Entity management in Master

sershe, v2, 2013-10-15; HBASE-5487

Table of Contents

Problems 1

Design approach 2

Implementation approach 2

Code 2

Persistent storage 3

State update coordination 3

Data structures 3

Table record 4

Region record 4

Note on user operations 5

Table state machine 5

Region state machine 6

Note on message semantics 6

Operations overview 6

Logical interactions 7

Operations details 8

Table operations 9

Region operations 11

Master startup 18

Balancer 18

Future work 18

Diagram sources 18

# Problems

* Current region management code is hard to reason about.
  + We historically find lots of bugs in code that manages regions (assignment manager, shutdown handling, etc.).
  + For the cases like split, merge, alter table, etc., we have separate logic to ensure consistency; these logics overlap, e.g. we have to see if regions are merging when we want to reassign them, handle parallel reassignment and split, etc.
* Semantics of the way different operations work together and resolve conflicts are not well defined. It becomes a worse problem if the number of operations increases (merge, etc.).
* Some pieces of state in HBase are managed poorly and don’t behave well if master fails in the middle of the operation (e.g. disabling the table, or user-triggered region move).
* The steps were recently made to have better management of the tables via table locks; however
  + it’s not clear if locks will scale, if reused for regions; especially in sufficiently large clusters where region opens/moves/splits can be frequent;
  + table locks themselves may also block table updates for unacceptable amount of time, especially in case of hardware/power/etc. failures where lock will not be released until session is closed in ZK after connection times out. This problem also increases with scale.

# Design approach

The following are general characteristics that we want the new design to have, in itself and as compared to the existing one.

* We should avoid two kinds of split-brain scenarios in this design:
  + Different parts of the state of the same region in multiple places (e.g. separate transient state machines for region operations) that needs to be reconciled.
  + Same region state in multiple places (e.g. master and ZK).
* Ideally, master should not be the separate service and the single point of failure.

# Implementation approach

This brief outline of the semi-consensus implementation choices is informed by the preceding design discussion before and during v1 of this document; it is here to put the detailed design that follows in context. They may still change.

## Code

* Implementation should be usable as a library (“master library”).
* Assignment logic should be easy to unit test by simulating various scenarios.
* The logic should be isolated from the persistent storage of state.

## Persistent storage

We will use special, collocated system tables as persistent storage. “Special” things about them are as such (note that this doesn’t apply to all system tables):

* They are hosted in the same process as the running instance of the master library. Presumably, it’s an RS process internally.
* They cannot be split (for v1), since there’s only one master in one place.
* Master library accesses the table via local method calls (thru an interface), w/o network roundtrips.
* checkAndPut operation, as well as the existing multi-row locks coprocessor will be used to ensure atomic updates (for operations like split and merge).

Alternatives to system tables that were considered are ZK, non-collocated system tables (hosted by some RS), and custom master WAL.

* Non-collocated system tables and ZK can both lead to some split-brain situations; esp. given that perf may not be as fast, so cached state in master would become necessary.
* ZK has many advantages w.r.t. scale, recovery, etc., but some client problems (delayed message delivery is an example) make it hard to use. We may consider it later.
* Custom WAL implementation does not have obvious advantages over just using in-process table with already-existing WAL.

## State update coordination

Master coordinates the state updates from outside (e.g. RS doing splits). Master knows of all external updates to the system store (thru being on the update path, or some hook into table, or ZK notification, …).

Note that external updates are not commands or notifications - they are state changes. Master must first apply (or fail) the state update, and only then may do additional operations based on the new state.

# Data structures

According to the design approach, each table and region has a single persistent record, with its entire state stored in one place. This record is never removed as long as the table/region exists. This record is atomically modifiable.

The record serves as the main and only source of truth about the table or region. Master and RS should not have any state that needs to persist across failure or transferred to a different party, that is not part of the record.

The operations, as described, appear to not suffer from ABA problem w.r.t records, if they do, we might add record version that will be incremented on any change and checked.

