Ozone Write Pipeline with Ratis Streaming

2021.6.9

The Ratis Streaming feature has been added by RATIS-979. It has been demonstrated that Ratis Streaming has outperformed Ratis Async API (RATIS-1209), the existing Ozone write pipeline implemented with Ratis Async API (RATIS-1234) and the HDFS write pipeline (RATIS-1312). Ratis Streaming can also scale to support more concurrent clients compared with the other implementations.

In this document, we discuss how to implement a new Ozone write pipeline with Ratis Streaming. For failure handling, see Ozone Write Streaming Pipeline in HDDS-4454.

Write Pipelines

Ozone write pipeline currently is implemented with the Ratis Async API. Although the implementation can achieve a high performance, it does not optimize the network resources and the datanode memories when writing large data objects.

Problem 1: The leader becomes a bottleneck. All the async requests have to be sent to the leader first. The leader forwards the requests to the followers via appendEntries. Therefore, the leader is a hot spot. A workaround in Ozone is to create more Ratis groups for the same set of datanodes in order to distribute the leaders evenly. For example, suppose there are 3 datanodes. Create 3 Ratis groups and assign a leader to each datanode. However, it requires more resources to create Ratis groups and the group management becomes more complicated.

Problem 2: It requires a large chunk size (e.g. 16MB) in order to have a high performance for writing large objects. When the chunk size is small, it will generate a lot of Ratis transactions and large Ratis log files since each write-chunk request generates a Ratis transaction. The datanodes become busy handling these transactions. However, a large chunk size means a large request size. It requires more memory to cache the requests in the datanodes.

Problem 3: Each chunk is copied multiple times in the client and servers. The Ratis Async API is implemented with Protobuf and gRPC. Unfortunately, Protobuf and gRPC require multiple buffer copying when sending and receiving data. In order to reduce the number of buffer copyings, Ozone has implemented a set of UnsafeByteOperations. However, it still cannot achieve zero buffer copying. Moreover, it is unsafe.

Write Pipelines with Ratis Streaming

Ratis Streaming can increase the efficiency of writing large objects in Ozone.

Solution to Problem 1: With Ratis Streaming, Ozone clients can stream data to any datanodes (leader or follower). Indeed, an Ozone client should stream to the closest datanode. Then the datanode will forward the stream to the other datanodes in the Ratis group. The leader is no longer a bottleneck. Ratis Streaming can also take the advantage of network topology to minimize the network traffic.

Solution to Problem 2: In Ratis Streaming, the chunk size can be small (e.g. 1MB) since Ratis Streaming generates a Ratis transaction per object, not per chunk. A small chunk size would not increase the number of transactions.

Solution to Problem 3: Ratis Streaming is implemented with Netty zero buffer copying.

Basic Implementation

We propose to add a new implementation of Ozone write pipelines with Ratis Streaming. Note that the new implementation is not a replacement of the existing implementation since the new implementation will be optimized for writing the large objects. For writing small objects, the existing implementation is more efficient.

Client-Side

The current client-side implementation has

1. OzoneOutputStream (and CryptoOutputStream),
2. KeyOutputStream,
3. BlockOutputStreamEntryPool,
4. BlockOutputStreamEntry, and
5. BlockOutputStream.

The OzoneOutputStream has a KeyOutputStream. The KeyOutputStream may possibly be wrapped by a CryptoOutputStream first. The KeyOutputStream has a BlockOutputStreamEntryPool which is logically a list of BlockOutputStreamEntry(s). BlockOutputStreamEntry is a wrapper of BlockOutputStream. Then, BlockOutputStream uses the XceiverClientSpi to send WriteChunk commands to the datanodes. The WriteChunk commands are wrapped as ContainerCommandRequestProto(s), which supports many different command types (ContainerProtos.Type). For WriteChunk, the XceiverClientSpi is implemented by XceiverClientRatis, which is further implemented by the Ratis Async API. Note that BlockOutputStream and some other classes extend java.io.OutputStream which writes with byte[] and requires buffer copying.

In the new implementation, the proposed new classes are

1. OzoneDataStreamOutput,
2. KeyDataStreamOutput,
3. BlockDataStreamOutputEntryPool,
4. BlockDataStreamOutputEntry, and
5. BlockDataStreamOutput.

The relationship between these classes are the same as before. In order to avoid buffer copying, the classes above would not extend java.io.OutputStream and the new request would not be wrapped as a ContainerCommandRequestProto. BlockDataStreamOutput will use Ratis DataStreamOutput to stream data. All these classes use java.nio.ByteBuffer (instead of byte[]) in the API so that zero buffer copying becomes possible (as an example, see the FileStore example in Ratis).

Datanode-Side

For the datanode-side, the ContainerStateMachine in Ozone already has implemented all the methods for the Ratis Async API. For Ratis Streaming, ContainerStateMachine should also implement the stream(..) method and the link(..) method in Ratis StateMachine.DataApi.

The stream(..) method is to create a StateMachine.DataStream for receiving incoming stream data from the client. Ozone should implement StateMachine.DataChannel, which extends WritableByteChannel, for writing incoming stream data to its local storage.

The link(..) method is to link the incoming stream with a Ratis log entry. Since the ContainerStateMachine already can handle WriteChunk requests for the Async API, the link(..) should use similar code to implement.

Note that the Ratis RaftServer handles the incoming network traffic. No change is required for receiving the Ratis Streaming requests.

Further Improvement

Sharing Executors between Async API and Streaming

The ContainerStateMachine has a list of executors to run commands. Ratis Streaming has its own writeExecutor (in DataStreamManagement) to write incoming stream data to the local storage via StateMachine.DataChannel. Ratis should provide an option to allow the StateMachine to pass an executor (this requires Ratis change). Then, ContainerStateMachine will be able to share executors between Async API and Streaming.

Topology Awareness

Ratis Streaming has a RoutingTable API in order to support topology awareness. A RoutingTable is to tell the Datanodes in a Ratis group how to forward the stream in order to minimize the network traffic. If Ozone can pass a RoutingTable when creating a DataStreamOutput, the Ratis will be able to stream the data optimally.

Note that the pipeline in XceiverClientRatis has the nodesInOrder information. We may be able to use it to build a RoutingTable.