# Optimise Bootstrap during Control Failover

*(Ayush Saxena)*

**HIGH LEVEL:**

**Problem Statement:**

Optimise the time taken for bootstrap from B->A in case of control failover.

**Present Solution:**

Drop the Initial source database(A) and do a normal bootstrap from B->A

**Probable Solution:**

The data & metadata copy can be optimised, if we can use the incremental mode of copy. But in case of unplanned failover, (DR scenario) the source(A) and target(B) cluster can loose sync, so Incremental flow can not be applied for all the tables, We have to identify the tables which need to be overwritten and on which Incremental flow can be applied.

The solution can be broken down into phases as below:

1. **Initial Phase:** The normal replication flow is going on from Source Cluster (A) -> Target Cluster (B)

   During this phase, We will start maintaining the event id to which B is replicated post every Load, That is the event ID with respect to cluster B's Notification Log, which can be used as marker when going B->A
2. **DR Phase:** Lost access to original source cluster (A), Cluster B, takes over as the new Prod cluster.

Once we get access back to cluster A, We block any further modifications either manual or by any background thread on cluster A to avoid any further divergence between cluster A & B. (Can be by means of ranger policies or better by having a feature to block entries into the Notification Log for the configured databases)

3. **Failover Preparation:** Generate the list of tables which got modified post last dump on the original source cluster A. Generate the list of tables and store it in the staging location.

The information can be fetched using the Notification Log on the source cluster, We would be having the last dumped event Id.

Some sample sql like: select TBL_NAME where ID > <Last Dumped ID> and DB_NAME=<source DB>

4. **Failover B->A:**

- **During Dump(On B):** The dump flow detects the availability of the table_diff file(Can be some other indicators as well) which denotes the availability of tables which needs to be overwritten and indicates it is the first B->A, after DR scenario.

For the tables mentioned it does a bootstrap dump and for others it proceeds with an Incremental mode of replication using the event Id stored as part of the last successful load.

- **During Load(On A):** During the load, the load flow detects the availability of the table_diff file, Indicating it is the first load after a DR scenario. Post successful load delete the table_diff file.

It overwrites the tables in the table_diff and does an incremental load for others.

**Overwrite Mechanism:** The basic mechanism can be a drop table and then load the bootstrapped table.
**Possible Optimisations:** Rather than dropping do an overwrite

5. **<u>Post Failover:</u>**

Everything back to normal, Can go ahead with normal A->B replication flow....

# Execution Steps*(LOW LEVEL DESIGN)*

**Source Cluster:** A
**Target Cluster:** B
**Initial Replication Flow:** A->B

## Replication Policies Names

| Policy Name | Flow | Initial/Reverse | Initial State |
|---|---|---|---|
| P1 | Dump A | Initial | Enabled |
| P2 | Load B | Initial | Enabled |
| R1 | Dump B | Reverse | Disabled |
| R2 | Load A | Reverse | Disabled |

## Deployment Model

| Model | Scope |
|---|---|
| On-Prem to On-Prem | Supported |
| Cloud to Cloud | Unsupported |
| On-Prem to Cloud | Unsupported |
| Cloud to On-Prem | Unsupported |

## Initial Assumptions:

- Non Loaded dump will not get loaded, in case of On-Prem setup, In case of On-Cloud setup, by use of staging directory it can work, but data-copy won't get optimised in that case. Staging needs to have data in a single directory.
- hive.repl.rootDir should be the different for initial and reverse policies.
- Background threads won't get active, unless the repl.target.for property is set, which will be removed after replication has been triggered and the dump has started. Once the dump of events on B is done.
- When going either B->A or A->B the writes will be blocked on the target of replication, before the start of replication.

## Background Threads:

The background threads on B(target), when it takes over, will be disabled by default as earlier it was the target for replication. When B->A replication is triggered, The background threads get enabled by default as part of the second dump.
If the background threads are supposed to be enabled even before that. They can be enabled by adding a dbProperty:
**repl.backgroundthread**=enable.

## Best Practices:

Keep the frequency of reverse policy less initially, since the first cycle just lands up creating a table_diff file, which will be very quick. And revert back to the original frequency after the original cycle has resumed.

