HBase

Random HBase topics

Wednesday, March 21, 2012

ACID in HBase

By Lars Hofhansl

As we know, ACID stands for Atomicity, Consistency, Isolation, and Durability.

HBase supports ACID in limited ways, namely Puts to the same row provide all ACID guarantees. (HBASE-3584 adds multi op transactions and HBASE-5229 adds multi row transactions, but the principle remains the same)

So how does ACID work in HBase?

HBase employs a kind of MVCC. And HBase has no mixed read/write transactions.

The nomenclature in HBase is bit strange for historical reasons. In a nutshell each RegionServer maintains what I will call "strictly monotonically increasing transaction numbers".

When a write transaction (a set of puts or deletes) starts it retrieves the next highest transaction number. In HBase this is called a **WriteNumber**.
When a read transaction (a Scan or Get) starts it retrieves the transaction number of the last committed transaction. HBase calls this the **ReadPoint**.

Each created KeyValue is tagged with its transaction's WriteNumber (this tag, for historical reasons, is called the **memstore timestamp** in HBase. Note that this is separate from the application-visible timestamp.)

The highlevel flow of a write transaction in HBase looks like this:

1. lock the row(s), to guard against concurrent writes to the same row(s)
2. retrieve the current writenumber
3. apply changes to the WAL (Write Ahead Log)
4. apply the changes to the Memstore (using the acquired writenumber to tag the KeyValues)
5. commit the transaction, i.e. attempt to roll the Readpoint forward to the acquired Writenumber.
6. unlock the row(s)

The highlevel flow of a read transaction looks like this:

1. open the scanner
2. get the current readpoint
3. filter all scanned KeyValues with memstore timestamp > the readpoint

4. close the scanner (this is initiated by the client)

In reality it is a bit more complicated, but this is enough to illustrate the point. Note that a reader acquires no locks at all, but we still get all of ACID.

It is important to realize that this only works if transactions are **committed strictly serially**; otherwise an earlier uncommitted transaction could become visible when one that started later commits first. In HBase transaction are typically short, so this is not a problem.

HBase does exactly that: All transactions are committed serially.

Committing a transaction in HBase means settting the current ReadPoint to the transaction's WriteNumber, and hence make its changes visible to all new Scans.
HBase keeps a list of all unfinished transactions. A transaction's commit is delayed until all prior transactions committed. Note that HBase can still make all changes immediately and concurrently, only the commits are serial.

Since HBase does not guarantee any consistency between regions (and each region is hosted at exactly one RegionServer) all MVCC data structures only need to be kept in memory on every region server.

The next interesting question is what happens during compactions.

In HBase compactions are used to join multiple small store files (create by flushes of the MemStore to disk) into a larger ones and also to remove "garbage" in the process.
Garbage here are KeyValues that either expired due to a column family's TTL or VERSION settings or were marked for deletion. See here and here for more details.

Now imagine a compaction happening while a scanner is still scanning through the KeyValues. It would now be possible see a partial row (see here for how HBase defines a "row") - a row comprised of versions of KeyValues that do not reflect the outcome of any serializable transaction schedule.

The solution in HBase is to keep track of the earliest readpoint used by any open scanner and never collect any KeyValues with a memstore timestamp larger than that readpoint. That logic was - among other enhancements - added with HBASE-2856, which allowed HBase to support ACID guarantees even with concurrent flushes.
HBASE-5569 finally enables the same logic for the delete markers (and hence deleted KeyValues).

Lastly, note that a KeyValue's memstore timestamp can be cleared (set to 0) when it is older than the oldest scanner. I.e. it is known to be visible to every scanner, since all earlier scanner are finished.

Update Thursday, March 22:
A couple of extra points:

- The readpoint is rolled forward even if the transaction failed in order to not stall later transactions that waiting to be committed (since this is all in the same process, that just mean the roll forward happens in a Java finally block).
- When updates are written to the WAL a single record is created for the all changes. There is no separate commit record.
- When a RegionServer crashes, all in flight transaction are eventually replayed on another RegionServer if the WAL record was written completely or discarded otherwise.

Posted by Lars Hofhansl at 2:20 PM

g+1 +9 Recommend this on Google

2 comments:

Jim Green May 23, 2014 at 3:19 PM

Nice post!

Reply

pranab Dash June 10, 2014 at 9:26 PM

Informative post, provides good insight on the transactional characteristics :)

Reply

Enter your comment...

Comment as: Select profile...

Publish **Preview**

Newer Post Home Older Post

Subscribe to: Post Comments (Atom)

Simple template. Powered by Blogger.