ROOT/*hbase:meta* Region Replicas

HBASE-18070 Enable memstore replication for meta replica

Huaxiang and Stack

ASK STACK OR HUAXING IF YOU WANT EDIT PRIVILEGES. OPEN WRITE IS ALLOWING SPAMMERS

Read Replicas on the hbase:meta *Table currently only does primitive read of the primary's hfiles refreshing every (configurable) N seconds. This issue is about making it so we can do the Async WAL Replication ability, currently only available for user-space Tables, against the hbase:meta system Tables too; i.e. the primary replica pushes edits to its Replicas so they run much closer to the primaries' state. If clients could be satisfied reading from Replicas, then we could have improved hbase:meta uptimes but also, we can distribute load off of the primary and alleviate hbase:meta Table (read)* ***hotspotting****. We could do the same for ROOT when/if it shows up as part of the split hbase:meta project. This feature needs to be explicitly enabled client-side. It is for application-side clients tolerant of an odd retry. It is not for use internally by the system client that runs in the HBase Master making operational decisions based on its read of hosted system tables.*

At the client side, a new mode (LoadBalance) is added on top of the existing HedgedRead mode. In HedgedRead mode, the focus is High Availability. It cannot increase the throughput of meta query as every query goes through the primary meta region. If the primary replica is overloaded, the whole cluster is in trouble so it does not help for the HedgeRead mode to request secondary replicas. In the new LoadBalance mode, meta query goes to meta replica regions first instead of the primary meta region, leaving the primary meta region mainly serves writes and queries from hbase servers. This removes the hotspot from the primary meta region. It increases the meta query throughput as queries are evenly distributed N replica regions.

1 Background

- HBASE-10070 *HBase read high-availability using timeline-consistent region replicas,* see attached PDF for general overview of Replica feature.
- HBASE-11183 *Timeline Consistent region replicas - Phase 2 design,* see the attached PDF for design of the ***Async WAL Replication*** feature implemented for user-space Regions only. It is a helpful background for this current document which is about adding the *Async WAL Replication* feature -- with some enhancements -- for *hbase:meta.*
- Async WAL Replication is a pointer to ref guide doc on the user-space feature: "*Setting hbase.region.replica.replication.enabled to 'true'...will add a new replication peer named*

region_replica_replication *...when you create a table with region replication > 1 for the first time."*

- Much of this document was taken from Huaxiang's *a draft implementation design*. This document subsumes it.

2 Noteworthy

Read Replica Replication is built on top of the inter-cluster Replication facility but it differs from inter-cluster Replication because Region Replicas reset, reloading hfiles on Primary reopen. Given this reset, there is no need for the replay of dead RegionServer's WALs when Replication is exclusively in-cluster: i.e. no need to support replay of "dropped queues"/'recovery'. Below we exploit this aspect so as to simplify/improve on the original Region Replica implementation.

We are adding Region Replica support to the *hbase:meta* Table. Replication explicitly does not replicate *hbase:meta* Table and will continue so.

3 Overview

HBASE-11568 implements phase two of the Read Replicas feature, *Async WAL Replication*, which enables almost near-real-time replication of primary regions' memstore to Region Replicas (See HBASE-11183 for the design).This feature reduces the gap between Region Replicas and their primary Region to sub-one second in the usual case. This feature currently only works on user-space Tables. HBASE-11568 explicitly skipped implementation for *hbase:meta*.

Here we fill out the gaps such that *Async WAL Replication* can be enabled on the *hbase:meta* Table also. The main motivation for Region Read Replicas feature is improved 'read' uptime -- if the primary is down, the client 'read' request can be satisfied at a Replica while the primary recovers -- but here we not only want improved uptime for *hbase:meta,* we also want to ***distribute*** the *hbase:meta* read load across replicas and primary to alleviate *hotspotting*; i.e. rather than the primary serving all reads, we would recruit the replicas to carry the *hbase:meta* read burden.

