Move replication queue storage from zookeeper to a separated HBase table

Previous design doc

Move HBase replication tracking from ZooKeeper to HBase

Background

Almost the same with the previous design doc so just copy it here
“
It is a long story that we want to depend less on zookeeper in HBase and finally purge it so we can easily deploy HBase on cloud.
Replication tracking system is one of the subsystems in HBase which heavily depends on zookeeper. In our(Xiaomi) deployment, we use replication heavily, and sometimes the replication will bring down the zookeeper cluster due to too many znode or too large znode, and cause a lot of trouble.
So here we propose to move the replication tracking system off zookeeper, to a system table in HBase.
”
In this design doc, we will focus on the replication queue storage, as for replication peer, the data is small and it is easy to find somewhere to store it.

The ‘deadlock’ problem

In the current HBase WAL and replication implementation, when replication is enabled, we always need to store the newly created WAL file in ReplicationQueueStorage. The work must be done before we allow actual WAL entries to go into the newly created WAL file.
But if we implement ReplicationQueueStorage based on a system table, obviously, the region of this table needs a WAL file to write WAL entries when we want to store something into it.
This introduces a cyclic dependency, think of we do not have any regionserver in the cluster yet, and once a region server is started, we will assign the system table to it, and during the assignment processing, we need to create a WAL file and write the open marker to it, but before we can writing anything into the WAL file, we need to store this WAL file to the system table. Deadlock.

Possible solutions

Introduce a special WAL file for system table

This is proposed in the previous design doc. The basic idea is to have a special WAL file, which does not need to be stored into the replication queue system table. This could break the tie.
But there are still some disadvantages:

1. Need to make a separate WAL provider for system tables too, so we can skip adding ReplicationQueueStorage to this WAL provider.
2. Need to add more steps in SCP to recover this system table first.
3. Need to implement special bootstrap logic to initialize this system table before serving.

And the most critical problem is that ZooKeeper is designed to be HA and its failover is pretty fast. But for HBase, if the region server which holds the region which stores the replication queue information crashes, we will hang the WAL rolling for the whole cluster for a 'long' time(usually tens of seconds or even several minutes). This will hurt the availability of the HBase cluster.

Track replication offset per queue instead of per file

The basic idea is that, for a normal replication queue, where the WAL files belong to it is still alive, all the WAL files are kept in memory, so we do not need to get the WAL files from replication queue storage. And for a recovered replication queue, we could get the WAL files of the dead region server by listing the old WAL directory on HDFS. So theoretically, we do not need to store every WAL file in replication queue storage. And what's more, we store the created time(usually) in the WAL file name, so for all the WAL files in a WAL group, we can sort them(actually we will sort them in the current replication framework), which means we only need to store one replication offset per queue. When starting a recovered replication queue, we will skip all the files before this offset, and start replicating from this offset.
For ReplicationLogCleaner, the logic is also straight-forward, all the files before this offset can be deleted, otherwise not.
In this way, theoretically we do not need to touch the replication queue system table every time when creating a WAL file, which could solve the most critical problem mentioned in the above solution.
Of course, the implementation in the real world is not easy, we need to deal with a bunch of corner cases. When implementing the POC of solution, I've already hit lots of problems, will list more detailed designs below.

The API and storage format

The data structure and API

The queue related methods in ReplicationQueueStorage

```
  void setOffset(ReplicationQueueId queueId, String walGroup, ReplicationGroupOffset offset,
    Map<String, Long> lastSeqIds) throws ReplicationException;

  Map<String, ReplicationGroupOffset> getOffsets(ReplicationQueueId queueId)
    throws ReplicationException;

  List<ReplicationQueueId> listAllQueueIds(ServerName serverName) throws
ReplicationException;

  List<ReplicationQueueId> listAllQueueIds(String peerId, ServerName serverName) throws
ReplicationException;

  List<ReplicationQueueData> listAllQueues() throws ReplicationException;

  List<ServerName> listAllReplicators() throws ReplicationException;

  Map<String, ReplicationGroupOffset> claimQueue(String peerId, ReplicationQueueId
queueId,
    ServerName targetServerName) throws ReplicationException;

  void removeQueue(ReplicationQueueId queueId) throws ReplicationException;
```

The methods for reading last sequence ids for serial replication, and HFile references are still the same so not posting it here.

