Ozone - Object Store for Apache Hadoop

- Arpit Agarwal (arp@apache.org)
- Anu Engineer (aengineer@apache.org)

This document is an Ozone design update that builds on the original Ozone Architecture [1] and describes in greater detail Ozone namespace management and data replication consistency. It concludes with a section that looks at the evolution of the design since the original architecture.

Acknowledgements

The ideas in this paper are based on conversations and work done by Jitendra Pandey, Suresh Srinivas, Sanjay Radia, Chris Nauroth, Jing Zhao, Tsz Wo Nicholas Sze, Enis Soztutar, Arpit Agarwal and Anu Engineer. We have also drawn inspiration from ideas in earlier HDFS

proposals: *HDFS-5477 - Block Management as a Service* [2]; and *HDFS-8286 - Partial Namespace In Memory* [3].

Introduction

Ozone is an object store like Amazon 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 (although neither limit is strictly enforced).

Keys/objects are organized into buckets; a bucket has a unique set of keys. Buckets exist in a *Storage Volume* and are uniquely named within a storage volume. A storage volume has a globally unique name. Further, a storage-volume can have quotas. 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.

Ozone's API is described in great detail in the *Ozone User Guide* [4].

In the original design proposal, Storage container manager was in charge of both Namespace management and block space management. This document is proposing that Ozone's name space be managed by a new service called Key Space Manager (KSM).

Shared Container Layer

Datanodes provide a shared generic storage service called the container layer[1]. The container layer is designed as a fully distributed service with no single points of failure and incorporates lessons learned from running HDFS at large scales. A similar idea was proposed by Bortnikov et al. in [2].

The container layer consists of two main components:

1. Storage containers which are the units of replication.
2. Storage container manager (SCM) - a replicated service which keeps track of the replicas of each container.

A container is a unit of replication and hence its size is bounded by how quickly it can be re-replicated following the loss of a replica. Each container is an independent key-value store with no requirements on the structure or format of individual keys beyond the requirement that no single object can exceed the size of the container itself. Keys are required to be unique within a container only.

[1] No relation to YARN containers or to Linux Containers (LXC).

The interpretation of the keys and values depends on the storage service that the container belongs to. E.g. For Ozone, keys and values can correspond one-one to Ozone keys and values. By encapsulating multiple keys+values within a single replication unit we can avoid block-space explosion that is seen is HDFS clusters since HDFS creates one block per object (i.e. file).

The design of the container layer is explored in greater detail in a later section of this document.

Ozone Namespace Management

KSM Service

Ozone's namespace operations will be managed by a Key Space Manager (KSM) service and SCM will be in charge of managing the containers. There are several advantages in splitting KSM out as an independent service.

1. HDFS always wanted to separate namespace management from block space management. We have learned that it is hard to do it retroactively.

2. While block space is accessed randomly, the namespace has a locality of reference. So the data structures and optimizations are different.

3. This split allows us in future to run many instances of KSM to address scalability issues. This is similar to federation, since Ozone does not support rename or move (at least not with the strict guarantees of a Posix file system). Hence it is easier to shard ozone's namespace across many instances of KSM.

4. The scalability of KSM makes it very easy to grow an ozone cluster. This is key to scaling up to trillions of objects.

Buckets Spanning Containers

When a bucket has more data than what can be held in a container, KSM allows a bucket to span multiple containers. This information is maintained inside KSM. In other words, a lookup of a name under ozone always returns path to a container. KSM is a name to container mapping service.

