OM HA Cache Design

Introduction

OzoneManager High-Availability deployments use the RAFT consensus protocol via Apache Ratis to achieve consensus between multiple OM replicas.

OzoneManager HA design document links:

- HA Design Document is attached to HDDS-505
- Handling Write Requests with OM HA

This document assumes some familiarity with the Ratis execution model, especially concepts like Raft server, state machine, startTransaction and applyTransaction.

Once Ratis has achieved consensus it calls back into the state machine on each OM replica to apply the transaction, at which point the OM replicas make the transaction persistent in their individual RocksDB stores. OMs can take advantage of the fact that the Ratis log is persistent to return success to the client before the transaction has been persisted to RocksDB aka early-return.

With the early return optimization, we need an in-memory cache of transactions that were acknowledged to a client but not yet written to RocksDB. Else we would break read-your-own-writes consistency. This document discusses about design and implementation details of Cache inside OzoneManager HA. And this document discusses about design and implementation of Double Buffer to commit transactions to OM DB, instead of performing put operation for each request.

First what is this cache. Cache is maintained per table. The cache for each Table is nothing but a map of key and value. For example for a KeyTable key can be KeyName and for value it can be KeyInfo along with additional fields.
This cache will be used for validation of information like key exists or not and also helps is serving read requests.

Flushing Transactions to RocksDB:

A cache entry can be purged once the corresponding write transaction has been written to RocksDB. To understand this better let's take a look at how the OM will asynchronously flush transactions to disk.

We propose using an implementation similar to the HDFS EditsDoubleBuffer. To keep the disk busy and reduce seeks, we will flush RocksDB transactions in batches. At a given time only one batch will be outstanding for flush while newer edits are accumulated in memory to be flushed later.

In DoubleBuffer it will have 2 buffers one is currentBuffer, and the other is readyBuffer. We add entry to current buffer, and we check if another flush call is outstanding. If not, we flush to disk Otherwise we add entries to otherBuffer while sync is happening.

In this if sync is happening, we shall add new requests to other buffer and when we can sync we use **RocksDB batch commit to sync to disk, instead of rocksdb put.**

Note: If flush to disk is failed on any OM, we shall terminate the OzoneManager, so that OM DB's will not diverge. Flush failure should be considered as catastrophic failure.

Cache Design and Implementation:

In OzoneManager, we have a Table for each Entity like Volume, Bucket and Key. The proposal is to have Cache for each entity. In this way by default the cache will be used during get Operations on the Tables, without any code changes required in classes using these tables. This also helps in future, if we add more Tables to OzoneManager. In Table interface we also need to expose api to add entries to cache and update the get() api to use cache first for getting, if it exists in cache return else perform a db.get() and return.

We implement cache as an integral part of the table, so users using this table does not need to do any additional work, except adding entries into the cache.

For each Table the Cache Key and Cache Value structure looks as below.
CacheKey<KEY> KEY - is the key for the table.

```
public class CacheKey<KEY> {

private final KEY key;

public CacheKey(KEY key) {
 this.key = key;
}

public KEY getKey() {
 return key;
}
```

```
}
```

CacheValue<VALUE> VALUE - Value we store for each key in the table.

```
public class CacheValue<VALUE> {

 private VALUE value;
 private OperationType lastOperation;


 public CacheValue(VALUE value, OperationType
lastOperation, long epoch) {
   this.value = value;
   this.lastOperation = lastOperation;
 }

 public VALUE getValue() {
   return value;
 }

 public OperationType getLastOperation() {
   return lastOperation;
 }

 /**
  * Last Operation performed on the key.
  */
 public enum OperationType {
  CREATED,
  UPDATED,
  DELETED
 }
```

As Bucket and Volume entries will not be very huge, on a typical ozone cluster this can be in the thousands. We shall preload this cache during OzoneManager startup. Cache will not be preloaded for the entities where there might be a millions/billions of entries in a table. Like keyTable, openKeyTable and also for prefixTable for OzoneFS and NativeAcl.

For example for VolumeTable Cache structure:
CacheKey: `CacheKey<String>` which is nothing but the key of the VolumeTable.

CacheValue:

```
 CacheValue<OmVolumeArgs> {
     OmVolumeArgs omVolumeargs; // value which need to be
persisted to OM DB.
     OperationType lastOperation;
}
```

OperationType {
CREATED, // set for create request
UPDATED, // set for update request like setVolumeProperty
DELETED // set for delete request
}

Why do we need to do this because, to return correct responses during read requests.
For example take requests are happening in below order:

1. CreateVolume /ozoneVolume(Write Op)
2. DeleteVolume /ozoneVolume(Write Op)
3. GetVolume /ozoneVolume (Read op)

Requests 1 and 2 will return to client after validation with cache/DB and generating the information required to return as response to client. And then we shall do batch commit later.

