IN4120 — File storage deduplication

Introduction

Both as developers and as users, it's not uncommon to run out of storage on both our desktop machines and on our servers. The need for persistent storage has grown increasingly complex, as different applications have unique needs for file and data storage. Furthermore, users and applications will frequently take copies or backups during migration or as fail-safes, which are often forgotten and left orphaned. In larger data cluster systems, we can also observe different users are storing identical files in different locations, creating unnecessary redundancy.

Having duplicate files may seem harmless at first. It's intuitive to assume that duplicate and unused files are merely occupying space on our persistent storage device, and nothing more. We might think we could just delete a few files or directories when space is low, and move on with our days, right? However, this is a naïve understanding.

Why bother?

Duplicate files, while benign on their own, introduce inefficiencies beyond simply occupying space. They increase the workload required for other processes like backups, indexing, virus detection..., essentially doubling the work for no added benefit. This problem propagates to networked systems, where bandwidth is not free, and can, in certain situations, increase the risk of data loss on filesystem formats with suboptimal transaction guarantees.

Moreover, duplicates complicate data management on two levels. First, from a systems perspective, having multiple versions of the same (or similar) data within a single system can make it challenging to identify which version is the original. Without taking extra care, one might accidentally override or delete the wrong version. On top of this, even if we can identify which version is the proper one, this requires extra overhead which slows down our systems (or brains).

At a lower layer, duplicates hinder compression. When a dataset contains redundant data, compression algorithms may struggle to find an optimal solution. Instead, it will waste its entropy resources to target deduplication tasks, and have little wiggle room to fine-tune our actual data. This issue extends into decompression too, since our index will spend more time dereferencing copies of the same data.

From a more technical perspective, many operating systems and filesystems leverage free storage space as a temporary resource to speed up other tasks. All major operating systems, for example, rely on swap space to supply more memory to the user, and certain filesystems will use its idle time to tidy datablocks in preparation for future use. In the past, this would have included defragmentation. Moreover, copy-on-write (CoW) mechanisms in certain filesystems may cause write-operations to introduce completely unnecessary latency.

This paper will explore methods for identifying, mitigating and eliminating duplicate files and directories. It is not in the scope of this paper to consider similarity between files, but we will briefly touch on the similarity between directories. We will also discuss the limitations of different approaches, and conclude with ideas for a paradigmatic shift which could potentially prevent duplication in the first place.

Terms and glossary

While we expect the reader of this paper to be vaguely familiar with most terms used in this paper, some of them are often used ambiguously. Having a consistent understanding of what we're referring to may be useful.

- **data**: A piece of information (text or binary)
- **dataset**: A collection of unique pieces of data
- **file**: An object to be stored on disk, containing data encapsulated by some metainfo.
- **filesystem, fs**: The format used to store our files. Examples include `ext4`, `btrfs`, `zfs`.
- **identical**: Whether there is no difference between the data in a file
- **metainfo**: Additional details related to a file, also stored on disk
- **system**: The overarching combination of tools and automated tasks we use to achieve our goals

Techniques

In order to find duplicates in our dataset, we must first be able to tell if two files are identical or not. While defining what it means for two files to be identical is easy enough, it is not obvious how one should compare this, especially at scale.

A first approach would load both files into memory, and then sequentially compare the bytes at each position of the files until we either find a dissimilarity, or until we reach the end-of-file marker. This works well enough in the simple case for small files, but as soon as we have larger files we would have to load chunks of each file to be able to fit it into memory, increasing complexity.

Building an index

Since we will rarely have to compare a single file with only one other file, a better approach might be to calculate the hash of the files instead, and treat each path as a posting in an inverted index using the hash as our index. The downside of this approach is that it requires us to actually compute the hash of all the files on our system, even the large ones which are unlikely to match anything.

Luckily, we don't need to measure the hashes of all files. We can look at other bits and pieces to check if the potential for overlap even exists. A good filter would be to compare the size of the files, since files can only be identical if they're equally big, and large files (which are the slow ones to compute the hash for!) rarely have the exact same byte count, unless we're working under special circumstances in which a less generic deduplication algorithm may be worth the investment.

For medium-sized files, we could additionally compute a hash for the first fixed-size chunk of the data. Finding a threshold for how big this chunk should be may depend on the filesystem, specific operating system parameters and specific hardware buffer/bus sizes. Luckily, finding a threshold for what counts as a small file, what counts as a medium file, and what counts as a big file would emerge by itself when our algorithm finds the need to compute the "bigger step".

While not suitable for all scenarios, there are many cases where we could also use the basename of the path as a filter.

Comparing directories

In order to compare directories, we could do something similar to what we did to build the inverted index of files, but using the hashes of the children as our data instead. This operation is destructive, since we lose information about which files were included, and we won't be able to distinguish a directory containing most but not all files another directory had. Ignoring this feature lets us cut some corners in how to optimise our algorithm, though.

Depending on the hashes of the contents of a directory requires us to recursively calculate the hashes of all the directories which are part of it. Similar to what we did with files, we can skip a lot of this work by first checking the metadata of our directory: checking the number of files and directories should be much faster than computing the hashes of all individual items. Furthermore, this lets us approach the problem in a top-bottom fashion.