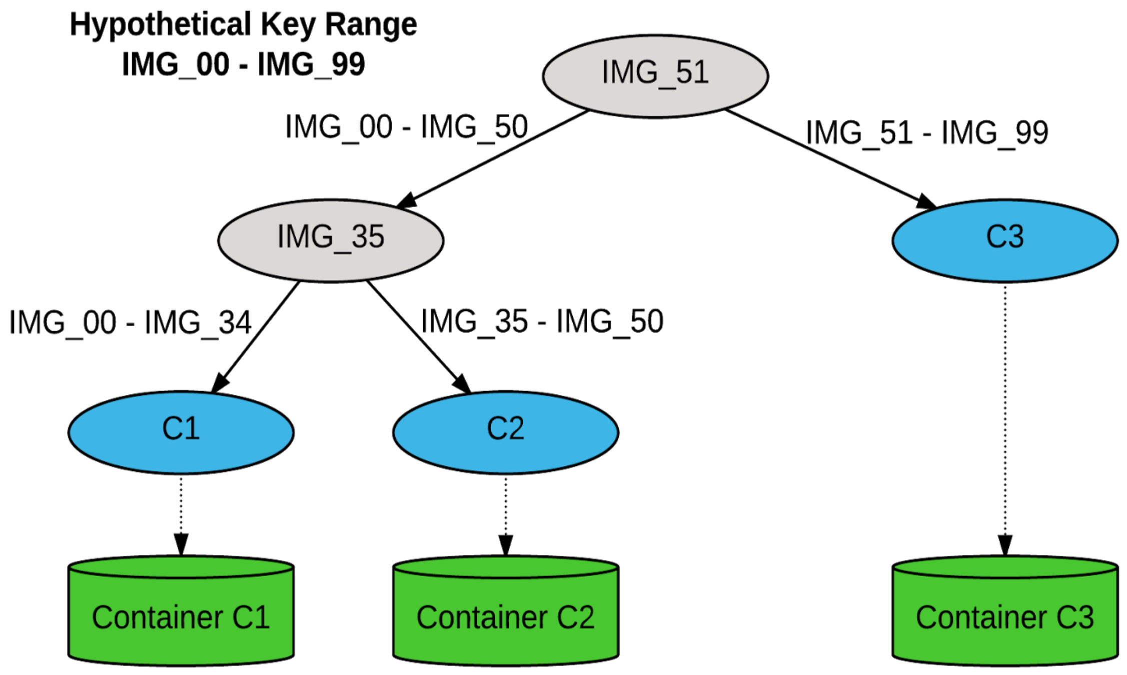


Figure 1: Bucket spanning containers

In steady state KSM maintains a sparse information tree of the namespace instead of a full blown namespace information. KSM never maintains information about keys themselves, since that is something that containers would know about. However if you look at the Figure 1, you will see that KSM also maintains state about the split points inside bucket. This means that KSM will be able to send an Ozone client to right container location.

Now let us look at some degenerative cases. A bucket can be continually split[2], the absolute worst case is that the KSM will have as many entries as as keys. This is no worse off than current Namenode (a little better in fact since the block info is in containers). Now that we have established that we can do no worse than steady state of Namenode, let us look at what would happen in a typical case like ETL, where someone uploads a bunch of files and deletes them. KSM might have split the bucket while upload was going on. We don't have to merge the bucket back when objects are deleted since any future put operations can go to either of the containers (based on the key name) of that bucket. This does increase the metadata on KSM side marginally, but with a B+ tree or equivalent the depth of tree will not increase too much.

If we are worried about the memory pressure, it is easy to spin up many KSMs and run KSM in a federated mode, so we do have a relatively easy solution. Due to these merging a container does not seem to be a phase one requirement for ozone. We also believe that the use of chunk references will make metadata merge a relatively fast operation.

[2] Say each object is close to 5GB or the container size.

The following DAG shows the conceptual distribution of state in an Ozone cluster with KSM in picture.

