HiveReplicationv2Development

This document describes the second version of Hive Replication. Please refer to the first version of Hive Replication for details on prior implementation.

This work is under development and interfaces are subject to change. This has been designed for use in conjunction with external orchestration tools, which would be responsible for co-ordinating the right sequence of commands between source and target clusters, fault tolerance/failure handling, and also providing correct configuration options that are necessary to be able to do cross cluster replication.

As of Hive 3.0.0 release : only managed table replication where Hive user owns the table contents is supported. External tables, ACID tables, statistics and constraint replication are not supported.

Issues with the Current Replication System

Some of the observed issues with the current replication implementation are as follows:

1. Slowness
2. Requiring staging dirs with full copies (4xcopy problem)
3. Unsuitability for load-balancing use-cases
4. Incompatibility with ACID
5. Dependency on external tools to do a lot (staging dir mgmt, manage state info, etc.)

We will thus first try to understand why each of these occurs and what we can do about them.

Slowness

Why is the first version of Hive Replication slow?

The primary reason for its slowness is that it depends on state transfer, rather than delta-replay. This means that the amount of data being funneled across to the destination is much larger than it otherwise would be. This is especially a problem with frequent updates/inserts. (Creates cannot be delta-optimized since the original state is null, and deletes are instantaneous.)

The secondary reason is that the original implementation was designed to ensure "correctness" in terms of resilience. We were planning optimizations that would drastically reduce the number of events processed, but these have not yet been implemented. The optimizations would have worked by processing a window of events at a time and skipping the processing of some of the events when a future event nullified the effect of processing the first event (as in cases where an insert followed an insert, or a drop followed a create, etc.). Thus, our current implementation can be seen as a naive implementation where the window size is 1.

Requiring Staging Directories with Full Copies (4xcopy Problem)

Again, this problem comes down to needing to do a state transfer, and using export and import to do it. The first copy is the source table, which is then exported to a staging directory. This is the second copy. It has to be dist-cp-ed over to the destination cluster, which then forms the third copy. Then, upon import, it impresses the data on to the destination table, becoming the fourth copy.

Now, two of these copies, the source table and the destination table, are necessary from the very nomenclature of replication - two copies are needed. The chief issue is that two additional copies are required temporarily in staging directories. For clusters without much temporary overflow space, this becomes a major constraint.

Let us examine each of these copies and the design reasoning behind each. Firstly, on the source, the reason the export is done is to give a stable point that will be resilient when we're trying to distcp to destination, in case the table should be modified during the attempt to copy it over. If we do not make a copy, the critical section of the source-side work is a much longer section, and any replay/recovery semantics get much more complex.

On the destination side, optimizations are certainly possible, and in fact, are actually done, so that we don't have an extra copy here. On import, import actually moves the files to the destination table, thus, our 4xcopy problem is actually a 3x copy problem. However, we've taken to calling this the 4xcopy problem since that was the first problem we hit and then solved.

However, what this optimization does mean is that if we fail during import after moving, then the redo of that import will actually require a redo of the export side as well, and thus, is not trivially retriable. This was a conscious decision as the likelihood of this happening is low, in comparison to the other failure points we have. If we were to desire to make the import resiliently retriable as well, we will have a 4x copy problem.

Unsuitable for Load-Balancing Use Cases

By forcing an "export" early, we handle DR use cases, so that even if the source hive wh should be compromised, we will not suffer unduly, and can replay our exported commands on the destination and recover. However, in doing so, we treat each table and partition as independent objects, for which the only important consideration is that we save the latest state each, without consideration to how they got there.

Thus, any cross-table relationships can be temporarily lost (an object can exist in one table that refers to something in another table which does not have the update yet), and queries that run across tables can produce results on the destination that never existed in the source. All this makes it so that the current implementation of replication is entirely unsuitable for load balancing.

