Hive Replication: ACID Tables

Non Acid hive replication is already implemented and as part of this document we are going to look at design approaches for replication for ACID tables. More information on ACID in hive can be found here.
ACID components that will affect replication are:

- Acid tables
- Compaction subsystem in ACID (contains three components Initiator / worker / Cleaner)
- Streaming Ingest Tables

Since currently support for replicating database is already present, on addition of support for ACID Tables, the replication system should be able to handle presence of ACID/non-ACID tables in a single database.

Use Cases

Backup for disaster recovery with failback

1. The target cluster may be equal to source cluster and able to run compaction for example
2. The target cluster may be mostly storage and very little compute and not have the resources to run full compaction
    a. Running compaction on source also speeds up failback since running compaction on target is not likely to produce base_N and delta_M_N with the same ID ranges as on source and so they will become incomparable with files in source on failback and have to be sent back. In a way this violates the 1 writer rule.
3. Failback should be as quick as possible.

Offloading primary cluster for scalability

1. The intent is 1 cluster for writes several for reads
2. Different secondary clusters may have different functions like ETL vs BI.

In both cases it's valuable to maintain the same transactional consistency on target side as on source side.

Key Assumptions

1. Only replicating ACID table to structurally identical ACID table.

2. A single cluster can write to a table. The table may be written by a cluster that contains the table or be written by replication (from 1 other cluster) but not both.
3. Target cluster may have tables written to by different source clusters.
4. Single table maybe be replicated to multiple targets.
5. When choosing target table for replication, it must be empty.
6. Source and target clusters must be on identical hive version. (For simplicity for now)
    6.1. For example, different versions of Acid may produce different data layouts/formats on disk so which may be a problem for file level replication.
7. This document ignores the fact that there exist non-acid tables which may need to be replicated.

Replication Model

Currently replication is **event based** i.e. each operation on hive, that changes the state of the database, either DDL / DML statements, generate events(capturing necessary information required for replication) which are captured using event listeners in the hive metastore rdbms. Replication subsystem will transfer these events to target warehouse and replay them there while talking to source warehouse to copy relevant data files for tables/partitions etc.

The goal is to see if we can tweak the same model for ACID tables as well. This will require us to add a additional concept of ***per table Write-Id*** which will now be associated with the current global transaction id used in ACID.

Replication Failback with ACID

A naive approach to Failback is to copy everything from Secondary cluster back to Primary and overwrite everything in Primary. This is easy but very expensive in time/network usage. To improve this we need an efficient way to find the minimal amount of data to transfer back on failover. Additionally, the primary cluster may create additional data after failover because it's in bad state or simply have data was never replicated to Secondary because of HW or Network issues. Additional files that got added in original Primary cluster after last replication would need to be cleaned up.

We can reduce the copying needed if we can determine which files need to be copied over by comparing the names and checksums. This is not possible in the current Acid architecture which imbeds global transaction IDs in the data and file names because each clusters transaction ID sequence moves independently. This can be solved (up to a point) by introducing a per table ID sequence to use in Acid ROW_IDs and file names instead of the global transaction ID. The following explains this in detail.

Per Table Sequences (Write-Id)

Having a per table sequence (Write-id) has other advantages other than efficient fail back since it allows tracking of transactions at the table level. For example, a open global transaction currently ends up blocking compaction across that open txnid boundary. With write-id, it is now possible to continue compaction for tables which don't have an open write-id corresponding to it. Similarly, it sets the stage to allow for future optimizations to avoid having global transaction id operations as a hotspot.

Currently every row in an Acid table includes a ROW_ID which consists of <transactionID, bucketProperty, rowid> triple. The transactionID is the global transaction ID that created the row. This ROW_ID is unique in a partition. Additionally all the data is laid out in directories such as base_N or delta_M_N where M and N are also transaction IDs.
We'll create a new per table Write ID sequence to be managed by the Transaction Manager to replace the Transaction ID in the ROW_ID and file names. The global transaction ID sequence remains in place as before. Write ID is a monotonically increasing sequence.

Anatomy of a Write

For example, *Insert into T select a,b from C* where T & C are unpartitioned. It proceeds as follows:

1. Open a transaction which allocates a transaction ID from a global sequence.
2. Compile the Query plan to identify all FileSinkDesc objects, in this case just 1 for T
3. For each distinct table allocate a Write ID and set it on FileSinkDesc. This replaces TransactionID on FileSinkDesc. All downstream processing write path remains exactly the same.
4. Allocation of Write ID is done by the Transaction Manager which records it durably and associates it with the global Transaction ID before returning to the client. Each txn can be associated with only one write ID per table throughout its lifetime. Below tables helps to manage this association.
    a. TXN_TO_WRITE_ID(db_name, table_name, txn_id, write_id) - ties TXN_ID with WRITE_ID.
    b. NEXT_WRITE_ID(db_name, table_name, next_write_id) - similar to NEXT_TXN_ID
5. Transaction commits and so all Write IDs are marked committed via their association with the global Transaction ID.

If several tables are written (e.g. multi-insert statement) each table gets its own Write ID from the per table sequence. All partitions of the same table written from a single statement use the same WriteID.