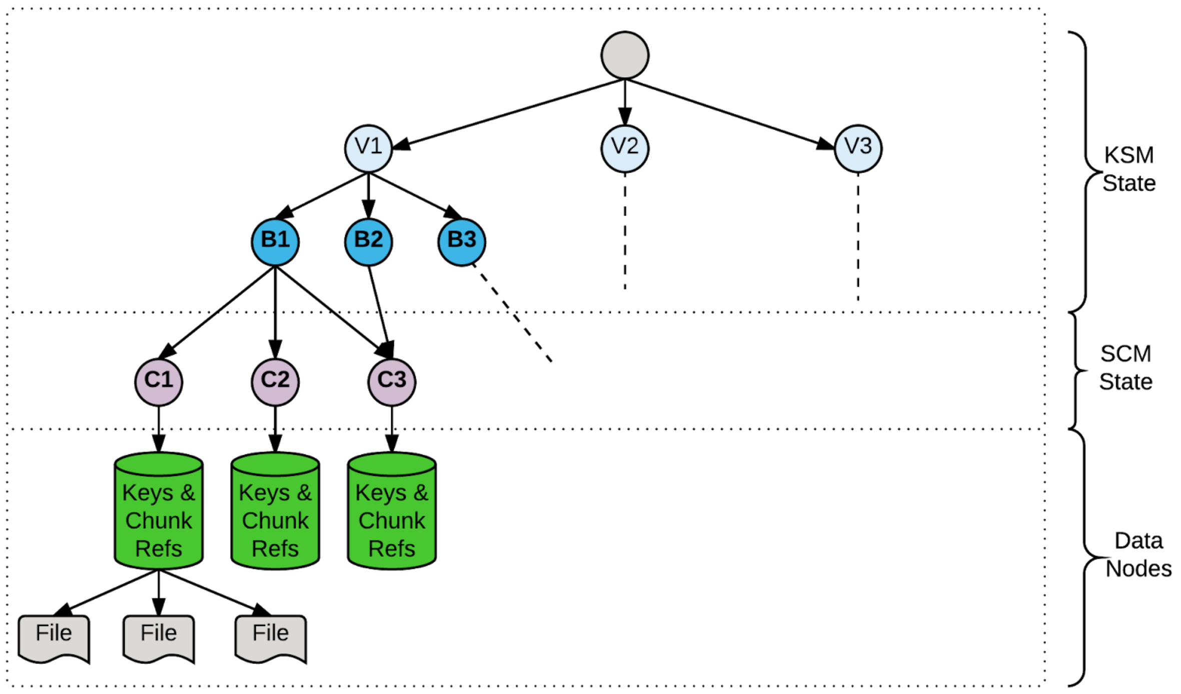


Figure 2: Logical view of namespace and containers

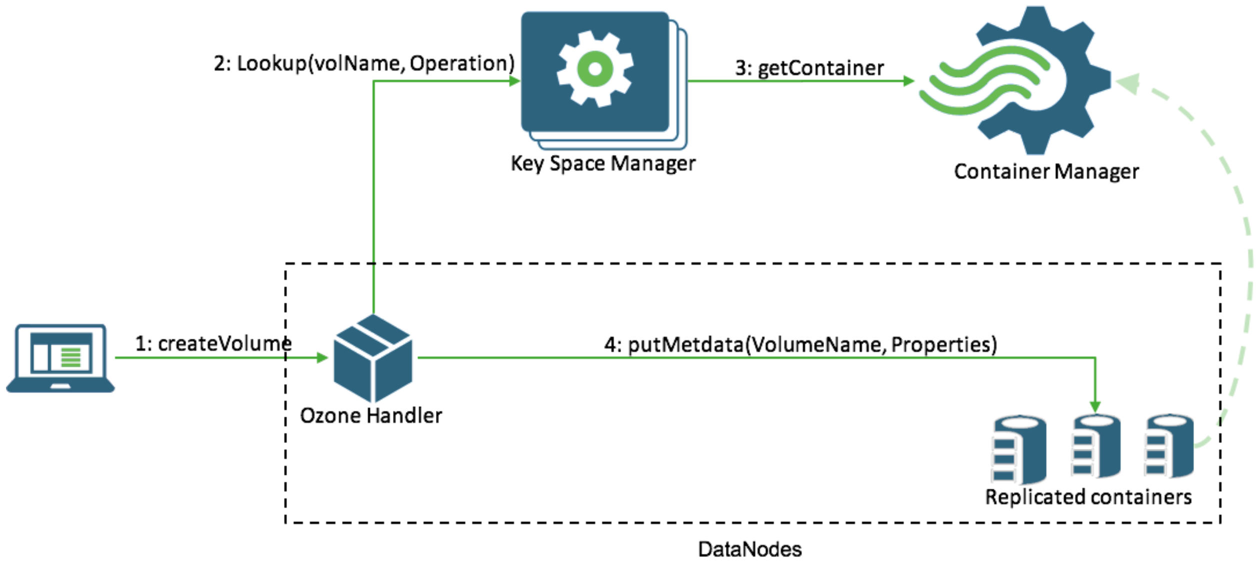
In ***figure 2***, KSM maintains the namespace – That is **mostly**[3] volume information and bucket information. The v1, v2, v3 corresponds to volumes in the ozone world. Volumes are root objects in the ozone namespace, hence the earlier reference to ozone's namespace as a forest instead of a tree. This representation is exactly same if we choose to do KSM or not. If we don't have KSM, then SCM will have to have the same tree, but inside a single service – more similar to namenode.

KSM Workflows

The SCM is in charge of container life-cycle and KSM requests access to these containers from SCM. SCM chooses the datanodes where a container would exist. These machines can be picked randomly or using a more sophisticated algorithm like copysets [5]. Our own preference is to use copysets since it reduces the probability of data loss if and when we lose more than 3 machines in a cluster[4]. The following figure illustrates how KSM will be used by Ozone.

[3] Please see the buckets that span multiple containers section, where we will maintain split points inside buckets.

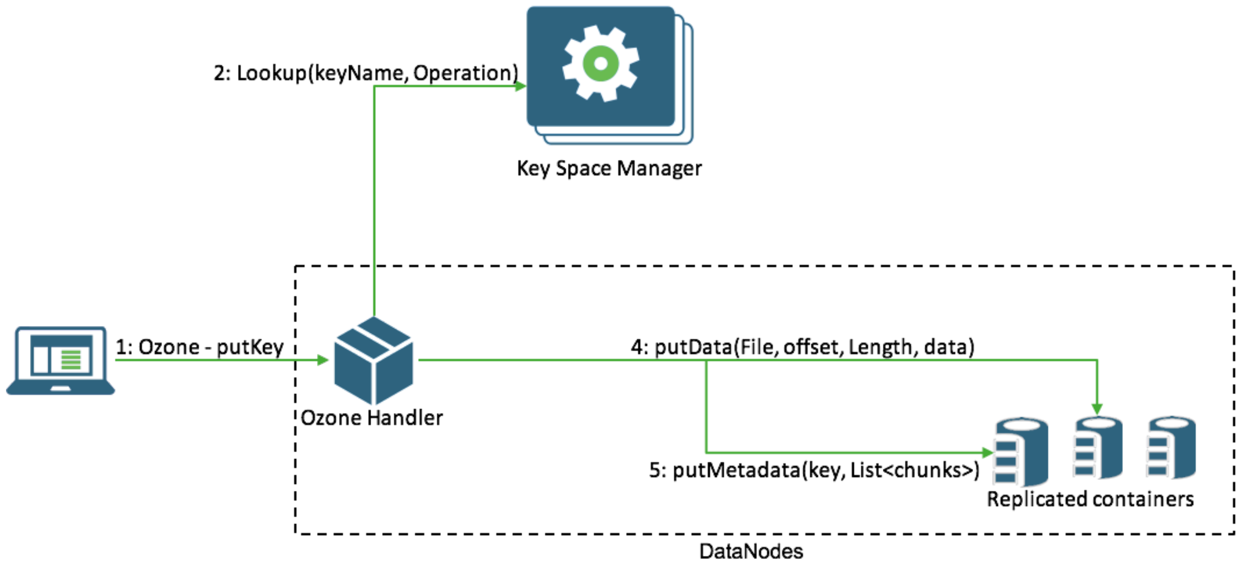
[4] Please also see the discussion in replication pipeline section about complete loss of containers.