## Side Effects:

The actual dumping & loading of events starts from the second cycle of replication, the first cycle is just suppose to identify the tables, which need to replicated

## Expected Don’ts

- Delete the notification log
- Play with replication related properties repl.source.for and repl.target.for
- Go and create fake failover related files in the rootDir, or delete them once we have created them.
- Follow steps sequentially, disable initial policies first and then only enable reverse replication policies, don’t run concurrently.**** Consequence
- After reverse replication is triggered, still letting A be writable when replication is running during B->A. (Target should be read-only, initial assumption for replication). If replication is aborted in middle or fails in middle, then A can be made writable.

## Terms Used:

- **event_ack:** File created after first dump on B, which contains the last loaded event id wrt A
- **table_diff:**WIP file created during load on A, which contains the tables which got modified on A.
- **table_diff_complete**: file created during load on A, which contains the tables which got modified on A. table_diif is renamed to table_diff_complete once it is completed.
- **repl.database.readonly:** Database property, which if set doesn’t allow entries to the notification log neither allow background threads to run, making the database read-only. Optional=3,all, bgthreds,nothing.***** Explore authoz, extra hive user.
- **repl.backgroundthread**=enable: To force enable the background threads, irrespective of repl.target.for, once B takes over as primary.
- **repl.source.for:** Database property, which signifies the database is the source of replication.

- **repl.target.for**: Database property. Which signifies that the database is the target of replication. It blocks background threads on the database.

## Post DR  scenario steps(<u>High-Level</u>)

**When B takes over as primary(Work on Target)**

| Step | Description |
|---|---|
| Disable P2 & P1<br><br>**(Mandatory Manual Step)** | Disable initial replication policies |
| Increase Events TTL & CM on both*<br><br>**(Recommended Manual Step)** | Since we tend to use incremental code flow increase the TTL and CM time-out, so that the events stay by the time we resume replication.<br><br>**Side Effect:** Increasing TTL will increase the load on Notification Logs & Increasing the CM Timeout, will increase the load on CM directories.<br>**Safe Calculation:** Time for A to come up + Time for 6 Cycles to Run + Safe margin |
| Enable R1 & R2<br><br>**(Mandatory Manual Step)** | Enable reverse replication policies |
| First  Execution of R1(Dump B) | Creates the **event_ack** file which contains the event to which B is loaded with respect to A |
| First Execution of R2(Load into A) | Will consume the event_ack file and create the **table_diff_complete** file with respect to that.<br>The process would be to block |

| | |
|---|---|
| | entries to notification log, then block the background threads and then start on table_diff file creation. **** blocks writes |
| Second Execution of R1(Dump B) | **table_diff_complete** file is found, it would do a bootstrap dump for those tables, and for the rest do an incremental. Remove the repl.target.for property.**** bootstrap happens before. |
| Second Execution of R2(Load into A) | **table_diff_complete** file is found it would do a bootstrap load for the tables in the list, incremental for others. For overwriting the tables it will overwrite the initial table metadata along with partitions. Once the load is complete it will delete the table_diff_complete file and remove the repl.source.for config. Explore (Alter)**** |
| Restore Events TTL & CM on both*<br><br>**(Recommended Manual Step)** | If done previously, restore back the values. |
| Further Execution | No table_diff_complete file and B has repl.source.for and A has repl.target.for we can continue normally now |

## Execution Flow Of Reverse Policies

**Success Case**:

1. R1(Dump B): Finds repl.target.for property set on the database and create an **event_ack** file, which contains the last event id with respect to A. If the even_ack exists, and contains the wrong event Id, it

deletes event_ack & table_diff_complete files (if exist) and re-writes it.

2. R2(Load A): Finds repl.source.for property set and checks for **event_ack** file, realises it needs to create the **table_diff** file. Sets **repl.target.for** property, which will block background threads. Starts creating the list of tables and adding it to **table_diff** file, once done renames the table_diff file to **table_diff_complete** file, marking the completion of generation of table diff
3. R1(Dump B): Finds **repl.target.for** property set on the database, looks for **table_diff_complete** file, validates the event_ack is there & correct, if incorrect throws unrecoverable exception(P1 & P2 are also running).If correct it starts the bootstrap dump for the tables in the list and for others goes incremental. Post successful dump of events, before dump_ack it removes **repl.target.for** property. Deletes any older dump directory if it exists.
4. R2(Load A): Finds **repl.source.for** property set on the database, finds the **table_diff_complete** file and finds a dump from B, starts incremental load from that. Post completion removes **repl.source.for** property properties, before ACK.
5. R1(Dump B): Doesn't find **repl.target.for** property. Goes to normal flow checks whether the last load is loaded or not. If loaded, back like normal incremental flow.
6. R2(Load A): Doesn't find **repl.source.for** property set. Goes to normal flow, check whether there is a new dump, if so, goes loading it normally.