Anatomy of a Read

For example, *Select S.a, T.b from S inner join T on …*:

1. Open a transaction which allocates a transaction ID from a global sequence.
2. Get ValidTxnList from the metastore. This is exactly the ValidTxnList object that we have today and locks in the snapshot for this transaction expressed in terms of global transaction IDs.
3. Compile the query to identify all table scans.
4. For each Acid table scan, get ValidWriteIDList from the Transaction Manager. This object has exactly the same structure as the current ValidTxnList except that it contains Write IDs and is built with respect to ValidTxnList of the reading transaction.
    4.1. In practice I think we can use ValidTxnList exactly as is here to minimize code changes.
5. For each table scan in the Query plan, ValidWriteIDList is encoded in the Configuration just like ValidTxnList is currently.
6. Each Acid reader uses ValidWriteIDList for the appropriate table to filter files/rows exactly as it currently use ValidTxnList.

Since ValidWriteID is obtained based on a Query plan we only need it for the Acid tables being read and will not bloat the Configuration. [We couldn't easily do the same with per partition WriteID sequences since a query may read 1M partitions and OOM.]

ValidTxnList should be stored on the TransactionManager for the duration of the transaction (in Snapshot Isolation) so that it's "locked in". ValidWriteIDList should be completely determined by ValidTxnList and current metastore information to ensure that it represents a stable snapshot of the table as concurrent transactions commit. [Alternative would've been to record ValidWriteIDLists for all tables in the system at the start of the transaction - prohibitive memory requirements.] This can be achieved as follows:

1. TXN_TO_WRITE_ID table keeps a mapping of Transaction ID to Write ID. The state of each Write ID (open, committed, aborted) is determined by the state of the parent transaction. In order to be able to get a ValidWriteIdList that is accurate wrt ValidTxnList that is locked in at the start of the transaction, we have to retain txnid<->writeid mapping even after the transaction ends. This is because a reader at Snapshot Isolation that started when transaction X was open, should continue to ignore the data written by X even after X commits.
2. Logic to build ValidWriteIdList for the given ValidTxnList,
    a. Find the writeId high-water mark (writeIdHwm) from TXN_TO_WRITE_ID using txn_id HWM (txnHwm) from ValidTxnList. It is possible that txn_id < txnHwm can have write_id(txn_id) > write_id(txnHwm). So, need to obtain writeIdHwm=max(write_id) from all txns under txnHwm. If not found from TXN_TO_WRITE_ID (possibly due to regular cleanup based on MIN_HISTORY_LEVEL or no writes on this table yet), then

(NEXT_WRITE_ID.nwi_next - 1) would be the writeIdHwm as it is the highest allocated write_id on this table.

b. Select all write_ids from TXN_TO_WRITE_ID for the relevant table which are under writeIdHwm. The status (aborted, committed, open, etc) of each write_id should be set to that of the corresponding txnid as recorded in the ValidTxnList.
c. It is possible that txn_id > txnHwm allocates write_id < writeIdHwm. As txn_id > txnHwm is marked as invalid in ValidTxnList, the corresponding write_id will also be marked as invalid in ValidWriteIdList.
d. Also, if any of the write_id < writeIdHwm maps to an aborted or open txn in the ValidTxnList, then those will also be marked as invalid in ValidWriteIdList.
e. So given a ValidTxnList of the active transaction we can always get a stable ValidWriteIDList

Cleanup logic for TXN_TO_WRITE_ID

As TXN_TO_WRITE_ID is populated each time a new write id is allocated for the table and will remain there until we drop the table/database. But, it is redundant to have all the entries if txns are committed/aborted and the relevant data are compacted. Also, too many entries will increase overhead to determine ValidWriteIDList. So, it is needed to clean this meta-table periodically.

1. Introduce MIN_HISTORY_LEVEL(current_txn_id, min_open_txn_id) meta-table - need this to ensure that we don't remove TXN_TO_WRITE_ID data before any operation that started while this txn was open/aborted is still live.
2. When txn X is opened, it records Y=select min(txn_id) from TXNS where txn_state='o' into MIN_HISTORY(txnid,opentxnid) table, i.e. it adds (X, Y) to MIN_HISTORY_LEVEL.
3. On commit (and abort) of X, it removes its own entry from MIN_HISTORY_LEVEL.
4. In the absence of Aborted transactions, MIN_HISTORY_LEVEL gives us the smallest open txnid across all active reader snapshots. Let Z=select min(opentxnid) from MIN_HISTORY. We can delete entries from TXN_TO_WRITE_ID for TXN_TO_WRITE_ID.T2W_TXNID < Z since every active reader sees txns < Z as committed.
5. If A is aborted txns, we retain the metadata about it in TXNS as long as any data written S may be visible to some reader in the system so that the reader knows to skip this data. The rules for when that is are complex but wrt to TXN_TO_WRITE_ID, if A=select min(TXN_ID) from TXNS where TXN_STATE='a', then it's safe to delete from TXN_TO_WRITE_ID when TXN_TO_WRITE_ID.T2W_TXNID < min(Z,A).
6. If no open or aborted txns exist in the system, then we need to enable cleanup using latest allocated value of NEXT_TXN_ID table. Delete condition would be TXN_TO_WRITE_ID.T2W_TXNID < min(Z,A,NEXT_TXN_ID.ntxn_next).
7. Also, it is proposed to trigger cleanup of TXN_TO_WRITE_ID from initiator immediately after cleaning up aborted txns metadata from TXNS table.