Essentially the primary problem is that the state transfer approach means that each object is considered independent, and each object can "rubber-band" to the latest version of itself. If all events have been processed, we will be in a stable state which is identical to the source, and thus, this can work for load balancing for users that have a pronounced "loading period" on their warehouse that is separate from their "reading period", which allows us time in the middle to catch up and process all events. This is also true at a table level. This can work for many of the traditional data warehousing use cases, but fails for many analytics-like expectations.

We will delve further into this rubber-banding problem in a separate section later, since it is a primary problem we attempt to solve in Replv2.

Incompatibility with ACID

Personally, I think that notions of replication should be developed on top of a robust transactional system(and it is easier to develop that too), and trying to manage replications without such concepts is what leads to problems like the ones we face in hive, where we have to do the best we can with what we have. Hive's flexibility in usage being one of it's major usability points, and having to solve the needs of the majority of the install base is what leads us to try to develop replication without assuming ACID.

And, having said that, due to the nature of compactions, our current approach of simply copying data and re-impressing metadata does not work without the need to copy over transaction objects, translate them and recompact on the destination as well, which does not work well without an idea of distributed transactions as well.

Thus, it is ironic that in our implementation of replication, we do not support ACID tables. We've considered what we would need to do to replicate ACID tables, and in most discussions, a popular notion seems to be one of using streaming to send deltas over to the destination, rather than to copy over the files and trying to fudge around with the transaction metadata. This, however, will require quite some more work, and thus, is not something we're planning on addressing in replv2 either. It is likely to be a major push/focus of the next batch of work we put into replication.

Dependency on External Tools To Do a Lot

Our current implementation assumes that we extend how EXPORT and IMPORT work, allow a Notification and ReplicationTask/Command based api that an external tool can use to implement replication on top of us. However, this means that they are the ones that have to manage staging directories, and in addition, have to manage the notion of what state each of our destination tables/dbs are in, and over time, there is a possibility of extensive hive logic bleeding into them. Apache Falcon has a tool called HiveDR, which has implemented these interfaces, and they've expressed a desire that hive take on some more of the management aspects for a cleaner interface.

To this end, one of the goals of replv2 would be that we manage our own staging directories, and instead of replication tools being the ones that move data over, we step in more proactively to pull the data from the source to the destination.

Support for a Hub-Spoke Model

One more piece of feedback we got was the desire to support a hub-spoke model for replication. While there is nothing in the current design of replication that prevents the deployment of a hub-spoke model, the current implementations by third party tools on top of Hive Replication did not explicitly support a 1:n replication, since they wind up needing to do far too much book-keeping. Now that we take on more of the responsibilities of replication on to hive, we should not have a situation whereby we introduce design artiacts that make hub-spoke replication harder.

One of the primary ways this consideration affects us is that we drifted towards a pull model where the destinations pull the necessary info from the source, instead of a push model. We might choose to revisit aspects of this, but in the meanwhile, this is worth keeping in mind as we design the rest of the system.

Rubberbanding

Consider the following series of operations:

```
CREATE TABLE blah (a int) PARTITIONED BY (p string);
INSERT INTO TABLE blah [PARTITION (p="a") VALUES 5;
INSERT INTO TABLE blah [PARTITION (p="b") VALUES 10;
INSERT INTO TABLE blah [PARTITION (p="a") VALUES 15;
```

Now, for each operation that occurs, a monotonically increasing state-id is provided by DbNotificationListener, so that we have an ability to order those events by when they occurred. For the sake of simplicity, let's say they occurred at states 10,20,30,40 respectively, in order.

Now, if there were another thread running "SELECT * from blah;" from another thread, then depending on when the SELECT command ran, it would have differing results:

1. If it ran before 10, then the table does not yet exist, and will return an error.
2. If it ran between 10 & 20, then the table exists, but has no contents, and thus, it will return an empty set.
3. If it ran between 20 & 30, then it will return { (a,5) }
4. If it ran between 30 & 40, then it will return { (a,5) , (b,10) } , i.e. 2 rows.
5. If it ran after 40, then it will return { (a,5) , (b,10) , (a,15) }.

