HCatalog Streaming Mutation API

A Java API focused on mutating (insert/update/delete) records into transactional tables using Hive's ACID feature. It is introduced in Hive 2.0.0 (HIVE-10165).

Background

In certain data processing use cases it is necessary to modify existing data when new facts arrive. An example of this is the classic ETL merge where a copy of a data set is kept in sync with a master by the frequent application of deltas. The deltas describe the mutations (inserts, updates, deletes) that have occurred to the master since the previous sync. To implement such a case using Hadoop traditionally demands that the partitions containing records targeted by the mutations be rewritten. This is a coarse approach; a partition containing millions of records might be rebuilt because of a single record change. Additionally these partitions cannot be restated atomically; at some point the old partition data must be swapped with the new partition data. When this swap occurs, usually by issuing an HDFS `rm` followed by a `mv`, the possibility exists where the data appears to be unavailable and hence any downstream jobs consuming the data might unexpectedly fail. Therefore data processing patterns that restate raw data on HDFS cannot operate robustly without some external mechanism to orchestrate concurrent access to changing data.

The availability of ACID tables in Hive provides a mechanism that both enables concurrent access to data stored in HDFS (so long as it's in the ORC+ACID format) and also permits row level mutations on records within a table, without the need to rewrite the existing data. But while Hive itself supports `INSERT`, `UPDATE` and `DELETE` commands, and the ORC format can support large batches of mutations in a transaction, Hive's execution engine currently submits each individual mutation operation in a separate transaction and issues table scans (M/R jobs) to execute them. It does not currently scale to the demands of processing large deltas in an atomic manner. Furthermore it would be advantageous to extend atomic batch mutation capabilities beyond Hive by making them available to other data processing frameworks. The Streaming Mutation API does just this.

The Streaming Mutation API, although similar to the Streaming API, has a number of differences and is built to enable very different use cases. Superficially, the Streaming API can only write new data whereas the mutation API can also modify existing data. However the two APIs are also based on very different transaction models. The Streaming API focuses on surfacing a continuous stream of new data into a Hive table and does so by batching small sets of writes into multiple short-lived transactions. Conversely the mutation API is designed to infrequently apply large sets of mutations to a data set in an atomic fashion: either all or none of the mutations will be applied. This instead mandates the use of a single long-lived transaction. This table summarises the attributes of each API:

Attribute	Streaming API	Mutation API
Ingest type	Data arrives continuously.	Ingests are performed periodically and the mutations are applied in a single batch.
Transaction scope	Transactions are created for small batches of writes.	The entire set of mutations should be applied within a single transaction.
Data availability	Surfaces new data to users frequently and quickly.	Change sets should be applied atomically, either the effect of the delta is visible or it is not.
Sensitive to record order	No, records do not have pre-existing lastTxnIds or bucketIds. Records are likely being written into a single partition (today's date for example).	Yes, all mutated records have existing `RecordIdentifiers` and must be grouped by [partitionValues, bucketId] and sorted by lastTxnId. These record coordinates initially arrive in an order that is effectively random.
Impact of a write failure	Transaction can be aborted and producer can choose to resubmit failed records as ordering is not important.	Ingest for the respective group (partitionValues + bucketId) must be halted and failed records resubmitted to preserve sequence.
User perception of missing data	Data has not arrived yet "latency?"	"This data is inconsistent, some records have been updated, but other related records have not" – consider here the classic transfer between bank accounts scenario.
API end point scope	A given `HiveEndPoint` instance submits many transactions to a specific bucket, in a specific partition, of a specific table.	A set of `MutationCoordinators` writes changes to unknown set of buckets, of an unknown set of partitions, of specific tables (can be more than one), within a single transaction.

Structure

The API comprises two main concerns: transaction management, and the writing of mutation operations to the data set. The two concerns have a minimal coupling as it is expected that transactions will be initiated from a single job launcher type process while the writing of mutations will be scaled out across any number of worker nodes. In the context of Hadoop M/R these can be more concretely defined as the Tool and Map/Reduce task components. However, use of this architecture is not mandated and in fact both concerns could be handled within a single simple process depending on the requirements.

Note that a suitably configured Hive instance is required to operate this system even if you do not intend to access the data from within Hive. Internally, transactions are managed by the Hive MetaStore. Mutations are performed to HDFS via ORC APIs that bypass the MetaStore. Additionally you may wish to configure your MetaStore instance to perform periodic data compactions.

Note on packaging: The APIs are defined in the `org.apache.hive.hcatalog.streaming.mutate` Java package and included as the `hive-hcatalog-streaming` jar.