Event Based Replication Model

A replication model for ACID tables based on the availability of Write-ID as discussed above.

- Events are generated for both DDL and Write (DML) operations on the entities such as Database, Table, Partition or Constraints.
- Generally each events can have 2 parts, such as, metadata and data.
    - Metadata is the serialized object of the operated entity itself.
    - Data is the list of data files associated with it. We just capture the data files names here.
    - Events such as ALTER, TRUNCATE, RENAME etc are metadata-only as it won't need data files information to replicate those events.
- Generated events are persisted in "MNotificationLog" table in metastore RDBMS.
- Point-in-time replication is achieved by replicating the events in the same sequence as it is generated. The Event ID generated using table "MNotificationNextId" helps to sequence the event.
- If replication (event generation) is enabled on a non-empty source warehouse, it is possible that there won't be any events mapping to their existing metadata/data. So, need to follow the below method to synchronize target with source.
    - Initialize the target by bootstrapping the entire database (both metadata/data) from source to target.
    - Trigger the catch-up incremental replication immediately after bootstrap to reach a consistent state at target
    - Only after bootstrap+catch-up incremental (first incremental) replication, the target would be ready to use.
    - Further sync-up can be achieved with incremental replication at regular intervals.

Reusability of existing events for ACID tables

Event Type	**Content**	**Remarks**
EVENT_CREATE_TABLE	- Table Object. - List of data files added in case of non-partitioned table.	
EVENT_ADD_PARTITION	- Partition object. - List of data files added to the partition.	

EVENT_CREATE_FUNCTION	- Function object. - List of jars associated with UDF or UDAF..	
EVENT_DROP_TABLE	- Table object.	Metadata-Only
EVENT_DROP_PARTITION	- Partition object.	Metadata-Only
EVENT_DROP_FUNCTION	- Function object.	Metadata-Only
EVENT_ALTER_TABLE EVENT_RENAME_TABLE EVENT_TRUNCATE_TABLE	- Before Table object. - After Table object.	Metadata-Only
EVENT_ALTER_PARTITION EVENT_RENAME_PARTITION EVENT_TRUNCATE_PARTITION	- Before Partition object. - After Partition object.	Metadata-Only
EVENT_INSERT	- Table/Partition object - List of data files added/overwritten.	Same event reused for 2 types of inserts. INSERT INTO, INSERT OVERWRITE and LOAD DATA. Use replace flag to decide if any need to overwrite the data or just append.
EVENT_ADD_PRIMARYKEY	- SQLPrimaryKey objects array	Metadata-Only
EVENT_ADD_FOREIGNKEY	- SQLForeignKey objects array.	Metadata-Only
EVENT_ADD_UNIQUECONSTRAINT	- SQLUniqueConstraint objects array.	Metadata-Only
EVENT_ADD_NOTNULLCONSTRAINT	- SQLNotNullConstraint objects array.	Metadata-Only
EVENT_DROP_CONSTRAINT	- Constraint Name.	Metadata-Only

1. All the existing DDL events (except EVENT_INSERT) are compatible with ACID Tables as they are independent of ACID txns.
2. From replication perspective, CTAS (Create Table As Select) operation shall be treated as 2 operations which is Create Table and Insert rows. For non-ACID tables, only EVENT_CREATE_TABLE is generated including data files.But for ACID tables, EVENT_CREATE_TABLE will contain only metadata changes but the insert rows should

generate the set of events for open txn, allocate write ID, write and commit txn events which is detailed out in below sections.

3. Similarly, for dynamic partition creation, currently, only EVENT_ADD_PARTITION is generated that includes the data which are inserted. But, for ACID tables, need to separate this into 2 operations that is Add partition and Insert rows.
4. TRUNCATE operation is implemented as INSERT OVERWRITE with 0 rows which generates an empty base file to retain the snapshot isolation semantics. So, EVENT_TRUNCATE_TABLE and EVENT_TRUNCATE_PARTITION events won't be used in case of ACID tables.
5. INSERT operation is performed within a transaction and hence EVENT_INSERT cannot be used for ACID tables. New set of events are added to handle ACID table writes which is discussed below.

New Events for ACID Tables Replication

Event Type	Content	Remarks
EVENT_WRITE	- Transaction ID. - DB Name. - Table Name. - Table Write ID. - Table/Partition - List of files and their checksum that are added as part of this operation	Synonymous to EVENT_INSERT but this type of events are persisted into another metastore table (Say, MTxnWriteNotificationLog) that maps these events to a txn ID. It is sufficient to capture the latest state of the table/partition object within the txn. So, if there are multiple writes into same table/partition within one txn, then just overwrite the table/partition object in the existing event instead of creating new one.
EVENT_OPEN_TXN	- Transaction ID.	Metadata-Only Event generated for both Read/Write txns. This type of event is required to track the sequence of open txns to avoid violation of snapshot isolation when 2 txns commits in different sequence wrt open txn.
EVENT_ALLOCATE_TABLE_WRITE_ID	- Transaction ID. - DB Name - Table Name	Metadata-Only Event generated only for Write txn when allocate a table write

	- Table Write ID	ID before writing into a table/partition.
EVENT_ABORT_TXN	- Transaction ID.	Metadata-Only Event generated for both Read/Write txns.