## Table record

Table record contains:

* Table descriptor (info).
* Table state (see state machine below).
* Table descriptor version.
* Additional information; logical columns:
  + Create spec - while table is creating, contains the spec necessary to finish the create, other than table descriptor (e.g. region splits). Blob.

Table “version” is bumped on every table-wide operation like alter. This allows us to apply table alterations “lazily” to regions currently in transition/split/merge/offline. See region record below.

For the current “reopen regions on change” approach, the version usage is simple - if regions is out of date, it should be closed and reopened. It can also be used to update regions w/o reopen, if such updates existed.

The additional information will be specific to the state the table is in (for example creation parameters).

## Region record

Region record contains:

* Region descriptor.
* Region state (see state machine below).
* Server that has last opened the region.
* Table descriptor version last used to open the region.
* Additional information, logical columns (only the relevant ones are present):
  + Splitting children (in Opened state when splitting).
  + Merge siblings and child (in Opened, Closing and Closed state when merging).
  + Region closed by user (shell), don’t reopen (in Closing and Closed state).
  + Current opening state start time (in Opening).
  + Current closing state start time (in Closing).
  + Target server to open on (in Closing, Opening).
  + Current user operation (see below).
  + TBD extend?

The table version used doesn’t allow any sort of “time travel” to previous versions – region internals will be initialized without regard for table version. It only allows us to detect when region is out of date (see below in operations).

To make recovery easier, this state machine has “transition” states that are used while some operation is ongoing (e.g. “opening”). These can be implemented as separate states, or as pairs of “from” and “to” states.

## Note on user operations

Region record has a column for the current goal state of the region set by the user via shell commands:

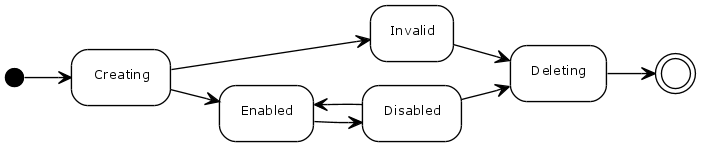
* Open/move to RS.
* Close (and stay closed).
* Split/merge (with other regions).

For v1, there will only be one operation present at a time; any user operation that successfully starts will install itself there, “forgetting” the previous operation. Depending on the operation and timing, the previous one may complete, or be canceled (see operation details).

For example, if region was closed to fulfill user move command, but then the user issued a split before it was opened, the move destination will be “forgotten”, and the region will be split. But if region was about to split and we issued a move, region split may finish and make move irrelevant.

## Table state machine

Table state machine is very simple. There are no separate “disabling”/”enabling” states, because this may conflict with the information already present in regions’ state for the table. Alter is handled thru versions and also doesn’t need separate states; several alterations can proceed in parallel.



## Region state machine

This doesn’t capture the full state w.r.t. merges and splits; region can be Opened and Splitting, or Closed and waiting to be merged. This is stored in separate columns (see descriptions below). Additional states can be added as an alternative.

## http://yuml.me/diagram/boring/activity/9a2749e9.png

## Note on message semantics

In central state machine approach, the messages from master to server serve as notifications of the current state, rather than commands.

Thus, sending multiple messages for the same transition is equivalent to sending one message and should not be an error. During master recovery, it can resend all state transitions to RS without regard to whether they were sent before.

# Operations overview

Region operations have the following external triggers:

* Initialization.
* Shell commands
  + Table commands: create, drop, alter, disable, enable.
  + Region commands: split, assign, close, unassign, move, merge.
* Balancer (regular or from shell).
* RS recovery (“server shutdown handler”).
* RS-triggered operation: split.
* Periodic check (for RS operations that take too long, e.g. opening/closing).

These result in things happening. One trigger, e.g. startup or server shutdown, can consist of several other operations.

## Logical interactions

Describes what **logically** happens when some other operation runs when one is already running. I made a matrix but it’s hard to read. If the interaction is not described, it means the operations do not logically interact.