1. Client makes a createVolume request to the Ozone front end.
2. Ozone Handler or the REST front end takes that request and looks up KSM (in the current code in the Apache HDFS-7240 feature branch, this request is send to SCM) for the current location of the container.
3. SCM returns a container location to KSM. This is because datanodes communicate to SCM. Datanodes send heartbeats as well as container reports to SCM.
4. Based on container reports - which is a summary information containing size, keys and number of operations - SCM chooses a container that KSM can use.
5. KSM updates the tree that it maintains, at this point a volume create is complete. This same information is written to a container, so that we can reconstruct the state of the ozone cluster even if we have a disastrous loss of Key Space Manager(KSM) or Storage Container Manager(SCM).

A very similar work flow exists for a bucket create.

In case of ozone key write there is an extra step. Ozone handler will write the chunks and then update the key.



2: Lookup(keyName, Operation)
Key Space Manager
1: Ozone - putKey
Ozone Handler
4: putData(File, offset, Length, data)
5: putMetadata(key, List<chunks>)
Replicated containers
DataNodes

1. Ozone client does a put key[5].
2. Ozone handler asks the KSM which container contains this key
3. Ozone handler writes a set of chunks - This data is replicated via simplified data pipeline.
4. Ozone handler writes the metadata - This gets replicated via RAFT and when the update is complete this key is visible in the Ozone namespace.

In relation to the original design proposal - Addition of KSM merely adds a namespace service which can be scaled independently of the container service.

KSM is responsible for Quotas, Key Space to containers mapping and is a critical component of Ozone. Since KSM is required for ozone's operation, high availability of this component is important and it is achieved via RAFT. In other words, all decisions made by a KSM is replicated via RAFT.

Quotas under Ozone

Ozone supports quotas at the volume level. However a volume is spread across many containers. KSM is not involved in the write path, in the sense that KSM just tells which container to write to. The individual allocation of blocks etc are done in the container level.

The absence of a centralized namespace manager like namenode makes it hard to implement centralised control of quotas like HDFS. Under HDFS if a new block allocation has the potential to violate the quota then it is assumed that action would violate quota and limits are enforced.

Under ozone, SCM would get container reports which tell us how many bytes of data exist in each container. KSM would periodically read this information and aggregate over volumes. This

[5] Please look at the next section to understand what a container looks like.

leads to a subtle issue that there can be a lag between when KSM learns that a quota violation occurred and actual quota violation occurring.

Hence quotas under Ozone are <u>soft quotas</u>, it is possible for end-user to violate quota, eventually KSM will learn about it and disallow further writes to that volume. User can still delete data and get volume back to an operational state. This delete and recovery is also slow async process since the same mechanisms must kick in for the computation of new quota values.

Container Splits

We have already looked at why we would need container splits, a bucket can have large number of keys and the total size that we store can far exceed the container size. In those cases we would have to split the container based on the key range. A typical example would be a user trying to upload lots of image files to a bucket. As the user progresses with his/her uploads we will reach the container size limit. This can be any configured size (based on network and number of nodes in the cluster) but by default ozone assumes that container size is 5 GB.

So when we go over the 5GB size on that container, the KSM will split that container. When the client is uploading the key that violates the size constraint the container client gets an error which says that the partition is required. The client reports this error to the KSM which initiates a split operation on the container. The container proceeds with the actual split and updates the SCM with the new container information. KSM will learn that information from SCM and update the KSM state.

The actual container split location is chosen by the container itself, and KSM learns this information via client as well as from SCM[6]. This is similar to the close operation in HDFS.

Container Replication Pipeline

As discussed in the original design document, containers are responsible for the metadata and data storage. This pipeline is called *simplified pipeline* since it is simpler compared to the current HDFS replication pipeline.