On the client-side, we modify clients so they distribute reads across Replicas and primary. At the server-side, we leverage the existing infrastructure built by HBASE-11568 with a few *enhancements*, described in the following section.

3.1 An aside on *Async WAL Replication* Uptake

Async WAL Replication has not seen much by way of uptake since originally committed ~5 years ago. The machinery is coarse: the replicating RegionServer reads its local WAL and then has the local ReplicationEndpoint plays the edits to each Replica in-cluster. Available cluster

RAM and Region count are cut in half or more depending on how many Replicas configured (each Replica occupies a Region slot and could have a memstore grow as large as that of the primaries'). Each inbound edit gets replayed in-cluster N times per replica occupying network. For a big table, any one RegionServer will be replicating to all other cluster members keeping an account on a per-Region basis. Replayed batches will be small. Overhead is high. *async WAL Replication* does not scale linearly as hbase does natively.

Some of the above could be improved upon but given little interest in this feature, it has not been a focus.

But for a small, critical Table such as *hbase:meta*, given the benefits (improved uptime AND read load distribution), the *async WAL Replication* overhead looks like a price worth paying especially when mitigated by the enhancements listed below.

Give the above, this document solves for the special case of catalog tables only. If *async WAL Replication* can be made work satisfactorily for the special case, it may revive interest in this facility for user-space Tables. At that time, we'll can come back to work on the general case.

4 Enhancements

Here we list enhancements. The first of the below could be implemented generally for user-space and *hbase:meta* Read Replicas while the remainder are for catalog tables exclusively.

4.1 Skip maintaining zookeeper replication queue (offsets/WALs)

Phase one, included as part of initial feature commit

The purpose of maintaining a persistent queue of WALs to replicate (and replication offsets) in zookeeper is to avoid replication data loss in case of RegionServer crash. For (in-cluster) replica replication, this is not needed. If the RegionServer hosting the primary region crashes, replication stops. The primary will be opened at another RegionServer and the system will recover WALs via ServerCrashProcedure (SCP). On recovery, all replicas will be synced with the primary via a flush event or a region open event. Because of this latter re-sync of Replicas, possible because we are in-cluster, we do not need to run the zookeeper-based replication queue recovery mechanism nor do we have to keep up running replication offsets in zookeeper; they can be in-memory only[1].

[1] Not keeping up WAL accounting will free the effort at purging zookeeper from Replication.

The only nit is that during primary region recovery, replica regions may have stale data as replication has stopped and updates lag; replicas may be missing some of the last edits made against the primary. This is not a big deal, as today, if the meta RegionServer crashes, read/write is not available at all until the primary meta region is reassigned (Replicas 'reset' on re-open of primary -- see `HRegion#replayWALRegionEventMarker`).

Removing zk replication queue upkeep reduces overhead. We can remove accounting for all (in-cluster) region replica replication, both for *hbase:meta* and for user-space tables. As part of this project, we will only undo zk accounting for *hbase:meta*. Removing accounting for user-space Tables will take extra work -- and it is work we may not want to bother with (see below).

4.2 Break replication barrier across all replicas HBASE-25125

Phase 2, not included in initial feature commit

To avoid HDFS overhead, the current implementation tails the WAL once. After a set of WAL edits are read from the WAL, it creates threads to send the WAL edits to all replica region servers concurrently. It then waits for ACKs from all replica region servers before it moves forward to read the next set of WAL edits. Essentially, it replicates WAL edits in lockstep across all RegionServers carrying Replicas. With this implementation, the Replication rate will be determined by the slowest Replica.

We want to break this synchronization.

We want to maintain one in-memory replication queue for each replica region server. When a set of WAL edits is read, it is queued to a replica RegionServer's own queue. Each queue can replicate at its own pace. The queue will be reset when flush events are queued. Since there is normally only 1 primary meta region per RegionServer -- the balancer may need amending to ensure this is the case -- and given that the region is flushed every 256M or 5 minutes, the memory consumption is not a concern here as there will be at most 256M memory (roughly) taken at the primary region server.