Now, the problem with the state transfer approach of current replication as it occurs is that if we replicate these sequence of events from source to a destination warehouse, it is entirely likely that the very first time EXPORT runs on the table is quite a while after the event has occurred, say at around event 500. At this point of time, if it tries to export the state of the partition (p="a"), then it will capture all the changes that have occurred till that point.

Let us denote the event of processing an event E to replicate E from source to destination as PROC(E). Thus, PROC(10) would denote the processing of event 10 from source to destination.

Now, let us look at the same select * behaviour we observed on the source as it occurs on the destination.

1. If the select * runs before PROC(10), then we get an error, since the table has not yet been created.
2. If the select * runs between PROC(10) & PROC(20), then it will result in the partition p="a") being impressed over.
    a. If PROC(20) occurs before 40 has occurred, then it will return { (a,5) }
    b. If PROC(20) occurs after 40 has occurred, then it will return { (a,5) , (a,15) } - This is because the partition state captured by PROC(20) will occur after 40, and thus contain (a,15), but partition p="b" has not yet been re-impressed because we haven't yet re-impressed that partition, which will occur only at PROC(30).

We stop our examination at this point, because we see one possible outcome from the select * on the destination which was impossible at the source. This is the problem introduced by state-transfer that we term rubber-banding - nomenclature coming in from online games which deal with each individual player having different latencies, and the server having to reconcile updates in a staggered/stuttering fashion.

An example:

State-transfer has a few good things going for it, such as being resilient and idempotent, but it introduces this problem of temporary states that are possible which never existed in the source, and this is a big no-no for load-balancing use-cases where the destination db is not simply a cold backup but a db that is actively being used for reads.

Change Management

Let us now consider a base part of a replication workflow. It would need to have the following parts:

1. An event happens on the source that causes a change (at, say, t1)
2. A notification event is generated for it (at, say, t2)
3. That notification event is then processed on source to "ready" an actionable task to do on the destination to replicate this. (at, say, t3)
4. The requisite data is copied over from the source wh to the destination warehouse
5. The destination then performs whatever task is needed to restate

Now, so far, our primary problem seems to be that we can only capture "latest" state, and not the original state at the time the event occurred. That is to say that at the time we process the notification, we get the state of the object at that time, t3, instead of the state of the object at time t1. In the time between t1 and t3, the object may have changed substantially, and if we go ahead and take the state at t3, and then apply to destination in an idempotent fashion, always taking only updates, we get our current implementation, with the rubberbanding problem.

Fundamentally, this is the core of our problem. To not have rubberbanding, one of the following must be true of that time period between t1 & t3:

1. No other change must have happenned to the object - which means that we do the equivalent of locking the object in question from t1 to t3. Such an approach is possible if t1 & t3 occur in the same transaction interval.
2. If changes to the object between t1 & t3 are inevitable, we must have a way of recording each state change, so that when t3 rolls around, we still have the original t1 state available somewhere.

Route (1) is how we should approach ACID tables, and should be the way hopefully all hive tables are accessed at some point in the future. The benefit of the transactional route is that we would have exactly the delta/change that we're applying, and we would save that delta to pass on to the other side.

In the meanwhile, however, we must try to solve (2) as well. To this end, our goal with replv2 is to make sure that if there is any hive access that makes any change to an object, we capture the original state. There are two aspects to the original state - the metadata and the data. The metadata is easily solvable, since t1 & t2 can be done in the context of a single hive operation, and we can impress the metadata for the notification and our change to the metadata in the same metastore transaction. This now leaves us the question of what happens with the backing filesystem data for the object.