Replication pipeline makes are two assumptions about container framework :

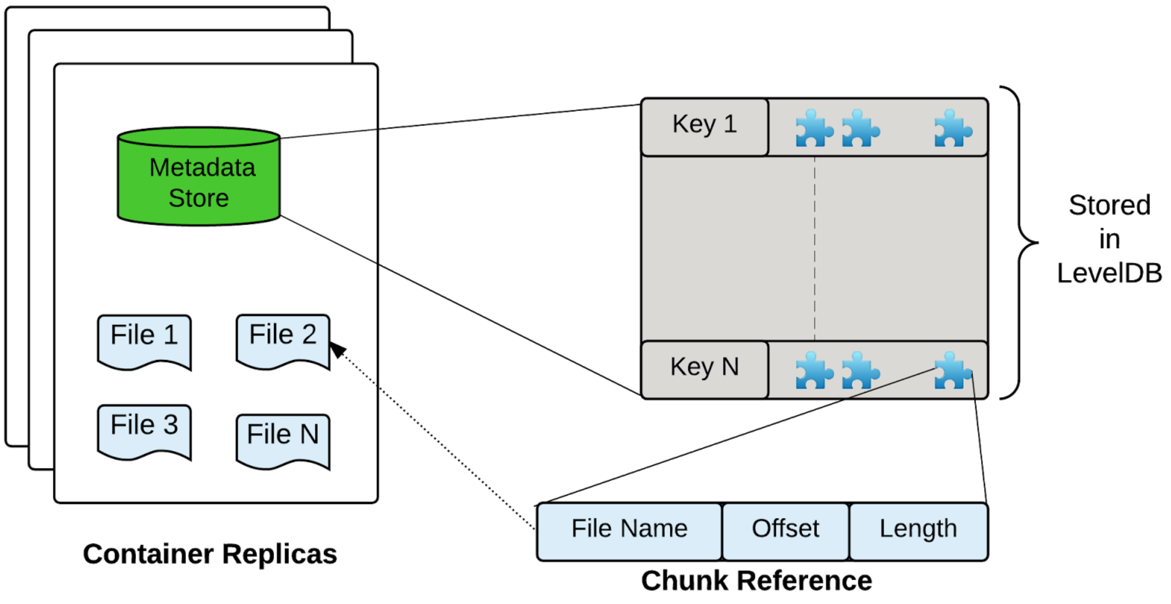
1. Chunks are immutable.
2. Keys offer serializable consistency. This is achieved via RAFT.

For the purposes of this discussion, the second assumption provides us with atomicity and isolation.

[6] A detailed explanation of this is provided in the ApacheCon 2016 , Vancouver, BC - ozone slides. Please look at the bonus slide section.

Ozone Key Write

First let us look at how *Put Key* works without RAFT and then we will see how this process is extended to achieve consistency using RAFT. A *Key* in a container (For example, “Key 1” in the following figure) points to a list of chunk references. A chunk reference (called *ChunkInfo* in source) is a pointer to a block of data.



In order to write an Ozonekey[7] we will write the chunks (i.e. the user data) and then update the key which will point to the list of chunks that make up the key. Writing the key makes the user data visible in the container's namespace.

Let's illustrate this with an example:

1. Write 100 bytes of data to a chunk. Let us write to a file called “block1” at offset 0 and length is 100 bytes. This does not alter the name space yet, since the metadata is not changed yet.
2. Update the key with this information . putKey(OzoneKey -> { {filename:block1, offset:0, length:100}}. This completes an OzoneKey write, we have written the data to a chunk

[7] To avoid confusion between a Key inside container and ozone key, an ozone key is always referred to as OzoneKey in this document.

and then updated the database with KeyEntry which tells us where the datablocks (chunks) live.
3. This means OzoneKey is now visible and can be read by any following request (Strong consistency).

If another client wants to read the OzoneKey, they read the metadata of the Ozonekey, which tells them which chunks to read. This is how a single node in ozone reads and writes data to a container (part of current code in HDFS-7240).

We will now extend this to see how to do this via a consistency protocol like RAFT. With the simplified pipeline and RAFT support, this logic would change to:

1. Write 100 bytes of data to a chunk.
2. Replicate that data into 2 other machines that form the pipeline.
3. If successful[8], perform a putKey operation.
4. putKey is propagated via RAFT. It means that metadata is in a replicated state machine.
5. When the RAFT state machine calls back the container with newly committed key, Container looks up the data block. If data block is not found or checksums do not match then that datanode will try to copy this chunk from one of the other machines. This is a best effort, it is possible for a container client to write a key without any data chunks. Since containers do not try to enforce any correctness.

Ozone Key Read

A client "read key" from the container will return a list of chunk references. Using this list of chunks, we can read the data from the actual chunk file. The only change with RAFT is that the client finds the RAFT leader to retrieve the key-chunks mapping.

Ozone Key Overwrite

We have already claimed that chunks are immutable in this model. Even though we don't intend to support overwrites or partial updates in any near future, it is an interesting problem to discuss because of the immutability assumptions.

For the sake of illustration, let us assume that we have a 16 MB file. We want to overwrite 4 KB in this file starting at offset 8MB. The file sizes and offsets are irrelevant to this problem.

Originally the container metadata or the key for the file looked like this.

Key:OzoneKey> {FileName:Block1, Offset:0, Len:16 MB}

1. We first write the 4 KB to another chunk – let us call this chunk Block2
2. The simple replication pipeline will try to write it to 3 replicas.

[8] We discuss later what success means in context of a pipeline write. Right now, we are focused on the big picture.

3. We do a putKey – with new metadata.

```
Key:OzoneKey ->        {
                       {FileName:Block1, offset:0, Len:8MB},
                       {FileName:Block2, offset:8MB, Len:4KB}
                       {FileName:Block1, offset 8MB + 4 KB, Len: 8 MB – 4 KB}
                   }
```

This does create 2 problems; one is the space is not really reclaimed when we overwrite blocks and second is the if we have full chunk delete how do we delete it.