This enhancement we'd do for the special catalog/meta tables only; maintaining separate queues for all replicas, while possible, would be involved and prohibitively costly RAM-wise.

4.3 Avoid WAL tailing

Phase 2, not included in initial feature commit

Another concern is WAL tailing. HDFS does not natively support tailing a file. HBase ‘tails’ WALs by opening, reading till EOF, closing, and then reading from the last read-point till a new EOF, and so on. This causes undue pressure on the HDFS NameNode.

In a second phase, we’d avoid WAL ‘tailing’ developing a coprocessor for WAL events. This coprocessor will directly queue WAL edits/events to the in-memory replication queues.

This should make replication more lightweight and more prompt. We can achieve sub 100ms latency for most WALedits replay (See HBASE-25063 for an alternate, possible solution[2]).

Similarly, we’d implement this enhancement on catalog tables only; doing it generally would be costly keeping RAM queues per Replica.

5 Non-Goal

Today, there are more families in meta table, such as rep_barriers, table state, and namespaces. These families will not be replicated as part of this effort; Region Replicas will not replicate these families (We may reexamine at a later date; in particular, replicating table state).

5.1 User-Space Regions

See 3.1 An aside on Async WAL Replication Uptake .

---

[2] a framework to replicate the edits to master local root to backup masters.
The implementation is straight-forward, change the postSync method of WALActionListener and pass in the synced WAL entries by this sync operation, so you can do replication in this method.
a framework to replicate the edits to master local root to backup masters.
The implementation is straight-forward. Change the postSync method of WALActionListener and pass in the synced WAL entries by this sync operation, so you can do replication in this method. There are two problems that need to be addressed if we want to make it available to a general table (rather than ROOT only as in HBASE-25063).
1. FSHLog does not store the unsynced WAL entries so this approach can not work with FSHLog. I do not think this is a big problem and it should be a bug for FSHLog, as when sync fails we should rewrite the unacked entries again. We can fix it, not very hard.
2. The cells in the WALEdit are stored on off-heap and will be released after the rpc call is done. This problem can be solved by delay the call to FSWALEntry.release. Maybe we could add a boolean return value to the postSync method, which means whether the upper layer should call FSWALEntry.release.

6 Implementation Design

6.1 Configuration

Just as we have *hbase.region.replica.replication.enabled* to enable *async WAL Replication* on user-space Tables, here we introduce a new configuration, *hbase.region.meta.replica.replication.enabled*. Set it to true to enable *async WAL Replication* on the *hbase:meta* Table.

Related, use *hbase.meta.replica.count* configuration to set the Replica count. This configuration already exists. It is a configuration for setting the Replica count on *hbase:meta* Table. Default is 1. It was introduced when the Region Replica feature was originally introduced. Usually you set Region Replica count on Table schema but you currently are not allowed disable *hbase:meta* so we have this system property instead.

6.2 Client

Currently, the hard-coded behavior of Region Replica clients is to read from the primary first and if no answer inside a configurable amount of time, go to all Replicas. Clients then select the first response. If the response is from a replica region, it will set the stale bit to indicate the result is from a replica region.

Load balancing logic needs to be added at the client to distribute reads across the primary and its replicas. When a client needs to do a meta location lookup, it will randomly choose a meta replica and send the scan request to this replica's RegionServer. If the location info is out-of-date, the client will get either a *RegionMovedException* or a *RegionNotServedException*. For *RegionMovedException*, the client does not usually need to query *hbase:meta* again because *RegionMovedException* contains the new location of the moved Region. For *RegionNotServedException*, the client has two options: it can go to the primary to get the most-recent location update or go to the same replica region with the hope that the replica region location has since been updated.

In the initial implementation, we will read round-robin from Replicas first. If any error, we will fall back to read from the primary. For the case of a severly lagging Replica where the query returns stale data/Exception, this will result in added reads against the primary -- at least until the Replica manages to catch up again.

The Master hosted client that reads hbase:meta will ALWAYS read the primary to avoid making changes based of possibly stale state.

