EXIM Replication Replay Protocol

TABLE object:

Events that modify table object:

CREATE TABLE event:

a) Exports definition only - i.e., not data. This is consistent with what create table itself is doing. (Exception : Unpartitioned tables - here, we do a data&metadata export)

b) At export time, if export finds that it cannot find the table, it simply logs a "noop" in the export definition.

- This protects against any drops that have happened since the time the create table happened, followed by the table not being recreated
- On the import side, the import sees the "noop" and simply exits without doing anything. No error is raised.

c) At export time, if it finds a table definition, it logs the repl.last.id at the time it does the export as part of the export.

- This works for the create table itself as-is, and also works for cases where a drop table happens after the create table, followed by yet another create table before the create table event is processed.
- This also works for cases where ALTERs were interspersed in between, in that it will capture the latest valid state.

d) Also, let's consider one important aspect - that of what an "undo" of a create table looks like. The "obvious" undo of a create table is a drop table. This should be okay irrespective of order of processing, because if we're processing a CREATE TABLE, then the state of the destination is acceptable as the state before the CREATE TABLE, i.e. we can drop the table as the undo, expecting that this command will be redone, thereby creating the table again.

Note:

Question : Is the idea that the export repl id (the id at which export happened) is in a way considered to be the repl id of all table creation DDLs?
Answer : The repl.last.id is a "state" counter of the source warehouse. And all DDL that happens against the source warehouse result in an event being fired which has a 1:1 mapping with that DDL statement.

There, however, is not a 1:1 mapping between the event related to that table mapping and the export that runs, since the export will capture the latest state of that table object(metadata) - this means that we will be slightly out of sync until the replication processing catches up to that point, with all intermediate operations that affect that same object effectively export, but do no-ops on the import.

DROP TABLE event:

a) Maintains the event id of the replication event that spawns it, and drops destination table iff the table's repl.last.id is older than the eventId of the DropCommand.

Note:

Question : Is there an undo for the drop table?

Answer : There is no undo for a drop table - it's a destructive operation. There is, however, an idempotent retry, since it will be the equivalent of a "DROP IF EXISTS (IF OLDER)". If the destination DROP TABLE "should not run" because the destination table is newer, then that should result in a success. However, if we should drop it, and we don't manage to drop it because of a connection glitch or something, we should attempt to retry the DROP without any undo.

Question : Why did we need to follow a DROP-IF-OLDER semantic instead of simply DROP-ing? Wouldn't a future create event wind up recreating the object anyway?

Answer : To make it more resilient in cases of parallelization of events (in the cases of a worker that times out and does not respond back, for eg., but might still be running, albeit slowly in the background), one of the goals of all Commands generated by Replication is that they should be idempotent, and reprocessing of events older than the state of an object should not cause any error. So, if one drone that's processing events (41,42,43) might perform 41 and then not respond back for a significant amount of time, causing Falcon to queue another HiveDR job that starts performing (41,42,43), and 43 might return successfully before the other job performs 42, and then failing. So, one of the early design goals was that all commands should be idempotent and resilient to repeats/replay.

Also, it helps from a performance optimization perspective to not have to drop if it's not needed, because then we wouldn't need to re-import after this.

ALTER TABLE event:

a) Exports definition only, exactly similar to the Create Table event.

b) The only difference here is that an Undo for an AlterTable event does not exist, and an alter table is not seen as being undoable - it is, however, idempotently retriable in that it doesn't need an undo to be done to support retrying.

Cases considered :

Let's say PROC(x) is the moment when event "x" is processed by the replication subsystem, and a notation A->B to denote sequence of actions happening, i.e. B happens after A happens. A->B->C then means that A happens, then B happens, and then C happens.

CREATE->PROC(CREATE)

: straightforward

CREATE->DROP->PROC(CREATE)

: handled by the fact that PROC(CREATE) here will generate a noop export

CREATE1->DROP->CREATE2->PROC(CREATE1)

: handled by the fact that PROC(CREATE1) will capture CREATE2 state, which is the eventual state of the table.

: the future processing of the DROP will not happen because the repl.id on the destination table will be newer

CREATE->ALTER->PROC(CREATE)

: straightforward in that it'll capture the latest state, and PROC(ALTER) will also happen, but since both are metadata-only operations, there's no eventual state change.

CREATE->ALTER->DROP->PROC(CREATE)

: similar to CREATE->DROP->PROC(CREATE), the intermediate ALTER does nothing.