**NOTE:** Tables present in A which aren't part of B, Will get dropped on A as well.

## Failure Scenarios

1. A->B replication successful, while coming back to B(B->A) one stage fails with solvable error and we want to continue B->A
2. A->B replication successful. Unsolvable errors during B->A

3. A->B replication successful, while coming back to (B->A) we abort/fail and decide for A->B
4. A->B replication failed, and We need to go for B->A
5. Accidental operations on A & B when A->B is going on.
6. Miscellaneous issues.

**A->B replication successful, while coming back to B(B->A) one stage fails with solvable error and we want to continue B->A**

- **First Dump on B failed:** The first dump on the B, is just supposed to create the event_ack file, by reading the database property and do nothing else, if it fails we are on stage-0(nothing done). So, it can retried after solving the reason for failure, (mostly can only be HMS or HDFS failure)
- **First Load on A failed:** The first load on A is supposed to consume the event_ack file and fetch the table_diff file using the notification log.
    - If it fails before starting to create table_diff, we can simply retry and table_diff file will get created and things will work normally
    - If it fails after starting to write the table_diff file, we will delete the table_diff file and restart from scratch, trying to build the file again. NOTE: if the table_diff file is completed in a run, we rename it to table_diff_complete. So, an existing table_diff just denotes that it is a failed attempt.
- **Second Dump on B fails:** The second dump will consume the table_diff_ack file, and do a bootstrap dump for those tables and incremental for rest. This is the same as an incremental run. If this fails, it can be resumed after correcting the reason of failure and normal checkpointing flow will take care.
- **Second Load on A fails:** The second load also consumes the table_diff_complete file to figure out which tables to drop and bootstrap load. If it fails, normal checkpointing flow will take care and it can be resumed normally.

**A->B replication successful. Unsolvable errors during B->A**

- **First Dump on B failed:** The first dump on the B, is just supposed to create the event_ack file, by reading the database property and does nothing else.There is no reason it can land up in an unrecoverable error state which can not be fixed.
- **First Load on A failed:** The first load on A is supposed to consume the event_ack file and fetch the table_diff file using the notification log. It can fail at an unrecoverable state, if the notification events expired. We have to go for the bootstrap solution**
- **Second Dump on B fails:** The second dump will consume the table_diff_complete file, and do a bootstrap dump for those tables and incremental for rest. This is the same as an incremental run. If this fails, due to non-recoverable error, like events expired. We have to go for the bootstrap solution**
- **Second Load on A fails:** The second load also consumes the table_diff_complete file to figure out which tables to drop and bootstrap load. If we fail here, may be because target got modifiable. So in that case. Go for bootstrap. Same as above.

**Working of BootStrap Solution****

- Drop database A(manual) & remove the non-recoverable file.
- The R2(Load on A) detects the database isn't present and there is an event_ack file. It creates a No_Db file marking the database has been dropped, and we need to go for bootstrap dump
- The R1(Dump on B) detects the No_Db file, Realises it needs to go for bootstrap dump. Removes repl.target.for property, the table_diff file_ack file if exists. and goes with normal bootstrap.

**Cases for Unsolvable errors:**

- Events expired in the Notification Log for either A or B.
- CM Timed-out
- Target for Replication(A) wasn't made read-only
- Target Database doesn't exist or got dropped.

**A->B replication failed**

- **If it fails during dump on A**, we don't care, we just have a half-baked dump directory, which we will eventually clean up in our next A->B run irrespective of whether that was a successful dump dir or not.
- **If it fails during load on B**, We maintain the event-id till which B was loaded as part of database properties already for checkpointing purpose as well as the id with respect to B, post every event load. So, we will be using that event id as a marker and continue.

**A->B replication going On, Accidental Dump/Load on wrong sites.**

- **Accidental Dump on B in case of A->B**

The B cluster will just create an event_ack file without doing anything else, which will also get deleted in case of successive future dump.

