Lineland

Everything's a dot.

Saturday, January 30, 2010

HBase Architecture 101 - Write-ahead-Log

What is the Write-ahead-Log you ask? In my previous post we had a look at the general storage architecture of HBase. One thing that was mentioned is the Write-ahead-Log, or WAL. This post explains how the log works in detail, but bear in mind that it describes the current version, which is 0.20.3. I will address the various plans to improve the log for 0.21 at the end of this article. For the term itself please read here.

Big Picture

The WAL is the lifeline that is needed when disaster strikes. Similar to a BIN log in MySQL it records all changes to the data. This is important in case something happens to the primary storage. So if the server crashes it can effectively replay that log to get everything up to where the server should have been just before the crash. It also means that if writing the record to the WAL fails the whole operation must be considered a failure.

Let"s look at the high level view of how this is done in HBase. First the client initiates an action that modifies data. This is currently a call to `put(Put)`, `delete(Delete)` and `incrementColumnValue()` (abbreviated as "incr" here at times). Each of these modifications is wrapped into a KeyValue object instance and sent over the wire using RPC calls. The calls are (ideally batched) to the HRegionServer that serves the affected regions. Once it arrives the payload, the said KeyValue, is routed to the HRegion that is responsible for the affected row. The data is written to the WAL and then put into the MemStore of the actual Store that holds the record. And that also pretty much describes the write-path of

About Me

LARS GEORGE

I am a Software Engineer by heart and work on large scale, distributed, failover-safe systems. Especially "NoSQL" type storage architectures and their related projects like Hadoop with HBase, Hive, Pig. Or Dynomite, Voldemort, Cassandra and so on. I am offering consulting services in this area and for these products. Contact me at info@larsgeorge.com or read about my offer!

View my complete profile

Where I am as well

LinkedIn

Xing

Facebook

Twitter

This work by Lars George is licensed under a Creative Commons Attribution-Noncommercial-Share Alike 3.0 Germany License.

My Tweets

HBase.

Eventually when the MemStore gets to a certain size or after a specific time the data is asynchronously persisted to the file system. In between that timeframe data is stored volatile in memory. And if the HRegionServer hosting that memory crashes the data is lost... but for the existence of what is the topic of this post, the WAL!

We have a look now at the various classes or "wheels" working the magic of the WAL. First up is one of the main classes of this contraption.

HLog

The class which implements the WAL is called HLog. What you may have read in my previous post and is also illustrated above is that there is only one instance of the HLog class, which is one per HRegionServer. When a HRegion is instantiated the single HLog is passed on as a parameter to the constructor of HRegion.

Central part to HLog's functionality is the `append()` method, which internally eventually calls `doWrite()`. It is what is called when the above mentioned modification methods are invoked... or is it? One thing to note here is that for performance reasons there is an option for `put()`, `delete()`, and `incrementColumnValue()` to be called with an extra parameter set: `setWriteToWAL(boolean)`. If you invoke this method while setting up for example a `Put()` instance then the writing to WAL is forfeited! That is also why the downward arrow in the big picture above is done with a dashed line to indicate the optional step. By default you certainly want the WAL, no doubt about that. But say you run a large bulk import MapReduce job that you can rerun at any time. You gain extra performance but need to take extra care that no data was lost during the import. The choice is yours.

Another important feature of the HLog is keeping track of the changes. This is done by using a "sequence number". It uses an AtomicLong internally to be thread-safe and is either starting out at zero - or at that last known number persisted to the file system. So as the region is opening its storage file, it reads the highest sequence number which is stored as a meta field in each HFile and sets the HLog sequence number to that value if it is higher than what has been recorded before. So at the end of opening all storage files the HLog is initialized to reflect where persisting has ended and where to continue. You will see in a minute where this is used.

The image to the right shows three different regions. Each of them covering a different row key range. As mentioned above each of these regions shares the the same single instance of HLog. What that means in this context is that the data as it arrives at each region it is written to the WAL in an unpredictable order. We will address this further below.