Now, in addition to this problem that we're solving of tracking what the filesystem state was at the time we did our dump, we have one more problem we want to solve, and that is that of the 4x copy problem. We've already solved the problem with the extra copy on the destination. Now, we need to somehow prevent the extra copy on the source to make this work. Essentially, what we need, to prevent making an extra copy of the entire data on the source, we need to have a "stable" way of determining what the FS backing state for the object was at the time the event occurred.

Both of these problems, that of the 4x copy problem, and that of making sure that we know what FS state existed at t1 to prevent rubberbanding, are then solvable if we have a snapshot of the source filesystem at the time the event occurred. At first, this, to us, led us to looking at HDFS snapshots as the way to solve this problem. Unfortunately, HDFS snapshots, while they would solve our problem, are, per discussion with HDFS folks, not something we can create a large number of, and we might very well likely need a snapshot for every single event that comes along.

However, the idea behind the snapshot is still what we really want, and if HDFS cannot support the number of snapshots that we would create, it is possible for us to do a pseudo-snapshot, so that for all files that are backing hive objects, if we detect any hive operation would move them away or modify them, we retain the original in a separate directory, similar to how we manage Trash. This pseudo-trash like capturing behaviour is what we refer to as the "change-management" piece and is the main piece that needs to be in place to solve the rubberbanding problem as well as the 4x copy problem.

_files

Currently, when we do an EXPORT of a table, the directory structure created in this dump has, at its root, a _metadata file that contains all the metadata state to be impressed, and then has directory structures for each partition to be impressed.

To populate each of the partition directories, it runs a CopyTask that copies the files of each of the partitions over. Now, to make sure that we do not do secondary copies, our design is very simple - instead of a CopyTask, we use a ReplCopyTask, which, will, instead of copying the files to the destination directory, will instead create a file called _files in the destination directory with a list of each of the filenames of the original files.

Thus, instead of partition directories with actual data, we will instead have partition directories with _files files that then contain the location of the original files. (We will discuss and handle what happens when the original files get moved away or deleted later, for now, it is sufficient to assume that these urls will be stable urls to the state of the files at the time we did the dump, as if it were a pseudo-snapshot.)

Now, when this export dump is imported, we need to make sure that for each _files file loaded, we go through the contents of the _files, and apply the copy instead to the underlying file. Also, we will wind up invoking DistCp automatically from hive when we try to copy files over from a remote cluster. (Again, this can be optimized and will be discussed in detail later, but for now, it suffices that we are able to access it.)

With this notion of EXPORT creating _files as indirections to the actual files, and IMPORT loading _files to locate the actual files needing copying, we solve the 4x copy problem.

Solution for Rubber Banding

Here is a possible solution to the rubber banding problem described earlier:
For each metastore event for which a notification is generated, store the metadata object (e.g. table, partition etc), the location of the files (associated with the event) and the checksum of each affected file (the reason for storing the checksum is explained shortly). In case of events which delete files (e.g. drop table/partition), move the deleted files to a configurable location on the file system (let's call it $cmroot for purpose of this discussion) instead of deleting them.

Consider the following sequence of commands for illustration:

```
Event 100: ALTER TABLE tbl ADD PARTITION (p=1) SET LOCATION <location>;
Event 110: ALTER TABLE tbl DROP PARTITION (p=1);
Event 120: ALTER TABLE tbl ADD PARTITION (p=1) SET LOCATION <location>;
```

When loading the dump on the destination side (at a much later point), when the event 100 is replayed, the load task on the destination will try to pull the files from the <location> (the _files contains the path of <location>), which may contain new or different data. To replicate the exact state of the source at the time event 100 occurred at the source, we do the following:

1. When Event 100 occurs at the source, in the notification event, we store the checksum of the file(s) in the newly added partition along with the file path(s).
2. When Event 110 occurs at the source, we move the files of the dropped partition to $cmroot/database/tbl/p=1 instead of purging them.
3. When Event 120 occurs at the source, in the notification event, we again store the checksum of the file(s) in the newly added partition along with the file path(s).