EVENT_COMMIT_TXN	- Transaction ID.
	- List of table and partition write events.
	  - Table/Partition object
	  - Table Write ID
	  - List of base/delta files added (both insert and delete delta files) within current table/partition.
		Event generated for both Read/Write txns.
		For read txns, this event will be a metadata only event which just have txn ID same as EVENT_ABORT_TXN.
		To generalize, this event contains a bundle of EVENT_WRITE type events that maps to given txn. All these events are obtained from "MTxnWriteNotificationLog" table.
		The list of base/delta files corresponding to a table write ID (from EVENT_WRITE) will be computed during commit txn [Base files would've been created in case of INSERT OVERWRITE or TRUNCATE table scenarios].

EVENT_COMPACT_TABLE	- Table object after compaction. - List of data files (both base/delta) along with their checksums as created after compaction.	This event is generated only for compaction on non-partitioned table. Same event used for both minor and major compaction.
EVENT_COMPACT_PARTITION	- Partition object after compaction. - List of data files (both base/delta) along with their checksums as created within the partition after compaction.	This event is generated only for compaction on partitioned table. Same event used for both minor and major compaction.

Incremental replication

Steps to apply all these new events on target warehouse

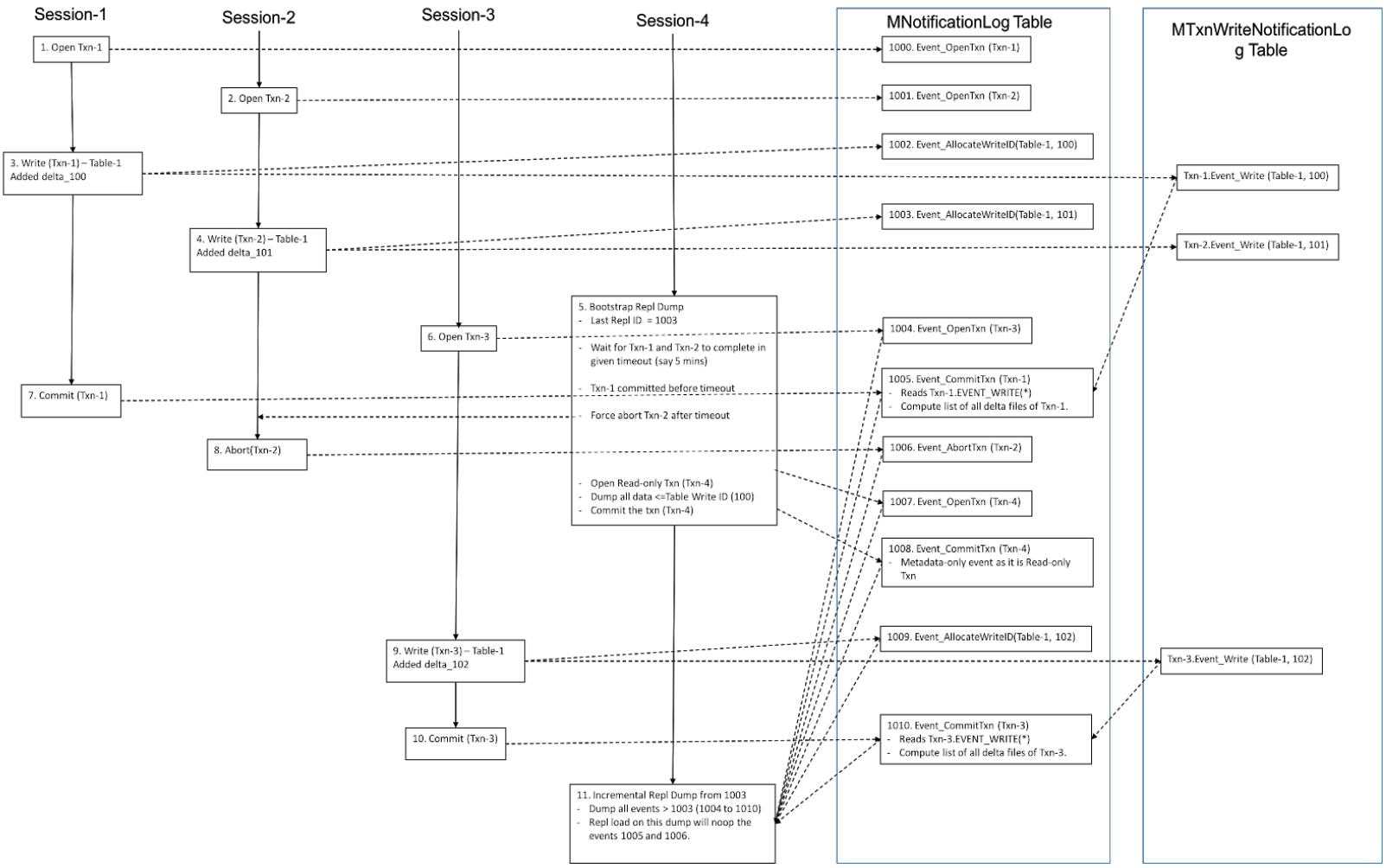
1. EVENT_OPEN_TXN
   1.1. Check if any open txn exist in target that maps to given source txn ID. If, yes, then ignore this event.
   1.2. Opens a txn in target. Txn ID generated need not be matching with source.
   1.3. Add an entry into the Map for source to target txn IDs and persist it in a new metastore table. This is needed to identify the txn ID when Abort or Commit event comes. [Need to maintain different map for different source. This can be achieved by having one map for each replication policy(*dbname.tablename*, where *tablename* will be null for DB level replication). Replication policy can be identified from the dump of REPL LOAD command line arguments]
2. EVENT_ALLOCATE_TABLE_WRITE_ID
   2.1. Identifies the corresponding txn at target using source txn ID. If not found in the map, then just ignore the event.
   2.2. If the current Table Write ID is already ahead of incoming Table Write ID, then just ignore the event and return.
   2.3. Else, Allocate the Table Write ID in the target and associate this Table Write ID with the target txn ID.
3. EVENT_ABORT_TXN
   3.1. Identifies the corresponding txn at target using source txn ID. If not found in the map, then just ignore the event
   3.2. If the target txn is not in open state, then just ignore this event as it might be already aborted and return.
   3.3. Abort the txn in target.
   3.4. Remove the corresponding entry from txn ID map.
4. EVENT_COMMIT_TXN
   4.1. Identifies the corresponding txn at target using source txn ID. If not found in the map, then just ignore the event as it was already applied.
   4.2. If the target txn is not in open-state, then just ignore it as it was already applied.
   4.3. Atomically copy all delta files and also overwrite the metadata objects.
   4.4. Commit the txn at target.
   4.5. Remove the corresponding entry from txn ID map.
5. EVENT_COMPACT_TABLE / EVENT_COMPACT_PARTITION
   5.1. If table/partition to be compacted doesn't exist, then just ignore the event.
   5.2. Using the list of compacted files from source, build the metadata (write ID range) needed to generate same set of files. Add this compaction task (as per metadata) to the compactor queue in target.