### Tables

* Alter table
  + Fails if the table is being created, or dropped.
  + Multiple alter-s are applied in logical sequence.
* Disable table
  + Fails if the table is being created, or dropped.
  + Ongoing open, move, split and merge operations may fail if the table is being disabled.
* Enable table.
  + Fails if the table is being created, or dropped.
  + Logically sequential with disable (may run in parallel).
* Drop table.
  + Fails if enable or drop is running.
  + Logically sequential with disable (may run in parallel).
  + Cancels any other table operation, including create.
  + Most ongoing region operations may fail while table is being dropped (some, e.g. user-triggered close, may succeed).

### Region operations

All fail if the table is being created, or dropped.

* User-triggered force close (from shell).
  + Cancels in-progress split, merge, and region open.
  + Cancels the “open” part of an in-progress move that is closing the region.
* User-triggered assignment or move.
  + If region is force-closed, this flag is removed.
  + Take over any in-progress move that is closing the region, by overwriting target server.
  + Cancels in-progress open if it’s not on the target RS.
  + Cancels in-progress split or merge.
  + Fails if table is being disabled.
* User-triggered split.
  + If region is force-closed or table is being disabled, fails.
  + Goes after any in-progress move or open.
  + Cancels in-progress merge.
  + For in-progress split, marks it as user-triggered and succeeds.
* User-triggered merge.
  + If region is force-closed or table is being disabled, fails.
  + Goes after any in-progress move or open.
  + Cancels in-progress split, or merge with different regions.
  + For in-progress merge w/the same regions, marks it as user-triggered and succeeds.
* System move, split, merge.
  + If region is force-closed or table is being disabled, fails.
  + Fails if the region is splitting, merging, opening; or being closed by user.
  + Updates target server for any region that is being closed by system.

# Operations details

Overview:

* Each operation is a table, which is sequence of steps. Each step has:
  + a service that performs it,
  + the step action,
  + how is unexpected state handled,
  + how are service failures handled.
* If unexpected state failover says “master aborts”, it is akin to “assert(false);”. HBCK would be needed to repair.
* Operations may trigger other operations (in *cursive*). Sub-operations don’t need to be tracked, recovery is always possible based just on the state.
* The loose table record format is {descriptor, state, version, additional info}; region record format is {descriptor, state, server last open*ed*, table version last used to open, additional info}.

If some parts are omitted, that means they are not relevant.

* Step that alters a system table are marked in **bold**. They are assumed to be atomic, and check the previous state of the record if indicated. E.g. New -> {, Closed, “server” } means go from “New” state to “Closed” state, and set the server field.

## Table operations

All operations are called by a user. All failover pertains to master failure unless otherwise specified.

### Create table

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | **Who** | **Step** | **Unexpected state** | **Failure handling** |
| **1** | Master | **Tables**: add {Descriptor, Creating, 0, {region details (e.g. splits)}}. | Create fails (name collision). | Create fails. |
| **2** | Master | Create FS structures, etc. not specific to regions. | - | For “Creating” table spec, step 2. |
| **3** | Master | Generate region specifications based on splits/etc. | - |
| **4** | Master | For each region, **Regions**: add {Descriptor, Creating, null, -1}. | Master aborts. | For “Creating” table w/”Creating” regions; delete all FS state, for Creating regions, then delete all creating records, then go to step 3. |
| **5** | Master | For each region, create initial FS state. | - |
| **6** | Master | For each region, **Regions**: Creating -> {, Closed}. | Master aborts. |
| **7** | Master | **Tables**: Creating -> {, Enabled, , {}}. | If state is invalid, go to cleanup; otherwise master aborts. | For “Creating” table w/”Closed” regions, go to step 7. |
| **8** | Master | *Assign* all regions normally (see below). | - | See *assign*. |

Cleanup from failed create (see step 7 above); table is in Invalid state.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | **Who** | **Step** | **Unexpected state** | **Failure handling** |
| **1** | Master | For all Creating regions (can happen only due to failover)  **Regions**: Creating -> {, Closed} | Master aborts. | For invalid table, step 1. |
| **2** | Master | **Tables**: Invalid -> {, Deleting} | Master aborts. |
| **3** | Master | Run *drop table*. | - | See *drop table*. |

