Container Location Cache in OM

Introduction

Ozone clients (S3G or OFS) need to get the container location information to read the data from the correct data nodes. This is achieved by the *lookupKey* API call implemented by OM. OM as part of the lookup does the following

1. Calculate the security tokens that clients can use to read data from the data nodes
2. Get the latest up-to-date container location from SCM StorageContainerLocationProtocol::getContainerWithPipelineBatch.

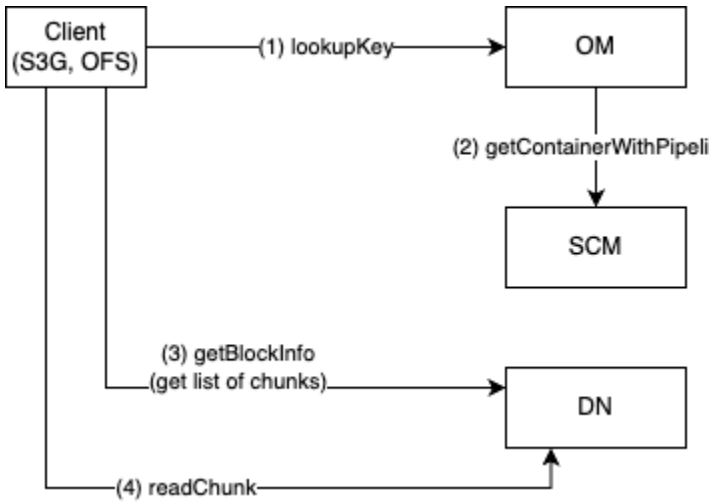


Figure 1: Current workflow for clients to read a key

As a rough calculation, in other to have a performance similar to HDFS NameNode (hundred thousand(s) queries per second), the key lookup operation in OM has to perform with a submillisecond latency. And as shown in the following graph, the mentioned two steps dominate the latency of lookup calls with latencies as a matter of milliseconds. They can be very well defined as the 2 blockers preventing OM from reaching max performance.

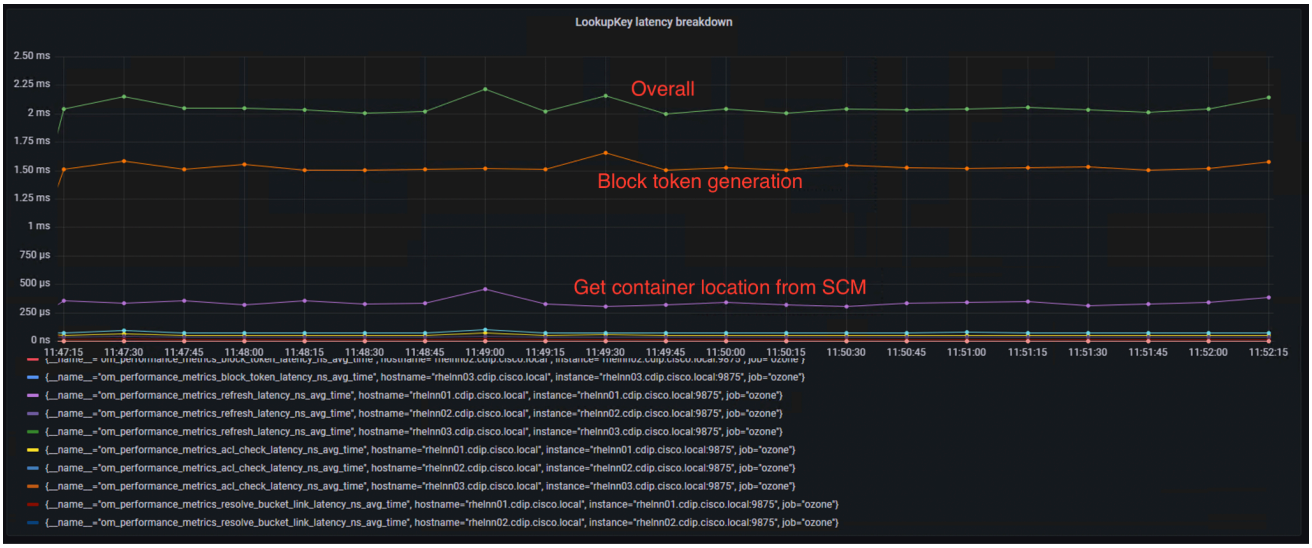


Figure 2: lookupKey latency breakdown

We did a measurement on an internal NVMe cluster that sequentially removes block token generation and prototypes a container location cache in OM, the result shows that they significantly improve the read OM throughput.

