Ozone Architecture

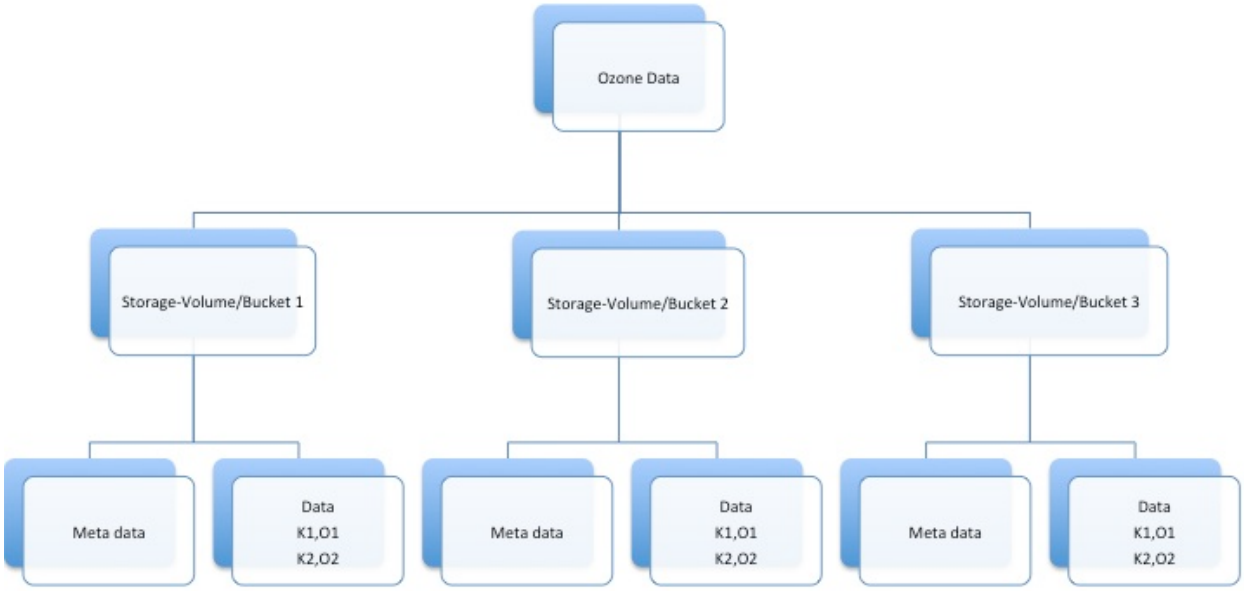
Introduction

Ozone provides a key-object store like AWS's S3. The keys and objects are arbitrary byte arrays. A key size is expected to be less than 1K, while values are expected to range from a few hundred kilobytes to hundreds of megabytes. Keys/objects are organized into buckets; a bucket has a unique set of keys. Buckets exist in a Storage-Volume[1] and are uniquely named within a storage-volume. A storage volume a has globally unique name. Further, a storage-volume has a quota for number of buckets and amount of data. In a private cloud, storage volumes are created for users (like home dirs), for projects, and for tenants. The admin can assign quotas for each user's private storage volume and also for each shared project volume. In a public cloud, multiple storage volumes can be created for each cloud account along with quotas.

A bucket is identified by a two part name: ***storage-volumeName/bucketName***. An object is identified as s***torage-volumeName/bucketName/objectKey***

Our scheme is similar to the Azure Blob Storage (WASB) where where buckets are unique within an account. However, instead of an account we have a storage volume. In contrast, S3 buckets are unique across all accounts.

The data organization can be viewed as shown in the following picture.



Basic Requirements

The basic operations for Ozone are (the API is at end of this document)

- Admin creates Storage Volumes.
- Create/delete buckets. A bucket has a unique URL. Buckets cannot be renamed. Only the owner or group owner of the storage volume are allowed to create or delete a bucket in that storage volume.
- List buckets for a Storage-volume.

- Create/delete objects within a bucket, given its key. The data or value of the object can be streamed to the Ozone service. The object is available for read only if fully written, no guarantees on partially written objects.
- List the contents of bucket.
- Create/Update/Delete Buckets ACLs.

Size requirements

This section lists a few limits that characterize initial set of requirements. This is what we are targeting for 1st version.

