**Scalable Data Deduplication using Similarity Matching and in-memory indexes**

Described is an approach to Data Deduplication that uses data segmentation and chunking coupled with MinHash based Similarity Matching to provide a high-performance mechanism to locate zones (or segments) of similar data and perform further chunk hash based identity deduplication.

Data Deduplication in backup and primary storage typically is based on looking up cryptographic hashes. Once the data is split into chunks, a high-security cryptographic hash (like SHA256) is computed for each chunk. These hashes are then inserted into an index. If the hash lookup in the index returns a duplicate then the current chunk is considered a duplicate. Instead of storing the current chunk data, only a reference to the existing entry is stored.

The big bottleneck with the above simple approach is the size of the index which can itself become terabytes in size if the dataset is in the order of hundreds of terabytes. Looking up chunk hashes in such an index requires multiple scattered disk accesses and is a big bottleneck to performance. Existing research have focused on techniques like using larger chunks, using bloom filters, pre-caching chunk hashes based on stream locality and stratified sampling among other things.

The idea described here takes an alternative approach of using a relatively tiny similarity index to quickly locate regions of similar data which can then be deduplicated. The idea is to split data into relatively large segments of 4MB to 8MB in size and then calculate similarity min-hashes at various extents of the segment. If the dataset is small then smaller segments are also permissible. So as an example, min-hashes can be computed starting at 5% of the segment data and at intervals of 5% upto 95%, hereafter referred to as similarity levels. These hashes are kept in an index and can be used to lookup segment matches for the corresponding similarity level. So if two segments are found whose hashes corresponding to the 50% similarity level are same then the segments are 50% similar to each other. Further chunking and chunk matching within the segment can then be done to do the actual identity deduplication.

The following steps illustrate the process in detail. As before we assume that we are using a similarity interval of 5% and using minhashes for match similarity levels of 5%, 10%, 15% and so on upto 95%. In practice we will want to use a smaller similarity interval (2.5% for example) to get more fine grained similarity matching.

1. Split data into relatively large segments. Segement sizes range between 4MB to 8MB. These numbers are found to provide a balance between Deduplication effectiveness and performance for large datasets.
2. Each Segment is then split into chunks based on a rolling hash or at fixed boundaries. An average chunk size of 4KB can be used.
3. A strong cryptographic hash is computed for each chunk in the segment and stored in the segment's chunk table on disk.
4. Match similarity level hashes are then computed for the segment data. If we take the similarity interval to be 5%, we can compute hashes over the corresponding K-min-values sets of the segment data. Or in other words the following:
   - Consider the raw bytes in the segment data as a stream of 64-bit integers – 8 bytes each. On 32-bit CPUs, 32-bit integers, 4 bytes each can also be used for optimal performance.
   - Pick the lowest 5% of the integer values and compute a hash.
   - Pick the lowest 10% of the integer values and compute a hash.
- Continue this till the 95% point.
- So we get 19 similarity hashes for the segment.
- This computation can be made efficient by using a Min-Heap and expanding the heap in place to avoid repeated copying of the entire data.

5. Now start looking up the similarity hashes in the index. We start the lookup from the hash corresponding to the 95% similarity level.

6. If a match is not found the non-matching similarity hash is inserted into the index and then the hash corresponding to the next lower similarity level is looked up in the index.

7. Steps 5 and 6 are repeated till a match is found or all hashes have been checked.

8. If a match is found the matching segment's chunk table is loaded.

9. Then the cryptographic hashes in the current segment and the matching segment are compared to find chunks that match exactly.

10. Only references to the pre-existing segment number and chunk numbers are stored for matching chunks in the current segment thereby achieving the actual deduplication.

11. Matching of cryptographic hashes is considered sufficient to identify identical chunks.

12. In step #8, if a match was found at similarity level 70% or greater, then the current segment processing ends. The similarity hashes in the current segment for levels less than 70% are inserted into the index, if they are not already present.

13. As an optimization for improved deduplication ratio if a segment is found to match similarity level less than 70% then lower similarity levels are also processed (step #5 onwards). Only hashes corresponding to the last 100 – current match level values are looked up.

14. Processing continues with the next input segment.

See figure below for the detailed flow diagram. This process of segmenting the data and calculating minhashes for various similarity levels or percentages results in only a few hashes being generated per segment. Given enough RAM it is possible to hold the entire similarity index in RAM. For example if we consider a similarity interval of 2.5%, max similarity match level of 95% and a segment size of 8MB then we get 38 minhash values per segment. If we are using a 256-bit cryptographic hash for the minhashes, we need at least $38 \times 32 = 1216$ bytes of hash storage per 8MB segment (leaving out data structure overheads). This translates to just a 152GB index (maximum) for 1PB of data.

For optimum processing the Chunk Data and Metadata must be kept separate. Chunk Metadata consists of the cryptographic hashes, chunks sizes and offsets. Since this metadata is accessed frequently it can be stored on high-performance Solid State Drive media. This metadata typically is rarely deleted but frequently appended to. So solid state drives are the ideal media to store this component.
We can further optimize and reduce space requirements by the following two options:

1. Use a 128-bit checksum for the minhash, like MD5 or truncated BLAKE2. Potential for random hash collision is increased especially with MD5. However the the cost of a hash collision during similarity matching is just reduced deduplication ratio. There is no data corruption since identity deduplication eventually uses the Rabin chunk’s 256-bit cryptographic checksum.

2. Use a separate index for each of the match similarity levels to split up the match domains and reduce chances of hash collisions during similarity matching.

3. Using separate indexes also allows us to do parallel index lookups where needed.

4. Use hardware offload for hashing and Rabin chunking.

By halving the hash size just 76GB of index is sufficient for 1PB of data. These index sizes are small enough to be held in RAM making similarity lookup extremely fast. For an 8MB segment size and 4KB chunks, it takes either one or very few disk reads for every 2048 chunks to do the actual identity deduplication. In most mainstream data deduplication software, larger chunks are used to keep an upper bound on the index size and have effective caching strategies. This approach allows using chunk sizes as low as 4KB and still remains performant. The 8MB example segment size used here is not a fixed quantity. Smaller segments up to as small as 512KB can be used for smaller datasets.

The approach described here also allows for a different variant of similarity matching to be used.
Instead of doing cumulative similarity matches (between segments) at 5%, 10%, 15% and so on, we can do range similarity matches. Assuming, as before, a similarity match interval of 5% we consider the following steps:

1. Split the segment data into fixed-size blocks where each block size is 5% of the segment size. If the segment size is 8MB then each block is 409KB in size.
2. Compute one 5% minhash for each block. We use the same value as the match interval here.
3. All these block hashes are looked up in the similarity index.
4. Segments that have maximum blocks matching in common with the input segment are considered for the next step of identity deduplication.
5. We can use some thresholds here, like top 5 matching segments or segments matching at least some X number of blocks (subject to experimentation).
6. If we are using separate similarity indexes, one for each block position in the segment then the lookups can happen in parallel.

See figure below for a diagram of this. This variation on the original approach can potentially provide a better deduplication ratio and the parallel lookup can be scaled across multiple cores/cpus and even within a cluster of nodes.