Test	LookupKey OPPS
Baseline	**13.4K**
Remove block token generation	**29K**
Prototype container location cache	**44K**

The block token generation will be addressed by a separate effort to redesign the usage of key to sign the block identifier.
The step to acquire block location today sends an RPC call to SCM which results in reducing overall performance for OM and introduces a read load on SCM that is directly proportional to the read load on OM. This document describes a proposal to eliminate latency by avoiding the direct call to SCM for every lookup.

High-Level Proposal

To avoid the extra RPC calls to retrieve container pipeline information, OM can have an eventually consistent read-through cache designed to locally store all containers' pipeline information.

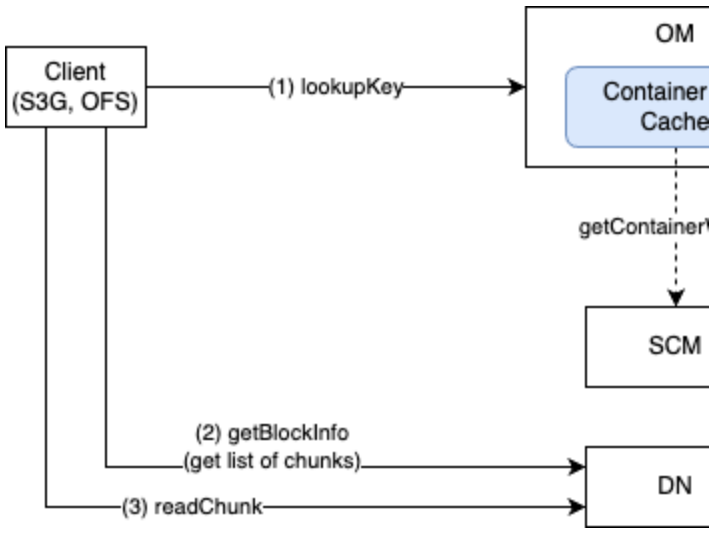


Figure 3: Proposed workflow for clients to read a key

This cache will eliminate almost all traffic from OM to SCM to retrieve container locations. OM will still occasionally call SCM in case of cache misses or when OM clients realize that a block location of a given key is outdated and specifically ask OM to refresh the key location. The next few sections will lay out the implementation details of the cache and how it is used.

Cache Implementation

Interface

The cache will be a pluggable component of ScmClient which will provide the following methods to allow container location lookups.

```
ScmClient {
  Cache<ContainerID, Pipeline> containerLocationCache;

  // get container location from the cache if it exists, otherwise call SCM.
  Pipeline getContainerLocation(ContainerID containerID);
  Map<ContainerID, Pipeline> getContainerLocations(Collection<ContainerID>
containerIDs);
}
```

Cache Population

For simplicity, we propose to start with an on-heap cache whose items are populated on cache-miss only. This will leave a concern of slow performance when OM fresh starts or on failover to followers. In the Optimizations section, we'll introduce improvements to address this concern.

Cache Size Estimation

The cache key is *ContainerID* and the cache value is *Pipeline* enclosing all information needed by a client to connect to and read data from data nodes. Typically, a *Pipeline* instance created for reading purposes contains the following fields.

```
Pipeline {
   // each pipeline has 3 data nodes, typically.
   DatanodeDetailsProto members = 1;
   ReplicationType type = 3;  // 4 bytes
   ReplicationFactor factor = 4; // 4 bytes
   PipelineID id = 5; // 8 bytes
   ECReplicationConfig ecReplicationConfig = 11; //16 bytes
}
```

