Better Write Predictability For HBase

Himanshu Vashishtha (himanshu@cloudera.com)
Jonathan Hsieh (jmhsieh@apache.org)
12/19/13

Problem statement

A RegionServer currently has a single WAL file where it appends all edits for all of the regions hosted by a particular RS. When there is any abnormal latency in the write pipeline, the overall write latency and throughput of the RegionServer is negatively affected. Also, since current commodity level nodes typically have 6/12 disks, we are currently under-utilizing available hardware resources.

Design Goals

In HBASE-5699, there has been discussion about adding **Concurrent Multi HLog** functionality. We considered two alternatives concurrent hlog designs, **Sharded HLogs** and **Time-multiplexed HLogs** before deciding to focus on **Single-Active Multi HLog** (SAMHLog) with the goal of improving write latency predictability. We chose to implement the SAMHLog approach first to avoid much of the added complexity the concurrent HLog designs incur and because the

SAMHlog serves as a desired building blocks for follow on work that would implement the concurrent multi hlog designs. (see appendix for details)

Our immediate goal is to have a ready and extra WAL file that we can switch to help in reducing the latency throughput effects in such scenario (WAL Switching). This will help in overcoming intermittent network hiccups and issues such as straggling WALs.
Our follow-on goal is to implement a concurrent multi WAL based on the above switching feature. With both these pieces in, we will have lower write latency (due to WAL switching), and higher throughput (concurrent writes).

The following are some of the key design goals:

- Improve the predictability of write latencies by improving the 90%tile, 95%tile or 99%tile latencies of a write heavy load.
- Lay down the necessary groundwork for future HLog implementations that can re-use this low write-latency feature (Extensible interface for other HLog impls).
- Minimal impacts on current replication or log recovery code paths.
- Easy to configure (single xml option / table option)

Analysis

To understand how we can improve latencies we start by measuring the latencies of operations used with typical WAL Log operations -- WAL rolls, appends and syncs.

We show that

1. Open and Close associated with rolling operations take significant time.
2. The sync call is the costliest operation
3. A sync op time depends on the chunk of data it is syncing.
4. Majority of ops (for chunk <= 100k) is done with in 10ms.(>99.7% ops)
5. Any operation that takes >1000ms is 2 orders of magnitude above normal and are the most offensive contributors.

HLog usage of HDFS APIs

Below is the breakdown of various HDFS APIs currently used by HBase's HLog:

1. Open/Close a writer/reader: (NameNode op)
    a. Called whenever we create a new WAL or open an existing WAL file (recovery and HBase replication)
2. Read/write/sync: (DataNode op)
    a. Sync is called after appending a WALEdit (and also periodically by LogSyncer). A sync call involves all the DNs in the pipeline.
3. Block allocation: (NameNode op)
    a. Whenever the current block is full and we need more; this is called automatically by DN.

Experiments

The cost of WAL rolls can be simulated and measured by performing open/close operations. The cost of an append workload of an regionserver on the NN + DN can be simulated and measured by performing successive write+sync operations.

Set up:

- 1 NN + 4 DN. (Hadoop-2.1.0-beta); Client running on 1 DN.
- Each DN has 3 disks.
- DFS Block size = 128 MB

Cost of HLog Rolling's Open / Close with data

We measured the time and variance associated with opening a file; writing/syncing a chunk of "log data" (~1 MB) and closing the file. In this experiment 1 Op ⇒ Open/Write a chunk/Sync/ Close.

Results:

- Open/1k write/Close 1 file: ~340 ms (avg).
- Open/1k write/Close 1000 files concurrently: ~300ms
    - 4 sec; 568ops takes > 1sec (2,3,4 sec)
    - 56.8%tile is >1sec

We can see that the cost of rolls are expensive -- taking >300ms on average with a large number taking >1s. Masking this would provide significant latency and throughput wins.

Cost of HLog Append's Syncing

We measure the cost of the normal hlog append operation by performing multiple write-sync operations. Open/(Write-Sync)^N/Close -- Write/Sync a chunk of data.

- 1 HDFS block = 128 MB.
- 1 op = write a chunk (size varies) and sync. For each iteration, client did 10k ops (N = 10,000).
- Client is co-located with a DN. So, for every block, there is 1 local block.

Results:

To get latency numbers, we varied the chunk size from 1k - 16MB. For each chunk, it writes-syncs 10k times (i.e., no of ops = 10k).

