store metadata about the LBNs that are written after the previous iteration of the off-line deduplication. We create in-memory bitmap data structure that stores this information. Each LBN has a bit corresponding to itself in this bitmap. Whenever a block is written to the disk, the corresponding bit is set in this bitmap. We check the existence of set bit in the bitmap for each block during the off-line deduplication process. Since the bitmap is maintained in the RAM, it is comparatively fast to detect new writes as compared to the lookups in the metadata maintained in the disk. This bitmap approach helps us to avoid checking the blocks that have been already deduplicated. This results into a faster completion

of the deduplication. The bitmap is created during the device mapper creation. This bitmap is never flushed to the disk and persists till the device mapper is running. The disadvantage of this approach is that once the system restarts, we need to perform the deduplication on all the LBNs again in the first iteration.

The approach works in the following way. During the off-line deduplication, for each LBN we check the presence of set bit corresponding to the LBN in the bitmap. If that bit is not set, we move forward to check the next LBN. When the bit is set, we deduplicate the LBN. At the end of each run of offline deduplication for a LBN, we clear the bit in the bitmap.

The advantages and disadvantages of each of the three modes of deduplication (incremental, full and bitmap-based) are pictorially mentioned in the Figure 3.

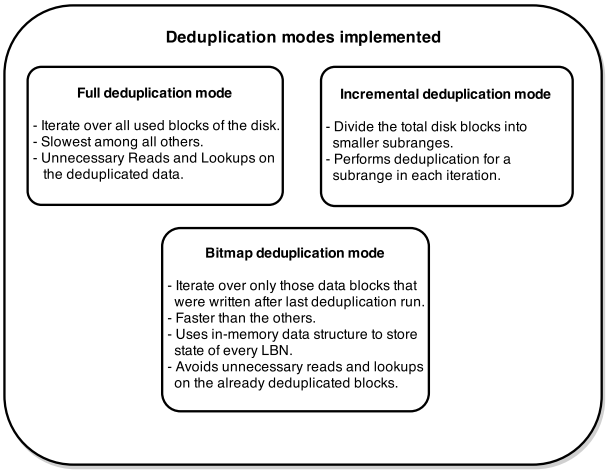

Figure 3: Offline Deduplication Modes

2.8 Deduplication Automation: Kernel Threads

In all previous modes of off-line deduplication that we discussed, user action is required to start the deduplication (by issuing message command). We implemented a kernel thread which acts as a background worker thread to trigger off-line deduplication based on some heuristics mentioned below. Because of the CPU-boundedness and expensiveness of deduplication process, heuristics are to be applied such that no other activity of high priority is hindered or any user should not face problems in using the system while deduplication is running. Keeping these points in mind, we applied the following criterias:-

- Created a kernel thread with highest *nice* value (lowest priority). This way, the kthread will only be scheduled when no other higher priority task is running (or the system is near to idle state).
- When the kthread is scheduled, it checks whether some significant amount of write operations have been to the disk after the last off-line deduplication. If the number is above a specific threshold value (tunable), the Dmdedup API to start off-line deduplication is triggered.
- Kthread worker function is implemented with an infinite loop which wakes-up in every 15 minutes (can be tuned to any value) and checks for other conditions. If all the conditions are fulfilled, deduplication is triggered. Once completed, the kthread again goes to sleep for 15 minutes.

3 Implementation

When constructing a Dmdedup instance, the user specifies the data and metadata devices, cache size, metadata backend type, hashing algorithm, in-line or off-line deduplication method, etc. We trigger off-line deduplication by issuing a message command or by using a kernel thread. We provide the deduplication statistics via the Dmdedup's STATUS ioctl. Primarily we provide the total number of blocks deduplicated during a run. The code was tested on the Linux 3.17. Dmdedup implemention has nearly 200 LOC.

4 Evaluation

In this section, we describe the goals of our evaluation followed by details and results of our experiments.

Evaluation Objectives. We evaluated the performance of various off-line Dmdedup modes. The time taken to perform deduplication on different workloads for all modes was measured. We experimented with the workloads involving all duplicates and unique data. We also measured the time taken on the workloads with different duplication ratios.

Experimental Setup. Deduplication system's performance is sensitive to data content. Having

same data in different blocks leads to high deduplication. We used Linux provided *DD* command-line utility to perform Unique and All-duplicate dataset I/O operations in our experiments. Each dataset represents different content and helped us to better analyze the system. *Unique dataset* test is performed using input file as */dev/urandom* with specific number of blocks. *All-duplicate dataset* test consist of data obtained from */dev/zero* as the input file.

