Introduction

This document gives the design for a new Hive Transaction Manager. This is part of the project to add insert, update, and delete with ACID properties to Hive. For details on the project as a whole see HIVE-5317.

Overview

The approach has the following pieces

1. A new implementation of `HiveLockManager` will be implemented that stores the lock information in the metastore database.
2. A new interface `HiveTransactionManager` will be created that provides open, commit, and rollback methods. Implementations of `HiveTransactionManager` will be expected to contain and use a `HiveLockManager`.
3. The metastore thrift interface will be changed to support new calls and structures for locking and transactions.
4. Tables will be added to the metastore database to store lock and transaction information.

The metastore was chosen as a place to store this information for several reasons:

1. Storing the information in a durable store provides durability in the face of client or metastore server failure, as well as supports scaling the metastore by adding multiple thrift servers.
2. All Hive users have an RDBMS that they store metadata in. Comparatively few users run ZooKeeper.
3. The transactional nature of the RDBMS makes implementation much easier.

This design makes the following assumptions:

1. Only auto-commit transactions are supported at this time. This is not inherent to the design; this design can be extended to support being/commit/rollback functionality.
2. The isolation level is always snapshot. This is not inherent to the design; this design can be extended to support serializable, dirty read, and dirty write functionality.
3. DDL operations are not part of a transaction. DDL operations done in transactions will not be done in a transactional nature. This is inherent to the design.

One goal of this design is to allow users who do not wish to use the new transaction functionality to continue to use Hive as they do today with no impact.

This design does not change based on whether we decide to start with a single delta file per writer or one per transaction.

The rest of this document will cover changes to the metastore database, to the thrift interface, to `HiveLockManager`, and the new `HiveTransactionManager`. It will

then walk through how clients will use the functionality.

Changes to the Database

In all of the database changes, bigint is used for timestamps to avoid cross database issues (as some databases implement TIMESTAMP but not DATETIME, and vice versa).

The tables proposed in this document will be kept in a separate schema (Postgres, Derby) or database (MySQL) than the metastore database (since Oracle ties schemas to users we will keep these tables in the metastore schema in Oracle). The connection to them will not use JDO but rather direct JDBC. This implies we will need a separate installation script for them. The reason for this is JDO will be a very bad model for the operations that need to be done on these tables. The schema/database for this will not be auto-created. A configuration value will be used to store the JDBC URI for the database. If it is defined, the application will assume the tables have been created. If it is not defined, any attempts to use transactions will fail.

Transaction Table

This table will track all current transactions. It will also track aborted transactions until any locks they hold and data they have generated can be cleaned up.

```
create table txns (
    txn_id bigint not null,
    txn_state char(1) not null, -- a = aborted, -- o = open
    txn_started bigint not null,
    txn_last_heartbeat bigint not null,
    txn_finalized bigint, -- will be null unless txn_state = a
    primary key(txn_id)
);
```

Transaction Components Table

This table will track database entities involved in a transaction (tables and partitions). This is required so that the compactor can know when it can and cannot clean aborted transactions out of the `txns` table.

```
create table txn_components (
    tc_txnid bigint not null,
    tc_database varchar(255) not null,
    tc_table varchar(255) not null,
    tc_partition varchar(255)
    foreign key (tc_txnid) references txns(txn_id)
);
```

Transaction ID Table

This table will track the next valid transaction ID. It will only have one row, which will contain the id of the next transaction. When the database is first created it will have a value of 1.

```
create table next_txn_id (
    ntxn_next bigint not null
);
```

Lock Table

This table will track current locks.

```
create table hive_lock (
    hl_lock_id bigint not null,
    hl_txnid bigint,
    hl_db varchar(255) not null,
    hl_table varchar(255) not null,
    hl_partition varchar(255),
    hl_lock_state char(1) not null, -- 'a' = acquired, 'w' = waiting
    hl_lock_type char(1) not null, -- 'r' = read, 'w' = write, 'x' =
exclusive
    hl_last_heartbeat bigint not null,
    hl_acquired_at bigint,
    hl_object_data clob,
    index (hl_lock_id),
    index (hl_txnid)
);
```

Lock ID Table

This table will track the next lock id. It will have only one row. The initial value when the database is created will be 1.

```
create table next_lock_id (
    nl_next bigint not null
);
```

