HBase Table Locks

Enis Soztutar
enis@hortonworks.com

Issues

New features:
HBASE-7305 ZK based Read/Write locks for table operations
HBASE-7546 Obtain a table read lock on region split operations
HBASE-7976 Rest of master operations (move, assign, unassign, offline region) should also acquire table locks
HBASE-7977 Online merge should acquire table lock
HBASE-7965 Port table locking to 0.94 (HBASE-7305, HBASE-7546, HBASE-7933)
HBASE-7848 Use ZK-based read/write lock to make flush-type snapshot robust

History:
HBASE-5494 (0.89-fb) Introduce a zk hosted table-wide read/write lock so only one table operation at a time
HBASE-5991 (0.89-fb) Introduce sequential ZNode based read/write locks

Bugs:
HBASE-7933 NPE in TableLockManager

Why do you need table locks?

HBase has the following operations from master: create, delete, disable, enable, add column, modify column, delete column, modify table, which we will call master table operations. There are also region splitting and merging going on at the same time. All of the operations, including online merge, are initiated and tracked by the master. However, region splits are initiated by the region servers.

First we need some kind of locking mechanism for master operations, so that two users are not trying to create two tables with the same name, or alter table is not running concurrently with disable table. This can easily be achieved with master-side in memory locking.

However, for all of the above master table operations, we do have to open/close or reopen all the regions in the table so that the changes take into effect, but region splits changes the set of regions. We have to synchronize on the agreed list of regions, so that we can ensure that the changes to the table/column metadata are seen by all the regions.

Since the splits are initiated by region servers, and master is not involved in the decision, the best way to sync is to use zookeeper locks.

Ordinary Lock vs ReadWrite lock

An ordinary lock guarantees that only one process can enter it's critical section. However, what we want is to have concurrent splits going on, but not concurrent splits and master operations. That is why we are using a ReadWrite lock instead of a ordinary boring lock.

Table locks vs Region locks

This implementation acquires a read or write lock for the table, not for individual regions. An alternative implementation might be done to use ordinary locks per region, and master operations acquiring locks for every region, and region splits acquiring the region lock for their region. We think that, since there might be a large number of regions, per-region locks are more costly.

Read lock vs Write lock

Read lock means that it is a shared lock, while write lock is an exclusive lock. We want region splits to acquire a read lock on the table, since we want concurrent splits happening at the same time.

On the other hand, we want master table operations to acquire write lock, since we want to exclude other master table operations for the same table, and also exclude region splits.

If we did the other way, region splits acquiring the write lock, and master operations acquiring the read lock, then we cannot have concurrent splits, and more dangerously, we would not exclude master operations from each other (disable / alter at the same time).

In this design, snapshot operations, take snapshot, clone, restore, should also take write locks to the table, since we want to exclude other master operations and region splits.

Implementation overview

Zookeeper lock recipe

The actual lock implementation is a more-or-less direct implementation of the zk recipe found here: http://zookeeper.apache.org/doc/trunk/recipes.html

For the lazy:
Clients wishing to obtain a lock do the following:

1. Call **create( )** with a pathname of "_locknode_/guid-lock-" and the *sequence* and *ephemeral* flags set. The *guid* is needed in case the create() result is missed. See the note below.
2. Call **getChildren( )** on the lock node *without* setting the watch flag (this is important to avoid the herd effect).
3. If the pathname created in step **1** has the lowest sequence number suffix, the client has the lock and the client exits the protocol.
4. The client calls **exists( )** with the watch flag set on the path in the lock directory with the next lowest sequence number.
5. if **exists( )** returns false, go to step **2**. Otherwise, wait for a notification for the pathname from the previous step before going to step **2**.

The unlock protocol is very simple: clients wishing to release a lock simply delete the node they created in step 1.
Here are a few things to notice:

- The removal of a node will only cause one client to wake up since each node is watched by exactly one client. In this way, you avoid the herd effect.
- There is no polling or timeouts.
- Because of the way you implement locking, it is easy to see the amount of lock contention, break locks, debug locking problems, etc.

