Project Report
for
Deduplication

Submitted By

Venkata Ramana Sai Malladi (vmallad)

Hari Padmakar Kancharlapalli (hpkancha)

Abstract

In the deduplication process, unique blocks of data, or byte patterns, are identified and stored during a process of analysis. As the analysis continues, other chunks are compared to the stored copy and whenever a match occurs, the redundant chunk is replaced with a small reference that points to the stored chunk. In our implementation, each file is divided into blocks, which are then hashed and noted in a block hash database. In the event that a block is written whose hash is already in the database, this block does not have to be stored - a pointer to the previously existing block will suffice.

Introduction

In applications where many copies of very similar or even identical data are stored on a single disk, same byte pattern may occur dozens, hundreds, or even thousands of times. With deduplication, the amount of data that must be stored or transferred can be greatly reduced.

While compression is a great way to save space in general, it is often prohibitively complex or resource intensive to do on live read/write data, especially at the file system level. Instead, deduplication can be leveraged with much lower overhead.

One typical example of such an application is a web-based file hosting system, where several users upload the same files. Using deduplication, we store only one copy of each such file and all subsequent instances are referenced back to the saved copy. Consider a 1MB file uploaded by 100 users; we save 99MB of storage space using this technique.

We are building a FUSE-based file system with custom implementations for make, open, close, read, write, getattr, access, rmdir, mknode, unlink, release and truncate operations. We divide the file into fixed-size blocks as opposed to variable-sized blocks. We compute the hash values of each block using SHA-1 algorithm and compare them with the existing hash values. This could be done each time a new file is written (eager) or it could be done periodically (lazy). We are using the eager approach. The trade-offs involved will be discussed in detail in later sections.

Technical Sections

Implementing FUSE-based file system:

Provide custom implementation for the file system functions like make, open, close, read, write, getattr, access, rmdir, mkdir, mknode, unlink and truncate operations.

FUSE is a three-part system. The first of those parts is a kernel module which hooks into the VFS code and looks like a filesystem module. It also implements a special-purpose device which can be opened by a user-space process. It then spends its time accepting filesystem requests, translating them into its own protocol, and sending them out via the device interface. Responses to requests come back from user space via the FUSE device, and are translated back into the form expected by the kernel.

FUSE lets you develop a fully functional filesystem that has a simple API library, can be accessed by non-privileged users, and provides a secure implementation. And, to top it all off, FUSE has a proven track record of stability.

With FUSE, you can develop a filesystem as executable binaries that are linked to the FUSE libraries -- in other words, this filesystem framework doesn't require you to learn filesystem internals or kernel module programming.

In user space, FUSE implements a library which manages communications with the kernel module. It accepts filesystem requests from the FUSE device and translates them into a set of function calls which look similar (but not identical) to the kernel's VFS interface. These functions have names like open(), read(), write(), rename(), symlink(), etc. For example, when the user requests a write operation on a file in a file system mounted with FUSE, this request is captured by the write function we implement and the kernel's write is substituted by our custom implementation.

Finally, there is a user-supplied component which actually implements the filesystem of interest. It fills a fuse_operations structure with pointers to its functions which implement the required operations in whatever way makes sense.[1]

In order to implement the file system created, we need to select a mount point. A mount point is a directory in the currently accessible filesystem on which our custom filesystem is "mounted" (i.e., logically attached). After mounting the directory, the mount point becomes the root directory of the newly added filesystem. This filesystem becomes accessible from that directory. Any original contents of that directory become invisible and inaccessible until the filesystem is unmounted (i.e., detached from the main filesystem).

Divide the file into blocks:

The input file has to be divided into blocks. This can be done in two ways - fixed-size or variable-sized blocks. Though fixed-sized blocks fail on a common use case: when a character is added at the start of one of the deduplicated files, none of the blocks will match; the major concern is to deduplicate files with the same content. Hence, we plan to use fixed-size blocks.

Fixed Block deduplication involves determining a block size and segmenting files into those block sizes. Then, those blocks are what are stored in the storage subsystem[2]. We plan to divide the file into blocks of size 4096 bytes each.

[1] FUSE - implementing filesystems in userspace, http://lwn.net/Articles/68104/

[2] Fixed Block vs Variable Block Deduplication – A Quick Primer, http://virtualbill.wordpress.com/2011/02/24/fixed-block-vs-variable-block-deduplication-a-quick-primer/

The code for dividing the file into blocks is written in write function. Whenever a file is written, the entire contents of it are stored in a buffer and this buffer is divided into blocks of size 4096 bytes.

Map the blocks to hash values:
For each of the blocks obtained in the above task, we compute the hash value. Once the hash value is obtained, we store this block in a file named with its hash value. All such blocks are stored in a folder named ‘datastore’. If a file with the same name already exists, it means that the block already exists. Hence, we ignore this block. This could be done each time a new file is written (eager) or it could be done periodically (lazy).