- **Storage Volume Name:** Between 3 and 64 bytes
- **Bucket Name:** Between 3 and 64 bytes
- **Key Size** : 1KB
- **Object Size** : 5G
- **Number of buckets system-wide :** 10 million
- **Number of objects per bucket:** 1 million
- **Number of buckets per storage volume** : 1000

The metadata of Ozone consists of following:

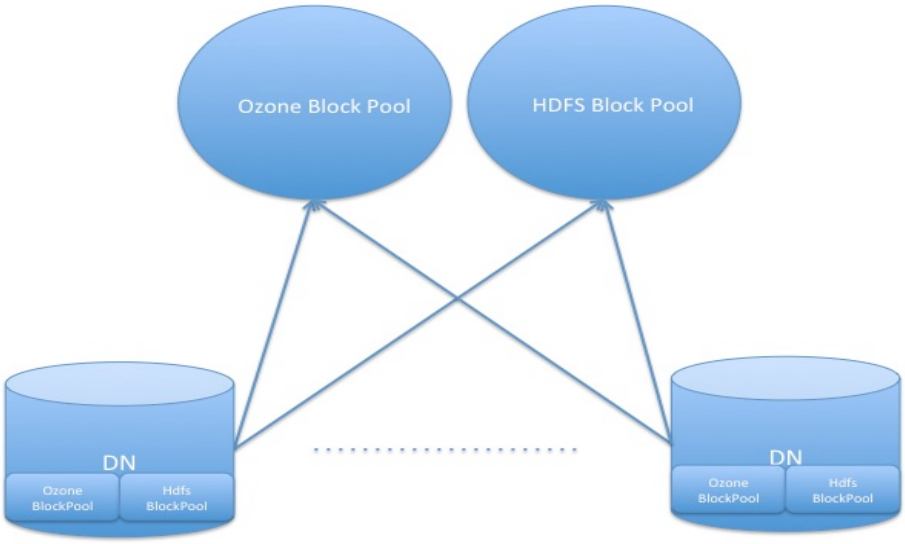
- Storage Volume level metadata
    - Storage volume owner
    - Storage volume name
- Bucket level metadata
    - Bucket owner: This is the user that owns the data in a bucket.
    - Bucket ACLs
    - A Bucket name that is unique globally.
    - Bucket Id: A numeric system-generated identifier associated with the bucket

High Level Design

Shared Datanode Storage With HDFS

DataNodes store data for both HDFS and Ozone. Ozone data is in separate block pools using separate blockPool-Id[1]. Just like there can be multiple independent namespace volumes, each in its own block pool, there can be multiple independent Ozone namespaces each in its own block pool. A DN can store multiple HDFS and multiple Ozone block pools. This is depicted in the picture below.

[1] Strictly speaking it is a container-pool-id but we have reused HDFS/DN's existing block-pool-id.



Storage Containers

A storage container is a unit of storage that is used to store both the Ozone data (ie the bucket data) and the Ozone metadata. Storage containers are stored in DataNodes (along with the HDFS blocks). Unlike HDFS, Ozone does not have a central NN with its separate metadata storage; instead Ozone's metadata is distributed across storage containers. A storage container is replicated as a whole (like an HDFS block). We offer strong consistency for container replicas for successful operations. The maximum size of the storage container is determined by ability to easily replicate and recover from node failures. The maximum size will be configurable, but it should be more than the maximum size allowed for a single object.

A bucket can have millions of objects and could grow up to terabytes in size and is much larger than a storage container. Therefore a bucket is divided into ***partitions*** where each partition is stored in a single container. (A storage container can contain a maximum of one partition and hence objects from only one bucket). In our initial implementation an object is fully stored in a single container; this may be relaxed later.

The storage containers are implemented by datanodes (the feature can be disabled via configuration for customers that need only hdfs blocks). This section defines the requirements and semantics for storage containers.The storage container needs to store following kinds of data, each requiring slightly different semantics:

- Bucket metadata
    - Individual units of data are very small - kilobytes. A storage container stores metadata for millions of buckets.
    - This requires update, because a bucket's ACL can be updated. An atomic update is important.
    - get/put API is sufficient.
    - Listing operation to list all buckets in the container.
- Bucket data
    - The storage container needs to store object data that can vary from a few hundred KB to hundreds of megabytes.
    - The data is accessed via its object key (ie the container storing the data has to maintain the index of its stored object)
    - Must support a streaming APIs to read/write individual objects.
    - Append or in-place update for data in objects is ***not*** supported.

In a later section we describe how a datanode implements storage containers to meet above requirements.

