LiveSchemaUpdates

Modifying Schema on a Live Cluster

This page discusses features available in 0.7.

Under the Hood

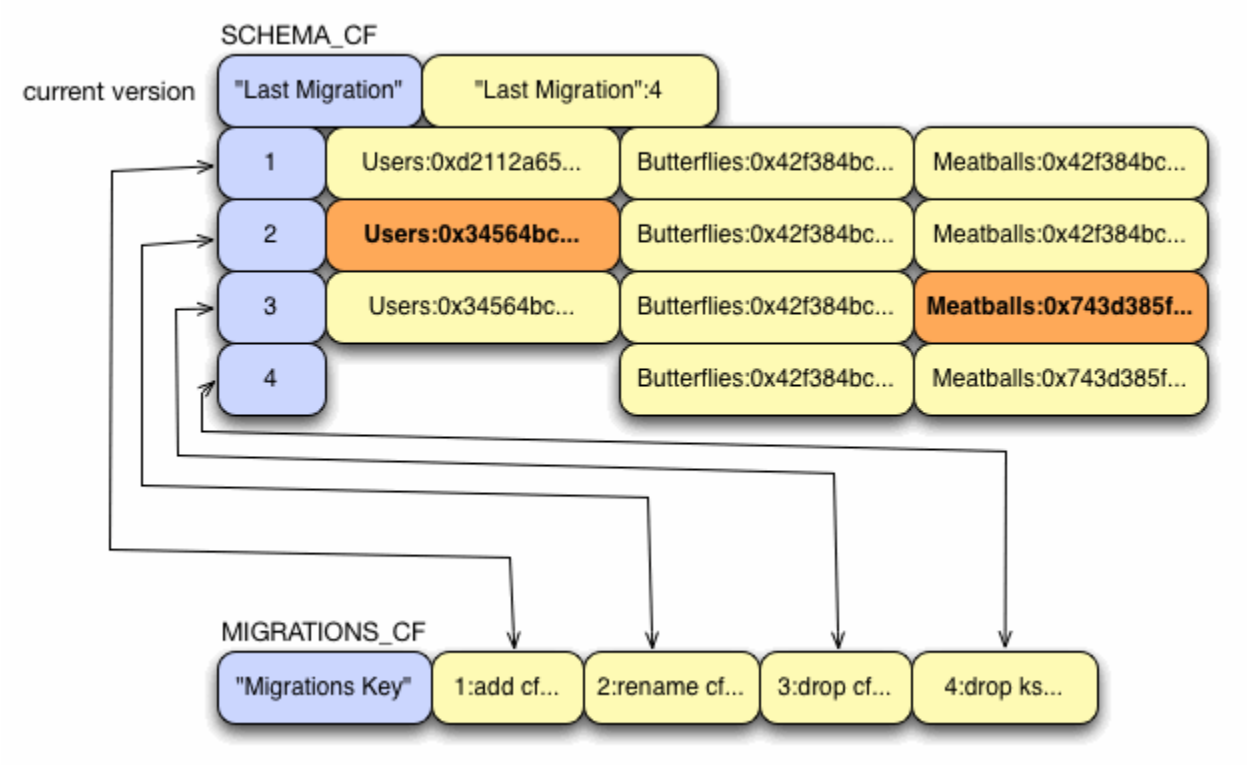
A new system table called `definitions` keeps track of two things: keyspace definitions (`SCHEMA_CF`) and keyspace changes (`MIGRATIONS_CF`). TimeUUIDs are used throughout to match migrations up with schema and vice-versa.

Keyspace Definitions (SCHEMA_CF)

The current set of keyspace definitions are stored in a single row, one keyspace per column with a TimeUUID as the row key (also serves as version identifier), keyspace name as column name, and definition serialization as the column value. There exists a special row, keyed by `"Last Migration"` that contains a single column indicating the current schema version UUID. This makes it easy to look up the version and then retrieve it.

Migrations (MIGRATIONS_CF)

`MIGRATIONS_CF` tracks the individual modifications (add, drop, rename) that are made to the schema. It consists of a single row keyed by `"Migrations Key"` with one column per migration. Each column has the migration version UUID as its name, with the serialized migration as its value.



Operations

Client Operation

- Add column family or keyspace
- Drop column family or keyspace
- Rename column family or keyspace

These are all executed via the Thrift API. It is expected that you have `ALL` access if you are using security. For rename and drop operations the client will block until all associated files are renamed or deleted.

Server Migration Process

Applying a migration consists of the following steps:

1. Generate the migration, which includes a new version UUID.
    a. {{DROP}}s only: snapshot the data that is going away.
2. Update `SCHEMA_CF` with a new schema row.
3. Update `MIGRATION_CF` by appending a migration column.
4. Update the `"Last Migration"` row in `SCHEMA_CF`.
5. Flush the definitions table.
6. Update runtime data structures (create directories, do deletions, etc.)