The size of a cached Pipeline instance can be relatively small, as much as a matter of 50 bytes, if it doesn't include *DatanodeDetails* instances. The size of a *DataNodeDetails* instance is variable and depends on the length of host names and network topology. One instance of the type can be somewhere between 200 bytes to 1KB.

```
DatanodeDetailsProto {
   string ipAddress = 2;      // IP address, eta 50 bytes?
   string hostName = 3;       // hostname, eta 50 bytes?
   repeated Port ports = 4;           // has a name and value, eta 20 byte
   optional string certSerialId = 5;   // Certificate serial id.
   // network name, can be Ip address or host name, depends, eta 20 bytes?
   string networkName = 6;    // eta 20 bytes
   string networkLocation = 7; // Network topology location, eta 20 bytes?
   NodeOperationalState persistedOpState = 8; // The Operational state
persisted in the datanode.id file
   int64 persistedOpStateExpiry = 9; // The seconds after the epoch when the
OpState should expire
   UUID uuid128 = 100; // UUID with 128 bits assigned to the Datanode, 8
bytes.
}
```

Note: In Java, String instances created via (de)serialization are not interned. If it can be enabled for Protobuf, that can be a significant improvement in the memory footprint.

Based on the above estimation, the total memory needed to cache 1 million containers can be around **650MB to 3GB**. And it can be higher for an on-heap cache with Java Object overhead. We'll see how to improve the memory footprint in the Optimizations section.

Cache Library

Below is the list of basic requirements for a Cache Library to match the use-case:

Bounded size, with LRU eviction policy	Must
Automatic refresh	Must
License	Apache?
Ability to switch to off-heap?	Optional

We'll go with Caffeine, a successor implementation of Guava cache, for interface familiarity, performance, and active community.
We can also make the cache implementation (like Caffein, Ignite, EHCache…) pluggable by defining a standard cache interface to depend on. That would allow the community to extend the Cache implementation options when they find it necessary.

Client Interactions and protocol changes

With the OM container cache in place, OM will always return the location information it finds in its cache first with the risk of that container cache information being stale. A client can request OM to force refresh the container location information from SCM if it detects an error with reading from DN.

Since older clients do not have an option or the code in the lookup request processing to indicate that it needs a refresh for the container location information. A new API will be implemented that allows for the updated interaction between client and OM. This would keep older clients using the slower read path with a refresh from SCM for each look up. This new protocol would also allow clients to skip seeking volume info and/or bucket info before getting the key information. (https://jira.cloudera.com/browse/CDPD-43095)

```
@@ -864,6 +864,8 @@ message KeyArgs {
     // When it is a head operation which is to check whether key
exist
     optional bool headOp = 18;
     optional hadoop.hdds.ECReplicationConfig ecReplicationConfig =
19;
+    // Force OM to update container cache location from SCL
+    optional bool forceUpdateContainerCacheFromSCM = 20;
 }

+message GetKeyInfo {
+  required KeyArgs keyArgs = 1;
+}
+
+message GetKeyInfoResponse {
+  optional KeyInfo keyInfo = 1;
```

```
+  optional VolumeInfo volumeInfo = 2;
+  optional string UserPrincipal = 3;
+}
```

1. Client calls OM `GetKeyInfo`, with `forceUpdateContainerCacheFromSCM = false`, to get key location. OM populates location information based on container location in its cache.
2. Client retrieves the list of chunks by calling Datanode's *getBlock* RPC.
3. Client reads chunks data by calling Datanode's *readChunk* RPC.
4. During (2) & (3), if the RPC request results in one of the following conditions, the client will repeat step (1) but with `forceUpdateContainerCacheFromSCM = true` to inform OM to refresh the relevant container location.
    a. Datanode is unreachable.
    b. Datanode returns a *StorageContainerException*, indicating that it got a problem reading the block/chunk from the requested container. The list of possible cases can be found here.
5. When OM sees `forceUpdateContainerCacheFromSCM = true`, it forces a refresh of the relevant container locations by calling SCM, updates its cache, and returns the updated location to the client.

