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) 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 :

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

Any read transaction opened above concurrently with the above timeline, should have same data visibility on both source and target warehouse.

Solutions

Approach 1 (Atlantic plan):

1. Add an additional attribute in the transaction commit event to denote if it is part of a streaming transaction batch.
2. The check for files size + checksum.
3. On the target cluster on receiving the txn commit event do the following:
    a. Check if the data file associated with the Txn commit event is already present on target warehouse HDFS.
        i. Copy the entire file for every commit txn if there is a difference in size or checksum.
        ii. ~~If present find the length of data in file and use that as an offset, to read from the source warehouse till the end of file and append (hadoop append of files is allowed) the relevant data to the file on the target warehouse.~~
        iii. If not present, copy the current state of file source warehouse to target warehouse.
    b. Copy the latest available side file (i think that is what its called, denoting the offset in the data file where the last valid ORC footer is written to) from source warehouse to target warehouse. This file should be copied over before the data file processing is started.

4. Sequence of steps when copying data from source to target, copy **side** file to staging directory -> copy **data** file to actual location -> copy **side** from staging directory to actual location.
5. Will have to do an append to the final location for data from staging even though we are copying the complete data file to staging directory, This is to allow existing readers to work.

Side file is also having offsets appended not overwritten.

Cons:

- Since data copy is till the end of file, different formats should not identify the file as corrupted if it's not completely written, as there is high possibility that part of the format might be missed and come as part of a later transaction.
- 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.

Pros:

- Implementation is simple for replication
- complexity of reading incomplete file's have to be handled by the underlying file format implementation.

Other Considerations:
For replication between two different hdfs zones (encrypted zones) or from encrypted to unencrypted zones the above strategy will work without any problems and hence would be the most likely approach taken for now. Additionally we are banking on the fact that the transaction batch and incremental replication will not overlap each other significantly to cause major network overhead if we copy the data/side file for every commit transaction (will be done only when there is a difference in size + checksum)

Approach 2:

1. Introduce a new Event type to capture the start and end offset of the ORC footer written for the data corresponding to the current transaction. It will be better to capture this as part of txn commit event else we have to solve for the case where load starts with the commit event and does not have the ability to traverse back.
2. As part of commit txn event capture the start and end offset for the data associated with the current transaction, written to the file.
3. Assuming the new event type in 1 above falls within the scope of transaction boundary and not outside.
4. On the target cluster when receiving transaction commit event, look for an associated event type denoted in 1 above :
    a. If found, append the relevant data + ORC footer using the offsets from source warehouse to target warehouse. This will be done directly at target location and staging directory will not be used, Also update the relevant offset in side file.

b. If not found treat as a regular acid table replication.

Cons:

- Introduction of new event types to identify the footer offsets, some other information might be required for different file formats.
- Copy utilities like distcp might not work with file copy along with offsets and the problem will be harder to solve for cases where the encryption is happening between encrypted and unencrypted zones or two different encryption zones.

Pros:

- Capturing only relevant portions of the data file as transactions commit so underlying file formats would not be required to handle files with incomplete information.
- Easier to reason about during debugging as we would precisely know as part of each event replication what was looked at and copied over.