Both of these issues can be solved by a sweeper thread that will compact data as well as delete unreferenced chunks. However in the Ozone world, we don't support overwrites and by convention we are mapping all chunks of an OzoneKey to a single physical file. Files (chunks) less than 1KB can be kept directly in LevelDB. This is a simple optimization.

Append to Keys

For the sake of completeness, let us discuss how appends will look like in this model. We do not not yet see a need for Ozone to support appends.

Append is a special case of overwrite where the blocks are added at the end of the metadata chain. Imagine a case where someone wants to "tail a file" (Please note: this is not a feature that is supported by ozone, This is purely a theoretical discussion). As seen above, the only limitation in a chunk file is that we need to update the key info for a file to visible to other process. Let us examine a case where we want to see every 100 bytes written to a file. Let us start with 100 bytes of data.

1. OzoneKey -> { {filename:block1, offset:0, length:100}}
2. 100 bytes gets written, and we do another putKey , but this time length is different.
    a. OzoneKey -> { {filename:block1, offset:0, length:**200**}}
3. Again we write 100 bytes more to the chunk, with an putKey update.
    a. OzoneKey -> { {filename:block1, offset:0, length:**300**}}

The frequency in which we can do a putKey will determine how fast a newly written data is visible to the user. That is user will be able to see each 100 bytes update as long as we update the key metadata.

Failures and recovery under Simplified Pipeline Model

The simplified pipeline model avoids the complex recovery that we have in HDFS pipeline. In the simplified pipeline chunks have only two states, they exist or do not. That is, they are immutable and visible only when completely formed. The definition of what is completely formed is defined by the metadata written in a key.

With a classical pipeline – a block moves thru various states. They are Replica Being Written (RBW), Finalized Replica, Replica Waiting to be Recovered, Replica under recovery. Most of the complexity in the classical pipeline comes from error recovery. So let us examine how error recovery would happen in case of simplified pipeline.

Recovery from Disk Failure

Assuming that a datanode has a mechanism to detect disk failures, there are two distinct cases that we need to worry about. The case where a disk loss causes us to lose the levelDB which was replicated via RAFT and case where we lose only lose data chunks.

The first case is when we have lost the container metadata. This case will be treated as complete replica loss. Please see the next section on how we will recover from this case.

In the case where we lose only chunks of data -- that is we have some blocks missing, but no metadata loss, we can copy the blocks from other nodes. This is very simple and localized healing of a given node. One extension that we have to worry about is the case where the loss of disk cannot be healed by the datanode, since it is out of disk space on all other healthy disks. In that case, instead of doing anything complicated, the data node reports the loss of the replica(s) to the SCM. This will cause a new replica allocation by SCM where all data container data will be reconstructed.

Recovery from Node Failure

Recovery from a node failure happens via two distinct mechanisms.

1. A container RAFT leader loss is detected via RAFT heartbeat mechanism and a new leader is elected.
2. A machine loss is detected by SCM - via heartbeat from datanode to SCM. The SCM will instruct the RAFT leader to recruit a new machine by sending the address of the new machine to RAFT leader.

Dealing with Container Loss

In the unlikely scenario of losing all 3 replicas of a container, that is we get a triple fault and lose both data and metadata of a container, it is important to report to users what data has been lost. To address this scenario, we propose a separate scribe service. This service can get key reports -- and it would log that data to persistent store. This service can be used to learn what data was lost in the unlikely event of losing a container. Avoiding a full container loss is an important concern in ozone, and also one of the motivators for using copySets algorithm.

Pipeline Architecture

The classical pipeline architecture uses a complex recovery mechanism when a node in the pipeline does not respond. It tries to find another node to write to. In the simplified pipeline we assume that a write is successful if we have a **quorum of blocks** written out. That is if 2 replicas out of 3 is written out with an update to Raft - then it is considered a reliable write.

If we are <u>not</u> able to write to 2 machines out of 3, that write has failed. We will not try to re-construct the pipeline in any fashion. Instead we will attempt to make a parallel (much slower) write. Any intermediate failure is treated as complete failure. For example, let us say we have a pipeline of A->B->C and the write from A->B fails. We will treat this as a full failure and error is returned to the client. At this point client will attempt a parallel write to A, B, and C. <u>Please note</u> : This is a slower form of write, but we will write from the client directly to A, B and machine C in parallel. If at least 2 machines acknowledge this write, we can write the key info to RAFT ring. <u>Please also note:</u> due to immutability of chunks *write chunk* is an idempotent operation.

If that parallel write also fails, the client is free to go back to KSM/SCM and and ask for a different container and if it is possible KSM would remap the current write to a new container. This raises some subtle namespace explosion issues. To address those KSM could maintain a log and move this write to appropriate location when possible.

Over Replication

Ozone does not support over-replication as a feature. If we decide to support it in the future, this is how we could to address it[9].