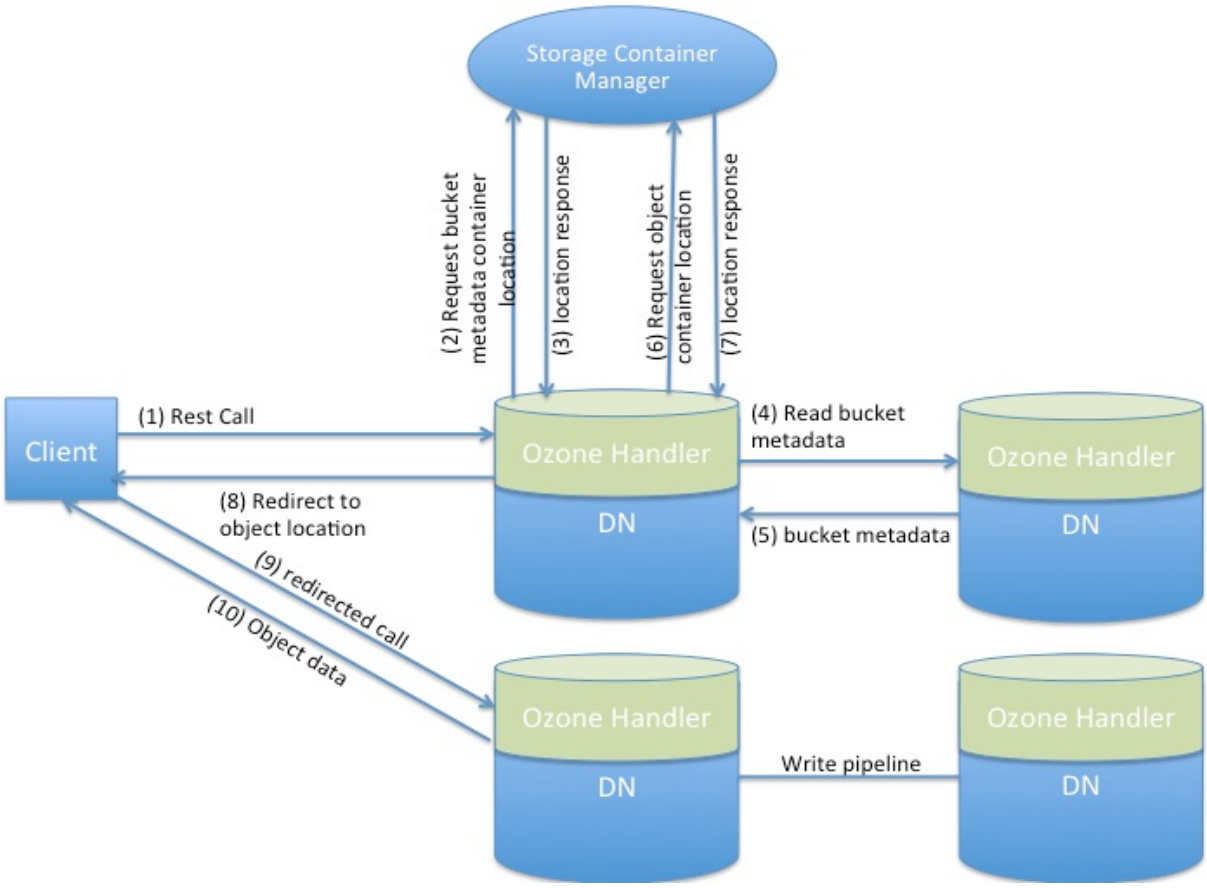
Storage Container Identifier

Each storage container is identified by a unique storage container identifier; it is a logical identifier (like HDFS's BlockId) and does not contain the actual location of the container on the network.

The object-key is mapped to a storage container identifier, and this identifier is passed to the **storage container manager** to locate the DataNodes that store the storage container containing the object. Similarly, a bucket name is also mapped to a storage container identifier that stores the bucket's metadata. In a later section we mention how the mapping is implemented. The storage container identifier is 64 bit, similar to the hdfs block id. In future we would like to extend this to 128 bit, but that would be a major change given we want to re-use hdfs's block management code.