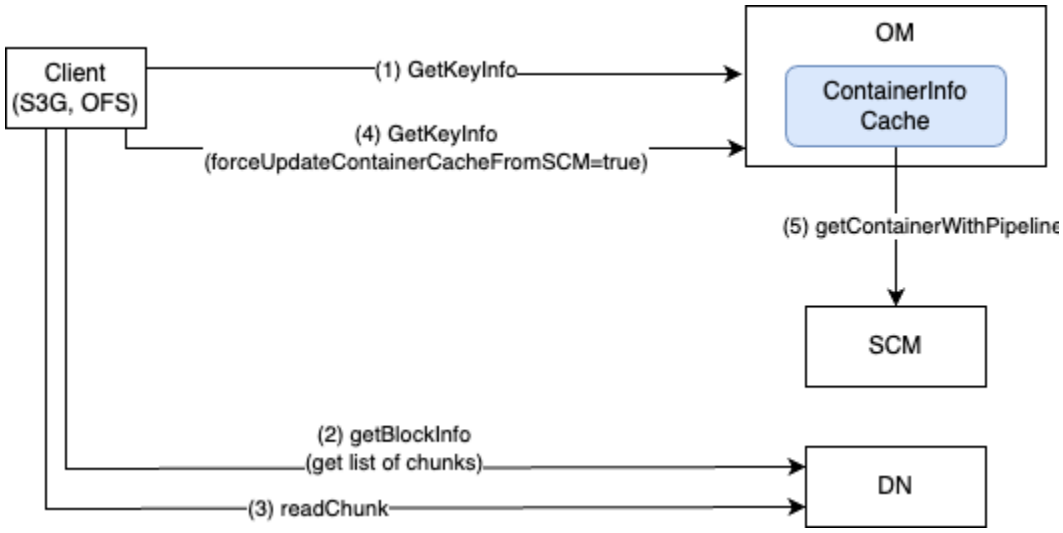


Figure 4: Proposed client interactions to make the container info eventually consistent

Optimizations

The previous section describes how the components layout and their interactions. Yet, there're still a couple of concerns like the cache memory footprint or performance after a fresh OM start or failover. This section describes some optimizations to address those concerns.

Optimization 1: Periodic refresher to reduce the chance of inconsistencies and cache misses

This can be logically split into 2 sub-objectives.

A. Refresh current keys in the cache

This is to reduce the risk of returning inconsistent container location data to customer while they are updated in SCM. It can be simply implemented by the chosen Cache Library feature. E.g. Caffein allows specifying refresh policy, e.g. after 5 hours from the last time a key is written.

B. Pre-populate keys not currently in the cache

As mentioned in the previous section, OM will start right away and the cache will be populated on-demand. However, to reduce the risk of cache misses, an asynchronous refresher can be implemented to 1) prepopulate the cache when OM fresh starts or 2) keep the cache up to date with new containers being allocated in SCM.
A new SCM API needs to be added to allow this happens:

- *getContainerLocation(lastUpdate: timestamp)*: return all container location created since the *lastUpdate* timestamp. The API will support pagination, with 1000 as the default page size, when the list of results is huge.

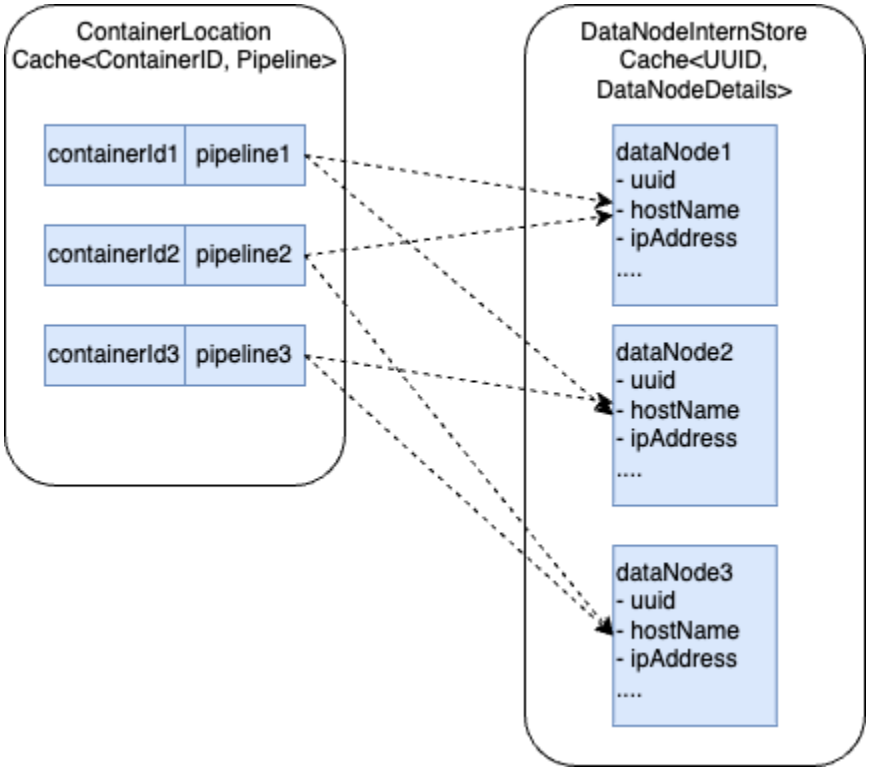
After a fresh OM start, refresher calls *getContainerLocation* with 0 as the timestamp and this will fetch all the container locations known by SCM. Assuming it takes *100ms* for OM to load and process a page of 1000 container locations, it will take less than 2 minutes for the refresher to pre-populate the full cache in the background.