Data Requirements

Generally speaking, to apply a mutation to a record one must have some unique key that identifies the record. However, primary keys are not a construct provided by Hive. Internally Hive uses `RecordIdentifiers` stored in a virtual `ROW__ID` column to uniquely identify records within an ACID table. Therefore, any process that wishes to issue mutations to a table via this API must have available the corresponding row ids for the target records. What this means in practice is that the process issuing mutations must first read in a current snapshot of the data and then join the mutations on some domain specific primary key to obtain the corresponding Hive `ROW__ID`. This is effectively what occurs within Hive's table scan process when an `UPDATE` or `DELETE` statement is executed. The `AcidInputFormat` provides access to this data via `AcidRecordReader.getRecordIdentifier()`.

The implementation of the ACID format places some constraints on the order in which records are written and it is important that this ordering is enforced. Additionally, data must be grouped appropriately to adhere to the constraints imposed by the `OrcRecordUpdater`. Grouping also makes it possible to parallelise the writing of mutations for the purposes of scaling. Finally, to correctly bucket new records (inserts) there is a slightly unintuitive trick that must be applied.

All of these data sequencing concerns are the responsibility of the client process calling the API which is assumed to have first class grouping and sorting capabilities (Hadoop Map/Reduce etc). The streaming API provides nothing more than validators that fail fast when they encounter groups and records that are out of sequence.

In short, API client processes should prepare data for the mutate API like so:

- **MUST:** Order records by `ROW__ID.originalTxn`, then `ROW__ID.rowId`.
- **MUST:** Assign a `ROW__ID` containing a computed `bucketId` to each record to be inserted.
- **SHOULD:** Group/partition by table partition value, then `ROW__ID.bucketId`.

The addition of bucket ids to insert records prior to grouping and sorting seems unintuitive. However, it is required to ensure both adequate partitioning of new data and bucket allocation consistent with that provided by Hive. In a typical ETL the majority of mutation events are inserts, often targeting a single partition (new data for the previous day, hour, etc.). If more that one worker is writing said events, were we to leave the bucket id empty then all inserts would go to a single worker (e.g: reducer) and the workload could be heavily skewed. The assignment of a computed bucket allows inserts to be more usefully distributed across workers. Additionally, when Hive is working with the data it may expect records to have been bucketed in a way that is consistent with its own internal scheme. A convenience type and method is provided to more easily compute and append bucket ids: `BucketIdResolver` and `BucketIdResolverImpl`.

Update operations should not attempt to modify values of partition or bucketing columns. The API does not prevent this and such attempts could lead to data corruption.

Streaming Requirements

A few things are currently required to use streaming.

1. Currently, only ORC storage format is supported. So '`stored as orc`' must be specified during table creation.
2. The Hive table must be bucketed, but not sorted. So something like '`clustered by (colName) into 10 buckets`' must be specified during table creation. See Bucketed Tables for a detailed example.
3. User of the client streaming process must have the necessary permissions to write to the table or partition and create partitions in the table.
4. Hive transactions must be configured for each table (see Hive Transactions – Table Properties) as well as in `hive-site.xml` (see Hive Transactions – Configuration).

Note: Hive also supports streaming mutations to **unpartitioned** tables.

Record Layout

The structure, layout, and encoding of records is the exclusive concern of the client ETL mutation process and may be quite different from the target Hive ACID table. The mutation API requires concrete implementations of the `MutatorFactory` and `Mutator` classes to extract pertinent data from records and serialize data into the ACID files. Fortunately base classes are provided (`AbstractMutator`, `RecordInspectorImpl`) to simplify this effort and usually all that is required is the specification of a suitable `ObjectInspector` and the provision of the indexes of the `ROW__ID` and bucketed columns within the record structure. Note that all column indexes in these classes are with respect to your record structure, not the Hive table structure.

You will likely also want to use a `BucketIdResolver` to append bucket ids to new records for insertion. Fortunately the core implementation is provided in `BucketIdResolverImpl` but note that bucket column indexes must be presented in the same order as they are in the Hive table definition to ensure consistent bucketing. Note that you cannot move records between buckets and an exception will be thrown if you attempt to do so. In real terms this means that you should not attempt to modify the values in bucket columns with an `UPDATE`.

Connection and Transaction Management

The `MutatorClient` class is used to create and manage transactions in which mutations can be performed. The scope of a transaction can extend across multiple ACID tables. When a client connects, it communicates with the metastore to verify and acquire metadata for the target tables. An invocation of `newTransaction` then opens a transaction with the metastore, finalizes a collection of `AcidTables` and returns a new `Transaction` instance. The ACID tables are light-weight, serializable objects that are used by the mutation writing components of the API to target specific ACID file locations. Usually your `MutatorClient` will be running on some master node and your coordinators on worker nodes. In this event the `AcidTableSerializer` can be used to encode the tables in a more transportable form, for use as a `Configuration` property for example.

