A new framework for region replication

HBASE-26233 The region replication framework should not be built upon the general replication framework

Background

The current region replication framework is built upon the general replication framework, there are basically these problems:

1. Need to add a special peer, which causes additional operational complexity, and easy to be misconfigured.
2. For enabling meta region replication, we have done some hacks to also make the meta WAL work together with replication, which introduces more cyclic dependencies between the WAL system and replication system, and make it more difficult to resolve HBASE-15867.
3. Also need to record the progress on ZooKeeper which increases the pressure on ZooKeeper. For meta region replication, we have done some hacks to skip the persisting to ZooKeeper, which is good. But we should skip the persisting for all tables, not only meta.
4. If the primary region is flushed, all the unreplicated WAL edits are useless, but in the general replication framework, there is no way for us to drop these WAL edits.

So in general, we should introduce a more light-weighted way to do region replication.

Requirements

Do not read data from WAL file

This is important for keeping the framework light-weighted.
The benefits are:

1. Less I/O pressure on the underlying HDFS. Tailing an opened file on HDFS is costly and error prone, you can see the magic in the ReplicationSourceWALReader implementation. And if we want to make the delay small, we need to more aggressively try to tail the file, but actually, region replication is usually used for ‘Read More Write Less’ scenario, which means we will waste a lot of CPU and I/O as there will be not too many edits.
2. Decouple with the general replication framework so we will not block other works for refactoring the WAL and Replication systems, such as HBASE-15867 and HBASE-20952.

Low latency

The old implementation which depends on the general replication framework, could have minutes latency if something goes wrong, and since it is designed to maintain correctness, which means we can not drop any edits, if we go into this scenario, it will last for quite a while and can not recover.
But for region replication, since the underlying HDFS is shared between all replicas, sending edits to the remote side is not the only way to fix the inconsistency, we could just do a flush, and then replicate the flush marker to the remote side to let them load the new store files. This means we have more flexible ways to catch up quickly.

On by default

The implementation should be stable and suitable for most cases, so when users set region replication to a value greater than 1, it will be enabled by default.
This could be done in a newer version of HBase when we can confirm that the feature is stable enough.

Non Goal

High throughput

The region replication feature is for the scenario where we have more reads than writes, so typically we do not need to perform very well when there are lots of writes to the region, i.e, maintain high throughput.
Of course, we should not crash the region server or block any requests if there are lots of writes, this is the bottom line.

Design

How to get the WAL edits

Through WALActionListener.postSync

The current version of WALActionListener.postSync does not include the edits synced, but it is easy to add this support, at least for AsyncFSWAL. Just see this commit when implementing HBASE-24950.

Pros

- Single source of truth. All edits will go through here so we only need to inject this only place, and it is likely that in the future this is still the only place, so we do not need to worry that modifications in the future will break us.

Cons

- Need to filter out lots of unnecessary edits, which may waste a lot of CPU cycles and may slow down the write speed. Unless we do some magic in the WALProvider implementation to let all regions with region replication enabled log to a separate WAL instance. It will also have other impacts as users may want to configure their own WALProvider implementation.
- Need to find a way to get some table/region specific properties. For example, on the replica number, it will never change unless the region is reopened, so in general, when opening a region, we could load this value and make use of it, as it will never change during the whole life cycle. But if the place where we want to use it is in another place, it will be hard to keep these properties in sync. One example is the current implementation in HBASE-18070, where we need to cache the table descriptors and also clear the location cache sometimes.

Inject in HRegion directly

In this way, we need to handle all the sync WAL edits places in HRegion. For now, there are basically two places:

1. HRegion.doWALAppend
2. The writeMarker methods in WALUtil

And for region replication, no matter what the actual marker is, the only action for us is to trigger a refreshStoreFiles call at secondary replica side, so maybe we even do not need to send the actual WALEdit for the event marker, just a special WALEdit to tell the secondary replica to call refreshStoreFiles, and we can load the actual sequence id after reopening all the store files, so even the sequence id for the marker is not important. This will give us a more flexible way to implement, as we do not need to only focus on the WAL anymore, just focus on the region operations, such as flush, compaction, and bulk load.

Some notes when implementing a POC

If we inject in HRegion directly, then the replication order will be a problem, as we could write to HRegion with multiple threads and there is no way to force them keeping the same order with txid order.
So first, if we still want to keep the order, then we still need to inject in the WAL related classes, as in WAL append we could make sure that we have the same order. This could make the implementation of the replay method at the remote side easier. But this will lose the ability to enable region replication for regions with SKIP_WAL. But anyway, this is only nice to have.
If we think the order is not important since the remote side can handle it correctly(we have this logic for now), then maybe we could just ignore the order problem. But there is still a problem that, when replication fails, we will request a flush, and before the flush marker arrives, we will

drop all the edits for the given replica. But if the flush marker comes after an actual data edit which has a greater txid, we will lose data. So maybe order is still a problem.

Pros

- Can make use of the properties for this region directly
- Easy to manage the lifecycle. For example, once the region is closed, we could drop all the buffered edits at the same time.
- Can even work for SKIP_WAL. We only need to change the code to also build the WAL edits but just do not append it to the WAL instance.

