Description

Many customers want to be able to insert, update and delete rows from Hive tables with full ACID support. The use cases are varied, but the form of the queries that should be supported are:

- INSERT INTO tbl SELECT …
- INSERT INTO tbl VALUES ...
- UPDATE tbl SET … WHERE …
- DELETE FROM tbl WHERE …
- MERGE INTO tbl USING src ON … WHEN MATCHED THEN ... WHEN NOT MATCHED THEN ...
- SET TRANSACTION LEVEL …
- BEGIN/END TRANSACTION

Use Cases

1. Once an hour, a set of inserts and updates (up to 500k rows) for various dimension tables (eg. customer, inventory, stores) needs to be processed. The dimension tables have primary keys and are typically bucketed and sorted on those keys.
2. Once a day a small set (up to 100k rows) of records need to be deleted for regulatory compliance.
3. Once an hour a log of transactions is exported from a RDBS and the fact tables need to be updated (up to 1m rows) to reflect the new data. The transactions are a combination of inserts, updates, and deletes. The table is partitioned and bucketed.

Areas of Development

There are four major areas of development that need to be addressed: transaction management; input format and output format; command parsing, planning, and processing; and compaction and garbage collection.

Transaction Management

Managing transactions is a critical part of adding inserts, updates, and deletes to Hive. The metastore will track the read and write transactions that are currently in progress. To provide a consistent view to read operations, the metastore will create a **transaction id** for each write operation. The transaction ids are strictly increasing from 1 and are global within a given metastore. There are no capabilities for cross-metastore transactions.

When a read operation starts, it provides the tables it is reading and requests the current set of committed transaction ids. This set will be represented as the maximum committed transaction id and the list of transaction ids that are still in flight. Each MapReduce job launched as part of the query will have exactly the same set of valid transaction ids provided in the JobConf and thus the reader will present a consistent snapshot view of the input tables from the time at the start of the command.

Write operations will provide the list of tables being read and the list of tables being written and get a list of valid transaction ids to read from and a write transaction id. Any data that is read will only include the valid transaction ids or their write transaction id. All data that is written will be tagged with the write transaction id.

In both cases, when the command is finished, it should notify the metastore. Since the metastore must identify (and abort) abandoned transactions, the Hive client must notify the metastore that the command is still running every 10 minutes or the transaction will be aborted. The metastore will maintain the list of tables being read and the corresponding transaction ids so that it can schedule compacted data to be deleted after the last reader finishes.

Before a write transaction is committed in the metastore, all of the HDFS files for that transaction must be in their final location. Similarly, before a write transaction is aborted in the metastore all of the files must have been deleted from HDFS. This ensures that any valid transaction id that is given to a reader will only have correct HDFS files associated with it.

InputFormat

The heart of the approach is the client-side merge of the HDFS files. The layout of the Hive warehouse will remain the same at the root and will look like *$database/$table/$partition*. Inside of each partition (or the table directory, if the table isn't partitioned), instead of a set of files named with the bucket id (eg. *000000_0*) there will be a directories with the base rows and the deltas that have modified that base. The base files will be stored in a directory named *base_$etid* and the deltas will be stored in directories named *delta_$btid_$etid*, where $btid is the first transaction id included in the file and $etid is the first transaction id not included in the file. This naming scheme will enable us to leave both the pre and post compaction files while the clients using the pre compaction files finish. Within each directory, the rows must be consistently bucketed so that each bucket can be processed independently.

In theory the base can be in any format, but ORC will be required for V1. Every row in the table is uniquely identified by the transaction id of the transaction that inserted it (or 0 for the original load), the row id, and the implicit bucket number. The implementation of delta files will be ORC files with a row format of:

```
create table deltaFile (
  operation int,      /* 0 = insert, 1 = delete, 2 = update */
  transactionId long, /* transaction that inserted the row */
  rowId long,         /* row id */
  rowData struct(...) /* for insert and update, the columns */
);
```

For sorted tables, the base and delta files will be sorted by the sort columns, while unsorted tables will be sorted by descending transaction id and ascending row id. Because of the integer run length encoding in ORC, the delta files will compress very effectively.

When the read query starts, it will get the valid transaction ids and put them into the

MapReduce's JobConf. The InputFormat's getSplits will get the directory listing of the parent directory and the valid transaction ids to find the required base and delta files. The locations of the input split will be the locations of the base file. The InputSplit will look like:

```
create table InputSplit (
  baseFile string,
  startTransactionId long,
  startRowId long,
  endTransactionId long,
  endRowId long,
  deltaDirectories list<string>
);
```

The RecordReader will open the base file and each of the delta files and do a merge sort to provide a single view to the reading operator.

When the read query is done, it needs to inform the metastore that it is no longer reading the tables so that obsolete versions of the tables can be garbage collected.