Thrift Changes

New Enums

```
enum TxnState {
    COMMITTED = 1,
    ABORTED = 2,
    OPEN = 3,
}

enum LockLevel {
    DB = 1,
    TABLE = 2,
    PARTITION = 3,
}

enum LockState {
    ACQUIRED = 1,
    WAITING = 2,
    ABORT = 3,
}

enum LockType {
    SHARED = 1,
    SEMI_SHARED = 2,
```

```
    EXCLUSIVE = 3,
}
```

New Structures

```
struct TxnInfo {
    1: required i64 id,
    2: required TxnState state,
}

struct ValidTxns {
    1: required i64 txn_high_water_mark,
    2: required list<TxnInfo> invalid_txns,
}

struct LockComponent {
    1: required LockType type,
    2: required LockLevel level,
    3: required string dbname,
    4: required string tablename,
    5: required string partitionname,
    6: optional string lock_object_data,
}

struct LockRequest {
    1: required list<LockComponent> component,
    2: optional i64 txnid,
}

struct LockResponse {
    1: required i64 lockid,
    2: required LockState state,
}

struct Heartbeat {
    1: optional i64 lockid,
    2: optional i64 txnid,
}

struct TxnPartitionInfo {
    1: required string dbname,
    2: required string tablename,
    3: required string partitionname,
}
```

New Exception

```
exception NoSuchTxnException {
    1: string message
}

exception TxnAbortedException {
```

```
    1: string message
}

exception TxnOpenException {
    1: string message
}

exception NoSuchLockException {
    1: string message
}
```

New Thrift Calls

Valid Transactions

This call will get a list of valid transactions. It will be returned in the form of a high water mark and an exception list. Any transaction id that is less than or equal to the high water mark and is not on the exception list is a valid transaction.

```
ValidTxns get_valid_txns()
```

Pseudo code:

```
select ntxn_next -1 from next_txn_id;
select * from txns;
return results to caller in ValidTxns
```

Open a Transaction

The client will call this to initiate a new transaction. It will return a transaction id to be used in future calls.

```
i64 open_txn()
```

Pseudo code:

```
begin;
select ntxn_next from next_txn_id for update;
update next_txn_id set ntxn_next++;
insert into txns
    (txn_id, txn_state, txn_started, txn_last_heartbeat)
    values (next_txn_id from above, 'o', now(), now());
commit;
return the txn_id to the writer.
```

Rollback a Transaction

The client will call this to rollback a transaction. It will throw `NoSuchTxnException` if the transaction being rolled back is not open.

```
void abort_txn(1:i64 txnid) throws (1:NoSuchTxnException)
```

Pseudo code:

```
begin;
update txns set txn_state = 'a' and txn_finalized = now()
    where txn_id = txnid;
delete from hive_locks where hl_txnid = 'txnid';
commit;
if (no rows updated) throw NoSuchTxnException;
```

Commit a Transaction

The client will call this to commit a transaction. It will throw `NoSuchTxnException` if the transaction to be committed does not exist. It will throw `AbortedTxnException` if the transaction has already been rolled back.

```
void commit_txn(1:i64 txnid)
    throws (1:NoSuchTxnException, 2:TxnAbortedException)
```

Pseudo code:

```
if (transaction aborted) throw TxnAbortedException;
begin;
delete from txns where txn_id = txnid;
delete from txn_components where txn_id = txnid;
delete from hive_locks where hl_txnid = txnid;
commit;
if (no rows updated) throw NoSuchTxnException;
```

Lock

The client will call this to lock databases, tables, or partitions. It will throw `NoSuchTxnException` if a non-existent transaction is referenced. It will throw `AbortedTxnException` if an rolled back transaction is referenced. It will return a response that indicates whether the client has obtained the lock or needs to wait.

```
LockResponse lock(1:LockRequest rqst)
    throws (1:NoSuchTxnException, 2:TxnAbortedException)
```

Pseudo code:

```
if (rqst.txnid != null) {
    if (rqst.txnid aborted) throw TxnAbortedException;
    else if (rqst.txnid not in txns) throw NoSuchTxnException;
    for each entity in lock {
        add entry to txn_components
    }
}
begin;
select for update nl_next from next_lock_id;
update nl_next set next_lock_id++;
for each entity in lock {
    insert into hive_lock
        (hl_lock_id, hl_txnid, hl_db, hl_table, hl_partition,
hl_lock_state,
            hl_lock_type, hl_last_heartbeat)
        values (lockid from next, rqst.txnid or null, rqst.db,
                rqst.table or null, rqst.partition or null, 'w',
rqst.type,
                now());
}
return checkLock();
```