### Alter table

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | **Who** | **Step** | **Unexpected state** | **Failure handling** |
| **1** | Client | Get descriptor and version from the master; apply changes to descriptor and send new one w/version. | - | Alter fails. |
| **2** | Master | **Tables**: version -> {new descriptor, , version+1}. | Go to step 1. |
| **3** | Master | Scan all regions; run *reopen* for those in Opened state and not splitting/merging (later can also cancel Opening and reopen, for optimization).  For other states, handling will be done during *open.* | - | Finds regions with outdated version, reopens normally. |

### Disable table

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | **Who** | **Step** | **Unexpected state** | **Failure handling** |
| **1** | Master | **Tables**: Enabled -> {, Disabled}. | Disable fails. | Disable fails. |
| **2** | Master | Depending on region state, **Regions:**   * Child -> {, Deleted}. * Opened -> {, Closing, , , {close start time}}. Remove any user operation information, as well as split/merge information. | - | Finds non-closed regions in a disabled table, do step 2. |
| **3** | Master | Proceed like normal *close* for each region, from step 2; wait for all regions to be closed. |  |

### Enable table

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | **Who** | **Step** | **Unexpected state** | **Failure handling** |
| **1** | Master | **Tables**: Disabled -> {, Enabled}. | Enable fails. | Enable fails. |
| **2** | Master | Scan all regions; trigger *assign* for all Creating and Closed regions. | - | Finds unassigned regions in the enabled table, treats them as normal region. |
| **3** | Master | Wait for all regions to be assigned. |  |

### Drop table

Can be called by user, or create table code if drop was issued while create was in progress.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | **Who** | **Step** | **Unexpected state** | **Failure handling** |
| **1** | Master | Depending on table state, **Tables:**   * Disabled|Deleting -> {, Deleting} * or Creating -> {, Invalid}, drop succeeds. | Drop fails. | Drop fails. |
| **2** | Master | For all regions not in Closed trigger *close* w/ensure close. | - | For “Deleting” table, do step 2. |
| **2** | Master | Wait for all regions to close, set **Regions:** Closed -> {, Deleting}. | Master aborts. |
| **3** | Master | For all regions, remove FS contents, and set **Regions**: Deleting -> delete. |
| **4** | Master | **Tables**: Deleting -> delete |

## Region operations

### Open/assign

TBD verify consistency with conflict matrix/fix glitches

Assign can be initiated by some other master operation, or called by user. Assign can be called with or w/o a specific server. Finding specific server to assign to is out of the scope of this document, but it may be derived from Region record:

* User wants to move the region to some server.
* Previous location of the server.

Note that opening server name is stored in “additional information”, not in server field.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | **Who** | **Step** | **Unexpected state** | **Failure handling** |
| **1** | Master | Master checks whether:   * “force-closed by user” flag is not set; if set, only a user call can assign it; * the table is Enabled. | - | Assign fails. |
| **2** | Master | **Regions:** Closed -> {, Opening, , , {open start time, target server, user operation open}}**.** Clean up “force-closed by user” flag. If called by user, | Assign fails. | Assign fails. |
| **3** | Master | Send message to open to RS w/table descriptor and version. | - | Master: for Opening region, do step 3. |
| **4** | Server | Perform open activities. | - | Master: for Opening region, do step 3.  Server: full failure - handled by “*RS recovery*” or “*State timer – opening*”.  Partial failure – see next table. |
| **5** | Server | **Regions:** Opening, server name -> {, Opened, “server”, version,} Clean up all opening state. If user operation is “open on this server”, clean it up. | Abort the open. |
| **6** | Master | Check that:   * table version is the same as current table version (see Alter table); * the table is enabled (see Table operations); * there’s no user-triggered operation like split, merge, close, or open on a different server.   If so, execute *reopen, move, merge* or *split* as necessary*.* | - | Finds regions with outdated version/incorrect state, reopens or closes normally. |