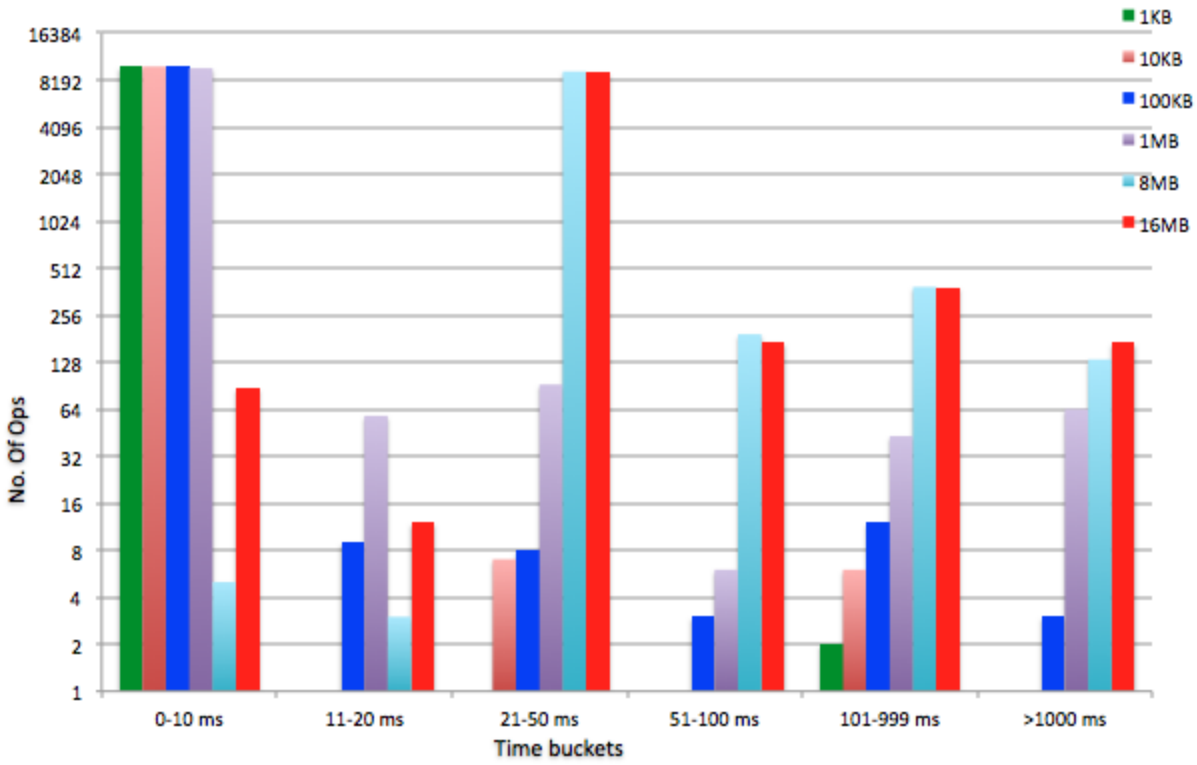


Fig 1. Latency of writing (flush + sync) 10k ops with different chunk size (1k to 16MB).

In the above graph, we bucketed the write-sync time for various chunk size of data for 10K operations. Here are few interesting observations:

- Upto chunk size of 1MB, ~99% ops are done within 10ms.
- Increasing chunk size increases sync time. As the chunk size increased to 8 or 16 MB, ~93% of the ops are done in 21-50ms bucket.

In this work, one of the primary goal is to improve HBase write predictability. From HBase point of view, the usual WAL edit size is about <~1MB, and could treat *sync ops taking more than a sec to be an outlier*. If we could reduce the frequency of ops taking more than a sec, it will improve the HBase write predictability. We propose the following design to achieve that objective.

Design

Along with looking at possible performance improvements (such as throughput, latency), we also considered factors such as simple design, MTTR, HBase Replication, WAL Straggling, and compatibility with 0.96/0.98.
We want to have minimal impact on MTTR. To do so, we should avoid out-of-order edits in a WAL file. Otherwise, we need to sort out all the edits before replaying them during recovery,

which significantly increases the recovery time. We also want this feature to be easily compatible with 0.96/0.98.