ReplicationQueueId

```
public class ReplicationQueueId {

  private final ServerName serverName;

  private final String peerId;

  private final Optional<ServerName> sourceServerName;

  public ReplicationQueueId(ServerName serverName, String peerId) {
    this.serverName = serverName;
    this.peerId = peerId;
    this.sourceServerName = Optional.empty();
```

```
  }

  public ReplicationQueueId(ServerName serverName, String peerId, ServerName
sourceServerName) {
    this.serverName = serverName;
    this.peerId = peerId;
    this.sourceServerName = Optional.of(sourceServerName);
  }

  @Override
  public String toString() {
    StringBuilder sb = new StringBuilder().append(serverName.toString()).append('-
').append(peerId);
    sourceServerName.ifPresent(s -> sb.append('-').append(s.toString()));
    return sb.toString();
  }
}
```

The serverName field is always the region server which is currently replicating this queue, the optional sourceServerName is the region server which creates and owns this replication queue at the beginning. For a normal replication queue, the sourceServerName is not presented, for a recovered replication queue, the sourceServerName is presented. Notice that, we do not need to record all the region servers which have replicated this queue in the history, as we only need to replicate the WAL data of the first region server.

ReplicationGroupOffset

```
public class ReplicationGroupOffset {

  public static final ReplicationGroupOffset BEGIN = new ReplicationGroupOffset("", 0L);

  private final String wal;

  private final long offset;

  public ReplicationGroupOffset(String wal, long offset) {
    this.wal = wal;
    this.offset = offset;
  }
}
```

The meaning of this class is straight forward. As said above, we can sort all the WAL files in a replication queue, so the offset of a replication queue is just a WAL file and the offset in this file. We do not implement Comparable here because the code for splitting the WAL file name is in the hbase-server module while this class is in the hbase-replication module.

The schema in hbase:replication

Replication Queue

Let's name the family 'queue'.
The row key will be in format
<PeerId>-<ServerName>[-<SourceServerName>]
The WAL group will be the qualifier, and the serialized ReplicationGroupOffset will be the value.
We only need to retain the newest version of the value.
Here we place the peer id first because

1. Since peer id can not contain '-' but server name can, it will be easier to parse the rowkey back when peer id comes first and we use '-' as the separator.
2. We can group all the replication queues for a replication peer together, so we can split the hbase:replication table. See the Split Policy for more details.

Of course, this will increase the difficulty on implementing listAllQueueIds, but it is still possible as we know the format of the row key, so it is easy for us to jump directly to the next peer id in the table, and then construct the correct prefix to scan for the replication queue. And we could also add a method which accepts two parameters, peer id and server name, to list queue ids, as we can get all the replication peers from other places.

Last Sequence Ids

Let's name the family 'sid'
The peer id will be the row key.
The encoded region name will be the qualifier and the last sequence id will be the value.
We only need to retain the newest version of the value.
The reason why we put all the data in one row is for simplifying the remove peer logic, as we can delete all the data with one delete family.

HFile References

Let's name the family 'hfileref'
The peer id will be the row key.
The hfile ref will be qualifier and we just need to store a dummy value.
The reason why we put all the data in one row is for simplifying the remove peer logic, as we can delete all the data with one delete family.

Split Policy

Since the atomic update of replication queues are always within a replication peer, and all the row keys start with peer id, we can make use of DelimitedKeyPrefix SplitRestriction where the delimiter is '-' to make the hbase:replication splittable.

The 'claim queue' operation

For the storage implementation, the claim queue operation is just to remove the row of a replication queue, and insert a new row, where we change the server name to the region server which claims the queue. If sourceServerName is not present, i.e, we are claiming a normal replication queue, then we fill the old leading server name as sourceServerName.

On the whole claim queue operation in HBase, one of the problem is that, since now we will not always record the replication queue offset when there is a new WAL file in the replication queue, it is possible that when we arrive the claim queue stage of SCP, there is no replication queue record in hbase:replication but actually, we need to 'claim' the replication queue, otherwise there will be data loss in the peer cluster.

The idea is straightforward, in the claim queue stage of SCP, we list all the replication peers, create a replication queue for it if it does not exist yet, and then let other region servers to claim it. This will introduce a dependency on hbase:replication table for SCP, but when arriving at the claim queue stage, we can make sure all the regions on the dead server are already online, so it will not introduce the 'deadlock' problem back.

There are mainly two problems for this solution:

1. A region server may have a bunch of WAL files already, and when adding a new peer, only the newest WAL file needs to be replicated. But if we haven't replicated anything yet, there will be no replication queue record for this peer, and then the region server crashes, with the above solution, we will replicate all the WAL files for this region server.
2. During the claim queue stage, we may also add or remove peers at the same time, which may lead to race and cause incorrect results.

For #1, the solution is to add a fence when adding a replication peer. That means, when adding a replication peer, we need to make sure that on all the region servers, we create the replication queue for the peer and store the initial offsets too. In this way, for the region servers which are still alive when adding the replication peer, we will have a replication queue for it. If a region server does not have a replication queue for the replication peer, we can make sure that the region server is started after we add the replication peer, so we need to replicate all the WAL files of it.