Cons

- Lack of global information, so it will be more difficult to do global resource limitation, for example, how to limit the total amount of heap memory consumed by the sending queue.
- We could also append to the WAL instance at other places in the future, which will break the implementation.

Attach an action to WALKeyImpl

In this way, we do not need to do filtering in the WALActionListener, just call the action when appending the WALEntry. In the action, we can implement the region replication logic, i.e, adding the entry to the send queue, and try to send them out.
The assumption here is that, we can safely replicate the WALEntry out after appending but before syncing, in a region replication scenario. It is because:

1. For normal cases, we will finally sync it out, so eventually we are consistent.
2. If the region server crashes before syncing, finally the primary region will online and do a flush, and cause the secondary replicas to refresh store files, so eventually we are consistent.
3. If sync fails, typically we will abort the region server, so it is same with #2.

So in this way, we could solve the order problem, but do not need to inject a WALActionListener. But there is still a problem that the WAL append thread is already busy enough, and since we will use the same WAL instance for all the regions(usually), it will slow down the write speed for regions which do not enable region replication.

Use MVCC.complete to trigger replication

A very rough idea:

1. If region replication is enabled, we will attach a special action to MVCC.WriteEntry.
2. Once the WriteEntry is completed, i.e, removed from the writeQueue, we invoke this special action.
3. In the action, we will add the WALEdit to the replication queue, under the lock of MVCC.

Since we have the same order for WAL and MVCC, in this way we could solve the order problem. And every region has its own MVCC instance, and it is usually called from the rpc handler, so we could also avoid the performance bottleneck if we attach the action to WALKeyImpl.

How to manage the send queue

The idea is to just use an in-memory queue. We should have a global limitation of the total size of the WAL edits in memory, and once it reaches the limit, we drop the edits for a region which has the most biggest size, and trigger a flush of the region, so it will then send the flush marker to secondary replicas to let them reload store files to catch up with the primary replica.

How to send WAL edits out

We can still use the AsyncClusterConnection.replay method. But we should have separated rpc timeout and operation timeout configurations since here we assume the rpc should be finished very soon. If it fails, it usually means the replica is not online yet, but it will trigger a flush on the primary region when it is reopened, so no problem, we do not need to wait for it online.
But for the actual rpc method, first we could still make use of the replay method, but it is not perfect, as it was designed to support DLR in the past. We should have our own version for region replication. But anyway, during rolling upgrading, the replay method can be a fallback option.

Failure recovery

There are basically two problems which could cause a failure:

1. Fail to replicate to some secondary replicas
2. We reach the global size limit and have to drop edits

In both cases, our choice is to stop sending new edits(to the specific replica or all replicas), and then trigger a flush. Once we hit a prepare flush edit, which is for flushing all the families, we will resume sending. And we will treat can not flush the same way, as we will only write a can not flush edit when there is nothing in the memstore when flushing. So when receiving this edit, the secondary replicas can make sure that, after loading all the store files, it can catch up with the primary replica, which is the same with a commit flush.
Why we need to resume sending when hitting a prepare flush but not a commit flush is that, for a prepare flush edit, we can make sure that all the edits before it will be included in the flushed storefile(s). For a commit flush edit, although it can trigger a refreshStorefiles immediately, which seems good. But the problem is that the flushed storefile(s) does not contain the edits between the prepare flush edit and commit flush edit. So if we only resume sending after hitting commit flush edit, we may still miss some edits after recovery.

Implementing a special method for replay region replication edits

We have some new assumptions which make the old replay method not work under some corner cases. For example, now if we meet an error when sending edits to a replica, we will just stop sending it until the next prepare flush marker, but the replay method expects us to send all the edits. So we need to implement our own method for replaying the edits.
In general, the logic should be simpler than the old replay method. The basic ideas are:

1. Like the replaying recovered edits method, we could just add the cell to memstore directly.
2. If we meet a meta edit, for most cases(except prepare flush), we just need to call refreshStoreFiles to load the new store files and drop the memstore content if possible.
3. For prepare flush, we still need to do an actual prepare flush operation, i.e, snapshot the current memstore and do a switch. This is because when the later commit flush marker arrives, we need to drop the whole memstore snapshot. If we do not prepare the flush first, we may find out that the memstore contains some cells whose sequence id is greater than the flushed sequence id, which means we can not drop the memstore. This is not acceptable as maybe it means we can never reclaim the memstore memory.

And as described in the above failure recovery section, if we hit an error we will stop sending until reaching the next prepare flush edit, so we can make sure that, once we receive a commit flush edit, there will be a prepare flush edit pending. And it is possible that when we receive a prepare flush edit, there is already a pending prepare flush, for this case we can just ignore the latter prepare flush edit, as once we receive the final commit flush edit, no matter which prepare flush operation we choose, its max sequence id must be smaller than the sequence number of the final commi flush edit, so we can safely drop the snapshot.

Reference Counting

The WALEdit may reference some off heap buffers if they are come from a rpc call, and since we will send the edits in an asynchronous way, we also need to deal with reference counting otherwise when we actually want to send them out, they have already been freed by the rpc framework because the rpc call has been finished.

And after HBASE-24984, we can reference a rpc call multiple times by calling retainByWAL, so here we could just make use of this ability, to increase the rc count when adding WAL edit to the send queue, and decrease the rc count after we actually send the WAL edit out.