   A detail section gives move details on how compaction is handled.

Key design points

- We copy the data only as part of EVENT_COMMIT_TXN event to avoid copying data files related to aborted txns. Also, this ensures better throughput for distcp by enabling parallel copy of data files.
- EVENT_WRITE events are never propagated to target as it will be bundled along with EVENT_COMMIT_TXN.
- Expired EVENT_WRITE events will be cleaned from “MTxnWriteNotificationLog” table using the cleaner thread that cleans the expired events from “MNotificationLog” table.
- EVENT_WRITE cannot be reused to replicate the allocation of write ID because, for writes, the sequence of operations are as follows,
- Allocate Write ID -> Perform Write -> If success, then log EVENT_WRITE.
- If write operation fails for some reason, then allocated write ID will never gets replicated. That's why we need to have separate event EVENT_ALLOCATE_TABLE_WRITE_ID to notify allocation of write ID irrespective of whether the write is success or failure. Also, this helps to minimize the number of events propagated to target.
- All the new events shall provide idempotent behavior same as existing events.
    - If the corresponding entity (DB/Table/Partition) have the repl_state (last applied event ID) which is later or equal to current event ID, then ignore the event as it was already applied.
    - If the data files to be copied already exist, then just overwrite it.
    - If the entity to be operated is missing in the target, then just ignore the event.
- DDL operations take exclusive lock on the entity operated and hence assumed it won’t affect an on-going txn or it may abort the txn. For instance, DROP TABLE is not allowed when there is an ongoing txn on this table. This means, DDL events captured in-between any txn won’t have any side effects while applying the txn events.
- The transactions opened by the replication tasks are supposed to be marked special and shouldn’t timeout due to missing heart-beat. So, ACID semantics have to provide additional option to set this status when open a transaction.
- As compaction is triggered at target explicitly and also replicated tables/partitions have only one writer, it is safe to allow compaction even if any txn is open on target. So, need changes in locking criteria for compaction method in target.

Bootstrap+Catch-up Incremental



Bootstrapping

- Perform **bootstrap dump within a read txn** which helps to get a snapshot of database/tables/partitions. This doesn't have to take any locks on the replicated tables but read txn will ensure a consistent state is replicated by bootstrap. For each table in the database,
    - Dump the metadata object (Table/Partitions).
    - Dump the list of all the valid data files (base+delta+delete_delta) for this txn using the same logic used for any read txn. [This step should exclude the data files of opened txns and also all the files waiting for cleanup such as aborted files, old files before compaction, old files before insert overwrite/truncate etc]
        - If this list gets a data file being written by streaming ingest, then the same data file shall have both aborted and committed data and also the file is

not even be closed yet. So, it is impossible to filter it out unless we rewrite the files.
        - To handle this case, we expect, all on-going txns which are started before triggering bootstrap should be force aborted after a configured timeout. This will abort even the txns batch which is opened by a streaming ingest connection. This approach is discussed in detail in next section.
        - Once we force abort all open txns, it is guaranteed that resultant list of data files will have only committed and aborted data (only in case of streaming ingest). So, it is enough to recreate the state of aborted txns/writeids in target during REPL LOAD.
    - Dump the ValidWriteIDList to transfer the txn/writeid state across clusters.
- **Handling On-going Txns:** When replication is enabled on the source for the first time with several ongoing txns, it is possible that some of the **events** [such as open_txn, write, allocate_write_id] **are not generated/captured**. This leads to **data loss** as we won't be able to replicate those changes during catch-up phase. This may also cause data non-sync if a data file being written by streaming ingest is copied which has both aborted and committed data. The proposed approach for this case is as follows.