After 1, 2 requests OzoneManager Cache and VolumeTable:

Cache after 1st request : (As there will be no sync to disk happening, assume that sync to disk is happening in parallel and it has completed after processing 2nd request)

Key	**Value**
ozoneVolume	CacheValue { OmVolumeArgs : volume:ozoneVolume creationTime:134555555 .

	. lastOperation: **CREATED** }

OM DB Table after 1st request:

Key	**Value**
ozoneVolume	OmVolumeArgs : volume:ozoneVolume creationTime:134555555 . .

Cache after 2nd request: (When 2nd request is being processed, let's assume sync is going, so we have not committed to table)

Key	**Value**

ozoneVolume	CacheValue {
	OmVolumeArgs :
	volume:ozoneVolume
	creationTime:134555555
	.
	.
	lastOperation: **DELETED**
	}

VolumeTable:

Key	**Value**
ozoneVolume	OmVolumeArgs : volume:ozoneVolume creationTime:134555555 . .

Now we are handling read request, 1 is committed to OM DB and 2 is not committed to OM DB, only cache has the updated entry. So, now when getVolume happens, we return for that request bucket does not exist, as in cache for this volume it has lastOperation set to DELETED. We can remove this entry from cache once after we commit to OM DB, so that deleted request key entries will be deleted from the cache.

Handling Read requests:

Now with cache and double buffer implementation, read APIs will be affected. As read API's previously used to read from OM DB. Now we cannot do that, as write request information is not being immediately applied to OM DB.

1. As shown in the above example when getVolume request is received, we use table.get(volumeName) to return VolumeInfo to the user. As we shall modifiy get() api of the Table to use both cache/DB and return value, the readPath for these kind of requests will require no changes.
2. Few APIs like listVolumes, listKeys, listBuckets which used to iterate the DB and return the response, these API's logic needs to be modified. Question here is how we can effectively use cache and DB to return the response for requests like listKeys of a bucket after a specified keyname.

Memory Usage:

As discussed above for Volume and Bucket Table we store full table information in memory. This will help in validation of the requests very quickly. As for every request Ozone Manager receives the mandatory check is volume/bucket exists or not.

On a typical Ozone cluster Volumes can be in number of thousands. (Considering this as an admin level operation in a system where each team/organization gets a volume for their usage). And for each volume we can expect 1000 to 10000 buckets. These are considered just for calculation purpose.

Let's assume each VolumeInfo and BucketInfo structure consumes 1KB in memory. Then,

Volume cache memory usage can be 1000 * 1KB = 10 MB.
Bucket cache memory usage can be 1000 * 1000 * 1KB = 1GB.

We can make the Volume and BucketTable caches partial if the number of buckets and volumes are very high in the system. This can be given as an option to end user. For now we assume that the entire list of volumes and buckets can be safely cached in memory.

How this will be used in OzoneManager HA

At a very high level how above double buffer and cache design will be used in OM HA.

1. In applyTransaction in OzoneManager StateMachine, when a OM request is received. Following are the steps to handle the request.
    a. we shall use get() which checks cache and DB to return the value.

b. Validate bucket/key/volume exists
c. Generate required information to add to OM DB.
d. We add to the cache and then return response to client.
e. Then asynchronously use doubleBuffer to flush to disks.

This is also explained in the Approach 2 section of [Handling Write Requests with OM HA Design Doc]

Restart

On OzoneManager restart, volume/bucket Table cache is preloaded. For other tables, the cache is not preloaded on a restart.

On restart, Ratis will take care of applying any pending transactions and we will rebuild the in-memory cache as these transactions are re-applied.

Parallelizing Apply Transaction

To speed up processing in OM, plan is to have multiple executor services for multiple volume requests, so that requests for multiple volume can go in parallel, and the requests for same volume will be serialized.

Example: If we have 10 Executors with singleThreadExecutors

```
List<ExecutorService> executors = new ArrayList<>();
executors.add(Executors.newSingleThreadExecutor());
```

To find which executor for request Pseudo code:

```
executors[computeHash(getVolumeName(omRequest)) %
executor.length]
```

Note: We can also computeHash and have hashTable for all Volumes instead of computing hash for each Volume in applyTransaction for every request.