Call Flow in the Storage Container Service

The following diagram shows a typical call flow for ozone which introduces Storage Container manager and Ozone Handler that are described below:



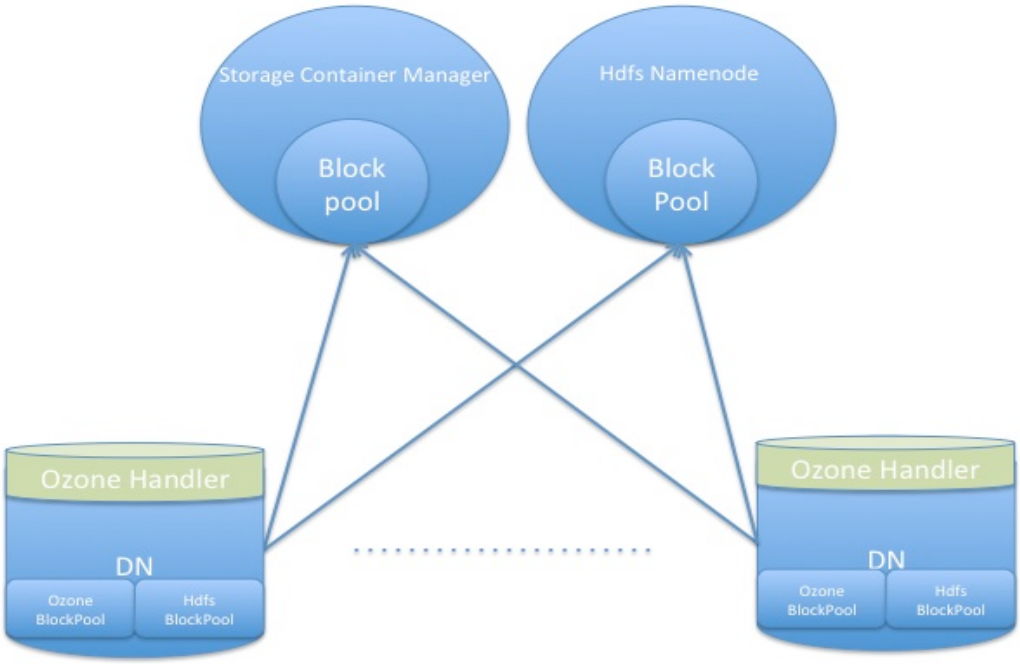
Ozone Handler in the DataNode

Ozone handler is the Ozone module hosted by each DataNode to provide the Ozone service. The handler consists of an http server and implements the Ozone REST APIs. Ozone handler interacts with the storage container manager to look up a container location, and interacts with the storage container implementation on the datanodes for various operations. This module can be disabled if a customer only needs HDFS and does not need Ozone.

Storage Container Manager

The storage container manager is very similar to the block management functionality of HDFS. The storage container manager collects heartbeats from datanodes, processes storage container reports and tracks the location of each storage container. It maintains a storage container map, that provides prefix matching lookup for storage containers. Note that

the DNs store data for both HDFS and Ozone. Ozone's data is in separate block pools using separate blockPool-Id[2] served by a separate storage container manager. This is depicted in the picture below. We plan to re-use Namenode's block management implementation for container management, as much as possible. This also allows to re-use the block pool service implementation in DataNodes.



Implementation

Mapping an object-key to a storage container

In this design we propose to use a **hash partitioning** scheme to map a key to its storage container. The key is hashed and prefixed with its bucketId, and the resulting value is mapped to the storage container using a prefix matching. As a container gets bigger we split it and use **extendible hashing**[1] to re-map the keys to their containers. Storage container manager stores the mapping in a trie for efficient prefix matching. There is one trie for each bucket.

[2] Strictly speaking it is a container-pool-id but we have reused HDFS/DN's existing block-pool-id.

Range Partitioning vs Hash Partitioning

Range partitioning is another popular technique for partitioning keys in databases or object stores. Following is a brief comparison of the two approaches. We are inclined to use hash partitioning as it is simpler to implement and covers our requirements.

- The range partitioning scheme requires range indexes. Range indexes are also big and need to be stored in the distributed fashion. This complicates the operation of splitting the partitions as it needs indexes to be updated as well.
- The range partitioning introduces an additional hop in the lookup, where first the range index needs to be read.
- Range partitioning also suffers from hot spots. Similarly named keys could cause large number of concurrent access to same ranges.
- However, range partitioning does offer the advantage of ordered access and ordered listing. We think an ordered access is not an important requirement for the object store. Ordered listing can be implemented using secondary indexes in later phases. We also believe that the design is flexible enough to add range indexes in future.

Mapping a bucket to a storage container

The bucket metadata is also stored in a storage container. The bucket name is used as the key for hashing. The bucket metadata for several buckets can be present in a storage container. We designate a special bucketId (e.g. 0) to prefix the container Ids for all bucket metadata related storage containers.

Storage Container Requirements

Storage Container is stored on datanodes, and we have following requirements:

- Storage Containers are replicated for reliability.
- Storage Container replicas are kept strictly consistent.
- Storage Containers provide an efficient lookup for the keys/objects stored within.
- Storage Containers must allow streaming writes and reads to/from objects.
- Storage Containers must support a get/put interface as well, to store and update bucket metadata.
- Storage Containers storing the bucket metadata must provide an atomic update.
- Storage Containers are split when they reach a size limit.