Now when Event 100 is replayed at the destination at a later point, the destination calculates the checksum for file(s) in the partition path. If the checksum differs (for example in this case due to Event 110 and Event 120 that have occurred at the source), the destination looks for those files in $cmroot/database /table/p=1, and pulls them from this location to replicate the state of the source, at the time Event 100 had occurred on the source.

_metadata

Currently, when we EXPORT a table or partition, we generate a _metadata file, which contains a snapshot of the metadata state of the object in question. This _metadata is generated at the point the EXPORT is done. For the purposes of solving rubberbanding, we now also have a need to be able to capture the metadata state of the object in question at the time an event happens. Thus, DbNotificationListener is being enhanced to also store a snapshot of the object itself, rather than just the name of the object, and at event-export time, it takes the metadata not from the metastore, but from the event data.

This then allows us to generate the appropriate object on the destination at the time the destination needs updating to that state, and not earlier. This, in conjunction with the file-pseudo-snapshotting that we introduce, allows us to replay state on the destination for both metadata and data.

A Need for Bootstrap

One of the requests we got was that by offloading too much of the requirements of replication, we push too much "hive knowledge" over to the tools that integrate with us, asking them to essentially bootstrap the destination warehouse to a point where it is capable of receiving incremental updates. Currently, we recommend that users run a manual "EXPORT ... FOR REPLICATION" on all tables involved, set up any dbs needed and IMPORT these dumps as needed, etc, to prepare a destination for replicating into. We need to introduce a mechanism by which we can set up a replication dump at a larger scale than just tables, say, at a DB level. For this purpose, the best fit seemed to be a new tool or command, similar to mysqldump.

(Note, in this section, I constantly refer to mysql and mysqldump, not because this is the only solution out there but because I'm a little familiar with it. Other dbs have equivalent tools)

There are a couple of major differences, however, between expectations we have of something like mysqldump, and a command we implement:

1. The scale of the data involved in an initial dump is orders more for a hive warehouse as compared to a typical mysql db.
2. Transactional isolation & log-based approaches means that mysqldump can have a stable snapshot of the entire db/metadata during which it proceeds to dump out all dbs and tables. So, even if it takes a while to dump them out, it need not worry about the objects changing while it gets dumped. We, on the other hand, need to handle that.

The first point can be solved by using our change-management semantics that we're developing, and using the lazy _files approach rather than a CopyTask.

The second part is a little more involved, and needs to do some consolidation during the dump generation. We will discuss this in short order, after a brief detour of new commands we introduce to manage the replication dump and reload.

New Commands

The current implementation of replication is built upon existing commands EXPORT and IMPORT. These commands are semantically more suited to the task of exporting and importing, than of a direct notion of an applicable event log. The notion of a lazy _files behaviour on EXPORT is not a good fit, since EXPORTs are done with the understanding that they need to be a stable copy irrespective of cleanup policies on the source. In addition, EXPORTing "events" is something that is more tenuous. EXPORTing a CREATE event is easy enough, but it is a semantic stretch to export a DROP event. Thus, to fit our needs better, and to not have to keep making the existing EXPORT and IMPORT way more complex, we introduce a new REPL command, with three modes of operation: REPL DUMP, REPL LOAD and REPL STATUS.

REPL DUMP

Syntax:

```
REPL DUMP <repl_policy> {REPLACE <old_repl_policy>} {FROM <init-evid> {TO <end-evid>} {LIMIT <num-evids>} } {WITH
('key1'='value1', 'key2'='value2')};

Replication policy: <dbname>{{.[<comma_separated_include_tables_regex_list>]}{.
[<comma_separated_exclude_tables_regex_list>]}}
```

This is better described via various examples of each of the pieces of the command syntax, as follows:

(a) REPL DUMP sales;
    REPL DUMP sales.['.*?']

Replicates out sales database for bootstrap, from <init-evid>=0 (bootstrap case) to <end-evid>=<CURR-EVID> with a batch size of 0, i.e. no batching.