Of course, there could still be race, as during the adding replication peer operation, a region server could die too and start to claim queues, which is similar to #2 above. So here, we need some fencing mechanisms between replication peer related procedures and SCP.

For RemovePeerProcedure, since in the claim queue stage of SCP, we may insert new replication queue for the dead server, we need to make sure that it does not

insert a new replication queue for the deleted replication peer, or at least, we need to have a way to delete these replication queues properly. The idea is to wait for all the ongoing SCPs to finish after deleting the replication peer from replication peer storage but before deleting all the replication queues for this peer. To be more specific, in the postPeerModification method of RemovePeerProcedure, collect all the ongoing SCPs, store them as a class field so next time we do not need to collect again(this is important, otherwise if there are new SCPs scheduled, it could make us stuck here forever, which is not necessary), and then check whether all of them are finished.If not, throw a ProcedureSuspendException to release the procedure worker and try again later.
For AddPeerProcedure, replicating more files then expected is not a big problem, it will not cause any data inconsistent problem, so in general, there is no special fencing needed.

The ReplicationLogCleaner

The basic idea to test whether a WAL file under oldWALs directory is to check whether the file is before all the replication offsets in all the replication queues which contain this file. If so, we can delete it, otherwise not.
There are several problems when we want to actually implement this ‘simple’ idea. First, since now we will not insert data to the replication queue storage immediately when there is a new WAL file in the replication queue, it is possible that when we want to check whether a file can be deleted, some of the replication queues which contain this file are still missing. The basic idea here is that, if there is a missing queue, then we do not delete the file. But notice that, by just listing all the replication queue ids for the region server is not enough to find out that we miss a queue, as the replication queues for some other replication peers may present and we do have some return values when calling listAllQueueIds. So the idea here is that, we first list all the replication peers, and get replication queues for each of the replication peers, to see whether we miss some replication queues.
And then consider dead region servers, their replication queues will be claimed by other region servers, so we can not list their replication queue ids easily. Here we need to scan all the replication queues for a replication peer to filter out the replication queues for a dead region server.
And here comes another problem, if a region server is dead, and all its replication queues are claimed by other region servers and are finished replicating, then these replication queues will be deleted. The WAL files belonging to these replications can be deleted if they are not referenced by other queues, but based on our above rules, if there are missing replication queues, we should not delete a WAL file. So in ReplicationLogCleaner, we need to know whether a region server is dead or not, and then choose different rules to determine whether a WAL file can be deleted.
Notice that, a ‘dead’ server means the region server is dead, and its associated SCP has also been finished. If SCP is not scheduled or not finished, we should still treat it

as a live region server. **So we need to find a way to expose this information to ReplicationLogCleaner.**

About sync replication

The problem is about the drain source operation. For sync replication, when converting from STANDBY to DOWNGRADE_ACTIVE, we need to make sure we do not replicate the data which belongs to the previous DA state. In the past, this is done by roll the WAL writer when transiting to S, this is because when transiting to S, we will remove all the WAL files in queue, so when can also remove the current WAL file after rolling the WAL writer, in this way we can make sure that there will no data belongs to the previous DA state in queue.
The problem here is that we do not store all the WAL files in the replication queue now, so the above solution is not applicable any more. First, in the S state, we should still keep updating the WAL offset when there is a new WAL file, so we can know that the WAL file is not needed any more. Or maybe we could just store a special offset which is after all possible valid offset, and when transiting to DA, we write a normal offset to restore the replication queue.
Considering the race between SCP, we need to find a way to make sure that all the recovered replication sources are processed. The idea here is similar to RemovePeerProcedure. For transiting sync replication state, we will do a two step transiting, and in the middle we will send a remote procedure to all the region servers to drain sources or do something similar. So before we commit the final state changes, we also need a step to wait until all the SCPs are finished. The difference comparing to RemovePeerProcedure is that, for RemovePeerProcedure, once we updated the replication peer storage, the newly dead region servers will not claim the queue for this replication peer any more, so we are safe to only take care of the ongoing SCPs, which may have already loaded the replication peers, and then deleting all the replication queues for the peer without any races. But for TransitPeerSyncReplicationStateProcedure, the peer is still there, and we can only rely on the region servers to write the replication offset, so we need to make sure that, the replication queues for all the regionservers(live, or ‘dead’ but the SCP for it is not finished yet) are processed. So the correct step for waiting SCPs to finish is that, we first get all the region servers and also the ‘dead’ region servers where the associated SCPs are not finished yet, and then send remote procedure to the live ones, and finally, we need make sure that, either the remote procedure is successfully done, or the SCPs for the region server is done. After this condition is met, we can finally commit the state transition and finish the TransitPeerSyncReplicationStateProcedure.