6.3 Server side

6.3.1 Region Server hosting Primary meta region

1. When RS opens the *hbase:meta* primary region, it will check if *hbase:meta* Region Replica is enabled. If it is enabled, it will create an *hbase:meta* ReplicationSource (*HBaseMetaReplicationSource -- HBASE-25055)* which knows how to read and forward *.meta* WAL content. This ReplicationSource will not persist state to the Replication backing store (offsets, list of WALs); i.e. 4.1 Skip maintaining zookeeper replication queue (offsets/WALs)
2. Add a *hbase:meta*-specific RegionReplicaReplicationEndPoint to implement other Enhancements.
3. *Primary meta region moves from RS1 to RS2.* The primary meta region is closed first. With primary meta region closed, the replication will be closed as well. Most of the logic has been handled nicely in the current code. When Region is opened at RS2, ***replication will restart with an open Region WAL edit added to the WAL.*** The Replicas will notice this new edit and reopen all the primaries hfiles (waiting on flush first before fully opening to make sure they are completely up-to-date).
4. *RS hosting primary meta region crashes:* Since this is a push model, nothing needs to be done at RegionServer hosting replica meta region. The producer is gone, there will be no meta replication. When master assigns the primary meta region to another region server, it is the same flow as 3. The replication model is exactly as intended: i.e. replicating data in memstore; if RS crashes, memstore is gone, no replication will happen. One nit is that for the WAL edits not yet replicated, it has to wait until primary meta region is assigned for edits to be available at replica regions[3].

6.3.2 Region Server hosting Replica meta region

For replay of replicated edits, it is same as the current Region Replica replication:

1. In case of RS hosting replica meta region crashing, the flow is same as the current replica replication: i.e. the replica region is moved to another RS, during replica region open, it will ask the primary to do a flush (these flush events are sent to all RSs hosting meta replicas, essentially sync'ing all meta replicas to the same snapshot), and it will only fully open to serve reads after the flush event is processed and it 'sees' the flush marker come in on its replication stream.
2. When a flush request is received from a replica region server (with `writeFlushWalMarker` set to true), it is possible that the primary RegionServer has not received the new assigned replica server event yet. It is critical that the new replica server receives these flush

[3] See the 'Failure Handling' section in the design attached to HBASE-11183 for fuller exposition of what happens around primary and replica Region crash.

events. What we can do is to refresh replica server list upon receiving these flush events.

3. Possible way to avoid “flush amplification” is for master to send a bit in regionOpen to decide if it needs to ask for “flush”.

6.3.3 Meta region split/merge

We need to consider how this feature works in the context of split *hbase:meta* (HBASE-11288 Splittable Meta).

The normal flow of Region split with replica regions is to close primary region and replica regions first and then do necessaries. If the primary region is closed first, it is fine as replication source is closed as well. If replica region is closed first and the source side is still shipping edits to this replica region, it will get RegionNotServedException. Primary should check for Region close before retrying and quit shipping edits immediately.

6.3.4 Bulkload support

Assume that there is no bulkload support for *hbase:meta* Table Regions.

6.3.5 Balancer and AssignmentManager enhancement

When meta regions are assigned, the AssignmentManager needs to make sure that no two primary meta regions are assigned to the same region server. More than one primary on a server will happen but there needs to be an anti-affinity built into the balancer. Similar for meta replica regions; the balancer should not for example colocate a primary with its replica.

Enhance balancer logic to not move meta replicas around (less possibility than normal user regions), so cache can keep warm for meta replica regions and serve better.

7 Issues yet to resolve

7.1 Observability

Observability? How to know the current state of Replication?

7.2 Replicas flushing the Primary too much

During replica region open, it will ask the primary to do a flush and it will wait for the “flush commit” before it will flip to read-ready. For a normal start case, let’s say, there is

1 primary and 4 replicas, then each replica will ask for a flush and the primary will end up flushing 4 times. Can we do better to avoid this case?