In the eager approach, deduplication is done whenever a block comes in, i.e., while the block is in RAM, the program runs and decides if it has seen this block before. If it has not seen this block, it writes to disk. In this method, most of the work is done in RAM and thus decreases the I/O operations. The advantage is that it works with the data only one time. The drawback is that, depending on the implementation, it could slow down the incoming backup.

In the lazy approach, each block of data that comes in is directly written to the disk. Then, there is another process running periodically on each of the blocks in the disk and then decides if it has seen that block before. If it has seen, it deletes this block and updates the reference to point to the similar block. If no, it does nothing. The advantage to this is that you can apply more parallel processes (and processors) to the problem, whereas the eager approach can apply only one process per backup stream. The disadvantage is that the data is written and read more than once, and the multiple reads and writes could cause contention for disk. In addition, the lazy approach requires slightly more disk than an eager approach because a lazy system must have enough disk to hold the latest set of backups before they're deduplicated.

We implement the former approach (eager) in order to maintain optimum space efficiency and decrease the number of I/O operations.

Hash function:
We plan to use the SHA-1 hash function as opposed to MD5 algorithm. SHA-1 takes a message of length at most 2^{64} bits and produces a 160-bit output. It is similar to the MD5 message digest function, but it is a little slower to execute and presumably more secure. Also, SHA1 is more secure that MD5.

The ability to forcibly create a hash collision means absolutely nothing in the context of deduplication. What matters is the chance that two random chunks would have a hash collision. SHA-1 produces a 160-bit hash. That is, every message hashes down to a 160-bit number. Given that there are an infinite number of messages that hash to each possible value, there are an infinite number of possible collisions. But because the number of possible hashes is so large, the odds of finding one by chance is negligibly small (one in 2^{80}, to be exact). If you hashed 2^{80} random messages, you'd find one pair that hashed to the same value.[3]

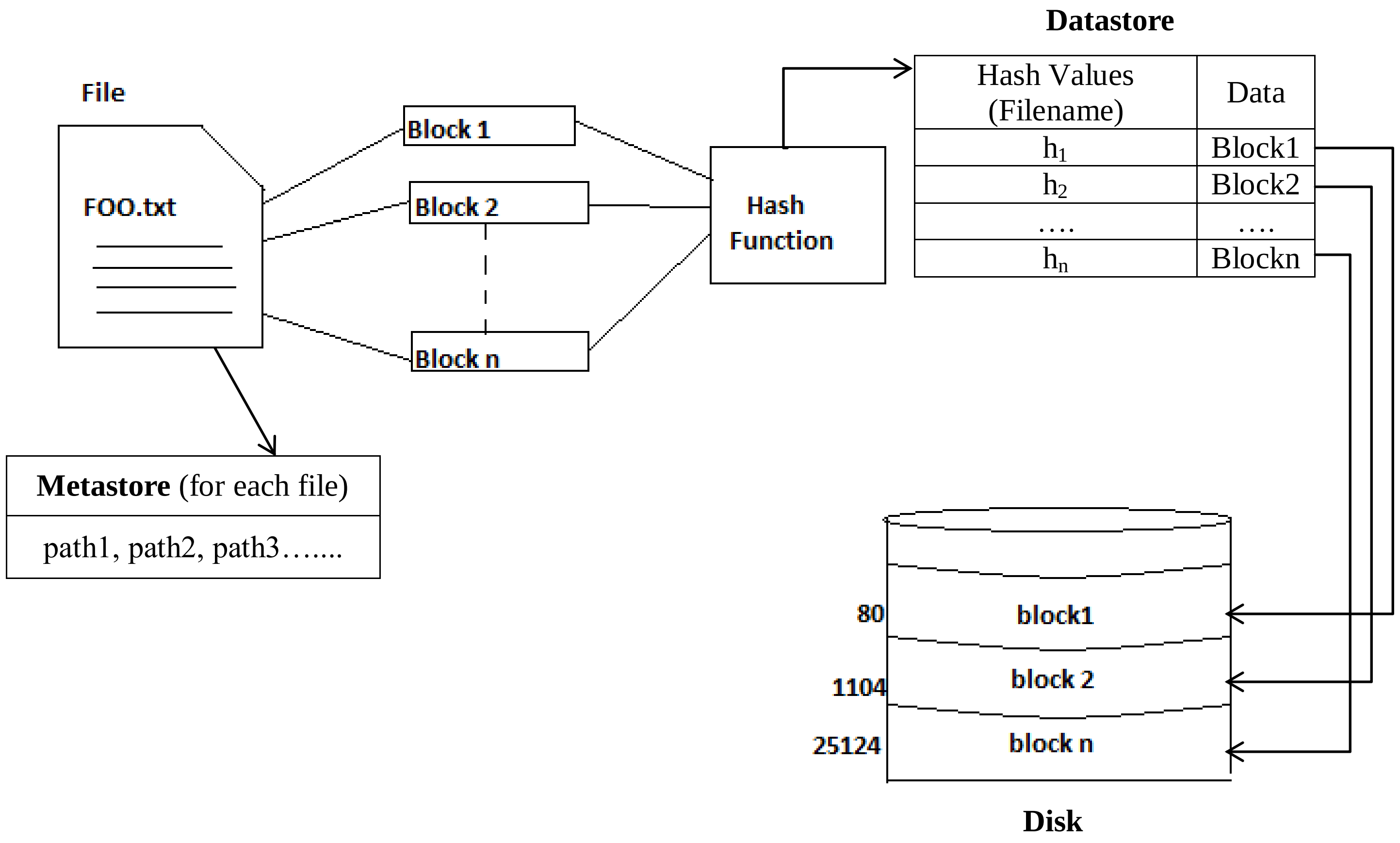
[3] B. Schneier, Cryptanalysis of SHA-1, http://www.schneier.com/blog/archives/2005/02/cryptanalysis_o.html