Check For Lock

If the client does not initially obtain the lock, it will need to poll to see if the lock has been obtained. It will throw `NoSuchTxnException` if a non-existent transaction is referenced. It will throw `AbortedTxnException` if an rolled back transaction is referenced. It will throw `NoSuchLockException` if a non-existent lock is referenced. It will return a

response that indicates whether the client has obtained the lock or needs to wait.

```
LockResponse check_lock(1:i64 lockid)
    throws (1:NoSuchTxnException, 2:TxnAbortedException,
            3:NoSuchLockException)
```

Pseudo code:

```
return checkLock();
```

checkLock()

Both lock and check_lock thrift calls will call a new server function checkLock. This function will have the following responsibilities:

1. Remove timed out locks.
2. Determine who has the lock for any given database, table, or partition.
3. Detect and break deadlocks (future).

Pseudo code:

```
// without this our locks could time out while we wait for them
begin;
update hive_lock set hl_last_heartbeat = now() where hl_lock_id =
lockid;
if (txnid provided) {
    update txns set txn_last_heartbeat = now() where txn_id = txnid;
}
commit;

// Remove timed out locks
begin;
n = select now();
delete from hive_lock where tl_last_heartbeat < n - timeout;
commit;

begin;
select * for update from hive_lock;
load into into queue, sorted by lock_state (acquired, then waiting),
lock_id;
if (for all entities to lock, lock is in acquired state) {
    rollback;
    return acquired;
}
if (for all entities to lock there does not exist any entries in front
        (including checking for tables locked when we need partitions
and
            dbs when we need tables or partitions)) {
    return acquire();
}
// If we've gotten this far then something is in our way
if (all entries in front are shared and this lock_id is shared or semi-
shared) {
    return acquire();
}
```

```
if (all entries in front are semi-shared and this lock_id is shared) {
    return acquire();
}

if (all entries in front are semi-shared and this lock_id is semi-shared
and
        shares the same txn_id as all entries in front) {
    return acquire();
}

return wait; // in the future when we allow users to do 'begin'
             // we'll need to check for deadlock here, as the user may
already
             // be holding locks and waiting on additional locks which
are
             // waiting on them.

acquire() {
    update hive_lock set hl_lock_state = 'a' and hl_acquired_at = now()
and
        hl_last_heartbeat = now() where hl_lock_id = lock_id;
    commit;
    return acquired;
}
```

Lock Interaction Matrix

Lock Held	Lock Requested	Result Request
SEMI_SHARED	SHARED	acquired
SEMI_SHARED	SEMI_SHARED	wait
SHARED	SEMI_SHARED	acquired
SHARED	SHARED	acquired
SEMI_SHARED	EXCLUSIVE	wait
SHARED	EXCLUSIVE	wait
EXCLUSIVE	SHARED	wait
EXCLUSIVE	SEMI_SHARED	wait
EXCLUSIVE	EXCLUSIVE	wait

If in the future we extend this to implement a serializable isolation level than `SEMI_SHARED` followed by `SHARED` will need to wait rather than acquire when running at that isolation level. If we someday implement dirty write then `SEMI_SHARED` followed by `SEMI_SHARED` would acquire rather than wait when running at that isolation level.

Unlock

The client will call this to release a lock. The client should not call this if it has opened a transaction. Rolling back or committing the transaction will automatically release the locks. If the provided lockid does not exist, `NoSuchLockException` will be thrown. If the lock referenced is part of an active transaction, `TxnOpenException` will be thrown.

```
void unlock(1:i64 lockid) throws (1:NoSuchLockException)
```

Pseudo code:

```
if (txn open) throw TxnOpenException;
delete from hive_lock where hl_lock_id = lockid;
if (no rows deleted) throw NoSuchLockException;
```

Heartbeat

The client will call this to inform the server that it is still actively holding locks and/or holding open a transaction. If a lockid is provided and that lockid does not exist, `NoSuchLockException` will be thrown. If a txnid is provided and that txnid does not exist, `NoSuchTxnException` will be thrown.

```
void heartbeat(1:Heartbeat ids)
    throws (1:NoSuchLockException, 2:NoSuchTxnException)
```

