# Design: Storage API Changes to use timestamps

Author: Eric Milkie

## Summary & Motivation

Change the Storage API layer to support providing explicit transaction timestamps to storage engines. These changes are necessary to eventually support multi-document transactions across a sharded system. Code modifications for this project will mostly be limited to files in src/mongo/db/storage/.

## Behavioral Description

After this project is complete, the oplogHack logic used for oplog document visiblity will have been moved from the storage engine glue layer to the WiredTiger layer.

## Detailed Design

Note: SnapshotName is currently a class that wraps a uint64_t. It is the same size as a Timestamp. As part of code cleanup after the completion of this project, we should consider renaming and simplifying this class, as it was originally conceived for named snapshot support for the "read concern" majority project. This can be done as part of the snapshot thread removal, slated for work in the Replica Set Point In Time Reads project.
In addition, there is a desire to make the SnapshotName class large enough to support terms. We can easily make SnapshotName wrap 16 bytes instead of 8, but it will require a [easy] build configuration change to the WiredTiger Commit Timestamps project to support 16 byte timestamps there.

### New function: Supply a timestamp for a transaction

```
virtual Status RecoveryUnit::setTimestamp(SnapshotName timestamp);
```

- The body of this function will call `WT_SESSION::timestamp_transaction()`, passing in the timestamp.
- It is presumed that only transactions that perform writes will call this function, to assign timestamps. MongoDB aborts all read transactions anyway, so assigning timestamps for such transactions would have no effect.
- If this function is called multiple times in a WriteUnitOfWork, the timestamps may only move forward (or stay the same). An error will be returned if setTimestamp() attempts to assign a timestamp that is sooner than a previously assigned timestamp for a transaction.
- In addition to this work, add setCommitTimestamp(), a similar function to setTimestamp(). Instead of calling timestamp_transaction(), it will set a member variable. At commit time in _txnClose, the timestamp will be passed in, if the transaction commits.

### New function: Use a timestamp to read at a point in time

```
virtual Status RecoveryUnit::selectSnapshot(SnapshotName timestamp);
```

- This function may optionally be called prior to calling getCursor() on a RecordStore.
- This function could be called again after calling getCursor(); this would be useful to, say, change to a more recent snapshot at query yield time. This is the current behavior of read-concern-majority, for example.
- This function cannot be called while a transaction is active: `invariant(!_active)`
- The data returned from the cursor will reflect data from transactions that have committed with write timestamps up to and including `time`. If a transaction had multiple timestamps assigned to individual writes, such a transaction could be sliced by the read operation. Slicing is only expected to occur for transactions that were committed by the replication machinery for oplog application.
- The body of this function will set a member variable for the snapshot time in the RecoveryUnit. At the next begin_transaction, it will use the stored time as the read_timestamp parameter.

## Modify existing function: Update the "majority commit" snapshot

```
virtual Status SnapshotManager::setCommittedSnapshot(const
SnapshotName& name);
```

- This function is pre-existing in current code.
- Its behavior will need to change. It will need to inform the storage engine of the majority-committed snapshot time, so that the storage engine can clean up older snapshots.
- Informing the storage engine will consist of calling `set_oldest_timestamp()`, passing in the timestamp parsed out of `name`.

## Remove existing OplogHack visibility code

- This code currently exists to ensure oplog entries always become visible in order, despite their write transactions possibly committing out of order.
- WiredTiger will expose the maximum timestamp such that all lower timestamps are committed with the syntax "`get=all_committed`" to `WT_CONNECTION::transaction_timestamp`. That can be saved across `WiredTigerSessionCache::waitUntilDurable` to determine the known-durable point in the oplog.
- There is also code to ensure that the visibility logic still works in the case that the process crashes and restarts (since the visibility logic is not durable, but the transaction committing of oplog entries could be durable).
- This project work will be to see how much of this logic can be removed from wiredtiger_record_store.cpp.

## Out-of-order oplog entry application on secondaries

- Due to the way a secondary batches operations into multiple threads, replicated oplog entries on secondaries are not applied in the exact order they were applied on primaries.

- This can result in temporary violations of unique key constraints for secondary indexes (the _id index's constraint will never be violated because individual document writes are still serialized in their original order).
- This is currently handled in WiredTiger via a method of storing multiple documents in a single entry in a unique index.
- This method will be problematic for reads that read at a timestamp between the time when the first conflicting document originally got removed and the second document originally got inserted.
- This design is now complete.  It will be included in the Single Shard Point-In-Time Reads project.

### Upgrade/Downgrade Design

No consideration is necessary for upgrading, as this project does not change the on-disk format or the user-facing API.

## Open Questions

- What happens in the following case: Transaction A sets ts 1. Transaction B sets ts 2 then commits. Transaction A raises its ts to 3 then commits. A reader asks to read ts <= 2. Is it able to see the effects of Transaction B even though it can't see Transactions A's write from ts1=?
    - Answer: I am going to assume that the transactions in this question all do writes immediately after setting timestamps (otherwise, the question doesn't make sense, so this is a safe assumption.)  The answer is that readers at point-in-time "2" will see Transaction A's write for ts 1 but not for ts 3, and it will also see Transaction B's write at ts 2.  This will mean that Transaction A is sliced for this particular read.  This situation is only expected to occur for transactions executed by the replication subsystem on secondaries, where multiple timestamps will be assigned in each transaction.