1. One of the architectural <u>simplification</u> we have made is to have the blocks associated with the container on the same machine. In other words, for each container with metadata, the associated data blocks ***must*** exist on the same machine. This allows us to immediately know from RAFT what is the health of a given key (collocation[10] requirement).
2. A set of data blocks can exist by itself, that is on machines without container metadata - Let us call them **remote blocks**. These blocks map to the over-replication factor specified by the user. With RAFT, each chunkInfo -- will contain the location where these chunks live. To illustrate with an example, let us say a key was written to a Raft ring, R1, R2, R3. Due to the invariant 1, we know that data blocks (chunks) must exist on machines R1, R2 and R3. But the user is free to specify Replication factor 10 for that file, and we will write those chunks to a set of machines {M4-M10), which will be the <u>remote</u>

[9] Over-replication is a feature in HDFS, especially given the fact that each file's replication factor can be controlled individually. In theory we could support the same capability with the simplified pipeline.

[10] In the EC world we might have to break this requirement, and assume that all blocks are always remote.

blocks. when we schedule jobs on M4 to M10, the metadata for these remote blocks will have to be read from the RAFT ring.

3. If we decide to support remote blocks, SCM would send a “state of the world” report to the data nodes. That is if a datanode is failed or in missing state, SCM would send that info to all datanodes. Each datanode can identify which remote blocks are missing and decide to re-replicate those blocks as needed. Instead of KSM/SCM being responsible for block loss identification and re-replication, we are suggesting that RAFT leader in coordination with KSM/SCM be in charge of such activity. This will help us scale instead of having the block report processing done by a single service.

SCM picks which containers to put the data into, But SCM only knows about how much data exists in each container and frequency of access to that container from container reports. It does not know about the keys, the information about keys is inside a container. So BlockPlacemenPolicy in the classical sense does not exist under ozone. It is a shared responsibility between KSM/SCM and the container itself.

Once again, if we decide to support remote blocks (which at this point Ozone does not need), we will not have to heartbeat remote containers. We will rely on SCM notifying us about the loss of datanodes via a “state of the world” report..

Health of a Container

Because of the presence of Raft protocol, a container is self-aware of its health. That is a node loss is auto-detected either by the leader or by the followers and they can heal themselves. Adding a new node to RAFT ring is the only external action needed. This means that we don’t need container reports between the machines. In case of disk failure, we are assuming that node is capable of detecting it and copying the data from the external sources.

Only case where we will need container reports is to let the SCM /service know how much data we have inside the containers.

For the scribe service, We will send key reports so that we can maintain an offline info about the namespace in the cluster.

The health of container can be thought of as similar to what we do for blocks now. We can classify them as over-replicated, replicated and under-replicated.

Comparison with original Ozone architecture proposal

This section summarizes the evolution of the Ozone design since the v1 architecture [1].

KSM changes

KSM is a new component, however the original ideas are embedded in the first proposal.

Page 6: Ozone Architecture.

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. .

We are simply proposing that we move the namespace management from SCM to KSM. With no additional changes, this changes allows us to scale ozone's namespace operations easily.

If you look at how KSM maintains the information about volumes, buckets and container mapping, they are exactly same as the earlier architecture.

Page 4: Ozone Architecture.

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.

One change is the relaxation of the constraint that a storage container can have only one partition. In the implementation phase of containers, we realized that this constraint is not buying us anything, and it should be the decision of SCM/KSM on where the data should land. This also allows us to deal with the edge case of lots of volumes and buckets without any data on an ozone cluster.

Another subtle change is in terminology – instead of saying partition/storage containers we have started using range/containers as the standard terminology. This is due to the fact that ozone had favoured hash partitions initially but based on community feedback we have moved to using range partitions.. Also instead of using the word blocks, we have started using the word chunk.

Page 8: Ozone Architecture.

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.

Metadata Management

Since we are proposing the KSM and SCM spilt, let us compare how the new architecture will look like in comparison to old architecture.