Recoverable Errors and the GUID

- If a recoverable error occurs calling **create()** the client should call **getChildren()** and check for a node containing the *guid* used in the path name. This handles the case (noted above) of the create() succeeding on the server but the server crashing before returning the name of the new node.

Shared Locks

You can implement shared locks by with a few changes to the lock protocol:

Obtaining a read lock:	**Obtaining a write lock:**
1. Call **create( )** to create a node with pathname "`guid-/read-`". This is the lock node use later in the protocol. Make sure to set both the *sequence*	1. Call **create( )** to create a node with pathname "`guid-/write-`". This is the lock node spoken of later in the protocol. Make sure to set both

and *ephemeral* flags. 2. Call **getChildren( )** on the lock node *without* setting the *watch* flag - this is important, as it avoids the herd effect. 3. If there are no children with a pathname starting with "`write-`" and having a lower sequence number than the node created in step **1**, the client has the lock and can exit the protocol. 4. Otherwise, call **exists( )**, with *watch* flag, set on the node in lock directory with pathname staring with "`write-`" having the next lowest sequence number. 5. If **exists( )** returns *false*, goto step **2**. 6. Otherwise, wait for a notification for the pathname from the previous step before going to step **2**	*sequence* and*ephemeral* flags. 2. Call **getChildren( )** on the lock node *without* setting the *watch* flag - this is important, as it avoids the herd effect. 3. If there are no children with a lower sequence number than the node created in step **1**, the client has the lock and the client exits the protocol. 4. Call **exists( ),** with *watch* flag set, on the node with the pathname that has the next lowest sequence number. 5. If **exists( )** returns *false*, goto step **2**. Otherwise, wait for a notification for the pathname from the previous step before going to step **2**.

Notes:

- It might appear that this recipe creates a herd effect: when there is a large group of clients waiting for a read lock, and all getting notified more or less simultaneously when the "`write-`" node with the lowest sequence number is deleted. In fact. that's valid behavior: as all those waiting reader clients should be released since they have the lock. The herd effect refers to releasing a "herd" when in fact only a single or a small number of machines can proceed.

TL;DR; There is a parent lock znode, read or write lock requests SEQUENTIAL_EPHEMERAL child znodes, and you look at the other children to see whether you have the lock or wait for your nearest neighbour to release the lock by putting up a watcher. Releasing the lock is deleting your znode.

The implementation is not reentrant and not revocable.

The zk RW lock implementation is generic, and haven't been polluted by table locks. We can use this for other types of locks (per-region locks, etc).

Main classes (sorry no fancy class diagrams)

InterProcessLock
ZKInterProcessLockBase (main implementation)
ZKInterProcessReadWriteLock / ZKInterProcessReadLock / ZKInterProcessWriteLock

Table locks

Table locks sits on top of zk RW locks. They abstract away the lock implementation details, and provide utilities related to table lock management. The main entrance to the code is TableLockManager which also hosts concrete implementations NullTableLockManager and ZKTableLockManager. NullTLM is a stub implementation for tests, and when table locks are disabled. NullTLM does no-op for every operation.

TableLock interface provides an interface for the rest of HBase for the table locks. It is a wrapper for the underlying lock, and knows about the table, and lock timeout.

ZKTableLockManager implements table locks using ZKInterProcessReadWriteLock. It provides lock reaping functionality for all write locks, which is needed when the master fails over, and has to invalidate locks from the prev master, and ensures that parent znode for the table is deleted when the table is deleted so that we do not leak out znodes for deleted tables.

Configuration Settings

```
  <property>
    <name>hbase.table.lock.enable</name>
    <value>true</value>
    <description>
    Set to true to enable locking the table in zookeeper for schema
change operations.
    Table locking from master prevents concurrent schema
modifications to corrupt table
    state.
    </description>
  </property>
  <property>
    <name>hbase.table.write.lock.timeout.ms</name>
    <value>600000</value>
    <description>
    Configuration key for time out for trying to acquire table write
locks. 10 min by default
    </description>
  </property>
 <property>
    <name>hbase.table.read.lock.timeout.ms</name>
    <value>600000</value>
```

```
    <description>
    Configuration key for time out for trying to acquire table read
locks. 10 min by default
    </description>
  </property>
```