  Finalised Approach: [No need to find if any txn exist without open_txn event. Also, works well for the combination of ACID and non-ACID cases as well.]
    1. Get the **last replicated state** (**event ID**) as the current latest event ID from sequence table (same as current implementation).
    2. Get the list of all open txns when we start bootstrap dump.
    3. Shall wait for configured time (New configuration: **hive.repl.bootstrap.dump.open.txn.timeout)** to see if all these txns are either committed/aborted.
    4. If the txn is not committed/aborted within this timeout period, then just trigger force abort from REPL DUMP session. As REPL DUMP is triggered only by "admin" user, it is safe to trigger force abort the oldest txn.
    5. During step-2 to step-4, if any txns are newly opened, then it won't be included in this timeout scope. Those txns will be replicated as part of catch-up incremental phase.
    6. Perform regular bootstrap within read txn. The last repl ID is obtained from step-1.

  Alternate Approach: [Extra complexity to validate if open_txn events captured for all ongoing txns]
    - Bootstrap Repl dump need to **validate if any txn exist without corresponding open txn event**. This can be found by searching EVENT_OPEN_TXN in the MNotificationLog Table with the Txn ID of oldest on-going txn.
    - If open event is not captured, then, wait for configured time for the ongoing txns to commit. Need a new configuration parameter **"hive.repl.bootstrap.dump.open.txn.timeout"** for the same.

    - If the txn is not committed/aborted within this timeout period, then just trigger force abort from REPL DUMP session. As REPL DUMP is triggered only by “admin” user, it is safe to trigger force abort the oldest txn.
    - Unlike non-ACID tables *[The current event ID is the last replicated state as we just dump the current data set]*, the **last replicated state** (**event ID**) of database should be the event ID of captured EVENT_OPEN_TXN of oldest on-going txn.
- Commit the txn once bootstrap dump is completed.
- It is responsibility of **catch-up incremental** (first incremental) replication to replicate the **current open txns** and their corresponding data.
- The existing limitation of disallowing Rename table/partition operations at source when bootstrap dump is in progress will be valid for ACID tables too.
- Bootstrap load at target
    - Gets a consistent snapshot of the database.
    - Reads the ValidWriteIDList per table and recreate the transaction/writeid state as follows. If any writeid is marked as aborted, then
        - Open a new txn
        - Add an entry to TXN_TO_WRITE_ID map where map the new txn against the writeid which is aborted.
        - Abort the txn.
    - Make sure the data files are copied only after transferring the state.
    - Target database will be ready to use immediately after bootstrap load.
    - Change management should be in place at source to ensure files are available during bootstrap load.

Catch-up Incremental

- After bootstrapping, the ACID tables will be ready to use as we replicated a stable snapshot with a read txn at source.
- The significance of catch-up incremental is to assure idempotent behavior for all the txn level events such as open_txn, allocate_write_id, commit_txn and abort_txn. This means,
    - Commit/Abort a txn that doesn’t exist in the source-target txn map should be noop.
    - Commit/Abort a txn that exists but doesn’t do any writes. Just commit/abort the txn without any metadata/data copy tasks.
    - Commit a txn that exists but the metadata/data was already copied by applying the same event. This can be found by validating the table write ID of the tables being written and see if this is already in the commited txns list.
- All the existing logic for DDL events are applicable for ACID tables as well. For instance, if a Drop table event is applied on target where the table is not found, then the drop event will be ignored.

Non-ACID to ACID conversion

Non-ACID to Full ACID conversion

1. This just involve change in Table properties (transactional=true) which will be replicated using EVENT_ALTER_TABLE message.
2. All the existing non-ACID data files are treated as original files (readable by all transactions) and virtually mapped to writeid=0. Also, it is retained in the same location.
3. All the future writes would take usual ACID write flow where writeid is allocated and write the data into delta directory named as “delta_minwriteid_maxwriteid_stmtid”.
4. If this table/partition is major-compacted, then all these original files would be written to a base directory. This makes the original files obsolete and future readers will read from base files itself. Cleaner would remove these obsolete original files.
5. No additional handling needed to replicate this conversion.
    a. EVENT_ALTER_TABLE message would set the Table properties (transactional=true).
    b. Future writes would have EVENT_ALLOCATE_WRITE_ID and EVENT_WRITE events to replicate the writes into new deltas.
    c. Compaction is auto-triggered at target which would compact the original files and move it to base directory.

Non-ACID to MM (Insert-Only) conversion

1. This case involves two changes
    a. Table properties (transactional=true and transactional_props=”insert_only”).
    b. Allocate a writeid and move all the existing non-acid data files to the delta directory created using the writeid allocated.
2. Additional handling needed to replicate this conversion.
    a. EVENT_ALTER_TABLE message would set the Table properties (transactional=true and transactional_props=”insert_only”). Applying this event at target should take additional input from ReplicationSpec for what is the writeid allocated and the associated txnid at source. Now, need to allocate writeid same as source (which is mostly 1) and move the non-acid data files to delta directory.
    b. EVENT_ALLOCATE_WRITE_ID message would replicate allocation of writeid. But this should be no-op as ALTER event itself would’ve already allocated the same writeid for the same txnid. So, just ignore it.
    c. As non-acid data files are moved to delta directory, it is required to **archive/copy** these files to CM root.