Considering all the above factors, we propose a new HLog implementation, **SingleActiveWriterMultiHLog (SAMHLog).**
The basic idea of SAMHLog is there are two open writers, and only one is active at any time. If the current active one becomes slow, we switch to the other writer. We explain it in more detail in the next section.

Given the measurements above, it is clear that we want to mask the latency required for expensive sync ops. It is also clear that although "chunk size" has an influence on latency, ideally, all operations should take less than 1000ms in HBase. So for the first cut we just use a constant **WAL switch threshold** (configurable value, with default equal to 1 sec). This means if a sync op takes more than 1 sec, we switch the current WAL to a standby that should have better latency.

It is also worth mentioning here that the original BigTable also mentions a similar WAL switching feature to solve the WAL Straggling problem.

Regarding implementation, we intend to do minimal changes for replication or recovery code paths. This requires that some constraints be maintained with the operation and modification of the WAL. This leads to the following invariants:

1. No out of order WALEdits in any WAL file.
2. While switching the WAL file, maintain the same order of inflight edits when writing them to the new WAL file.
3. Switching of WALs re-use the **same** WALEdits object.

There are two interesting implementation concerns here. Firstly, in order to retain our ordering invariant, we will need to retry edits attempted on one log on the alternate log in order. This requires us to restructure the HLog such that entries can be replayed (should be able to retry syncing using the same wal edits). Secondly, we need a **switching policy** to decide when to move to the alternate file.

Sticking these constraints make the SAMHLog design and implementation significantly simpler than any of the Concurrent HLog designs. As we argue in the appendix, the SAMHLog is a building block that would be beneficial to any Concurrent Hlog implementation.

Structure

Structurally, SAMHLog contains two writers but maintains one shared write buffer. Only one of the two writer is active at a time and is flushing/syncing the edits.

1. Each writer writes to a distinct file on filesystem. However, only one is actively taking writes at a given time.

2. All the appendXXX calls does is add the entries to the shared buffer. It uses similar (existing) blocking semantics such that we have in-order sequence ids.
3. There is a set of syncer threads (Syncers), associated with every writer, which empties the buffer, write to local stream (flush), and invokes the sync API.
4. A HDFSSyncWatcher thread monitors all the inflight flush-sync calls. If any operation takes more than a user-defined threshold time (WAL Switching threshold), it kicks off the WAL switch operation. We explained the switching policy in the next section.

Switching policy

An hlog switch involves taking all *in-flight edits* of the old WAL, and appending them to the new WAL while maintaining the same order. Following are the steps involved:

1. Block any incoming writes (append/appendNoSync) call.
2. Hold the entries in the buffer of the current active WAL file (involves taking the updateLock, etc). Call them bootstrapping entries (entries which are required before making the new WAL file available for new appendXXX ops).
3. Write/flush these entries while maintaining their insertion order.
4. Enable the new wal file to start taking fresh writes
5. Roll the old writer, so that we don't have out of order edits in case the new WAL is also slow and we need to switch again. This could be a rare case, but we should handle it too.

Evaluation of Prototype

While working on design, we did experiments to do a feasibility study of various approaches.

We wanted to quickly verify the benefits of latency oriented multi WAL approaches before finalizing any approach. To that effect, we prototyped two variations of the proposed SAMHLog and ran various experiments to verify the effectiveness of the proposed approach.

In order to verify WAL switch effectiveness, we needed a way to simulate a bad HDFS pipeline so the sync call takes longer time. We inject **network hiccups** to achieve that effect. A hiccup is injected to simulate a bad HDFS pipeline, where the sync call on the filestream is blocked. This is done by instrumenting FSHLog such that after every 10k sync ops, it stops syncing for 'N' ms. This also blocks other threads pending on the sync call. For these experiments, we varied N from 100 ms to 2000 ms.

A slight variation of the proposed SingleActiveWriterMultiHLog is prototyped, where each writer is basically a FSHLog instance, and only one is active at any time. There is a set of Syncer threads in SAMHLog, which invokes the sync call to the underlying active FSHLog instance. This brings one level of in-direction b/w regionserver handlers and sync calls. This is an essential step for WAL switching because we want handlers to be unblocked to take new writes after the WAL is switched (and not blocked on the old sync call). A monitor thread, HDFSSyncLatencyWatcher monitors all the sync ops. If any sync op takes more than WAL switch threshold, it triggers the WAL switch. It uses the same switch policy as described earlier.

