RGW Dedup

- Limits:
  - Doesn't support encrypted objects (RGW_ATTR_CRYPT_MODE)
  - Doesn't support Copy between 2 remote objects
  - Doesn't support Copy between 2 different pools
  - Doesn't support Copy between objects with different placement rules
- Existing Server-Side Copy mechanism
  - Increment ref-count on the SRC tail-objects
  - If the TGT existed before -> move it to be BG-Deleted
  - Copy the Head Object with all its attributes and Manifest from SRC to the TGT
    - If copy fails it will rollback the ref-counts on the SRC tail-objects
- New Mode for Dedup:
  - Keep the existing TGT head-Object with all its attributes
    - No need to copy data from the SRC
    - Skip reading the SRC Object and only read its manifest from the OMAP
    - Only the manifest will be changed on the TGT head-object
  - Read SRC/TGT etag, SHA256, version, size, manifest,..
    - Do we need an CLS or can we trust rados get-attribute to run atomically?
  - Increment ref-count on the SRC tail-objects
    - Rollback on any failure
  - Need a CLS on the TGT to perform the followings **Atomically**:
    - Verify TGT has the correct etag/SHA256/version/size
      - Abort the copy on a mismatch and rollback the ref-count on the SRC tail-objects
    - Store the TGT manifest on a persistent store
    - Overwrite the TGT manifest in the head-object with the SRC manifest
    - BG-Delete of the tail objects of the TGT (using the stored manifest)
- Open Issues:
  - Do we have code to BG-Delete tail objects while keeping the object active?
    - Maybe we can use the LC code?
    - Need to protect against accesses to tail objects while they are being deleted
    - [update] according to C. B. the existing GC code handles this correctly
  - Future enhancement:
    Do we support a head object with **no data** and tail objects **with** data?
    - If so can we split the head-object into an empty head (with attributes only) and move the data to a new named tail-object on the same OSD???
    - Can try and choose a name according to rados hashing with explicit_objs

Division of labor

- Double sharding - Ingress sharding by S3 object-name and Egress sharding by MD5
- Ingress sharding by object-name
    - Shard the object name space into N shards - set N to a high value (e.g. N=128) and shard the objects-space into many small units
        - Each rgw will grab one shard (using compare-swap on rados obj)
        - Upon completion of the current shard try and grab the next available shard (using compare-swap on rados obj)
        - This means rgw will perform multiple listing for all buckets
            - [update] C. B. suggested passing sharding parameters to the listing call causing it to skip objs belonging to other shards
        - It can also lead to uneven execution with the last shards being handled by a sub-group of the rgw members
        - It has the benefit of being more robust and eliminates the need for an external coordination between rgw members
    - Each rgw will read attributes for all objects in its assigned/owned shard
    - Output the read records into M egress shards
- Egress sharding by MD5
    - Shard the MD5 space into M shards
    - Iterate over all buckets and read object-count from bucket stats
    - Set M to be global_object_count / 16M
    - Each shard will contain about 16M records
        - With 32 Bytes per table entry we will need 1GB to process a single shard
    - Each rgw will grab one shard (using compare-swap on rados obj)
    - Upon completion of the current shard try and grab the next available shard
- No Inline dedup operations/updates as request can arrive at any rgw, but only a single rgw holds the shard managing a specific MD5 ETAG
- All operations will be done periodically from a BG-Task
- Drop all state between cycles
    - Each cycle will collect MD5 ETAGS from **all** objects

Step 0: preparation:

- Shard the object space to **N** shards
- Each rgw process similar number of disjointed S3-Object
- Skip objects smaller than the threshold (current threshold is a full rados object of 4MB)
- For each S3-Object find its head-object and read attributes from it into a **record**
    - MD5
    - Length
    - Part-count (for multi-part)
    - SHA256 (if exists)
    - version
    - Manifest
    - **Shared**-Manifest-Object (SMO) or **Dedicated**-Manifest-Object (DMO)
    - The head-object rados-name (which we already got)
        - No need for head-object name if this is a **Shared**-Manifest-Object
- Shard the records by MD5
    - This is another shard simply dividing the MD5 space by a given count
- Maintain **M** output stream-buffers mapped to the **M** MD5 shards
    - Each stream is written to a 4MB buffer and when the buffer is full it is written to rados and a new buffer is opened
        - Can use a smaller buffer (1MB/512KB..) if memory is scarce
    - Each buffer is divided into 4-16 KB blocks (256-1024 blocks per 4MB buffer)
    - Each Block contains multiple Records (each record describes a single S3 Object) and a short block header
    - The Records are appended to the current open block without crossing block boundaries (the leftover space is padded with zeros)

Step 1: Table Construction:

- Each rgw reads all its designated streams created by the N rgw sequentially
- Preallocate an open addressing hash-table with enough space for all the entries
    - Read the summary table from each rgw member to calculate the entries count
- Hashtable entries format:
    - Key: 16B MD 5 + 4B Length + 2B Part_Count
    - Data: 4B disk-block-index + 1B flags +1B pad
        - Flags:
            - Shared-Manifest-Object or Dedicated-Manifest-Object
            - Singleton or Multicopy entry
            - Has valid SHA256 or need to be calculated
    - Hash-Links: 4B indices into the table instead of pointers (allow 4B unique MD5)
    - Total is **32B** per a unique MD5
- Read the 4MB buffers one after another and iterate over all the records in the 4MB buffer
- Lookup the MD5 in the memory-hashtable and if doesn't exist add a new entry
    - Set flags:
        - Shared-Manifest-Object or Dedicated-Manifest-Object based on the record values
        - Singleton (as it is the first time we saw it)
        - Has valid SHA256 or need to be calculated based on the record values
- If a hashtable entry already exists for the record's MD5:
    - If the hashtable-entry is a Shared-Manifest-Object:
        - If the new record is a Shared-Manifest-Object -> skip it
        - If the new record is a Dedicated-Manifest-Object -> turn off the hashtable-entry singleton bit (as we will dedup the entry)
    - If the hashtable-entry is a Dedicated-Manifest-Object:
        - If the new record is Shared-Manifest-Object -> change the hashtable-entry setting the new record index in the entry and updating the SHA256 bit based on the new record values (should probably be set)
            - Change the entry flag from Dedicated-Manifest-Object to Shared-Manifest-Object
            - Turn off the singleton bit if it is on (as we will dedup the entry)
        - If the new record is a Dedicated-Manifest-Object:
            - Turn off the singleton bit if it is on (as we will dedup the entry)

Step 2: Dedup:

- Purge all entries not marked for-dedup (i.e. singleton bit is set) from the system
- Iterate again over the disk-records reading 4MB buffers each time:
    - If the record is marked as Shared-Manifest-Object -> skip it
        - Assert that the SRC-entry in the table is marked as a dedup entry
    - Lookup the record MD5 in the hashtable:
        - if doesn't exist -> skip it (it is a singleton and it was purged)
        - If the record block-index matches the hashtable entry -> skip it (it is the SRC object)
        - All other entries are Dedicated-Manifest-Objects with a valid SRC object
    - If the SRC/TGT object or both doesn't have a valid SHA256 -> read the object data from disk and calculate the SHA256
        - Before reading the OBJ data verify that its MD5/version hasn't changed
        - After completing the SHA256 calculation update the SHA256 attribute in the head-object (using a CLS to verify that its MD5/vesion hasn't change)
        - Update the SHA256 on the object disk-record for SRC OBJ (if it didn't have SHA256 before) and set the SHA256 bit in hashtable entry
    - Read the SRC object attributes from the disk record using the stored block-index
        - Compare the SHA256 and if they differ skip the new entry
    - Take the manifest from SRC-Object disk-record and inc-ref for all the tail objs
        - If one or more of the tail objs changed since we created the disk-record abort the copy operation
            - Change the MD5 SRC obj in the table to the TGT entry properties
            - Update the SHA256 field in the disk-record of the new SRC entry (which used to be TGT before)
        - Atomically update the TGT head-object:
            - Verify the MD5/SHA256/version are the same
                - If not abort the operation and rollback the ref-inc on the SRC tail objects
            - Store the TGT manifest in a persistent-store
            - Overwrite the manifest in the TGT head-object with the one read from the SRC-Disk-Record
            - Piggyback 2 extra attributes updates for the TGT head:
                - SHA256
                - Shared-Manifest with an 8 Bytes hash of the manifest
        - If all went fine -> set the previous manifest of the TGT for BG-delete
    - When the operation completed successfully:
        - If the SRC Record was marked as Dedicated-Manifest-Object add a Shared-Manifest-Object attribute to its head-object with 8 Bytes hash of the manifest

Data Structures

- The disk is broken into fixed size blocks (8-32KB) with multiple records each
- Disk-Blocks
    - Fixed size unit identified by a 32 bit disk-block-index
    - Always read the full disk-block
        - Updates can be done by 4KB random-write
    - Starts with a block head:
        - 2 Bytes magic number
        - 2 bytes records count
        - 4 bytes CRC16 of the block data
- Disk-Record:
    - 2 Bytes record-len
    - 2 Bytes Part-count (for multi-part)
    - 4 Bytes obj-size
    - 8 Bytes obj-version
    - 16 Bytes MD5
    - 32 Bytes SHA-256 (set to all zeros until calculated)
    - 8 bytes **Shared**-Manifest-Object or all zeros for a **Dedicated**-Manifest-Object
    - Variable-length Manifest (probably no more than 1KB)
        - A few hundreds bytes normally
    - Variable-length head-object rados-name (for **Dedicated**-Manifest-Objects only)
        - Rados name is up to 1 KB long, but normally much shorter
- MD5 Hashtable:
    - Map from an MD5 to disk-block-index of its SRC object
    - Open addressing table hashing with preallocation for all entries
    - Key: 16B MD5 + 4B Length + 2B Part_Count
    - Data: 4B disk-block-index + 1B flags +1B pad
        - Flags:
            - Shared-Manifest-Object or Dedicated-Manifest-Object
            - Singleton or Multicopy entry
            - Has valid SHA256 or needs to be calculated
    - Hash-Links: 4B indices into the table instead of pointers (allow 4B unique MD5)
    - Total is **32B** per a unique MD5
- Open Tasks table (about 1 MB total space):
    - A table holding 256 open tasks
    - Tasks are indexed by the SRC Object MD5
        - Prevents multiple actions on the same SRC object
    - Each Task holds the full disk-record of both the SRC and TGT
    - State variables
    - Operation log for rollback

Glossary:

- Disk Stream - sequential disk data for a given MD5 shard
- Disk-Block - fixed size sequential disk unit (8-32KB) with multiple records
- Disk-Block-Index - a relative index for a block within a disk stream
- Disk-Record - metadata for a single S3 object
- Table-Entry - Describing the SRC object of a dedup for all objects sharing an MD5
- Singleton Object - Only a single instance of the MD5 has been observed
- Multicopy Object - Multiple instances of the same MD5 have been observed
- Shared-Manifest-Object (SMO) vs. Dedicated-Manifest-Object (DMO)