CREATE1->DROP->CREATE2->ALTER->PROC(CREATE1)

: similar to a combination of CREATE1->DROP->CREATE2->PROC(CREATE1) and CREATE->ALTER->PROC(CREATE)

CREATE1->ALTER->DROP->CREATE2->PROC(CREATE1)

: similar to a combination of CREATE->ALTER->DROP->PROC(CREATE) and CREATE1->DROP->CREATE2->PROC(CREATE1)

CREATE1->ALTER1->DROP->CREATE2->ALTER2->PROC(CREATE1)

: another extension of above, and the most complex case here, let's look at each event to satisfy ourselves that this works

: PROC(CREATE1) will create a destination table equivalent to the metadata after ALTER2.

: PROC(ALTER1) will come along, and will again export metadata with identical to above. The import of that export will do nothing, since the state of the table is the “same or newer”

: PROC(DROP) will come along, and not drop the table, since the table state is newer than the drop.

: PROC(CREATE2) will come along, and export definition at the time it runs, again something that is identical to that of after ALTER2. Again, the import of that export will do nothing because the table state is “newer or the same” as that of this export dump.

: PROC(ALTER2) : again, straightforward as above.

: Yes, it's possible we may have additional activities after ALTER2 that may have modified the state of the table between the time we do PROC(CREATE1) to when we process the further events, but we should still be okay, it's just that we will dump a further state as needed.

Note that an alter table can be an alter schema, and that still does not affect the above analysis that the "eventual" state of the table object is consistent. We just have a very loose definition of "eventual" consistency.

Note that we've not considered the effect of Partitions on the above, and yes, that adds some complexity, but we'll deal with that in the section describing the partition events. For now, we're satisfied that the table metadata object looks okay through these permutations of all the commands that can alter the state of a table metadata object.

Another note - DropDatabase can also drop the table object, but we're not worried about that because after a DROP DB, nothing can happen until a CREATE DB is processed, which we (a) currently don't, and (b) if we do begin doing it, then any intermediate inconsistency will be wiped out by the DROP DB, and correct state reinstated after the CREATE DB.

PARTITION object:

Events that modify partition object:

ALTER TABLE & CREATE TABLE events:

: act as metadata-only create and affect only the table object, and thus do not affect a partition object.

DROP TABLE

Will nuke partitions, but should result in all the table's metadata (&potentially data) being dropped.

Let us consider our two cases :

a) First : when PROC(DROP TABLE) does drop the table, because the table's repl.id is older than the eventid of the DROP TABLE event.

: This case is straightforward, in that the table is appropriately dropped, and any new events processed after this point will recreate any data/metadata in the table as needed.

b) Second : when PROC(DROP TABLE) does not drop the table in question, because the table's state is newer than the DROP TABLE event.

: If we have a scenario where the PROC(DROP TABLE) does not drop the table, because some of the state in the table is newer, then we run into an issue wherein a person might still have old partitions in the table, but a new CREATE or ALTER might have resulted in the table object's state being newer, and thus, the table might not be dropped, thus leaving behind the old partitions. Thus, we need one of the following to happen:

i) Every DROP TABLE event should imply and be preceeded by a DROP PARTITION event for each partition inside it. This then means that all "old" partitions will have been processed as DROP PARTITION EVENTS first, so will be consistent. Then, the DROP TABLE event can be processed as normal, and everything'd be good. The issue with this is that this could lead to a very large number of events being generated each time a table is dropped (not common, but spiky when it does happen). Also, this is not how it currently works, in that on the metastore, a drop_table does not result in a multitude of DropPartitionEvents getting fired.

ii) Drop Table's logic can be changed, so as to apply the same drop-if-older semantic that it does at the table level to be recursive on to the partitions. i.e. like this:

DropTable(eventid=47)
If table.repl.id < 47, drop table.
if table.repl.id > 47, iterate through all partitions in the table, and drop all partitions whose repl.id < 47.
(Note : none of those repl.ids can be equal to 47 since that implies they got created as the result of an import whose export ran when repl.id was 47, but that's impossible, since when the source repl.id was 47, this table did not exist. Thus, it must be strictly newer or strictly older.)

: Between approaches (i) and (ii) above, (ii) might be a little more complex in DDL, but more performant and require less metadata api change.

ADD PARTITION/ ALTER PARTITION / DROP PARTITITION events :

Follow the exact same logic we used for CREATE TABLE/ ALTER TABLE / DROP TABLE, at the table level, only they now apply at the partition level.