Deduplication

Deduplication is a form of compression.

At a high level, deduplication systems:

1. identify duplicate objects and
2. eliminate redundant copies of information

How the system defines and "object" and how the system defines a "redundant copy" is system specific.

Deduplication systems can be defined along several axes.

On-line vs. offline

In on-line dedup systems, the deduplication process happens at the time that data is written. A typical online deduplication system works as follows:

1. When the system receives a request to write some new object *O* (and before *O*'s data is actually written), the system checks to see if a duplicate copy of *O* already exists.
    - If the object is found to be unique, then the object is written as normal.
    - If a duplicate object is already in the system, the new object *O* is not written; instead, a *reference* to the original copy is stored.

In off-line dedup systems, the deduplication process happens after data is already persisted. A typical offline deduplication system will have a background process that scans data, and replaces duplicate objects with references to a common copy. This ultimately saves space because redundant copies are replaced with a reference, but there is a (potentially long) time window between when a copy is written and when the copy is replaced.

Fingerprinting

Fingerprints are unique content identifiers.

Scanning a storage system and doing pairwise byte-by-byte comparisons of all objects is impractical. Instead a cryptographic hash is used as a unique ID for an object's contents. Then, to compare objects (regardless of their size), you only need to compare their hashes (fingerprints): if the hashes are the same, then the objects have the same contents.

- For this to work, the probability of a *hash collision* must be close-to-zero. In other words, if two objects differ by even one byte, then their hashes should be completely unrelated. In a deduplication system, a hash collision introduces a correctness error.
- We often choose hash functions and hash sizes (i.e., the number of bits in the fingerprint) such that the probability of a hash collision is less likely than the probability of a hardware error. Since we assume that

the bits in a crytpographically strong hash function are uniformly random, each additional bit that is added to a hash will double the number of unique values that the hash function can represent.
- The "birthday paradox" is often used to analyze the collision probability and justify a particular hash size.

Chunking

Chunking is the process of breaking data into objects. Chunks can be whole-file objects, fixed-size chunks, or variable-sized chunks.

- Whole-file deduplication is simple and often has low overheads. If two files are exact copies, only one version is written. Subsequent files with matching contents are added to the system as references to the original.
  - The amount of metadata required to keep track of all objects in the system is quite low; it scales with the number of files.
  - Any modification to a file requires a unique copy to be made (breaking *all* sharing with similar files), which means that whole-file deduplication systems often have lower compression ratios than systems that use finer-granularity chunking schemes.
- Fixed-size chunks are often chosen to be sizes that are multiples of system hardware parameters, like memory pages or disk sectors.
  - The process of breaking an object into chunks is easy, since no computation needs to be performed.
  - If a single chunk is modified, common chunks can still be shared with other objects' chunks
  - If data *shifts* (for example, after the insertion of a byte at the head of a file), then all subsequent chunks will shift. Thus, local changes *can* falsely cause duplicates to be treated as unique
  - Since there are at least as many chunks as there are files, the amount of metadata required to keep track of all objects in the system scales with the amount of data (not number of files).
- Variable sized chunks are often defined by the contents of the objects using Rabin fingerprints or some other *sliding window* method.
  - If a single chunk is modified, common chunks can still be shared
  - If data *shifts*, then it is unlikely that nearby chunk boundaries are affected, since the boundaries are determined by the contents

Chunk Size

- Using large chunks creates lower metadata overheads (fewer chunks, so the fingerprint index is smaller), but large chunks usually result in lower deduplication ratios (coarser granularity of sharing).
- Using small chunks creates higher metadata overheads (more chunks, so the fingerprint index is larger), but small chunks usually have higher deduplication ratios (objects can share data at finer granularities).

Indexing

The chunk index can be a bottleneck in large deduplication systems, since it likely will not fit into RAM. Hashes are randomly distributed, so fingerprint index lookups often have no locality within the indexing data

structure, even if the workload has high locality.

- Bloom filters can be used to detect whether a fingerprint exists, eliminating the need for some unnecessary lookups.
- Some systems group fingerprints into groups on disk. If groups are defined by temporal locality, then caching and evicting based groups may improve cache efficiency
    - Question: how would you define the appropriate "group" for a very common chunk (e.g., the chunk is common to many unrelated files, each with their own fingerprint group)