- **Accidental Load on A in case of A->B**

The A cluster will do nothing as it will not find event_ack file

**Apart from this if any one executes a full cycle of dump & load, then it isn't protected. If we want to prevent that we would need custom configs to be set on R1 & R2 policies, then unset explicitly unset after their first execution**

## Abort Scenarios

A->B replication successful, while coming back to (B->A) we abort. In case there is an abort for B->A replication and we decide to go for A->B, There are couple of challenges in this case, First is the target B might have got modified, since we made B writable, and second is the swap of configurations repl.source.for/repl.target.for etc.
To counter such a situation we use our Reset-Solution**, The reset solution takes care of sorting out of the DB properties, correctly with respect to A->B, and if there are changes on B, It makes the A->B replication go towards preparing

table_diff and doing a bootstrap for the modified tables on B, so as we can resume A->B replication.

**Working of Reset-Solution****

- Disable R1 & R2(Reverse Policies)(manual)
- Alter P1 & P2 to set repl.failover.rollback=true(manual)
- Enable P1 & P2 (manual)
- First execution of Dump on A(P1): The reset property is set, it will remove the repl.target.for property
- First execution of Load on B(P2): Will remove the repl.source.for property and set repl.target.for property. Will check if the last loaded event id is the same as the present database event id. ***empty If not will create a table_diff_complete file with the tables modified.
- Second execution of Dump on A(P1): Consumes the table_diff_complete file and does a bootstrap dump for those tables.
- Second execution of Load on B(P2): Finds the table_diff_complete file and does a bootstrap load for those tables.
- Alter P1 & P2 to set repl.failover.rollback=false(manual)
- Further dump will clean up the table_diff_complete file.
- Back to Normal…..

- **Abort before First Execution of Dump on B:** This is the first step which creates the event_ack file. Without this nothing can happen. We haven't done anything, we can go back easily, without doing anything just disable the reverse policies and enable the forward ones.
- **Abort after First Execution of Dump on B:** This step will create the event_ack file, with the last loaded event id and do nothing. If we abort here, we would just have an event_ack file nothing more. The file will get deleted in the subsequent B->A run, whenever it happens. So, we can go back without any effort same as above. If B is modified use the **Reset-Solution****
- **Abort during First Execution of Load on A:** This means there is a failure during creation of table_diff file. Use the **Reset-solution*****.

The deletion on event_ack and table_diff_complete will be done in subsequent B->A run(whenever it happens)

- **Abort after First Execution of Load on A:** This means we are aborting after table_diff_complete file got created. Use the **Reset-solution*****.
- **Abort during Second Execution of Dump on B:** This will actually start dumping the events. It won't do any changes in the cluster(dump doesn't do that either). To resume back Use the **Reset-solution*****.
- **Abort after Second Execution of Dump on B:** The background threads would have been enabled, B would be modified, use the **Reset-solution*****.
- **Abort during/after Second Execution of Load on A:** Load started means we have modified B. So, now we have to use optimised-bootstrap cycle only. Use the **Reset-solution*****.
- **RollBack to A as prod after successful second execution of Load on A:** Follow the same steps, we are the same as A->B changing to B->A, with name flipped this time. Do failover in the reverse direction.

**MISCL Issues**

- **A comes back after TTL time:** We have a config preventing deletion of events after restart of HS2, so it won't get deleted for a configurable amount of time. In the mean time TTL can be increased accordingly. *** example while adding. By default enable.*** LOG***

**Snapshots Handling For External Tables:**

**Basic Design Change:**

- Rather than maintaining two snapshots always with same suffix, append the dump eventId with the snapshot name.

Example: Presently we maintain <DbName>replyNew & <DbName>replOld pair of snapshots. Now we maintain <DbName>replNew<EventId-2> & <DbName>replOld<EventId-1>
Where replication is from event id 1 to 2.

Basically the eventId are the fromEventId & ToEventId while we do a dump.

- In the target database on starting the load, maintain the toEventId of the Dump.

**Coming back from B->A, With**

- **Last Cycle of A->B successful:** Normal Handling as we do for bootstrap in case of Control Failover, that is while dump on B create the second snapshot, while copying delete the older snapshot on A and go ahead with normal flow.
- **Some Snapshots on A created, but not all:** Snapshots on A would have been created but on B the snapshot would be stale. So, for the paths where eventId matches, we go ahead like normal flow, as we would have done for above case, For the one which it doesn't match we do a rDiff to the correct older version.