Pseudo code:

```
begin
if (lock id provided) {
    update hive_locks set hl_last_heartbeat = now()
        where hl_lock_id = lockid;
    if (no such lockid) throw NoSuchLockException;
}
if (txnid provided) {
    update txns set txn_last_heartbeat = now() where txn_id = txnid;
    if (no rows updated) throw NoSuchTxnException;
}
commit;
```

Time Out Transactions

The compactor will call this before it begins compacting. It will be used to mark any open transactions that have failed to heartbeat as aborted.

```
void timeout_txns()
```

Pseudo code:

```
update txns set txn_state = 'a' and txn_finalized = now()
    where txn_heartbeat < now() - timeout;
```

Remove Aborted Transactions from Txns Table

After the compactor has finished compacting a partition, it will make this call to potentially remove references to aborted transactions from the txns table. This call will only remove references to a partition at a time. This is necessary because a transaction may span several partitions and/or tables, and it cannot be removed from the txns table until all of the involved partitions and/or tables have been compacted. This will keep the txns table from growing without bound.

```
void clean_aborted_txns(1:TxnPartitionInfo)
```

Pseudo code:

```
begin;
```

```
select distinct tc_txnid
    from txn_component tc join txns t on (tc.tc_txnid = t.txn_id)
    where tc_database = partition.dbname
        and tc_table = partition.tablename
        and tc_partition = partition.partitionname
        and t.txn_heartbeat < now() - timeout;
if (rows from select) {
    for each row {
        if (other entries in txn_component table) {
            delete just record for this partition
        } else {
            delete record for this partition
            delete from txns where txn_id = txnid;
            delete from hive_locks where hl_txnid = txnid;
        }
    }
    commit;
} else {
    rollback;
}
```

New Interface HiveTxnManager

A new interface `HiveTxnManager` will be created. Two implementations of this will be built, the `DbTxnManager` (which will be used by the functionality described in this document) and the `DummyTxnManager` (which will implement the current behavior).

Every transaction manager implementation will contain an implementation of `HiveLockManager`. The task of acquiring and releasing locks, which is currently invoked direclty in the classes `Driver` and `DDLTask` will be moved into implementations of `HiveTxnManager`. The choice to contain the lock manager in the transaction manager was made because:

1. It cuts the amount of thrift and database traffic in half (since we need to heartbeat for both).
2. It avoids the odd situation where a client is heartbeating for one and not the other, so locks are expiring and the transaction is not, or vice versa.
3. By making rollback/commit auto-unlock all resources it makes handling client failure much easier. Otherwise there are a number of situations where a client unlocks (or commits/rollsback, depending on which happens first) and then fails before it can commit/rollback (or unlock). By making commit/rollback and unlock atomic we avoid these issues.
4. Providing a dummy transaction manager allows for cleaner code (from Hive's viewpoint there's always a transaction manager) while supporting a backward compatible mode of no transactions.

This new interface will have the following methods:

```
openTxn()
```

Open a transaction by making the thrift `open_txn` call.

`acquireLocks(QueryPlan plan)`
Given a query plan, obtain the necessary locks to execute this plan. In a query with a transaction, this must be called after `openTxn`.

`getLockManager()`
This will be used to obtain the lock manager. This should be called rather than instantiating a lock manager directly, as the transaction manager will choose which lock manager to use.

`commitTxn()`
Commit a transaction by making the thrift `commit_txn` call. All locks held by the lock manager will be released. In the case of read only queries where a transaction was not started, the transaction manager will still need to release all held locks when this is called.

`abortTxn()`
Rollback a transaction by making the thrift `abort_txn` call. All locks held by the lock manager will be released. In the case of read only queries where a transaction was not started, the transaction manager will still need to release all held locks when this is called.

`heartbeat()`
This will give the transaction manager an opportunity to send a heartbeat to the metastore (via a thrift `heartbeat` call) so that the transaction is not timed out. Transaction managers that do not require heartbeat functionality can implement this as a no-op.

`getValidTxns()`
Get a list of valid transactions by making the thrift `get_valid_txns` call.

`timeoutTxns()`
Ask the server to abort timed out transactions by making the thrift `timeout_txns` call. This is intended for use by the compactor.

`cleanAbortedTxns()`
Ask the server to remove transactions for a given partition that has been compacted by making the thrift `clean_aborted_txns` call. This is intended for use by the compactor.

`showLocks(ShowLocksDesc lock)`
This will show the current set of locks and the transaction they are associated with.