To continue optimising, if we realise the number of files and subdirectories is equivalent, we could start by comparing only the direct children files, and only if that also is equivalent start recursing. In summary, a breadth-first-search approach here will quickly rule out most candidates. That said, our goal here is not to check for membership, but rather to find duplicates across the whole system. We will inevitably open that barrel at some point, so might as well get it over with. This leads us into our next topic.

Tailored data structures

A Merkle tree is a specific data structure used in many contexts where data consistency is meaningful. Examples of this include version control systems such as Git and Mercurial, networked file transfer protocols such as BitTorrent and IPFS, data-oriented file systems like btrfs and zfs, transaction technologies like cryptocurrencies, certificates, and many more.

This type of tree is also referred to as a hash tree, namely because of how it uses the hashes of each node as part of its structure. A parent node computes its own hash by hashing the sum of all its children nodes. This effectively means that if a child is modified, the hash of all its superior nodes will need to update its hashes too, in a bottom-up approach.

This approach would be ideal if hashing the contents of a file could be done in minimal and constant time. Potentially, hashes for files were computed by the filesystem intrinsically during write-operations, and the invalidation of hashes would be propagated up the filetree automatically. The reason why we invalidate a hash instead of recomputing it here is to avoid recomputing a directory hash over and over

again when nobody is actually going to read it. Furthermore, if we're walking up the tree, and find a directory which was already invalidated, we can stop early.

Probabilistic filtering

A Bloom filter is essentially a data structure which can tell us whether a file definitely does not exist in a directory, or whether there is a chance it *may* be part of it. In our case, this basically tells us whether we should keep recursing into subdirectories or not.

This type of filter has a fixed-size bit array, where each index into its array is initially set to 0. There is also a set of *k* unique hash functions (uniformly distributed and independent). To update the bloom filter to capture a new file, we will need to hash some attribute of the file for each hashing function of our bloom array. For each returned value, we will set the index of the bloom filter array to one. As a consequence, if a file is part of the underlying directory the bloom filter must return 1 at these same positions when testing.

Since a directory is likely going to have very many input files during deep recursion, we need to consider a combination of hashing functions which will not set too many bits high. It is suggested that for a given number of members n, and a bit length of m, our number of hash functions should be $k=\ln(2)\cdot m/n$. My home directory has 2.75M files. My system itself has 6.13M files. Most files seem to lay between 5 and 10 nodes deep (see figure below). From these numbers, and picking a false positive rate of 0.5%, we get an m of around 28M bits, which works out 2.7T bits, roughly 326GiB, only for depth=6. That's too costly.

```
~
doas python depthcount.py /
Depth      Files       Dirs
    1          0         19
    2        941      1_009
    3     43_405      8_330
    4    184_602     16_679
    5    321_794     50_555
    6  2_549_515     97_574
    7    681_773     54_903
    8    562_982     53_385
    9    428_951     87_255
   10    540_964     56_997
   11    439_686     51_933
   12    358_841     51_009
   13    334_918     40_280
   14    318_916     27_913
   15    208_358     17_960
   16    163_164     13_396
   17     80_622     10_462
   18     56_621      5_413
   19     22_449      3_396
   20     20_684      4_879
   21     10_604      2_404
   22      3_768      1_514
   23      2_122        163
   24        389         50
   25         74         24
   26         20          4
   27          4          0
~
```

```
~
python depthcount.py ~
Depth      Files       Dirs
    1        128         62
    2        258        420
    3      4_692      1_521
    4     31_605      4_051
    5    156_405     10_794
    6    120_215     36_468
    7    268_906     77_972
    8    443_650     48_714
    9    358_541     43_348
   10    307_816     42_707
   11    292_181     33_415
   12    290_550     24_693
   13    199_780     16_644
   14    158_242     12_939
   15     76_530     10_411
   16     56_449      5_385
   17     22_415      3_340
   18     20_598      4_879
   19     10_604      2_404
   20      3_768      1_514
   21      2_122        163
   22        389         50
   23         74         24
   24         20          4
   25          4          0
~
```

Instead, we should attempt to build a filter which reduces the amount of information into a smaller but deterministic set. For instance, we could look at the modulo 1024 or the magnitude of the values, as

an example, to drastically reduce the size required for our filter. This approach remains to be implemented and tested, but hopefully the values are sparse enough that we can benefit from such a data structure.

Avoiding the problem before it appears

There's at least three different ways one can build a system where none of this matters or makes sense. The first one is to build a system where duplicates may never appear. That is, require all inserted files to be validated on the index before they're added to the set. Of course, this adds overhead, but may be preferred to having to fix this after it becomes a problem.

An alternative to this, which is fairly different, is to think of filesystems, not as a tree, but as a history. Every modification to your filesystem is a transaction, and we're only storing the changes. When we look at our filesystem this way, the very concept of duplicate files loses its meaning. For some cases, this makes sense. While one might argue this is rarely the way to go, it's still an option worth considering.

The last proposal, and perhaps more realistic, I'd suggest here is to have very strict requirements about the layout of your files. If there's duplicates of some files, that's because they're required for some reason, and one will know exactly they're duplicates because of their location on the system.