(b) REPL DUMP sales.['T3', '[a-z]+'];

Similar to case (a), but sets up db-level replication that includes only table/view 'T3' and any table/view names with just alphabets of any length such as 'orders', 'stores' etc.

(c) REPL DUMP sales.['.*?'].['T[0-9]+', 'Q4'];

Similar to case(a), but sets up db-level replication that excludes table/view 'Q4' and all table/view names that have prefix 'T' and numeric suffix of any length. For example, 'T3', 'T400', 't255' etc. The table/view names are case-insensitive in nature and hence table/view name with prefix 't' would also be excluded from dump.

(d) REPL DUMP sales.[];

This sets up db-level replication that excludes all the tables/views but includes only functions.

(e) REPL DUMP sales FROM 200 TO 1400;

The presence of a FROM <init-evid> tag makes this dump not a bootstrap, but a dump which looks at the event log to produce a delta dump. FROM 200 TO 1400 is self-evident in that it will go through event ids 200 to 1400 looking for events from the relevant db.

(f) REPL DUMP sales FROM 200;

Similar to above, but with an implicit assumed <end-evid> as being the current event id at the time the command is run.

(g) REPL DUMP sales FROM 200 to 1400 LIMIT 100;REPL DUMP sales FROM 200 LIMIT 100;

Similar to cases (d) & (e), with the addition of a batch size of <num-evids>=100. This causes us to stop processing if we reach 100 events, and return at that point. Note that this does not mean that we stop processing at event id = 300, since we began at 200 - it means that we will stop processing events when we have processed 100 events in the event stream (that has unrelated events) belonging to this replication-definition, i.e. of a relevant db or db.table, then we stop.

(h) REPL DUMP sales.['[a-z]+'] REPLACE sales FROM 200;

REPL DUMP sales.['[a-z]+', 'Q5'] REPLACE sales.['[a-z]+'] FROM 500;

This is an example of changing the replication policy/scope dynamically during incremental replication cycle.

In first case, a full DB replication policy "sales" is changed to a replication policy that includes only table/view names with only alphabets "sales.['[a-z]+']" such as "stores", "products" etc. The REPL LOAD using this dump would intelligently drops the tables which are excluded as per the new policy. For instance, table with name 'T5' would be automatically dropped during REPL LOAD if it is already there in target cluster.

In second case, policy is again changed to include table/view 'Q5' and in this case, Hive would intelligently bootstrap the table/view 'Q5' in the current incremental dump. The same is applicable for table/view renames where

(i) REPL DUMP sales WITH ('hive.repl.include.external.tables'='false', 'hive.repl.dump.metadata.only'='true');

The REPL DUMP command has an optional WITH clause to set command-specific configurations to be used when trying to dump. These configurations are only used by the corresponding REPL DUMP command and won't be used for other queries running in the same session. In this example, we set the configurations to exclude external tables and also include only metadata and don't dump data.

Return values:

1. Error codes returned as return error codes (and over jdbc if with HS2)
2. Returns 2 columns in the ResultSet:
    a. <dir-name> - the directory to which it has dumped info.
    b. <last-evid> - the last event-id associated with this dump, which might be the end-evid, or the curr-evid, as the case may be.

Note:

Now, the dump generated will be similar to the kind of dumps generated by EXPORTs, in that it will contain a _metadata file, but it will not contain the actual data files, instead using a _files file as an indirection to the actual files. One more aspect of REPL DUMP is that it does not take a directory as an argument on where to dump into. Instead, it creates its own dump directory inside a root dir specified by a new HiveConf parameter, `hive.repl.rootdir`, which will configure a root directory for dumps, and returns the dumped directory as part of the return value from it. It is intended also that we will introduce a replication dumpdir cleaner which will periodically clean it up.

This call is intended to be synchronous, and expects the caller to wait for the result.