### Open failed

Happens on steps 4-5 of Assign/open.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | **Who** | **Step** | **Unexpected state** | **Failure handling** |
| **1** | Server | **Regions:** Opening, server name -> {, Closed,,, clean up all opening state}  If user operation is “open on this server”, clean it up. | Abort the open. | Master: after restart, will go to *open*, step 3 for opening region, which may fail again.  Server: full failure - handled by “RS recovery” or “State timer – opening”. Partial failure – abort open. |
| **2** | Master | Run *open* for the region, w/o target server. | See *assign.* | See *assign.* |

### Close/unassign

TBD verify consistency with conflict matrix/fix glitches

Close command can have the following flavors:

* Internal (e.g. from balancer-triggered move)
  + “Ensure close” true of false; changes how splits/merges are handled – canceled if true, close fails if false
* Coming from user
  + As part of a move or reopen.
  + “Force close” shell command. Region will not be reopened until the user says so w/another command.

In case of a move (internal or user-triggered), target server for future assignment can also be supplied.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | **Who** | **Step** | **Unexpected state** | **Failure handling** |
| **1** | Master | Depending on region state, do the following in **Regions.**   * If Opened   + If not splitting/merging:   Opened -> {, Closing, , , {closing start time, force close flag, target server if any, whether user indicated the target}},   * + If splitting/merging and called by user, or ensure-close:   Same + remove split/merge data; find children and siblings; for children: Child -> {, Deleted};  for merge siblings: Opened -> clean up merge data.   * If Opening   Opening -> {, Closed, , , {force close flag, target server if any, whether user indicated the target}}**.**   * Closed -> {,,,, {force close flag, target server if any, whether user indicated the target}}; close succeeds. * Closing -> {,,,, {force close flag, target server if any, whether user indicated the target}}; go to step 5. | Go to step 1 or fail (see description). | Master: close fails.  RS: handled by “RS recovery”. |
| **2** | Master | If Opening/Splitting/Merging/Opened state was cancelled in step 2, send message to RS to close the region (and cancel opening/split/merge, which is an optimization, RS will fail to update state and fail anyway even w/o message). | - | Master: for Closing region, do step 2.  Server: full - handled by “RS recovery” or “State timer – opening”, partial – ignore/force, go to step 4. |
| **3** | Server | Perform close activities. | - |
| **4** | Server | **Regions:** Closing, server name -> {, Closed}. | Finish closing; don’t touch state. | Master: -  Server: handled by “RS recovery” or “State timer – opening”. |
| **5** | Master | Wait for region to be Closed. | - | - |

### RS recovery

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | **Who** | **Step** | **Unexpected state** | **Failure handling** |
| **1** | Master | Find all regions associated with the server in Opening, Opened, Child, Merging, Splitting, Closing state. | - | Master startup finds regions on dead server and resumes.  RS: see *open*. |
| **2** | Master | For each region independently, **Regions:**  Opening|Opened|Merging|Splitting|Closing -> {, Closed}  If Merging|Splitting, find children (must be on the same server) and **Regions:** Child -> {, Deleting}in the same tx. | Go to step 1. |
| **3** | Master | Trigger “*open*” for each region. Servers to assign to can be found together by load balanced. Do cleanup for children. | - |

### State timer – Opening

RS is presumed to be alive, but takes too long to open the region; opening start time is stored in the record, so it can also be done across master restart.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | **Who** | **Step** | **Unexpected state** | **Failure handling** |
| **1** | Master | Choose different server to assign to, like in open. | - | Master startup finds region in opening state w/old start time. |
| **2** | Master | **Regions:** Opening, server name, start time -> {, Opening, , , {new opening start time, new target server}}**.** | Timer irrelevant, success. | Master startup finds region in opening state w/old start time. |
| **3** | Master | Send message to old RS to cancel open (optimization). | - | See *open*. Message to old server will not be sent, but it will fail when trying to update state to Opened; recover to step 4. |
| **4** | MasterServer | Same as open, from step 3. | See *open.* | See *open.* |

### State timer – Closing

RS is presumed to be alive, but takes too long to close the region; we might want to double-check if RS is alive.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | **Who** | **Step** | **Unexpected state** | **Failure handling** |
| **1** | Master | **Regions:** Closing, server name, start time -> {, Closed}**.**  All operations that set Closing wait for the region to be Closed; hence, they will trigger here. | - | Master startup finds region in closed state w/old start time. |

### Split

A RS or a user can trigger split. When client does so, it still goes to RS and not master, so server starts the operation.