Finally the HLog has the facilities to recover and split a log left by a

crashed HRegionServer. These are invoked by the HMaster before regions are deployed again.

HLogKey

Currently the WAL is using a Hadoop SequenceFile, which stores record as sets of key/values. For the WAL the value is simply the KeyValue sent from the client. The key is represented by an HLogKey instance. If you may recall from my first post in this series the KeyValue does only represent the row, column family, qualifier, timestamp, and value as well as the "Key Type". Last time I did not address that field since there was no context. Now we have one because the Key Type is what identifies what the KeyValue represents, a "put" or a "delete" (where there are a few more variations of the latter to express what is to be deleted, value, column family or a specific column).

What we are missing though is where the KeyValue belongs to, i.e. the region and the table name. That is stored in the HLogKey. What is also stored is the above sequence number. With each record that number is incremented to be able to keep a sequential order of edits. Finally it records the "Write Time", a time stamp to record when the edit was written to the log.

LogFlusher

As mentioned above as data arrives at a HRegionServer in form of KeyValue instances it is written (optionally) to the WAL. And as mentioned as well it is then written to a SequenceFile. While this seems trivial, it is not. One of the base classes in Java IO is the Stream. Especially streams writing to a file system are often buffered to improve performance as the OS is much faster writing data in batches, or blocks. If you write records separately IO throughput would be really bad. But in the context of the WAL this is causing a gap where data is supposedly written to disk but in reality it is in limbo. To mitigate the issue the underlaying stream needs to be flushed on a regular basis. This functionality is provided by the LogFlusher class and thread. It simply calls `HLog.optionalSync()`, which checks if the `hbase.regionserver.optionallogflushinterval`, set to 10 seconds by

Labels

hbase (19)

hadoop (16)

work (10)

linux (6)

java (4)

nosql (4)

openhug (3)

erlang (2)

music (2)

vserver (2)

apache (1)

aws (1)

bigtable (1)

couchdb (1)

ec2 (1)

eclipse (1)

fosdem (1)

home (1)

iphone (1)

katta (1)

lucene (1)

macos (1)

xen (1)

xml (1)

xsl (1)

xslt (1)

default, has been exceeded and if that is the case invokes `HLog.sync()`. The other place invoking the sync method is `HLog.doWrite()`. Once it has written the current edit to the stream it checks if the `hbase.regionserver.flushlogentries` parameter, set to 100 by default, has been exceeded and calls sync as well.

Sync itself invokes `HLog.Writer.sync()` and is implemented in `SequenceFileLogWriter`. For now we assume it flushes the stream to disk and all is well. That in reality this is all a bit more complicated is discussed below.

LogRoller

Obviously it makes sense to have some size restrictions related to the logs written. Also we want to make sure a log is persisted on a regular basis. This is done by the LogRoller class and thread. It is controlled by the `hbase.regionserver.logroll.period` parameter in the `$HBASE_HOME/conf/hbase-site.xml` file. By default this is set to 1 hour. So every 60 minutes the log is closed and a new one started. Over time we are gathering that way a bunch of log files that need to be maintained as well. The `HLog.rollWriter()` method, which is called by the LogRoller to do the above rolling of the current log file, is taking care of that as well by calling `HLog.cleanOldLogs()` subsequently. It checks what the highest sequence number written to a storage file is, because up to that number all edits are persisted. It then checks if there is a log left that has edits all less than that number. If that is the case it deletes said logs and leaves just those that are still needed.

This is a good place to talk about the following obscure message you may see in your logs:

```
2009-12-15 01:45:48,427 INFO
org.apache.hadoop.hbase.regionserver.HLog: Too
many hlogs: logs=130, maxlogs=96; forcing flush of region with
oldest edits:
foobar,1b2dc5f3b5d4,1260083783909
```

It is printed because the configured maximum number of log files to keep exceeds the number of log files that are required to be kept because they still contain outstanding edits that have not yet been persisted. The main reason I saw this being the case is when you stress out the file system so much that it cannot keep up persisting the data at the rate new data is added. Otherwise log flushes should take care of this. Note though that when this message is printed the server goes into a special mode trying to force flushing out edits to reduce the number of logs required to be kept.