Unique dataset resulted in poor system performance. It gave lowest deduplication ratio. In this case, we had to almost always make an entry into the HASH→PBN map. Inserting an entry in the HASH→PBN map is comparatively a slower process. CPU utilization was also high when the deduplication happens for the first time.

All-duplicates dataset test gave better system performance. When the deduplication was run for the first time, there were very few inserts into the HASH→PBN map. However, there were many updates in the LBN→PBN map. This slowed down the process and increased the CPU utilization. From the second iteration of deduplication, it was running comparatively faster as most of the mappings were already updated in the LBN→PBN map.

We measure the effectiveness of our system by counting the number of blocks that were deduplicated during the process. We count the deduplicated blocks for confirming the accuracy of results in different modes. We evaluated our system on COW B-Tree (CBT) metadata backend.

In order to verify the correctness of deduplication, we created a sample data file. In the file, we wrote first seven blocks with same data and next three blocks with different data. The file size was of exactly 10 blocks. After running the Dmdedup on it, the blocks 2-7 got deduplicated (as checked in kernel logs). The correctness of original data was checked by comparing the md5 checksum of the file before and after the deduplication process.

We performed our experiments on multiple linux kernel source codes to get more realistic results (practical dedup ratios). Our experimental setup involved performing deduplication on different number of kernel source codes (from 2.6.27.8 to 2.6.27.11 kernel version). The results were the number of blocks deduplicated and the dedup ratio in all cases.

	205MB	**410MB**	**819MB**	**1.6GB**
All Duplicates	10	20	42	90
All Uniques	11	19	50	105
50 % Duplicates 50 % Unique	10	19	41	85
80% Duplicates 20% Uniques	11	19	46	90

Table 1: Experimental results for bitmap dedup mode. 4 different workloads of 4 different sizes are used and results are mentioned in time (seconds) to perform each workload.

	205MB	**410MB**	**819MB**	**1.6GB**
All Duplicates	14	27	42	89
All Uniques	12	21	63	130
50 % Duplicates 50 % Unique	13	22	43	90
80% Duplicates 20% Uniques	12	21	42	88

Table 2: Experimental results for full dedup mode.

	1	**2**	**4**
Blocks deduplicated	0	1	1
Total blocks	19613	39226	58839
Dedup ratio	1.00	1.00	1.00

Table 3: Dedup ratio for 1, 2 and 4 different kernel source tarballs

Experimental Results. We performed testing by creating 4 different workloads: All-duplicates, All-Unique, 50% Duplicates - 50% Unique and 80% Duplicates-20% Uniques. These workloads were tested with 4 different file sizes (205MB, 410MB, 819MB and 1.6GB). We measured the time taken in each of the instance to compare the performance of deduplication system. We performed the experiments on bitmap and full deduplication mode.

The results for bitmap and full mode deduplication is listed in Table 1 and Table 2 respectively.

The results in terms of the number of blocks deduplicated when off-line deduplication is performed on 1-4 kernel source codes is listed in the Table 3. To confirm the validity of data in the original kernel code tarballs, we computed the md5 hash before the deduplication process and after the deduplication process. The values came out to be same.

5 Related Work

As mentioned in the design-classification section, deduplication can be implemented at the applica-

tion, file system or block level. Several studies and experiments have implemented deduplication into the existing file systems by modifying it or developed new file systems from scratch incorporating deduplication. Bereft of having some benefits of deduplicating at the file system level, there are major drawbacks involved in these approaches. Modifying an existing file system involves a significant amount of work and may lead to unstability and question its reliability. Developing new file system demands massive efforts and time. Moreover, the resulting file systems become limited to system's architecture. Dmdedup is implemented at the block level which makes it file system or architecture independent.

Some deduplication prototypes were developed for specific tasks and lack research and experimental versatility [15]. In contrast, Dmdedup is developed for experimentation with design that is flexible and modular. It provides API interfaces which can easily be used by users who wants to use their own backends. In addition, its modular design makes it easy to try different workloads [13].

6 Conclusions

Deduplication is a hot research topic in big database management systems. Depending on the applicability of the system, deduplication works in different modes. Off-line deduplication is particularly useful in the following scenarios; (1) disk operations should be very fast at the time of use (real time queries on big databases); (2) extra storage-space is available for temporarily storage of all data; (3) the system is idle for some part of the day; (4) CPU speed is not very fast for carrying out in-line deduplication. Because of such high practical usage of off-line mode, efficient implementation is utmost important.