OutputFormat

When one of the insert, update, or delete commands starts, it gets the valid transaction ids for reading and a transaction id to write to. When we implement multi-command transactions, the write transaction id will need to be included in the set of valid transaction ids so that each transaction will see its own previous updates.

The delta files are written with a smaller stripe size (32MB), a smaller buffer size (32K), and no compression. This is because many of the files will be very small and lowering the sizes will reduce the resource requirements for reading the files.

For the insert command, the tasks will write the delta file with the insert rows. For the delete command, the task will stream through the table base and deltas using the reader and write a new delta file with the transaction ids and row ids of the deleted rows. The update command will read the table base and deltas using the reader. For unsorted tables, the updates will be split into a delete of the old record and an insert of the new record. Because the delta file must be sorted by transaction id, the deletes, which are naturally smaller than the inserts, will be buffered and written at the end of the delta. For sorted tables, the updates will be encoded as updates to simplify the merge.

The update commands should use the underlying MapReduce OutputCommitter instead of Hive's NullOutputCommitter so that the task output can be promoted automatically when a task finishes and the job output can be promoted when a job completes.

While any query is running, the client will heartbeat to the metastore every N minutes to ensure the transaction isn't removed for inactivity.

Command Parsing, Planning, and Processing

The parser, planner, and operator tree need to be extended with the new insert, update, and delete commands. Reading from tables with delta files will be transparently handled by the record reader, so the read commands won't need to be modified.

Compaction

Periodically the system must rewrite the smaller delta files into larger ones (minor compaction) and rewriting the delta files into the base file (major compaction). Since the compactions may take relatively large amount of time, they should not block other updates. This is possible because although they are rewriting the data, they are semantically equivalent of the previous data.

To preserve read throughput over time, compactions must happen regularly and therefore automatically. In particular, since reading the table involves doing a N-way merge sort, we must bound the size of N to a relatively small number. Minor compactions should be automatically scheduled for a table when we have more than 10 committed deltas. The minor compaction asks the metastore for the smallest in-flight transaction id and will compact all transactions that are strictly less than it. Once the minor compaction is done, the directory with the output is moved into the partition's directory with the name *delta_$bid_$eid.* The previous delta directories will be scheduled for garbage collection once the transactions using the previous version have finished.

When more than 10% of the records are coming from delta files instead of the base, a major compaction should be triggered. I suspect that in V2 we will also need a time based trigger to do major compactions during times when the cluster is less busy. The major compaction will request the lowest in-flight transaction id and re-write the base with the merged transaction that are less than it. When the compaction is done, the output is moved to *base_$etid* and the old base is scheduled for garbage collection. Because compaction will change the row ids, the major compaction will also output a set of files that translate from the previous major compaction to the new one. The translation files will be stored in the base directory and be named *base_$etid/_translation_$bucketid*. To keep the translation files small, it is sufficient to write the old transaction id, row id, and the number of sequential rows. With this encoding, the translation files should be proportional to the size of the deltas rather than the size of the base. After the major compaction is finished, if there are deltas using the old base file the system should schedule a minor compaction to be run. When the minor compaction is done, the translation files can be scheduled for garbage collection. Until the minor compaction finishes, the record reader will need to merge the translation files together with the old delta files to find the updated or deleted records.

To support garbage collection, the metastore needs a job manager that can automatically submit MapReduce jobs and monitor when they complete or fail. Failed jobs need to be logged via log4j and resubmitted with an exponential backoff. Compaction jobs must be throttled to a specified

percentage of the MapReduce cluster.

Garbage Collection

The directories and files that are left over after compaction need to be deleted. However, if they are deleted while queries are reading them, the queries will fail. Therefore, the metastore will use the list of current read transactions to wait until after all of the read transactions that starting reading before the compaction are finished. Since garbage collection is relatively inexpensive, the metastore can use a thread to issue the HDFS deletes.

Phases of Development

Phase 1

1. Support insert, update, and delete commands.
2. Support for ORC as the base storage.
3. All insert, update, and delete commands auto commit their transaction at the end of the query.
4. Read commands will have a snapshot view of the tables at the start of the command.
5. Write commands will acquire an exclusive lock on the table, which implies serializable isolation.
6. Updates will be supported on partitioned, unpartitioned, sorted, and unsorted tables.

Phase 2

1. Support merge command.
2. Support begin transaction and end transaction commands and multi-statement transactions.
3. Tools to list and abort current transactions
4. Optimizations for commands that update every row, such “DELETE FROM tbl” and “UPDATE tbl SET X = …”.
5. Support for scheduling major compactions at particular times of day.

Later

1. Support isolation level selection of uncommitted, committed, snapshot, or serializable.
2. Other storage formats as the base storage
3. Row level update conflict detection
4. Update of sort column
5. Update of partition column
6. Support for updating tables with a total order bucket scheme. (dependent on hive having such a scheme of course)