As you would expect, a `Transaction` must be initiated with a call to `begin` before any mutations can be applied. This invocation acquires a lock on the targeted tables using the metastore, and initiates a heartbeat to prevent transaction timeouts. It is highly recommended that you register a `LockFailureListener` with the client so that your process can handle any lock or transaction failures. Typically you may wish to abort the job in the event of such an error. With the transaction in place you can now start streaming mutations with one or more `MutatorCoordinator` instances (more on this later), and then can `commit` or `abort` the transaction when the change set has been applied, which will release the lock with the metastore client. Finally you should `close` the mutation client to release any held resources.

The `MutatorClientBuilder` is provided to simplify the construction of clients.

WARNING: Hive doesn't currently have a deadlock detector (it is being worked on as part of HIVE-9675). This API could potentially deadlock with other stream writers or with SQL users.

Writing Data

The `MutatorCoordinator` class is used to issue mutations to an ACID table. You will require at least one instance per table participating in the transaction. The target of a given instance is defined by the respective `AcidTable` used to construct the coordinator. It is recommended that a `MutatorClientBuilder` be used to simplify the construction process.

Mutations can be applied by invoking the respective `insert`, `update`, and `delete` methods on the coordinator. These methods each take as parameters the target partition of the record and the mutated record. In the case of an unpartitioned table you should simply pass an empty list as the partition value. For inserts specifically, only the bucket id will be extracted from the `RecordIdentifier`, the transaction id and row id will be ignored and replaced by appropriate values in the `RecordUpdater`.

Additionally, in the case of deletes, everything but the `RecordIdentifier` in the record will be ignored and therefore it is often easier to simply submit the original record.

Caution: As mentioned previously, mutations must arrive in a specific order for the resultant table data to be consistent. Coordinators will verify a naturally ordered sequence of [lastTransactionId, rowId] and will throw an exception if this sequence is broken. This exception should almost certainly be escalated so that the transaction is aborted. This, along with the correct ordering of the data, is the responsibility of the client using the API.

Dynamic Partition Creation

It is very likely to be desirable to have new partitions created automatically (say on a hourly basis). In such cases requiring the Hive admin to pre-create the necessary partitions may not be reasonable. The API allows coordinators to create partitions as needed (see: `MutatorClientBuilder.addSinkTable(String, String, boolean)`). Partition creation being an atomic action, multiple coordinators can race to create the partition, but only one would succeed, so coordinators' clients need not synchronize when creating a partition. The user of the coordinator process needs to be given write permissions on the Hive table in order to create partitions.

Care must be taken when using this option as it requires that the coordinators maintain a connection with the metastore database. When coordinators are running in a distributed environment (as is likely the case) it is possible for them to overwhelm the metastore. In such cases it may be better to disable partition creation and collect a set of affected partitions as part of your ETL merge process. These can then be created with a single metastore connection in your client code, once the cluster side merge process is complete.

Finally, note that when partition creation is disabled the coordinators must synthesize the partition URI as they cannot retrieve it from the metastore. This may cause problems if the layout of your partitions in HDFS does not follow the Hive standard (as implemented in `org.apache.hadoop.hive.metastore.Warehouse.getPartitionPath(Path, LinkedHashMap <String, String>)`).