We designed and implemented off-line Dmdedup, a versatile and a practical deduplication block device that can be used by regular users as well as reseachers. We used Copy-on-Write B-Tree metadata backend which provides faster lookup and modification. We started by implementing typical off-line deduplication system that scans the whole disk in each iteration. This resulted in the slow operation of the deduplication process and produced unnecessary lookups and reads on the metadata and data device. We then added the functionality of incremental and full deduplication mode to the Dmdedup to solve the former problem mentioned in the previous line. In order to counter the problem of unnecessary lookups and reads, we implemented in-ram bitmap functionality that only deduplicates those blocks that are written after the last dedup run. We extended this to run on a kernel worker thread that wakes up at regular intervals to perform deduplication. We also provided the user with an API to initiate the deduplication process.

Future Work. In case of a crash, the off-line deduplication metadata backends can be reconstructed to the state before the crash. This is because we have a dedicated metadata device (persistent media) for storing the metadata, but appropriate remapping is to be done to reconstruct them. However, the mentioned functionality is not implemented. Our future work is to create APIs to reconstruct or recover the metadata structures to resume the deduplication from the state just before the crash.

At present, the garbage collection functionality is based on iterating over all the entries in HASH→PBN map and reclaim the PBNs which have refcount value of 1. However, this API is still not implemented properly and requires some stress testing for proper working.

In the present state of project, we have implemented bitmap and incremental modes for off-line deduplication. These modes are working properly with different kernel binaries (different compile-time flags are used to enable each mode). Since both modes have their individual benefits, we can combine them to enhance the efficiency of off-line deduplication. Our future task is to create a mode which incrementally scans the disk based on the bitmap indexes for each LBN and runs deduplication process on blocks which are dirtied after the last DmDedup run.

Acknowledgments. We express our sincere thanks to Ms. Sonam Mandal who helped us in each and every step throughout the project progress. We faced problems and every time received genuine help and guideline from her. We also thank Mr. Vasily Tarasov who helped us in the starting phase to create a preliminary design algorithm for off-line deduplication. In addition, we thank Prof. Erez Zadok who helped us in clearing doubts regarding

the implementation at various places. The novel ideas from all of the mentioned persons helped us in making our way to achieve the goals of the project.

References

[1] Acronis. When deduplication is most effective. *http://www.acronis.com/en-us/support/documentation/ABR11/index.html#18528.html*.

[2] A.More, Z.Shaikh, and V.Salve. Dext3:blocklevel inline deduplication for ext3 file system. *Proceedings of the 2014 Ottawa Linux Symposium (OLS12)*, 2012.

[3] Giridhar Appaji Nag Yasa and P. C. Nagesh. Space savings and design considerations in variable length deduplication. *SIGOPS Oper. Syst. Rev.*, 46(3):57–64, December 2012.

[4] R. E. Bohn and J. E. Short. How much information? 2009 report on american consumers. *https://git.kernel.org/cgit/linux/kernel/git/torvalds/linux.git/tree/Documentation/device-mapper/persistent-data.txt*, 2009.

[5] Druva. Data deduplication. *http://en.wikipedia.org/wiki/Data_deduplication*, 2009.

[6] Ravindra Mahabaleshwar. Effective data deduplication implementation. available at http://www.tcs.com/SiteCollectionDocuments/White

[7] D. Meister and A. Brinkmann. dedupv1: Improving deduplication throughput using solid state drives (ssd). In *Proceedings of the MSST Conference*, 2010.

[8] D. Meyer and W. Bolosky. A study of practical deduplication. In *Proceedings of the FAST Conference*, 2011.

[9] Opendedup. SDFS. *www.opendedup.org*, January 2012.

[10] Dave Raffo. Inline vs. offline deduplication. *http://searchdatabackup.techtarget.com/tutorial/Inline-deduplication-vs-post-processing-Data-dedup*.

[11] Jaspreet Singh. Understanding data deduplication. *http://www.druva.com/blog/understanding-data-deduplication*.

[12] Kiran Srinivasan, Tim Bisson, Garth Goodson, and Kaladhar Voruganti. idedup: Latency-aware, inline data deduplication for primary storage. In *FAST'12: Proceedings of the 10th conference on USENIX Conference on File and Storage Technologies*, pages 1–14, 2012.

[13] V. Tarasov, D Jain, G Kuenning, S Mandal, K Palanisami, P Shilane, S Trehan, and E Zadok. Dmdedup: Device mapper target for data deduplication. In *Proceedings of the 2014 Ottawa Linux Symposium (OLS14)*, July 2014.

[14] Joe Thornber. Persistent-data library. *http://hmi.ucsd.edu/pdf/HMI_2009_ConsumerReport_Dec9_2009.pdf*, December 2011.

[15] A. Wildani, E. Miller, , and O. Rodeh. Hands: A heuristically arranged non-backup in-line deduplication system. In *Proceedings of the International Conference on Data Engineering (ICDE)*, 2013.