Notes on Storage Container Implementation

- We intend to re-use HDFS's block pool management infrastructure as much as possible. Therefore we implement a StorageContainer as an extension of a Block. An

hdfs block consists of an identifier and generation stamp and size. All these three attributes are applicables to storage containers as well.

- For consistency and durability the storage containers implement a few atomic and persistent operations i.e. transactions. The container provides reliability guarantees for these operations. In the first phase we implement following transactions:
    - **Commit**: This operation promotes an object being written to a finalized object. Once this operation succeeds, the container guarantees that the object is available for reading.
    - **Put**: This operation is useful for small writes such as metadata writes.
    - **Delete**: deletes the object
- Each transaction in a container has a transaction ID that must be persisted.
- We plan to implement a new data pipeline for storage containers as it requires different kinds of update and recovery semantics. In the next section we outline the design of the data pipeline.
- We are considering a storage container prototype using *leveldbjni*[2] which is a key-value store that meets our storage container requirements.

Data Pipeline Consistency

The data pipeline chain replicates any data that flows to the containers. Container replicas have generation stamps similar to HDFS blocks. The generation stamp of each replica in the pipeline is updated during pipeline setup so any stale containers are discarded. HDFS additionally uses the block length during block recovery to determine which replicas are most up to date. Similarly storage containers will use a transaction ID to determine which container replicas are most up to date. More detailed design of the pipeline will be posted on the corresponding hdfs jira.

Future Work

We have not covered following requirements in this document, but which we intend to support in later phases of the work:

- **High availability**: Storage Container manager needs to be a highly available service. One solution is to use an Active/Standby using HDFS's QJM for the journal. Another approach could be to use all Active servers with a Paxos ring. We will bring that up for debate in the community in the second phase of the project.
- **Security**: Once again we have HDFS's approach based on kerberos to use.
- **Cross colo replication:** We would like to approach this feature in a way that can benefit both HDFS and Ozone.

Ozone API

Cluster Level APIs

- **PUT StorageVolume**
    - API - PUT /admin/volume/{StorageVolume}
        - Create a storage volume
        - Only admin is allowed this call
- **HEAD StorageVolume**
    - API - HEAD /admin/volume/{StorageVolume}
        - Check if the Storage Volume exists
        - Only admin is allowed this call

- **GET**
    - List all the Storage Volumes in the cluster

- **DELETE Storage Volume**
    - API - DELETE /admin/volume/{StorageVolume}
        - Deletes a volume is it is empty

Storage Volume Level APIs

- **GET Buckets**
- **API - *GET /***
    - *Signed with User Authorization*
        - Return the list of buckets owned by the sender of the request
- **Get User Buckets**
    - API - GET /admin/user/{userid}
        - Signed with User Authorization , and if he / she has permissions to read other users information, Returns a list of buckets owned by the specific user.

Bucket Level APIs

- **LIST objects in a bucket**
    - API - GET /{bucketName}
        - Return upto 1000 keys in the buckets

- **PUT bucket**
    - API - PUT /{bucketName}
        - Create bucket owned by the sender of the request

- **GET / PUT Bucket ACL**
  - API - /{Bucket}?acl
    - Allows user to get and set ACLs on a bucket
- **HEAD bucket**
  - API - HEAD /{bucketName}
    - Check if the bucket exists and the sender has permission to access it
- **DELETE bucket**
  - API - DELETE /{bucketName}
    - Delete the bucket if it is empty

Object Level APIs

- **GET object**
  - API - GET /{bucketName}/{key}
    - Return value for the given key if it exists
    - In phase 1 no ACL support, therefore only owner of the bucket can read and write to a bucket
- **PUT object**
  - API - PUT /{bucketName}/{key}
    - Create an object in the bucket
    - In phase 1 no ACL support, therefore only owner of the bucket can read and write to a bucket
    - Only full uploads of the value are considered successful, no guarantees for partial uploads.
- **HEAD object**
  - API - HEAD /{bucketName}/{key}
    - Check if the object exists
    - In phase 1 only owner of the bucket is allowed this call.
- **DELETE object**
  - API - DELETE /{bucketName}/{key}
    - Delete the object

References

1. Extendible Hashing - http://en.wikipedia.org/wiki/Extendible_hashing
2. leveldbjni - A Java Native Interface to LevelDB. https://github.com/fusesource/leveldbjni