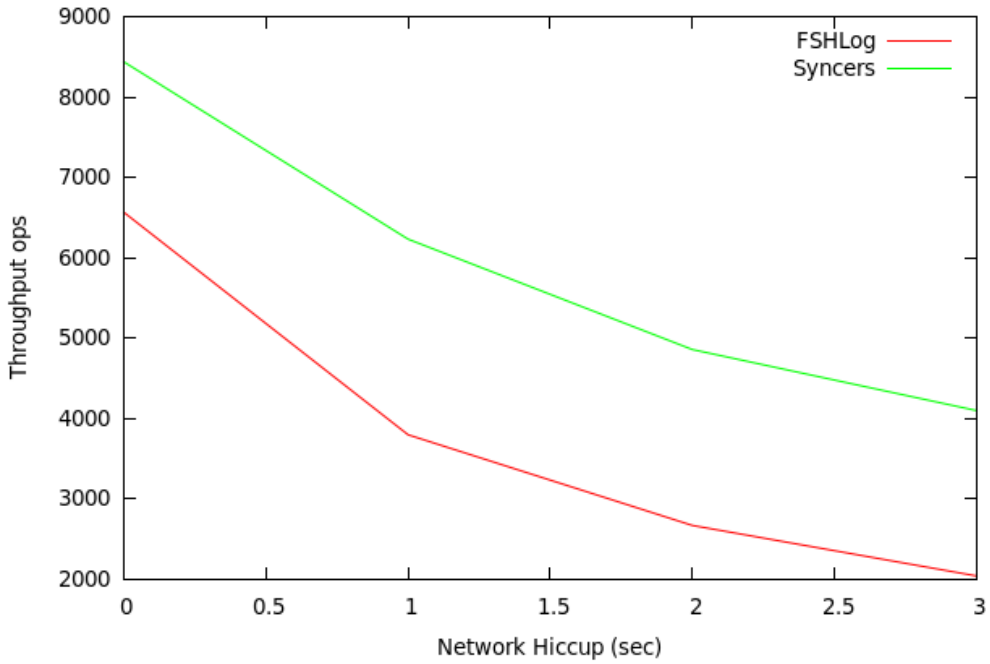
Experiment Set Up

We running HLogPE on the “instrumented” HLog impl.

- 5 node cluster 1 NN + 4 DN. (Hadoop-2.1.0-beta); Client running on 1 DN.
- Each DN has 3 disks.
- DFS Block size = 128 MB
- 5 client threads writing 10k ops each.

Infinite switch threshold

In this run, we evaluate current FSHLog and SAMultiHLog with WAL switching effectively **disabled.** This is done to get a base throughput comparison of two implementations.



This experiment shows that both the implementations suffer in their overall throughput due to the injected hiccups. The delta between the two lines remains almost constant in the overall run. SAMHLog (Syncers) gives better throughput because of the different syncing model (results in batched syncing).

In the next runs, we will enable WAL switches and see how the throughput tends to improve when switching is enabled.