`lockTable(LockTableDesc lock)`
This will issue a special, extra transactional lock. It is an optional method and is provided only for backward compatibility with existing lock managers and is not expected to be implemented by any transaction manager except the `DummyTxnManager`.

`unlockTable(UnlockTableDesc lock)`

This will release a special, extra transactional lock. It is an optional method and is provided only for backward compatibility with existing lock managers and is not expected to be implemented by any transaction manager except the `DummyTxnManager`.

A new class `TxnManagerFactory` will be built as well that will choose the correct transaction manager.

Which transaction manager to use in an instance of Hive will be configured via the configuration value `hive.txn.manager`. The default value will be `DummyTxnManager`. When the `DummyTxnManager` is in use, the configuration value `hive.lock.manager` will continue to be used to determine the lock manager to use. In general, whether to honor the value of `hive.lock.manager` or not will be at the discretion of the transaction manager, as some transaction managers will need to mandate the lock manager to use. The `DbTxnManager` will provide its own lock manager.

Changes To HiveLockManager

A new value `SEMI_SHARED` will be added to the enum `HiveLockMode`. The purpose of this value is to indicate a lock that can share with a `SHARED` lock but not with other `SEMI_SHARED` locks.

A database based implementation of `HiveLockManager` will be built, `DbHiveLockManager`. It will differ from the existing `HiveLockManager` implementations in:

1. Unless log level is set to debug it will ignore the data in `HiveLockObjectData` and not send it in the thrift call. This is to reduce the amount of data exchanged on each lock request, as this data is significant in size (several K) and necessary only for debugging.
2. It will not fail if a lock comes back with a status of wait. It will go into a loop of retrying to acquire the lock. It will use a standard back off algorithm when waiting for the lock.

TODO - Open question: should we offer a “no wait” mode for the lock where failure to lock immediately produces a query failure rather than waiting?

Scenario Walk Throughs

Read

When a client wishes to execute a read operation (e.g. SELECT) or an insert it will:

1. Request `SHARED` locks on each partition (possibly across multiple tables) it needs to read or write to by calling `HiveLockManager.lock()`.
    a. If it receives a wait response, `DbHiveLockManager` will call `check_lock` until it receives acquired.
2. Call `HiveTxnManager.getValidTxns()` so that the InputFormat understands which delta files (or portions of the delta file) to read.
3. While reading the data the client will occasionally call

`HiveLockManager.heartbeat()` to keep hold of its locks.
    a. If it fails to heartbeat any other call to `lock` or `check_lock` by another client will cause these locks to be released.
4. When finished reading, it will call `HiveLockManager.unlock()`.

Write

When a client wishes to execute a write operation (e.g. DELETE) it will:

1. Open a transaction by calling `HiveTxnManager.openTxn()`.
2. Request `SEMI_SHARED` locks on each partition it needs to write to (and possibly `SHARED` locks across each partition it needs to read) by calling `HiveLockManager.lock()`.
3. If reading any data, it will call `HiveTxnManager.getValidTxns()` so that the InputFormat understands which delta files (or portions of the delta file) to read.
4. While reading and writing data, the client will occasionally call `HiveLockManager.heartbeat()` to keep its locks and transaction alive.
    a. If it fails to heartbeat any other call to `lock` or `check_lock` by another client will cause these locks to be released. Its transaction will eventually be declared invalid by the next call to `timeout_txns`.
5. When finished reading and writing, it will call `HiveTxnManager.commitTxn()`. It will **not** call `HiveLockManager.unlock()`.
    a. If it fails rather than succeeds in its operation it will call `HiveTxnManager.abortTxn()`.

Compactor

When the compactor runs it will:

1. Call `HiveTxnManager.timeoutTxns()`.
2. Acquire a `SHARED` lock on each partition it is compacting.
3. Call `HiveTxnManager.getValidTxns()`. It can use the information about aborted transactions in this call to determine which delta files (or portion of the delta file) it can remove because their writers have or aborted their transactions. Even in the delta file per transaction case this is required because it is possible that a writer moved its files into the base directory and then died before committing.
4. Compact the delta files for the partition.
5. Release the lock on the partition.
6. Call `HiveTxnManager.cleanAbortedTxns()` with information about the partition being compacted.
7. Obtain an `EXCLUSIVE` lock on the partition and remove any delta files that are no longer necessary. It can determine these by looking at the transaction ids of the delta files.