The ReplicationSyncUp tool

Since the claim queue logic is completely different in the new implementation, the old compatible code in ReplicationSourceManager is not functional, which breaks the ReplicationSyncUp tool.
In general, we'd better not adding special logic in normal code, so I suggest we do not reuse the logic in ReplicationSourceManager, we should implement our own claim queue method in ReplicationSyncUp tool and then start RecoveredReplicationSource by our own.
The current ReplicationSyncUp tool mainly solves the problem that existing WAL data can be replicated to the Slave-Cluster after the HBase Master-Cluster crashed, but ZK, HDFS and the network are available. The principle is to implement a DummyServer. The DummyServer is used to claim the replication queues of all dead RegionServers, and then the DummyServer is used to copy the data that has not been replicated in the corresponding queue to the Slave-Cluster.
The info related to replication in the whole process is obtained from ZK. However, in the new implementation, it needs to be obtained from the HBase table. But if the HBase Master-Cluster has crashed, the info related to replication cannot be obtained from the HBase table.
After discussion, we all agree that ReplicationSyncUp tool can directly read the hbase:replication table data offline to implement the new ReplicationSyncUp tool.
When the ReplicationSyncUp tool is executed, the master cluster is in a down state. Because the hbase:replication table is flushed regularly, ReplicationSyncUp can directly read the hbase:replication table data offline. This way has no technical challenges and is simpler. Of course, the flush way and the snapshot way have the same problem, because flush is executed regularly, there is a certain delay time, which will also lead to redundant data being replicated to the slave cluster when ReplicationSyncUp is executed.
After ReplicationSyncUp is executed, the data that needs to be replicated by the master cluster has been replicated. Theoretically, the data in the hbase:replication table needs to be cleaned up. When ReplicationSyncUp is executed, a flag can be written to the file system, and the master cluster HMaster recovers, the data in the hbase:replication table can be cleaned according to this flag. After cleaning, we must delete this flag to avoid repeatedly cleaning the hbase:replication table.

Does the data cleaning of the hbase:replication table require that the HMaster be started before the RegionServer when the master cluster recovers to avoid inconsistency of hbase:replication data?

HMaster does not need to be started before RegionServer for two reasons:

a. If the RegionServer is started first, the RegionServer will be in the initialization state until the HMaster is started, no regions are assigned to it, so no data needs to replicated, and the hbase:replication table will not be modified;

b. If the RegionServer is started first, it will not claim the replication queue of dead RegionServer, because this process is launched in the ServerCrashProcedure, and ServerCrashProcedure is executed by HMaster.

The DumpReplicationQueues tool

The data in the replication queue is completely different as now we do not store every WAL file, so the output of DumpReplicationQueus tool should also be changed. The info output by the current version of DumpReplicationQueues includes:

1. replicatedTableCFs info: columns whose table column attribute scope is not 0
2. replicationPeer info: obtain data from ZK
3. queue info: includes RegionSever info, queueId, peerId, and offset of each wal file, etc. If '-- distributed' is set, it will poll each regionserver to obtain this information, otherwise we will obtain it from ZK.

In general, the output info of the new and old versions of DumpReplicationQueues is similar. In the new implementation, because we store the replication related info in the HBase table, DumpReplicationQueues needs to obtain data from the HBase table instead. At the same time, each queue in the new implementation has only one wal file and the corresponding offset. But we can follow the old way, to find out all the wal files need to be replicated and print them out.We can get the list of wal files through the wal directory.

It is recommended that the output of the wal file and offset info be consistent with the old version, to avoid the situation where users fail to upgrade the HBase version due to their dependence on the output info of the DumpReplicationQueues tool.

About data migration and rolling upgrading

The idea is simple and straightforward. Let’s disable all the replication peers while rolling upgrading, and start a background migration chore on master to migrate the data from zookeeper to the new hbase:replication table. Once the migration is done, we can enable all the replication peers.

One of the problems is that, during migration, the region server can not claim the replication queues too. But during rolling upgrading, we need to restart region servers, and claim replication queues is the last stage of SCP. The idea here is, in the claim replication queue stage of SCP, we

1. Check whether we have finished the data migration, if not, just suspend the SCP for a while and retry.
2. Once the migration is done, we can start to claim the replication queue, but we will only send claim queue procedures to the region server with the new version. This could be done in several ways, for example, we know the version of a region server, when selecting region servers we could do a filter, or we just introduce a new remote procedure callable(since the queue definition is changed), so when the region server with old version receives the request, it will fail to deserialize the data and fail immediately.
3. To avoid moving most of the replication queues to only a small set of region servers(the region servers which are upgraded first), we should set a threshold of the percentage of upgraded region servers among the whole

region server, for example, only after at least 30% region servers have been upgraded, we can start to execute the claim replication queue operations.