**Example(Normal A->B Complete & We move to B-A):**
**First Iter(A->B): Dump Till EventId=10**
A->B replication going On:
First Initial Copy: Creates Snapshot suffixed eventId 10 on A.<dbName>replNew10
Post Copy: Creates snapshot suffixed with eventId 10 on B, <dbName>replOld10
**Second Iter(A->B) Dump Till eventId=20**
First Diff Copy:
Renames <dbName>replNew10 to <dbName>replOld10
Creates <dbName>replNew20
Post Copy: Deletes replOld10 & creates replOld20
And So-On…
**First Iter(B->A) Dump Till eventId= 50**
Finds replOld20 & creates a replNew50 for the diff copy.
Post Copy: Finds replNew20 on A, renames to replOld20 and does a copy post that
Deletes replNew20 & replOld* & creates replOld50
Normal Flow Goes On……..

- **If Only Dump Was Done, NO Copy.(Stale Snap on one side)**

**Continuing from Second Iter**

Renames <dbName>replNew10 to <dbName>replOld10
Creates <dbName>replNew20
Post Copy: Deletes replOld10 & creates replOld20 ---> This Doesn't Happen
**First Iter(B->A) Dump Till eventId= 50**
Finds **replOld10** & creates a replNew50 for the diff copy.
Copy Phase: Find replOld10 & finds finds replNew20 as well, does a rDiff to replOld10 & deletes the other snapshot,
Flow stays the same thereafter.

**Snapshot Handling in case of Bootstrap Solution**
In case of bootstrap solution the target database is dropped, For external tables with purge true, the data won't be there, so we would see if the target doesn't exist, we go with Initial mode of copy.

**Fallback to normal copy: Snapshot Corrupts:Default True.

**Snapshot Handling in case of Abort Solution**
The handling will stay as in the normal cases.
If we abort post snapshot creation on Target(B): On the target there would be a renamed snapshot, we will clean up that which was created due to dump process, and follow the same flow.

** Since both A & B cluster are supposed to be moving, rDiff will always be used to restore back to earlier synced state.

## Scenarios & Corner Cases.

| Scenario | On-Prem/Cloud | Solution/Description |
|---|---|---|
| Modification of only 1 partition | On-Prem | Drop table without deleting the data, So only modified partition files get copied., but complete metadata |

| | | |
|---|---|---|
| | Cloud | Data is copied to staging location, so entire data copy. |
| Compaction & Cleaner | On-Prem | No impact on metadata copy. Would affect only few newly created partitions on data copy front. Only modified files will get copied |
| | Cloud | Entire table will get copied, due to use of staging directory and not directly copying to target location |
| Source has a non loaded dump | Both | Discard the non-loaded dump, the event id is picked from the database B, which is the last loaded event id. |
| TTL Expired | Both | Unrecoverable Error Case. |
| A failed and B took over. Coming back from B to A, B fails and A takes over. | Both | Chances of loss of of 'one' table.<br>In case the table was supposed to be overwritten and the cluster went off after we dropped the table and before we could load it. |
| Notification Log not enabled on B | Both | Do a normal drop & bootstrap |
| Planned Failover | Both | The table_diff file will be |

| | | |
|---|---|---|
| | | empty, it will do incremental load for all the tables. |
| A table created at A after the dump, which after DR and B takes over isn't created again on B.. | Both | The newly created table which isn't part of target, will get dropped on source cluster**** Load |

## Additional Codes:(Dev-Only)

- Prevent deletion of Notification Events on A post restart of HS2 for a considerable amount of time
- Use distcp atomic
- FileUtils handle small files update
- Take care of **repl.backgroundthread** and failover files clean up in bootstrap code
- Take care of Snapshots for external tables, refresh only if last copy was success. For failure cases, keep a track of directories copied, and verify from that, if it can be refreshed.
- Handle checkpointing with respect to tables being bootstraped.
- Maintain the last loaded event id on B, with respect to B as well.
- ***In Memory Partition***
- **Snapshot: ***

## Code-Time Checks(Dev-Only)

- Expose the failover specific dump/load and  operations in metrics. Metrics to have number of tables being bootstraped and the state of failover like created load_ack, table_diff created, first_failover_dump, first_failover_load.
- Status in Metrics:

EVENT_ACK: After generating the event_ack file
TABLE_DIFF_ACK: After generating the table_diff_ack file
OPTIMISED_DUMP: After the first dump on B
OPTIMISED_LOAD: After the first load on A.

- Clean-up of table_diff and event_ack file in case someone accidentally triggered dump or load on the wrong sites.