Optimization 2: Followers can replicate the container cache update

An optimization to reduce cache miss when OM fails over is to mirror every on-demand SCM read from the OM leader to its followers using Raft state machine. This will keep the followers updated without duplicating loads to SCM.
The implementation of this optimization will leverage the existing Ratis state machine infrastructure that is already built for OM metadata replication. We will add a new OMClientRequest, called *OMContainerCacheUpdate*, to replicate cache updates (after each cache miss or client force refresh).

Optimization 3: Reduce cache size by deduplicating cached DataNodeDetails

A simple on-head cache optimization to reduce the cache size is to introduce a DataodeDetails cache to deduplicate DatanodeDetails, assuming that data node information, identified by its UUID, wouldn't be modified during a data node lifecycle. When getting new Pipelines from SCM, we replace the DataNodeDetails instances with the ones already in the cache.
This would reduce the cache size of container location to **50MB** for **1M** containers. The size of DataNodeDetails cache is proportional to the number of data nodes in the cluster. 10K data nodes would take a cache of **2 - 10MB**.



The downside of this approach is that it evicts a number of DatanodeDetails instances which results in overhead for GC. However, this only happens when OM has to refresh container locations from SCM and that is not supposed to be the main path.
Moreover, this optimization is coupled with on-heap cache. On the other hand, it enables on-heap cache by reducing the total size long-live objects and on-heap cache indeed delivers a performance advantage by avoiding serialization.

Optimization 4: Reduce chatty communication in the container location RPC

The response of *getContainerWithPipeline* includes pieces of unnecessary or repeatable data:

- ContainerInfo: completely not used for this use-case.

- Pipeline: include a number (3) of full DatanodeDetails, which are highly repeatable and cachable.

One way to improve this is that SCM instead provides 2 APIs:

- *getContainerPipeline(ContainerID):* returns a Pipeline with data node UUIDs only.
- *getDatanodeDatails(list of dataNodeUUIDs):* which OM can call to fetch and cache *DataNodeDetails* . Whenever OM detects new data node IDs from the previous API response, it fetches and caches the missing data node details from SCM.

This will reduce the data footprint of a container location from **1KB to 50 bytes**, and eliminate the creation of many duplicated DatanodeDetails thus relaxing GC workload.

Conclusion

We can predict that the periodic refresher (optimization 1) and the replication of cache update (optimization 2) would not necessarily improve the read throughput, but rather improve latency at P99 or P999 (outliners). Even after fail-over or a fresh start, the cache will be quickly filled up with the necessary container locations, assuming that the number of keys per container is very big.

Optimization 3 seems to be more essential, yet only really compulsory when there're 100K containers or more, which is a huge number.

Finally, we will need proof proving the necessity of each mentioned optimization.

Test plan

- Integration tests to ensure key locations are refreshed when being stale.
- Performance test to verify the performance improvements.

Tasks Breakdown & Estimation

Phase 1: Adding container location key cache to improve core OM read path

- Implement the cache in ScmClient and update lookupKey and other relevant APIs. The *refreshLocation* param should be defaulted to *true* to keep it backward/forward compatible so that this task can be checked in independently.
- Cache size verification for 1 million containers.
- Implement client interaction, call OM with *refreshLocation=false* for the initial lookup and *refreshLocation=true* when hitting the discussed condition.
- Integration test cases.

- Performance test.

Phase 2: Optimizations (only if necessary).

- Optimization 1: Periodic refresher.
- Optimization 2: Followers can pre-populate the container cache
- Optimization 3: Reduce cache size by deduplicating cached DataNodeDetails.
- Optimization 4: Reduce chatty communication in the container location RPC.