If user triggers split when region is in certain states (like Closed, Opening, etc.), split may be delayed. Later when the region is Opened, master will resend the split (see *open*), pretending to be the user.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | **Who** | **Step** | **Unexpected state** | **Failure handling** |
| **1** | Server | Depending on region state: | Go to step 1. | Split fails. |
| If Opened, generate the minimum information for child regions.  **Regions:** Opened, server -> {, Splitting, , , {user-triggered split flag, children}}**.**  nothing -> {Creating descriptor, Child, , , {parent}} x 2 |
| If Opening, Closing, Closed or Splitting and not user-triggered, split fails; otherwise:  **Regions:** Opening|Closing|Closed|Splitting, server -> {,,,, {user-triggered split flag}}. |
| If Merging and not user-triggered, split fails; otherwise generate the minimum information for child regions and, in  **Regions:** Merging, server -> {,,,, {user-triggered split flag, new children}}.  nothing -> {Creating descriptor, Child, , , {parent}} x 2  delete old child of the merge;  Sibling of the merge – Merging, server -> {, Opened, , , {}}. |
| If Child or Deleted, split fails. |
| **2** | Server | Perform splitting activities. Get table descriptor and version. | - | Server: total failure covered by “*RS recovery*”*.* In future, we could recover the split.  Partial failure – see “Abort the split” below. TBD |
| **3** | Server | Parent: Splitting, server -> {, Deleting}  Children: Child -> {, Opening, , version, {target server – self}} | Abort the split (see below). |
| **4** | Server Master | Proceed like *open*, starting from step 4. | See *open.* | See *open.* |

### Merge

User command or a master operation can trigger merge. It operates on two regions;

TBD: no move, just close and reopen merged

### Reopen

Reopen can be issued as a user command, or used as part of alter table. It is the same as closing w/”ensure close” flag and specific target server, followed by opening on that server.

In case of failover of master or the server, opening the region on a different server also counts as “reopen”; hence, there are no special recovery considerations beyond the ones in close and open.

### Move

Move can be requested by a master operation (such as balancer), or by user.

TBD

## Master startup

On master initialization, in parallel, it:

* picks up all region servers (out of the scope of the document), and make a “live” and “dead” list;
* loads the regions and tables store (for the system tables - into block cache).

After the master has the live/dead list, it has to pick up the unfinished operations and restart them (see each individual operations for details; the cues for unfinished operations are specific states (“Creating” table), or table-region inconsistency).

## Balancer

Balancer works as before and generates a number of move requests. Move requests are independent and proceed according to the plan above. We assume most requests will usually succeed, so balance will be improved; we don’t need to make sure all or none are moved or otherwise coordinate.

# Future work

If we use system tables

TBD Shadow regions for MTTR, multiple RS-es for HA; may split system table with only one region collocated, most of the regions of the same table will be close? Then will need separate recovery.

# Diagram sources

Can be pasted at <http://yuml.me/diagram/boring/activity/draw>

Region:

(start)-s or m>(Child),(start)-create tbl>(Creating),(Creating)-create ok>(Closed),(Child)-s or m ok>(Opening),(Opening)-RS opened>(Opened),(Opening)open failed->(Closed),(Closed)assign->(Opening),(Creating)drop table->(Deleting),(Child)s or m failed ->(Deleting),(Closed)drop table->(Deleting),(Deleting)->(end),(Opened)close->(Closing),(Opened)recovery->(Closed),(Closing)-RS closed>(Closed),(Opened)s or m ok->(Deleting),(Creating)retry create->(end)

Table:

(start)->(Creating),(Creating)->(Invalid),(Creating)->(Enabled),(Enabled)->(Disabled),(Disabled)->(Enabled),(Disabled)->(Deleting),(Invalid)->(Deleting),(Deleting)->(end)