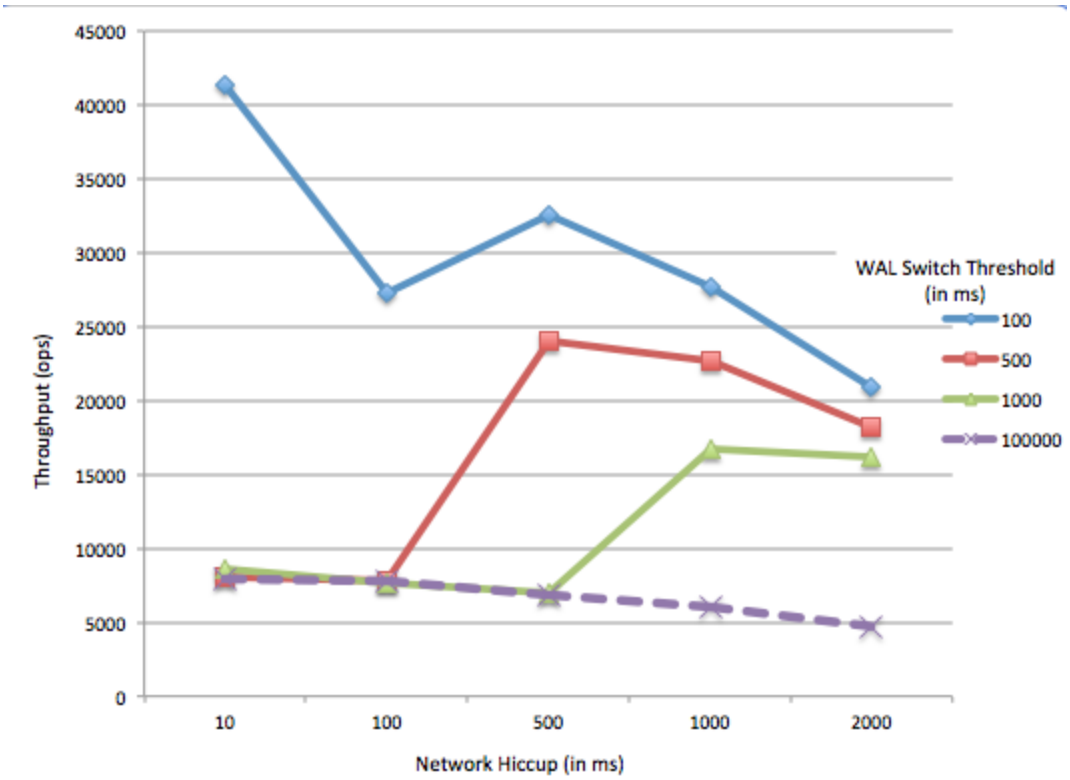
Switch Threshold vs N/W Hiccup

In the previous run, we compare FSHLog and SAMHLog without taking switching into account. As expected, there is a gradual decrease in throughput as the n/w hiccup interval is increased. In this run, we enable WAL switching. A n/w hiccup is injected after every 10k sync operation. The hiccup duration is varied from 10ms to 2000ms and the performance of HLogPE is checked

for various WAL Switch threshold values. The WAL switch threshold is varied from 100ms to 100,000ms.

In the below table, each cell shows time taken (in sec) / throughput/ (ops/sec) Number of WAL switches.

N/W Hiccup Switch threshold (ms)	10	100	500	1000	2000
100	12.081s / 41387.301 / 3	18.329s / 27279.174 / 22	15.351s / 32571.168/ 20	18.047s / 27705.434/ 23	23.964s / 20864.629 /21
500	62.331s/ 8021.69/ 0	63.958s/ 7817.630/ 0	20.762s/ 24082.459/ 13	22.067s/ 22658.270/ 21	27.406s / 18244.180/ 22
1000	58.068s / 8610.595/ 0	64.741s / 7723.082/ 0	71.519s/ 6991.14/ 0	29.943s/ 16698.393/ 12	30.978s/ 16140.486/ 21
100000	63.343s/ 7893.532/ 0	63.621s/ 7859.04/ 0	72.711s/ 6876.539/ 0	82.822s/ 6037.043/ 0	105.725s/ 4729.250/ 0



Observations:
This experiment shows the benefits of WAL switching. The lower the threshold value, the sooner it switches and improves the otherwise decreasing throughput. The switch occurs when the hiccup time is more than (or equal to) the WAL switch threshold.

Let's look at above numbers in more detail:

1) WAL Switch threshold 100000ms: This is the base line (represented as dashed in the above figure). There is no WAL switch, and all sync ops are blocked for the entire hiccup interval. The performance drop is very similar to the current FSHLog (minus the batched-sync effect, as discussed in the previous run).
2) WAL Switch threshold 500ms and 1000ms: The throughput shoots up when the network hiccup time is equal or larger than the WAL switch threshold. This is because the WAL switching kicks in, and the handlers start writing to the new WAL file instead of waiting on to the current blocked WAL.
3) WAL Switch threshold 100ms: We switch the WAL as soon as an op takes more than 100ms. The smaller threshold value is the reason for its better performance among all other variations. There were about twenty WAL switches happened when the hiccup >=100ms, which improves the overall throughput. Another interesting observation is WAL switched 3 times even when the n/w hiccup time is 10ms. This could be attributed to the better performance in this category, as compared to other runs. But, the

anomalously higher throughput value at 10ms hiccup needs more research. We plan to investigate it further while working on the final implementation.

Another interesting data point is the throughput reaching to 20k ops for all switch threshold switches lesser than 2 sec, when the network hiccup is at 2 sec. Theoretically, at 100 ms threshold, the throughput should have stabilized after switching kicks in (and not drop after 25k). One possible reason could be the way the test is set up. This is because during switching, we roll the old WAL file, which also syncs it. Since the sync is blocked for 2 sec, the old WAL file couldn't be rolled for 2 sec. Meanwhile, the new WAL file has already taken next 10k writes and is also blocked (and waiting to be switched).