Replicating Streaming Ingest Tables

Streaming in Hive : Client opens a transaction batch with *n* number of transactions either per table(no partitions) or per partition. Starts writing transactions one after another in sequential fashion within a transaction batch. Assuming there are 4 buckets in the partition/table, then a single file per bucket (*4 files in total : this will depend on what execution engine is being used, for MR it will be 4 for tez it will only create files in buckets where the data for the transactions is being written to*) will hold all data associated with the *n* transactions. All the file will be open till the transaction batch commits.

Given : we have write id associated per table in defining the primary key for a given data row.

Sample Use Case to solve, for events timeline as follows :

1. Source transaction batch of 10 transactions with Txn numbers (1,2,3…..10)
2. Source transaction committed after writing data for Txn Number 1,2,3.
3. *txn-> write id -> a unique column* per row,for the table
    3.1. 1 -> 19 -> 100,101,102
    3.2. 2-> 20 -> 103,104,105,106
    3.3. 3 -> 21 -> 107
4. Concurrent Transaction 11 for the same table updates row number 101 and commits.
5. Other transactions in the batch commit 4,5…..10.

Any read transaction opened above concurrently with the above timeline (specifically a read between step 4 and 5 above), should have same data visibility on both source and target warehouse. For streaming ingest on source this is achieved by having two sets of files the **data file** and **side file.**

- **Data file:** this file contains the actual data along with any metadata as required the respective file formats, for ex ORC footer.
- **Side file:** this file contains a number representing an offset in the **Data file** having the last valid footer. As transactions part of a transaction batch complete they append new offset values to this file.

There are two possible approaches to solving this as below. Either of the approaches will atleast require the following changes to EVENT_COMMIT_TXN

1. Add an additional attribute in the transaction commit event(EVENT_COMMIT_TXN) to denote
    1.1. if it is part of a streaming transaction batch: the identifier used to represent a streaming batch this will help us in book-keeping on the target.
    1.2. If the batch is complete : this is required to remove the side file that is generated as part of streaming ingest to denote the last valid ORC footer written to the data file.

Approach 1

1. When we receive the EVENT_COMMIT_TXN on target as part of copying files for this event (the below set of steps are very specific for events stating that they belong to a transaction batch), check for the file checksum on the source and target warehouse, if its different only then initiate a copy from the source to target warehouse.
    1.1. This is required to prevent copy of the same file for multiple transactions belonging to the same batch.
    1.2. Replication is lagging behind the source, in which case, when on souce a transaction batch is complete and then on target we receive the first txn commit event belonging to the above transaction batch, then we can get to the final state of the files in the first copy itself, additional transaction commit events belonging to the transaction batch will only be metadata events.
2. Copying of files as described in point 2 will have a few specific semantics as below w.r.t to both the data and side file:
    2.1. Copy the source file (data/side) to staging directory of the target warehouse
    2.2. If the file is already present, then append the remaining data by reading selective portion of the file copied in step a. and append to the actual file location in target warehouse
        2.2.1. We do the append of the differential data rather than replacing the complete file because when we are trying to do this operation if there are transactions open on target which are reading this file (since it contains data from multiple transactions in the same file) then we dont want to corrupt their reads as part of doing a replace.
    2.3. Since there are two files associated with a txn commit, a **data file** and a **side file** we have to follow a sequence of steps to copy relevant data as below
        2.3.1. Copy **side file** to staging directory on target from source.
        2.3.2. Copy **data file** to staging directory on target from source.
        2.3.3. Append the differential data to from **data file** in staging directory on target to actual location of data file (if no existing data file exists make sure to create an empty data file and proceed).
        2.3.4. Append the differential data to from **side file** in staging directory on target to actual location of side file (if no existing side file exists make sure to create an empty side file and proceed).
            2.3.4.1. We follow the above sequence of steps so as to make sure that when replication is happening in near real time then replication on target will read the source warehouse when an active transaction belonging to the transaction batch is modifying the **data** and **side file** on source warehouse. Since on source the **data file** is written *followed by* **side file** we want to make sure we copy in the reverse order for replication events on target, such that we dont copy data file with less data than what the side file points to.

Cons:

- Debugging might be difficult as the state of files being copied, will have their state associated with time which might make it difficult to reason about during debugging which is generally at a later point of time.

Approach 2

1. As part of the transaction commit event(EVENT_COMMIT_TXN) capture the *start* and *end offset* in the **data** and **side file** corresponding to this transaction.
2. On receiving this event on target read the relevant portion of the source warehouse **data** and **side file** and append it to the target **data** and **side file.**
3. The sequence in which the files have to be updated should be same as described in ***Approach 1*** above.

Cons:

- Easier to reason about during debugging as we would precisely know as part of each event replication what was looked at and copied over.
- Since we are copying only the differential data network overhead will be less if replication is near real time.

Transactions in a given transaction batch can either be committed or aborted. For aborted transaction events we do not capture any additional metadata and any gaps in the data file as a result of an aborted transaction between two committed transaction on the target side will be filled with garbage data.

Replication For Compaction Subsystem

Hive Compaction: There is minor and major compaction. Minor compaction being triggered when a configured number of delta files for a table is reached. Major compaction is triggered on various possibilities, like when the data in the delta files reaches a configured threshold size of the base file for the table/partition, or when the number of aborted transactions in delta files exceeds a value. Compaction subsystem has three sub components Initiator / worker(compactor) / Cleaner. Each component generates work the next component such that the workflow is from Initiator => worker (compactor) => Cleaner. The work is transferred between components using a db table (which acts a queue).