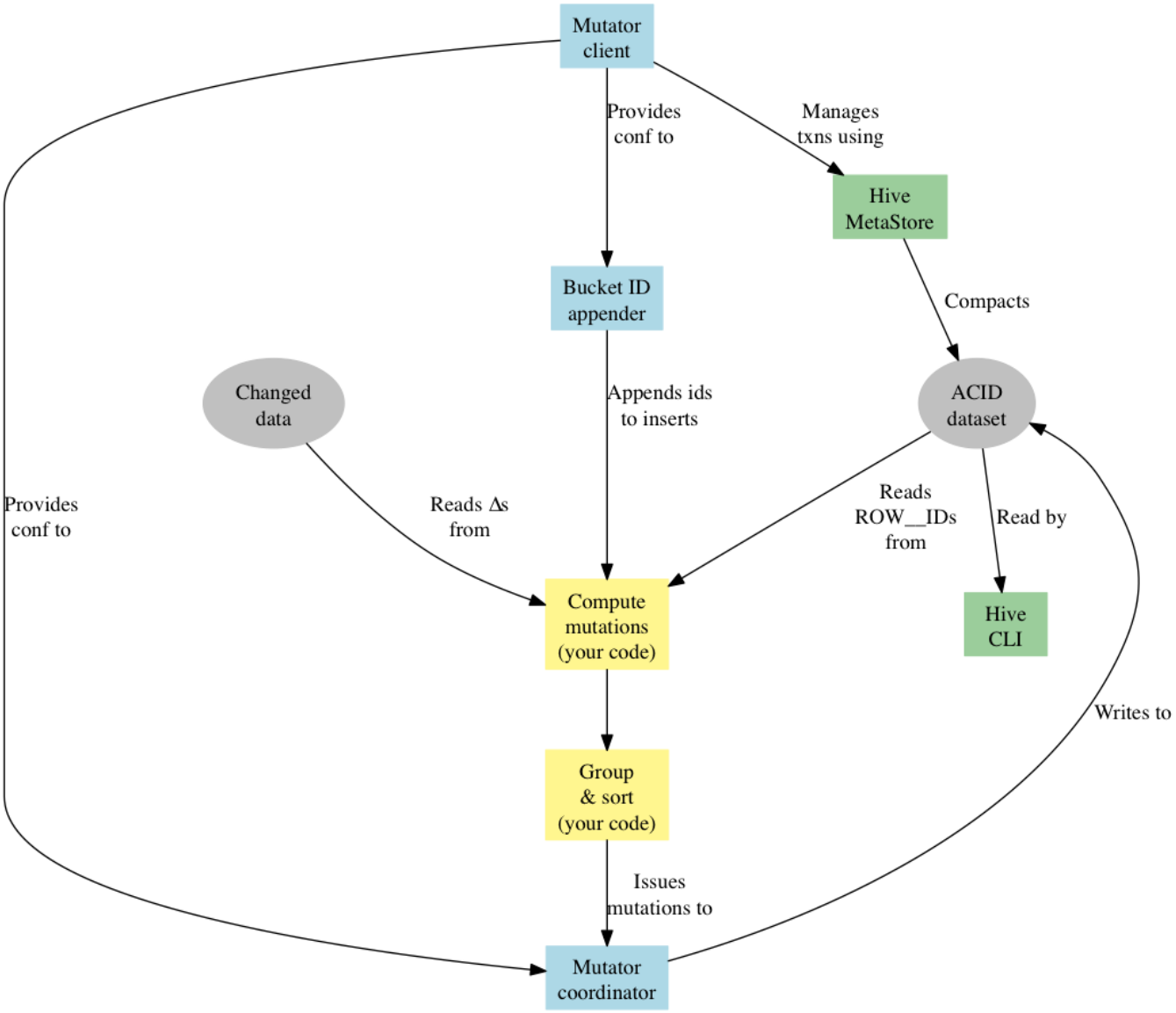
Reading Data

Although this API is concerned with writing changes to data, as previously stated we'll almost certainly have to read the existing data first to obtain the relevant `ROW__IDs`. Therefore it is worth noting that reading ACID data in a robust and consistent manner requires the following:

1. Obtaining a valid transaction list from the metastore (`ValidTxnList`).
2. Acquiring a lock with the metastore and issuing heartbeats (`LockImpl` can help with this).
3. Configuring the `OrcInputFormat` and then reading the data. Make sure that you also pull in the `ROW__ID` values. See: `AcidRecordReader.getRecordIdentifier`.

4. Releasing the lock.

Example



So to recap, the sequence of events required to apply mutations to a dataset using the API is:

1. Create a `MutatorClient` to manage a transaction for the targeted ACID tables. This set of tables should include any transactional destinations or sources. Don't forget to register a `LockFailureListener` so that you can handle transaction failures.
2. Open a new `Transaction` with the client.
3. Get the `AcidTables` from the client.
4. Begin the transaction.
5. Create at least one `MutatorCoordinator` for each table. The `AcidTableSerializer` can help you transport the `AcidTables` when your workers are in a distributed environment.
6. Compute your mutation set (this is your ETL merge process).
7. Optionally: collect the set of affected partitions.
8. Append bucket ids to insertion records. A `BucketIdResolver` can help here.
9. Group and sort your data appropriately.
10. Issue mutation events to your coordinators.
11. Close your coordinators.
12. Abort or commit the transaction.
13. Close your mutation client.
14. Optionally: create any affected partitions that do not exist in the metastore.

See `ExampleUseCase` and `TestMutations.testUpdatesAndDeletes()` for some very simple usages.