Consider the case that no two blocks have the same data, and 200 zetabytes of data is written each day, the probability that 2 blocks would have the same hash value is 0.0000403. Thus, it would take approximately 68 years before we find two different blocks mapping to the same hash value[4]. Thus, if two blocks have the same hash value, SHA-1 is robust enough to conclude that they both have the same information.

Metadata about the blocks:

Along with datastore, we maintain a folder named ‘metastore’. This contains one entry per each file created following the same directory hierarchy as that of the original file. In this file, we store the absolute paths to the blocks corresponding to the original file in the same order. After the hash values along with the corresponding blocks for each file are stored in the datastore, we then store their paths in the corresponding file in the metastore.

When a file is read by the user, we open the corresponding file from the metastore, open blocks from each of the stored absolute paths, append the contents into a single buffer and return the entire contents as a single file. If the user modifies the file, the hash values are computed again and are stored accordingly.



[4] http://en.wikipedia.org/wiki/Birthday_attack

Related work:

Content similarity in both memory and archival storage have been investigated in the literature. Memory deduplication has been explored before aiming to eliminate duplicate in-memory content both within and across virtual machines sharing a physical host. Data deduplication in archival storage has gained importance in both the research and industry communities. Current research on data deduplication uses several techniques to optimize the I/O overheads incurred due to data duplication. The initial approach was the use of an in-memory content-addressed index of data to speed-up lookups for duplicate content. Similar content-addressed caches were used in some data backup solutions. Content-based caching in I/O Deduplication is also implemented in some cases.[5]

Also, in archival storage systems, there is a huge amount of duplicate data or redundant data, which occupy significant extra equipments and power consumptions, largely lowering down resources utilization (such as the network bandwidth and storage) and imposing extra burden on management as the scale increases. So data de-duplication, the goal of which is to minimize the duplicate data in the inter-file level, has been receiving broad attention both in academic and industry in recent years. There are a few proposals on semantic data deduplication , which makes use of the semantic information in the I/O path (such as file type, file format, application hints and filesystem metadata) of the archival files to direct the dividing a file into semantic chunks (SC). While the main goal of SDD is to maximally reduce the inter-file level duplications, directly storing variable SCes into disks will result in a lot of fragments and involve a high percentage of random disk accesses, which is very inefficient. So an efficient data storage scheme is also designed and implemented: SCes are further packaged into fixed sized Objects, which are actually the storage units in the storage devices, so as to speed up the I/O performance as well as ease the data management.[6]

There is also a concept called stream deduplication by breaking up an incoming stream into relatively large segments and deduplicating each segment against only a few of the most similar previous segments. To identify similar segments, we use sampling and a sparse index. Then, a small portion of the chunks in the stream as samples is chosen; and a sparse index maps these samples to the existing segments in which they occur. Thus, avoiding the need for a full chunk index. Since only the sampled chunks' hashes are kept in RAM and the sampling rate is low, the RAM to disk ratio is dramatically reduced for effective deduplication. At the same time, only a few seeks are required per segment so the chunk-lookup disk bottleneck is avoided.[7]
In addition, work has taken place to determine how small changes affect the deduplication ratio for different file types on a microscopic level for chunking approaches and delta encoding. An intuitive assumption is that small semantic changes on documents cause only small modifications in the binary representation of files, which would imply a high ratio of deduplication. This work

[5] R. Koller, R. Rangaswami, *I/O Deduplication: Utilizing content similarity to improve I/O performance*, in ACM

[6] C. Liu, D. Ju, Y. Gu, Y. Zhang, D. Wang. DHC Du, *Semantic Data De-duplication for archival storage systems*.

[7] M. Lillibridge, K Eshghi, D Bhagwat, UCS Cruz, V Deolalikar, G Trezise, *Sparse indexing: large scale, inline deduplication using sampling and locality*, USENIX Association Berkeley.

shows that this assumption is not valid for many important file types and that application-specific chunking can help to further decrease storage capacity demands.[8]

[8] D Meister, A Brinkmann; *Multi-level comparison of data deduplication in a backup scenario*, SYSTOR '09 Proceedings of SYSTOR 2009