If HiveConf parameter `hive.in.test` is `false`, REPL DUMP will not use a new dump location, thus it will garble an existing dump. Hence before taking an incremental dump, clear the bootstrap dump location if `hive.in.test` is `false`.

Bootstrap note : The FROM clause means that we read the event log to determine what to dump. For bootstrapping, we would not use FROM.

When bootstrap dump is in progress, it blocks rename table/partition operations on any tables of the dumped database and throws HiveException. Once bootstrap dump is completed, rename operations are enabled and will work as normal. If HiveServer2 crashes when bootstrap dump in progress, then rename operations will continue to throw HiveException even after HiveServer2 is restored with no REPL DUMP in progress. This abnormal state should be manually fixed using following work around.

Look up the HiveServer logs for below pair of log messages.

> *REPL DUMP:: Set property for Database: <db_name>, Property: <bootstrap.dump.state.xxxx>, Value: ACTIVE*
>
> *REPL DUMP:: Reset property for Database: <db_name>, Property: <bootstrap.dump.state.xxxx>*

If Reset property log is not found for the corresponding Set property log, then user need to manually reset the database property <bootstrap.dump.state.xxxx> with value as "IDLE" using ALTER DATABASE command.

REPL LOAD

```
REPL LOAD {<dbname>} FROM <dirname> {WITH ('key1'='value1', 'key2'='value2')};
```

This causes a REPL DUMP present in <dirname> (which is to be a fully qualified HDFS URL) to be pulled and loaded. If <dbname> is specified, and the original dump was a database-level dump, this allows Hive to do db-rename-mapping on import. If dbname is not specified, the original dbname as recorded in the dump would be used.

The REPL LOAD command has an optional WITH clause to set command-specific configurations to be used when trying to copy from the source cluster. These configurations are only used by the corresponding REPL LOAD command and won't be used for other queries running in the same session.

Return values:

1. Error codes returned as normal.
2. Does not return anything in ResultSet, expects user to run REPL STATUS to check.

REPL STATUS

```
REPL STATUS <dbname>;
```

Will return the same output that REPL LOAD returns, allows REPL LOAD to be run asynchronously. If no knowledge of a replication associated with that db is present, i.e., there are no known replications for that, we return an empty set. Note that for cases where a destination db or table exists, but no known repl exists for it, this should be considered an error condition for tools calling REPL LOAD to pass on to the end-user, to alert them that they may be overwriting an existing db with another.

Return values:

1. Error codes returned as normal.
2. Returns the last replication state (event ID) for the given database.

Bootstrap, Revisited

When we introduced the notion of a need for bootstrap, we said that the problem of time passing during the bootstrap was something of a problem that needed solving separately.

Let us say that we begin the dump at evid=170, and by the time we finish the dump of all objects contained in our dump, it is now evid=230. For a consistent picture of the dump, we now also have to consolidate the information included in events 170-230 into our dump before we can pass the baton over to incremental replication.

Let us consider the case of a table T1, which was dumped out around evid=200. Now, let us say that the following operations have occurred on the two tables during the time the dump has been proceeding:

event id	operation
184	ALTER TABLE T1 DROP PARTITION(Px)
196	ALTER TABLE T1 ADD PARTITION(Pa)
204	ALTER TABLE T1 ADD PARTITION(Pb)
216	ALTER TABLE T1 DROP PARTITION(Py)

Basically, let us try to understand what happens when partitions are added(Pa & Pb) and dropped(Px & Py) both before and after a table is dumped. So, for our bootstrap, we go through 2 phases - first an object dump of all the objects we're expected to dump, and then a consolidation phase where we go through all the events that occurred during our object dump.

If the table T1 was dumped at around evid=200, then, it will not contain partition Px, since the drop would have been processed before the dump occurred, and it will contain the partition Pa, since that partition was added before the object dump occurred. In contrast, partition Pb will not be present in the dump, since Pb will have not yet been added, and also, it will still contain partition Py, since that partition will not yet have been dropped.