The other parameters controlling the log rolling are `hbase.regionserver.hlog.blocksize` and `hbase.regionserver.logroll.multiplier`, which are set by default to rotate logs when they are at 95% of the blocksize of the SequenceFile, typically 64M. So either the logs are considered full or when a certain amount of time has passed causes the logs to be switched out, whatever comes first.

Replay

Once a HRegionServer starts and is opening the regions it hosts it checks if there are some left over log files and applies those all the way down in `Store.doReconstructionLog()`. Replaying a log is simply done by reading the log and adding the contained edits to the current MemStore. At the end an explicit flush of the MemStore (note, this is not the flush of the log!) helps writing those changes out to disk.

The old logs usually come from a previous region server crash. When the HMaster is started or detects that region server has crashed it splits the log files belonging to that server into separate files and stores those in the region directories on the file system they belong to. After that the above mechanism takes care of replaying the logs. One thing to note is that regions from a crashed server can only be redeployed if the logs have been split and copied. Splitting itself is done in `HLog.splitLog()`. The old log is read into memory in the main thread (means single threaded) and then using a pool of threads written to all region directories, one thread for each region.

Issues

As mentioned above all edits are written to one HLog per HRegionServer. You would ask why that is the case? Why not write all edits for a specific region into its own log file? Let's quote the BigTable paper once more:

> If we kept the commit log for each tablet in a separate log file, a very large number of files would be written concurrently in GFS. Depending on the underlying file system implementation on each GFS server, these writes could cause a large number of disk seeks to write to the different physical log files.

HBase followed that principle for pretty much the same reasons. As explained above you end up with many files since logs are rolled and kept until they are safe to be deleted. If you do this for every region separately this would not scale well - or at least be an itch that sooner or later is causing pain.

So far that seems to be no issue. But again, it causes problems when things go wrong. As long as you have applied all edits in time and persisted the data safely, all is well. But if you have to split the log because of a server crash then you need to divide into suitable pieces, as described above in the "replay" paragraph. But as you have seen above as well all edits are intermingled in the log and there is no index of what is stored at all. For that reason the HMaster cannot redeploy any region from a crashed server until it has split the logs for that very server. And that can be quite a number if the server was behind applying the edits.

Another problem is data safety. You want to be able to rely on the system to save all your data, no matter what newfangled algorithms are employed behind the scenes. As far as HBase and the log is concerned you can turn down the log flush times to as low as you want - you are still dependent on the underlaying file system as mentioned above; the stream used to store the data is flushed but is it written to disk yet? We are talking about fsync style issues. Now for HBase we are most likely talking Hadoop's HDFS as being the file system that is persisted to.

Up to this point it should be abundantly clear that the log is what keeps data safe. For that reason a log could be kept open for up to an hour (or more if configured so). As data arrives a new key/value pair is written to the SequenceFile and occasionally flushed to disk. But that is not how Hadoop was set out to work. It was meant to provide an API that allows to open a file, write data into it (preferably a lot) and closed right away, leaving an immutable file for everyone else to read many times. Only after a file is closed it is visible and readable to others. If a process dies while writing the data the file is pretty much considered lost. What is required is a feature that allows to read the log up to the point where the crashed server has written it (or as close as possible).

Interlude: HDFS append, hflush, hsync, sync... wth?

It all started with HADOOP-1700 reported by HBase lead Michael Stack. It was committed in Hadoop 0.19.0 and meant to solve the problem. But that was not the case. So the issue was tackled again in HADOOP-4379 aka HDFS-200 and implemented `syncFs()` that was meant to help syncing changes to a file to be more reliable. For a while we had custom code (see HBASE-1470) that detected a patched Hadoop that exposed that API. But again this did not solve the issue entirely.

Then came HDFS-265, which revisits the append idea in general. It also introduces a `Syncable` interface that exposes `hsync()` and `hflush()`.