Compaction becomes highly relevant for replication when we have to do failback. **During failback we will do a bootstrap again from target to source cluster. As part of bootstrap we can implement an optimization which will compare the files on target and source directories and only copy changed files from target to source.**

There are various solutions to handle failback along with compaction as discussed:

- Any of the approaches below, running parts of the compaction sub system, will run them as its run on primary warehouse, i.e. asynchronously
    - This is primarily important because on the source warehouse compaction is a background operation and hence we dont want to make if a foreground operation on target warehouse as it will block replication of other events.
    -
- Intent is to make sure compaction on source and target produce files which as similar as possible

Default State With no additional Changes for Compaction

In this state all the components of the compaction subsystem will run both on source and target independently. In the best possible case they run at the same time and on similar start states such that they generate a **similar** set of files after compaction on either side even though they run independently. This best case run will make failback only a metadata operation since all the compacted file sets are exactly similar, or with very few differences.

Approach 1: Change compactor

This approach will actively try to make changes to the compaction subsystem such that the same set of files are generated on the target and source warehouse. This will eliminate the need for replication subsystem to convey any events from source to target warehouse relating to compaction.
This will require, that we remove any time dependencies on what gets compacted as part of compaction. What this means is there is a deterministic state transition from before to after irrespective of when compaction runs. The state here refers to, the set of files used for compaction in a partitions / table to get a result file.

Ex: assuming all other changes on the warehouse are same in both the cases below:

Non Deterministic changes:
State 1 -------> state 20 if compactor runs at time t1
State 1 -------> state 22 if compactor runs at time t2
State 1 -------> state 28 if compactor runs at time t3

Deterministic changes:
State 1 -------> state 20 if compactor runs at time t1
State 1 -------> state 20 if compactor runs at time t2
State 1 -------> state 20 if compactor runs at time t3

Then we should be able to run compactions independently on both source and target. Replication does not need to handle compaction separately and it becomes a property exclusive

to ACID.We can run all compaction related jobs initiator / compactor(worker) / cleaner on both source / target warehouse with same code paths. We only have to replicate events for tables / partitions.

On failback if we wait for compaction queue on target to complete before initiating the failback then, for a case where there are no writes to target warehouse on failover, followed by failback the optimized bootstrap will be a purely metadata only operation.

Approach 2: Copy compacted files from source

1. The compaction subsystem (initiator / compactor) on target warehouse will not run for replicated tables (*we have to define how do we implement this as on failover these tables will become primary*), for tables which are not target to any replication they will run as usual.
2. On the source warehouse a event will be generated once the worker(compactor) runs which will include the list of files produced as a part of that run + the checksum for each of these files.
3. Replication of these events on target leads to copy of compacted data files from source warehouse to target warehouse.
4. Since we are directly copying the relevant files from source in step 3, we will have to also generate the corresponding work object (for the replicated table/partition) for the cleaner subsystem of compaction running on target.
    4.1. We should be able to identify the files to be cleaned based on the name of the file we are copying from source. These file names should provide an information of what write ids are part of that file and we have to then do a reverse lookup of any delta files associated with those transactions on the target to create the cleaner work object.
    4.2. The cleaner work object representation might change slightly than what is currently implemented to include the details as generated above.

There are some additional steps required to be taken on failback and failover with the above approach:

- On failover we have to make sure that we enable the initiator / compactor(worker) on target warehouse w.r.t replicated tables.
- On failback we have to make sure that we disable the initiator / compactor(worker) on target warehouse w.r.t to replicated tables.

Cons:

- Non time series based tables having data being added to them will potentially lead to full copy data most of which is already present on target warehouse. High bandwidth usage in worst case scenario where there are lot up updates. Additionally PI data according to EU laws might require cleanup for even time series based data warehouse model.

- In worst case if replication is lagging behind significantly and the compaction sub system runs frequently enough to generate a number of base files then, change management will need to archive these base files and might lead to large disk usage.

Approach 3: Transfer compaction metadata only

This is the **preferred approach** as of now.

1. There is no initiator running on target warehouse w.r.t replicated tables(*need to identify how we are going to achieve this*). Compactor and cleaner are running target warehouse for all tables including replication targets.
2. Generate events once the compactor has finished relevant work on the source warehouse. As part of the event :
    2.1. Capture the file names included + their checksums which were included in this compaction.
    2.2. Capture the result of compaction, which is list of result file names + their checksum.
3. On the target warehouse once the event generated in step 2 is received, create a corresponding work object for the related table/partition and put it in the queue at target warehouse.
    3.1. This work object looks like is going to be significantly different than the work object that is created by Initiator.
    3.2. The compactor on the target warehouse for replicated tables will use the above work object to do its work.
    3.3. Compactor on target warehouse for replicated tables will ignore open transactions since we know they are going to be read only transactions.

Cons:

- On failover we have to make sure that we start the initiator on target warehouse w.r.t replicated tables.
- The compactor running for replicated tables will follow a slightly different code path than the compactor which is running on source tables, hence we will need to build the ability to run the respective code paths on source and target warehouse and change them on failover + failback.
- On failback we have to make sure that we stop the initiator on target warehouse w.r.t to replicated tables.