Handling Failure

A node can fail during any step of the update process. Here is an examination of what will happen if a node fails after each part of the update process (see Server Migration Process above).

1. Nothing has been applied. Update fails outright.
    a. Same. You will have an extra snapshot though.
2. Extra data exists in SCHEMA_CF but will be ignored because "Last Migration" was not updated.
3. Extra data exists in SCHEMA_CF and MIGRATION_CF but will be ignored because "Last Migration" was not updated.
4. **Broken**: commit log will not be replayed until *after* schemas are loaded on restart. This means that the "Last Migration" will be read, but will not be able to be loaded and applied.
5. Startup will happen normally.
6. Startup will happen normally.

If a node crashes during a migration, chances are you will have to do some manual cleanup. For example, if a node cashes after steps 4 or 5 of a `DROP` migration, you will need to manually delete the data files. (Not deleting them does no harm unless you 'recreate' the same CF via `ADD` later on. Then you have an instant database.)

Starting Up

When a node starts up, it checks `SCHEMA_CF` to find out the latest schema version it has. If it finds nothing (as would happen with a new cluster), it loads nothing and logs a warning. Otherwise, it uses the uuid it just read in to load the correct row from `SCHEMA_CF`. That row is deserialized into one or more keyspace definitions which are then loaded in a manner similar to the load-from-xml approach used in the past.

At the same time, the node incorporates its schema version UUID into the gossip digests it sends to other nodes. It may be the case that this node does not have the latest schema definitions (as a result of network partition, bootstrapping a new node, or any other reason you can think of). When a version mismatch is detected the definition promulgation mechanism described next is invoked.

Definition Promulgation

Definition promulgation consists of two phases: *announce* and *push*. *announce* is a way for node A to declare to node B "this is the schema version I have". If the versions are equal, the message is ignored. If A is newer, B responds with an *announce* to A (this functions as a request for updates). If A is older, B responds with a *push* containing all the migrations from B that A doesn't have.

When a schema update originates from the client (Thrift), gossip promulgation is bypassed and this *announce-announce-push* approach is used to push migrations to other nodes.

New Cluster (Fresh 0.7)

For new clusters, things will work best if you start with one node and apply migrations using Thrift until you get the schema you want. Then bring new nodes online and they will pull migrations from the first node (or each other in a large cluster).

Alternatively, you could then shut down the first node and manually copy its `SCHEMA_CF` and `MIGRATIONS_CF` to each new node in the cluster.

The simplest method of applying these schema changes is with `bin/cassandra-cli`. You can either do this interactively, or place the commands in a file and apply them in batch mode (type `help` and `help <command>` to see the available commands). For example:

```
$ cat schema.txt
/* Create a new keyspace */
create keyspace Keyspace1 with replication_factor = 3 and placement_strategy = 'org.apache.cassandra.locator.
RackUnawareStrategy';

/* Switch to the new keyspace */
use Keyspace1;

/* Create new column families */
create column family Standard1 with column_type = 'Standard' and comparator = 'BytesType';
create column family Standard2 with column_type = 'Standard' and comparator = 'UTF8Type' and rows_cached =
10000;
$ bin/cassandra-cli --host localhost --batch < schema.txt
```

Existing Cluster (Upgrade from 0.6)

To provide some backwards compatibility, we've provided a JMX method in the `StorageServiceMBean` that can be used to manually load schema definitions from storage-conf.xml. This is a one-shot operation though, and will only work on a system that contains no existing migrations. If you are upgrading a cluster, you will probably only have to do this for one node (a seed). Gossip will take care of promulgating the changes to the rest of the nodes as they come online.

For those who dont know how to do it (like me):

```
ps aux | grep cassandra # get pid of cassandra
jconsole PID
```

MBeans -> org.apache.cassandra.db -> StorageService -> Operations -> loadSchemaFromYAML

Lastly, there is a system tool that can poke the same JMX method without having to remember its location:

```
bin/schematool HOST PORT import
```

Concurrency

It is entirely possible and expected that a node will receive migration pushes from multiple nodes. Because of this, all migrations are applied on a single-threaded stage and versions are checked throughout to make sure that no migration is applied twice, and no migration is applied out of sync.

Each migration knows the version UUID of the migration that immediately precedes it. If a node is asked to apply a migration and its current version UUID does not match the last version UUID of the migration, the migration is discarded.

One weakness of this model is that it is vulnerable if a new update starts before another update is promulgated to all live nodes--only one migration can be active within a cluster at any time. One way to get around this is to choose one node and only initiate migrations through it.