One answer is we can have a bit in regionOpen RPC message. This bit will tell replica region to ask for primary flush. The purpose is to optimize cases during cluster startup or some other cases, where there is no need for each replica region open to ask for a flush.

Stack: I've seen that the primary will not flush if it just did; it just writes an empty flush marker which seems good to me. Not a problem?

7.3 Memory accounting

Replication queues on primary.

7.4 Primary and Replica on same server

Close Replica?

7.5 Flag Endpoint about open/close of Region

How do I tell the endpoint. Add Region, Remove Region, so can clean up Qs?. Let me see about passing open/close. Split?

7.6 Misc

HBASE-6617 flattens meta so it is in the same WAL group as all the others.

8 Questions

Duo: What if the region server which carries the secondary replica crashes? The secondary replica will have less data after reassigning? I know this is not your fault, it is the current implementation. Just asking.
Huaxiang: No, once the replica region is opened, it will ask primary to do a flush first. It waits for flush events before serving read requests.

Q: What happens if RS carrying Replica crashes? Stalls all Replicas?
Huaxiang: Not a concern as we decouple replication across replicas when we implement a queue per Replica.

Q: Will the WAL cleaner prematurely remove WALs currently being replicated?

Duo: “I do not think this is a problem. We can only move a WAL file to oldWALs when all the data in it are flushed to HDFS, which means we do not need to replicate the file any more?”

9 Meeting Notes 20200807

Clara, Huaxiang, Nick & Stack

Primary Region is open, RegionOpenHandler ...Create ReplicationSource. Adding ReplicationSource.

Primary Region open for write, should have Peer to replicate. After WAL Recovered. All Replica Regions will be up to date.

Master or ZK. From Master or ZK says RecoveredReplicationSource.

On open of primary meta, send a local peer modification to get the ReplicationSource added for meta.

Replicate to more than one Replicas to different RS. It reads a set of WAL edits and sends to all 5 RS and then waits for all to 5 RS to ACK before it proceeds. Need to optimize send to all 5. Has to wait for the slowest to complete. One option is all 5 tailing WAL Edits; i.e. pull. Or, have inmemory Qs. First phase takes current implementation. Improve later, need to improve RPC’ing.

Q: Unhitching from the peer system in phase2. Replication Peer.

On primary crash, after recovery, primary sends a flush event or reset event.

On replica crash, replica region and when open, it can ask primary to do a flush and when flush completes, then it opens.

Where the replicas are located? We have to add to it the replica locations. This is already there.

Out of order edits because incoming writes to primary. Is there a problem here?

Client side at crash time.

Observability? How can we tell how out-of-date Replicas are?

No disable.

The addPeer at regionServer is not from master. Instead, it is inlined at meta regionOpen, so there is no need to add zk peerNode.
Without it, it is not possible to hook in these replication monitoring tool. Need to implement new ways to hook in in-memory replication queue. Possible to add in-memory storage system?

One more reason we need this 4.2 is related with the enhancement 4.1. Since there is no persistent replication queue in zk. WalCleaner at master does not know meta wals are being replicated, it may delete meta wals earlier. If we maintain memory queue for each replica, this replication is going to be very fast, and we do not need to implement anything. With 4.3. Implemented, wal tailing is totally bypassed.

4.1 Skip maintaining zookeeper replication queue

10 Meeting Notes 20200923

Huaxiang, Stack
Q. Stack raised question that multiple primary regions could coexist in the same region server, in the case of region server crash etc. This is a good point and endpoint cannot fail in this case. Need to change the implementation.

Q. What will be the interface to notify the endpoint that a primary has been assigned/unassigned from the region server? The purpose is to create/clean meta resources for this specific region.

Stack mentioned that it is going to be hard to add extra methods for this purpose. Maybe another mechanism can be considered, timeout, i.e, when there is a new region is met from wal, it creates a new set of queues for this region. And if it idles (lets say 10 times), clean up starts from endpoint.

Have not thought about it. Main concern is that if a region is split or moved. There are some wal edits for this region, It will try to replay them. How to differentiate all failure situations and when to stop? Need time to think about it.