This is more of a synthetic issue in the current testing setup. But overall, we do see benefits of this work.

Appendix - Alternate designs

Adding **Concurrent Multi HLog** functionality in HBase has the primary focus of increasing write throughput. These alternate approaches also share several new problems:

- **Straggling WAL:** If any WAL gets slow, we would like to "switch" to some other good WAL. Otherwise, all handlers could just block on this WAL. In order to improve on individual write latency, one would want the SAMHLog switching capability.
- **Open WAL file limit:** While concurrent hlogs would make use of available disks and provide higher write throughput, all of the parallel hlog approaches must be wary of number of opened WAL files. With default HDFS replication being 3, one write in HBase affects three disks. Even if a node has 12 disks, one should be conservative in opening WAL file (not more than 4, for example).
- **More HDFS lease recovery on failure:** Opening more WALs concurrently means we would require more expensive lease recovery operations during a regionserver failover.
- **Out-of-order WALEdits**: While switching, this other WAL is also taking fresh writes in parallel, and there is high probability we would have out-of-order WALEdits in the WAL. This greatly increases MTTR effort, as we rely on in-order WALEdits while replaying the recovered.edits on recovery. Please note that HBASE-8701 tries to solve it, but we want this multi WAL feature to not depend on other configurations (such as using HFileV3).

Sharded HLogs

The Sharded HLogs design assigns writes to particular regions to a particular WAL instance. It is a generalization of the HBASE-7213 where we separate out the META log entries into a separate hlog. There can be multiple variants based on grouping schemes, each which have their own MTTR vs load-balancing throughput tradeoffs.

1. **A WAL per Region** potentially minimizes MTTR time because no splitting would be required. Unfortunately it would create a large number of WAL files causing write throughput to bottleneck due to increased seek work load on spinning disks. Our experiments suggests that opening too many WAL files may not be a good idea.

2. **A WAL per group of Region** potentially maximizes write throughput but introduces a load balancing problem. Ideally it would have 3-4 concurrently writing WAL files with load from different regions balanced across the WALs..
3. **A WAL per Namespace/Table** potentially improves MTTR for some tables and would not have as many WAL files as the WAL per region approach. It is an interesting alternative because it allows for some tables or namespaces to be recovered and become available before others.

Any implementation could benefit from using SAMHLogs to gain the latency consistency features in each shard.

Time Multiplexed HLogs

The time-multiplexed hlog approach would ping pong or round robin between multiple open hlogs, regardless of if there were a hiccup or not. This approach would likely provide the absolute lowest possible latency, but also incur the most expensive MTTR recovery path, due to the higher probability of Out-of-order edits and timestamp concerns.

Community Process

Interaction with HBASE-8755

HBASE-8755 gathered traction and got committed when the prototype for this work was being written. They both take similar approaches regarding syncing (batched syncing). We will determine and provide evidence of which implementation is more efficient and will have the final merged implementation use the most effective version.

Branching

We'll be implementing this on a branch. It potentially interacts with multiple subsystems (replication, MTTR for example), and we need to make sure it doesn't break any of these dependencies.

JIRA breakdown

We plan to create a new jira for this work and link it to HBase-5699. The reason is because this work provides a basic building block for getting a multi WAL feature: WAL Switching. This helps in improving write latency. We could have various concurrent WAL impls (as discussed in alternate design section) using WAL switching.

Here is the Jira breakdown for this work:

1. **Proof** that it helps: New HLog impl: SingleActiveHLogImpl, test cases, switch policy.
2. **Plumbing** code to instantiate HLog impl.
3. **Testability**: Make HLog/FSHLog a true interface (remove explicit type casting that is widespread in the test code; create TestFSHLog and make TestHLog abstract)

4. **Correctness**: Ensure MTTR works properly. If we roll the file on every switch, it eases out the MTTR side quite a bit. Otherwise, we need to ensure we don't lose edits. With multi WALs, there will be overlapping wal edits. Recovery needs to ensure it replays them in order (requires a merge sort of WAL edits while creating recovery edits files).
5. **Replication** should handle more than one open WAL file at a region server.
6. **Doc** how to enable/use multi wal.