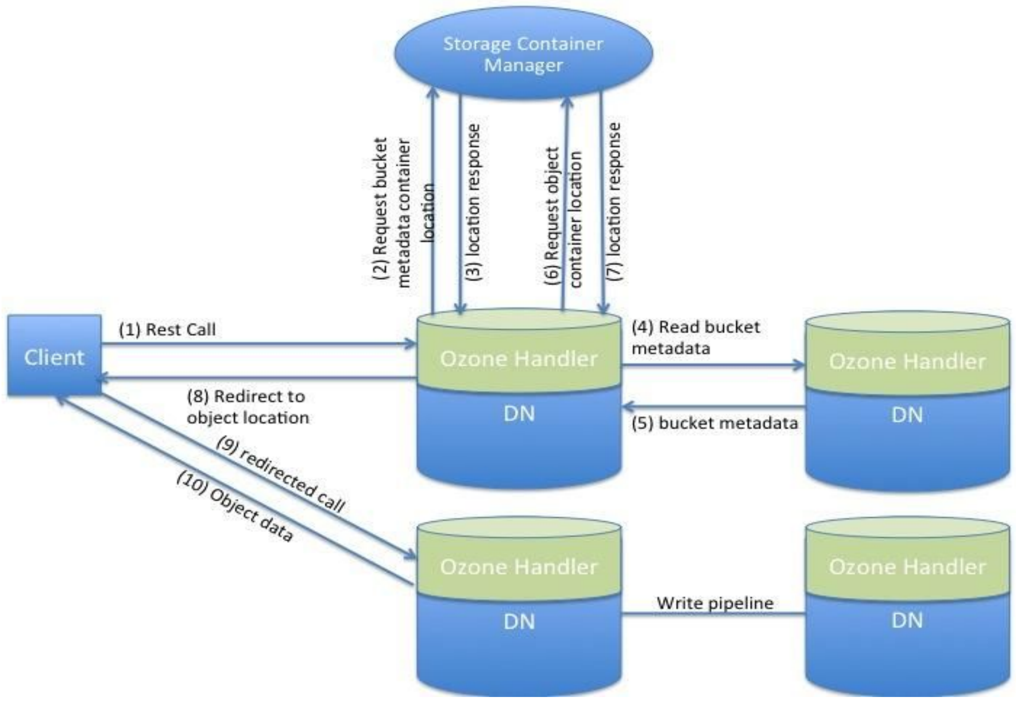


Figure 1 : Proposed Architecture from Ozone Architecture doc

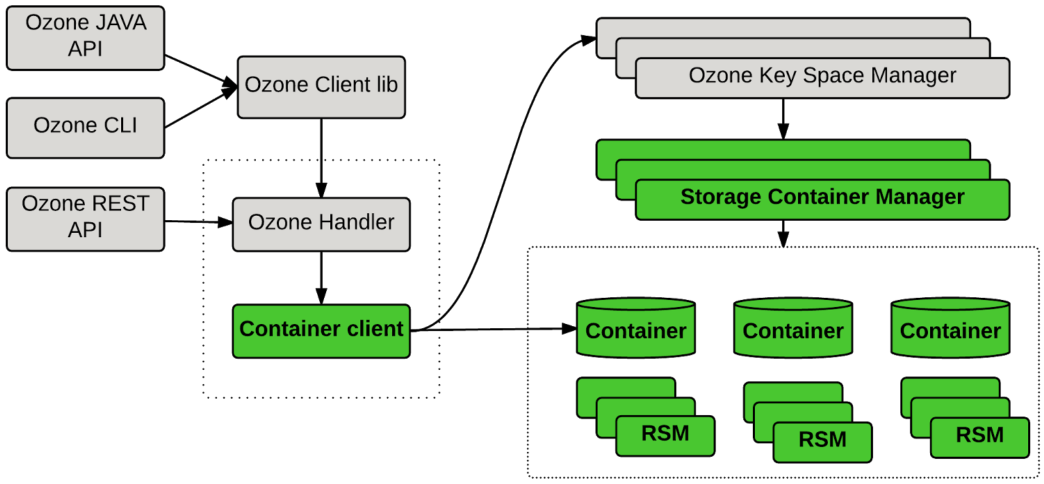


Figure 2: Updated Architecture

Even though the pictures look very different, because one is a sequence diagram and other is a block diagram (with all components broken out), If you compare them you will see the only real difference is the KSM existing as a new service and RSM.

The reasons of why we would like to have a separate KSM is already explained. Now let us briefly look at the RSM(Replicated State Machine) change instead of write pipeline. RSM uses write pipeline to replicate data. This is exactly same as proposed by the original design document. The original design document omitted some of the detail. However in the page: 4 of the original design document it says "*We offer strong consistency for container replicas for successful operations. ".* How that consistency will be achieved is not discussed in that document. In fact, on the section on data pipeline consistency – it says we will have a data pipeline and "*More detailed design of the pipeline will be posted on the corresponding hdfs jira*." (page 9: Ozone Architecture). This proposal spells out how that consistency would be achieved.

Page 8: Ozone Architecture

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.*

Storage containers are now called containers. No other change in relation to the original spec.

Page 8: Ozone Architecture

- *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*

To achieve this promised consistency and durability we are proposing that we use RAFT. This proposal have suggested the details of consistency protocol and how it would be used.

Here is the summary of changes and clarifications of ozone design from V1 to this version (V2).

Status	Original	Current
Change	SCM managed namespace	KSM, since it allows for more scalability. This is at the proposal stage.
Clarification	Data pipeline replication	We made a promise of serializable consistency on data node replication. We are proposing we use a replicated state machine to achieve that consistency.
Change	Hash Partition	**Range Partition**, but this was due to strong push back from Colin. So I am presuming that it is community driven. Even though I cannot find that info in the JIRA.

References

1. Jitendra Pandey. Architecture spec for an Object Store in Apache HDFS.
2. Edward Bortnikov et al. HDFS-5477: Block Manager as a Service.
3. Haohui Mai et al. HDFS-5389: A Namenode that keeps only a part of the namespace in memory.
4. Anu Engineer. Ozone User Guide.
5. Cidon et al. Copysets: Reducing the Frequency of Data Loss in Cloud Storage. USENIX ATC 2013.