Entity management in Master

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

Scope

This is a design spec for "new master". It covers the problems, the requirements to design, and the logical operations and how they interact; it also has an overview on semi-agreed-upon, high level implementation details. Operation implementation details are in a separate document.

Table of Contents

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.
- Many operations are impossible to monitor: for example, when user creates a table, the only way to find out it completed is monitoring that the table is created.

- 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 constraints

The design should:

- Have one source of truth and avoid split-brain scenarios such as:
    - 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, meta and ZK).
- Be resilient to master and RS failures.
- Have clear operation semantics, esp. in case of conflict.
- Not lose functionality compared to current AM (e.g. bulk region assignment).
- Allow users to wait on operations, including on failure.
- Ideally, move towards having non-single-point-of-failure master (distributed master, or otherwise fast recovery). Doesn't have to be in v1 but v1 should not preclude it.

Code should:

- Be usable as a library ("master library").
- Be understandable by mere mortals.
- Easy to unit test by simulating various scenarios w/o MiniCluster and all this stuff (AM logic).
- Have AM logic isolated from the persistent storage of state.

Master commands and operations (to design for)

The following internal and external triggers cause master to do stuff:

- Startup.

- Shell commands
  - Table commands: create, drop, alter, disable, enable.
  - Region commands: split, assign, close/unassign, move, merge, open/assign.
  - Balancer.
- Periodic balancer, potentially other checkers (e.g. auto-merge).
- RS recovery ("server shutdown handler").
- Split triggered by RS.
- RS operation timeout (for opening/closing).

From this, we identify the following operations:

- Table (all user-triggered): create, alter, disable, enable, drop.
- Region:
  - user-triggered: split, merge, move, close/unassign, open/assign.
  - system-triggered: split, merge, move, open/assign.
- Auxiliary: close (part of move, disable table, etc.), reopen (part of alter table).

User-facing semantics: completion, conflicting operations

Describes what should **logically** happen for user-triggered operation to succeed; as well as how operation conflicts are handled. If the conflict is not described, it means the operations do not logically interact.

Table operations

- Create table
  - **Complete**: table state and all regions are created.
    As a convenience, client can wait for regions to be *assigned*, but that's not part of create as such. Justification: see Enable table just below.
- Alter table
  - **Complete**: there are no open regions with old table descriptor.
  - Fails if the table is being created, or dropped.
  - Multiple alter calls are applied in sequence. Each table descriptor change is atomic and cannot accidentally overwrite parallel change.
- Disable table
  - **Complete**: the table state is disabled and it has no open regions.
  - Goes logically after concurrent Enable (see Enable completion).
  - 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.
  - **Complete**: the table state is set to Enabled.
    As a convenience, client can wait for regions to be *assigned*, but that's not part of enable as such. Justification: we can have enabled table with unassigned regions after failures/balance/etc. – as long as they are *being assigned* it's ok.
  - Cancels ongoing disable operations (they fail).

    - Fails if the table is being created, or dropped.
- Drop table.
    - **Complete**: table and all region records are persistently marked for deletion.
    - Fails if table is being enabled or dropped.
    - Goes logically after concurrent Disable (may run in parallel).
    - Cancels ongoing Create and Alter (they fail).
    - Any ongoing region operations *may* fail while table is being dropped.

Region operations

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

- User-triggered force close (from shell).
    - **Complete**: the region is closed, and has a force-close flag.
    - 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.
    - **Complete**: the region is opened (on target server if server is supplied), and does not have a force-close flag.
    - Fails if table is disabled or being disabled.
    - Cancels in-progress open if it's not on the target RS.
    - Cancels in-progress split or merge.
    - Takes over any in-progress move that is closing the region, by overwriting target server.
- User-triggered split.
    - **Complete:** the region is marked for deletion, child regions exist.
    - Fails if region is force-closed or table is disabled or being disabled.
    - Goes logically 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.
    - **Complete:** the regions are marked for deletion, child region exists.
    - Fails if any region is force-closed or table is disabled or being disabled.
    - Goes logically 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-triggered split or merge.
    - Fails if region is force-closed or table is disabled or being disabled.
    - Fails if the region is splitting, merging; or being closed by user.
    - Fails if the region is not Opened.
- System-triggered move or open (recovery).
    - Fails if region is force-closed or table is disabled or being disabled.
    - Fails if the region is splitting, merging, opening; or being closed by user.
    - May cause user-triggered open or move to fail if the region is opened on a different server. System may do it, e.g. if target server is dead.

- If the region is being closed by system, updates target server.

Implementation choices and details

This is a semi-consensus implementation outline, informed by the discussion in the JIRA. It is here to put the detailed design in context. **This is not yet prescriptive and may change.**

Persistent storage

We will use special, collocated system tables as persistent storage. "Special" things about them are as such (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.
- 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 coming from outside (e.g. RS doing splits). Master knows of all external updates to the system store (being on the update path, or thru some hook into system table, or ZK notification, …).

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

Master upgrade path

In v1, whatever we implement should be able to support both new APIs (supporting, for example, waitable user operations), and old APIs. Internally, it doesn't matter if it's an RS with master library running. If so, it will also not host any additional regions.

Master recovery

In V1, master can load hardcoded special-system-table regions. HA can also function as before.

Future improvements to master

If we use system tables, many improvements to master MTTR become possible. Improvements such as “secondary”/”shadow” region servers for the same region will help that.

In future, we can also split system table and have master hosted in multiple RSes. If we specify table name as “key prefix” and use KeyPrefix split policy to ensure that the rows for regions of one table in the “regions” table are always in the same region, we can split “regions” table into multiple regions (assuming “tables” table updates are rare, so it can be remote). In that case, we can also host master regions with normal regions on normal region servers. This will complicate master recovery.