Lastly `SequenceFile.Writer.sync()` is not the same as the above, it simply writes a synchronization marker into the file that helps reading it later - or recover data if broken.

While append for HDFS in general is useful it is not used in HBase, but the `hflush()` is. What it does is writing out everything to disk as the log is written. In case of a server crash we can safely read that "dirty" file up to the last edits. The append in Hadoop 0.19.0 was so badly suited that a `hadoop fsck /` would report the DFS being corrupt because of the open log files HBase kept.

Bottom line is, without Hadoop 0.21.0 you can very well face data loss. With Hadoop 0.21.0 you have a state-of-the-art system.

Planned Improvements

For HBase 0.21.0 there are quite a few things lined up that affect the WAL architecture. Here are some of the noteworthy ones.

SequenceFile Replacement

One of the central building blocks around the WAL is the actual storage file format. The used SequenceFile has quite a few shortcomings that need to be addressed. One for example is the suboptimal performance as all writing in SequenceFile is synchronized, as documented in HBASE-2105.

As with HFile replacing MapFile in HBase 0.20.0 it makes sense to think about a complete replacement. A first step was done to make the HBase classes independent of the underlaying file format. HBASE-2059 made the class implementing the log configurable.

Another idea is to change to a different serialization altogether. HBASE-2055 proposes such a format using Hadoop's Avro as the low level system. Avro is also slated to be the new RPC format for Hadoop, which does help as more people are familiar with it.

Append/Sync

Even with `hflush()` we have a problem that calling it too often may cause the system to slow down. Previous tests using the older `syncFs()` call did show that calling it for every record slows down the system considerably. One step to help is to implement a "Group Commit", done in HBASE-1939. It flushes out records in batches. In addition HBASE-1944 adds the notion of a "deferred log flush" as a parameter of a Column Family. If set to `true` it leaves the syncing of changes to the log to the newly added LogSyncer class and thread. Finally HBASE-2041 sets the `flushlogentries` to 1 and `optionallogflushinterval` to 1000 msecs. The `.META.` is always synced for every change, user tables can be configured as needed.

Distributed Log Splitting

As remarked splitting the log is an issue when regions need to be redeployed. One idea is to keep a list of regions with edits in Zookeeper. That way at least all "clean" regions can be deployed instantly. Only those with edits need to wait then until the logs are split.

What is left is to improve how the logs are split to make the process faster. Here is how is the BigTable addresses the issue:

> One approach would be for each new tablet server to read this full commit log file and apply just the entries needed for the tablets it needs to recover. However, under such a scheme, if 100 machines were each assigned a single tablet from a failed tablet server, then the log file would be read 100 times (once by each server).

and further

> We avoid duplicating log reads by first sorting the commit log entries in order of the keys (table, row name, log sequence number). In the sorted output, all mutations for a particular tablet are contiguous and can therefore be read efficiently with one disk seek followed by a sequential read. To parallelize the sorting, we partition the log file into 64 MB segments, and sort each segment in parallel on different tablet servers. This sorting process is coordinated by the master and is initiated when a tablet server indicates that it needs to recover mutations from some commit log file.

This is where its at. As part of the HMaster rewrite (see HBASE-1816) the log splitting will be addressed as well. HBASE-1364 wraps the splitting of logs into one issue. But I am sure that will evolve in more sub tasks as the details get discussed.

Posted by Lars George at 3:45 PM

Labels: hadoop, hbase

8 comments:

Hyunsik Choi February 11, 2010 at 9:48 AM

That's really informative!

Reply

Unknown September 13, 2010 at 2:48 PM

Great details! Thanks!

Reply

Unknown November 29, 2010 at 5:06 PM

Great!But I'm very curious about which tool you draw these pictures?

Reply

आशिष आल्मेडा (Ashish) January 19, 2011 at 2:01 AM

probably in inkscape...

Reply

Lars George January 25, 2011 at 6:41 AM

I am using OmniGraffle on my Mac. Great tool!

Reply