So, given this disparity, we need to consolidate this somehow. There are a couple of ways of consolidation.

Approach 1 : Consolidate at destination.

Now, one approach to handle this would be to simply say that we say that the dump is of the minimum state for the whole object, say 170, and let the various events apply on the destination as long as they are applicable, and ignore errors (such as when we try to drop a partition Px from a replicated table T1 which already does not have Px in it.)

While this can work, the problem with this approach is that the destination can now have tables at differing states as a result of the dump - i.e. a table T2 that was dumped at about evid=220 will have newer info than T1 that was dumped about evid=200, and this is a sort of mini-rubberbanding in itself, since different parts of a whole are at different states. This problem is actually a little worse, since different partitions of a table can actually be at different states. Thus, we will not follow this approach.

Approach 2 : Consolidate at source.

The alternate approach, then, is to go through each of the events from evid=170 to evid=230 in our example, which are the current-event-ids at the beginning of the object dump phase and the end of the object dump phase respectively, and to use that to modify the object dumps that we've just made. Any drops will result in the dumped object being changed/deleted, and any creates will result in additional dumped objects being added. Alters will result in dumped objects being replaced by their newer equivalent. At the end of this consolidation, all objects dumped should be capable of being restored on the destination as if the state for them was 230, and incremental replication can then take over, processing event 230 onwards.

This is the approach we expect to take. One further modification this will require from the current export semantics, is that currently, export exports only 1 _metadata file per table, which contains the list of all the partitions inside it in the _metadata file itself. Instead, now, we propose to split that up so that the _metadata level at an object level will contain only metadata for that object. Thus, _metadata at a table level will contain only the table object, and the individual directories inside it will contain all the required partitions, and each of those dirs will have a partition level _metadata.

Metastore notification API security

We want to secure DbNotificationListener related metastore APIs listed below by adding an authorization logic (other APIs not affected). These three APIs are mainly used by replication operations, so are allowed to be used by admin/superuser only:

1. get_next_notification
2. get_current_notificationEventId
3. get_notification_events_count

The related hive config parameter is "hive.metastore.event.db.notification.api.auth", which is set to true by default.

The auth mechanism works as below:

1. Skip auth in embedded metasore mode regardless of "hive.metastore.event.db.notification.api.auth" setting
   The reason is that we know the metastore calls are made from hive as opposed to other un-authorized processes that are running metastore client.

2. Enable auth in remote metastore mode if "hive.metastore.event.db.notification.api.auth" set to true
   The UGI of the remote metastore client is always set on metastore server. We retrieve this user info and check if this user has proxy privilege according to the proxy user settings. For example, the UGI is user "hive" and "hive" been configured to have the proxy privilege against a list of hosts. Then the auth will pass for the notification related calls from those hosts. If a user "foo" is performing repl operations (e.g. through HS2 with doAs=true), then the auth will fail unless user "foo" is configured to have the proxy privilege.

Setup/Configuration

The following parameters need to be setup in source cluster -

hive.metastore.transactional.event.listeners = org.apache.hive.hcatalog.listener.DbNotificationListener

hive.metastore.dml.events = true

//"Turn on ChangeManager, so delete files will go to cmrootdir."

hive.repl.cm.enabled = true

There are additional replication related parameters (with their default values). These are relevant only to cluster that acts as the source cluster. The defaults should work for these in most cases -

REPLDIR("hive.repl.rootdir","/user/hive/repl/", "HDFS root dir for all replication dumps."),
REPLCMDIR("hive.repl.cmrootdir","/user/hive/cmroot/", "Root dir for ChangeManager, used for deleted files."),
REPLCMRETIAN("hive.repl.cm.retain","24h", new TimeValidator(TimeUnit.HOURS),"Time to retain removed files in cmrootdir."),
REPLCMINTERVAL("hive.repl.cm.interval","3600s",new TimeValidator(TimeUnit.SECONDS),"Inteval for cmroot cleanup thread."),