# Distributed MVCC And Transactional SQL Design

Problem Description And Test Case

In Ignite 1.x implementation general reads performed out-of-transaction (such as getAll() or SQL SELECT) do not respect transaction boundaries. This problem is two-fold. First, local node transaction visibility is not atomic with respect to multi-entry read. Committed entry version is made visible immediately after entry is updated. Second, there is no visible version coordination when a read involves multiple nodes. Thus, even if local transaction visibility is made atomic, this does not solve the issue.

The problem can be easily described using a test case. Let's say we have a bank system with a fixed number of accounts and we continuously run random money transfers between random pairs of accounts. In this case the sum of account balances is a system invariant and must be the same for any getAll() or SQL query.

## General Approach Overview

The main idea is that every node should store not only the current (last) entry value, but also some number of previous values in order to allow consistent distributed reads. To do this, we need to introduce a separate node role - transaction version coordinators - which will be responsible for assigning a monotonically growing transaction version as well as maintaining versions of in-progress transactions and in-progress reads. The last committed transaction ID and IDs of pending transactions define the versions that should be visible for any subsequent read. The IDs of pending reads defines the value versions that are no longer needed and can be discarded.

## Version Coordinator(s)

In the initial version of distributed MVCC we will use single transaction coordinator that will define the global transaction order for all transactions in the cluster. The coordinator may be a dedicated node in the cluster. Upon version coordinator failure a new coordinator should be elected in such a way that the new coordinator will start assigning new versions (tx XIDs as well) that is guaranteed to be greater than all previous transaction versions (using two longs: coordinator version which is a topology major version and starting from zero counter).

## TxLog

To be able to determine a state of Tx that created a particular row, a special structure (**TxLog**) is introduced. **TxLog** is a table (can be persistent in case persistence enabled) which contains MVCC version to transaction state mappings.

TxLog is used to keep all the data consistent on cluster crush and recovery as well:

- If a particular Tx has ACTIVE or ROLLED_BACK state on at least one data node it marks as ROLLED_BACK on all nodes.
- If a particular Tx has COMMITTED state on at least one data node it marks as COMMITTED on all nodes.
- If a particular Tx has PREPARED state on all data nodes and all involved partitions are available it marks as COMMITTED on all nodes.
- If a particular Tx has PREPARED state on all data nodes and at least one involved partition is lost (unavailable) it is left in PREPARED state, all the entries are locked for updates until state is changed manually or lost partition become available.

## Internal Data Structures Changes

### BTree leafs structure is changed as follow:

```
|           key part          |           |         |
|-----------------------------|  lockVer  |   link  |
| cache_id | hash |  mvccVer  |           |         |
```

**mvccVer** - MVCC version of transaction which has created the row
**lockVer** - MVCC version of transaction which holds a lock on the row

other fields are obvious.

Rows with the same key are placed from newest to oldest.

### Index BTree leafs structure is changed as follow:

```
|    key part    |
|----------------|
| link | mvccVer |
```

**link** - link to the data
**mvccVer** - XID of transaction who created the row

### Data row payload structure is changed as follow:

```
|              |           |         |            |          |           |             |
|             |
| payload size | next_link | mvccVer | newMvccVer | cache_id | key_bytes | value_bytes | row_version |
expire_time |
|              |           |         |            |          |           |             |
|             |
```

**mvccVer** - TX id which created this row.
**newMvccVer** - TX id which updated this row or NA in this is the last row version (need to decide whether the row is visible for current reader).

other fields are obvious.

## Locks

During **DML** or **SELECT FOR UPDATE** statements Tx acquires locks one by one.

If the row is locked by another tx, current tx saves the context (cursor and current position in it) and register itself as a Tx state listener. As soon as previous Tx is committed or rolled back it fires an event. This means all locks, which are acquired by this Tx, are released. So, waiting on locked row Tx is notified and continues locking/writing.

**TxLog** is used to determine lock state, if Tx with MVCC version equal to row lock version (see BTree leafs structure) is active, the row is locked by this TX. All newly created rows have  lock version the same as its MVCC version, so, all newly created rows are locked by Tx, in scope of which they was created.

Since, as was described above, rows with the same key are placed from newest to oldest, we can determine lock state checking the first row only.

## Transactional Protocol Changes

All the changes are written into cache (disk) at once to be visible for subsequent queries/scans in scope of transaction.

**Two Phase Commit** is used for commit procedure but has no Tx entries (all the changes are already in cache), it is needed just to keep **TxLog** consistent on all data nodes (Tx participants).

Near Tx node has to to notify **Version Coordinator** about final Tx state to make changes visible for subsequent reads.

## Version Coordinator recovery

When MVCC coordinator node fails, a new one is elected among the live nodes – usually the oldest one.

The main goal of the MVCC coordinator failover is to restore an internal state of the previous coordinator in the new one. The internal state of MVCC coordinator consists of two main parts:

- Active transactions list.
- Active queries list.

Due to Ignite partition map exchange design all write transactions should be finished before topology version is changed. Therefore there is no need to restore active transactions list on the new coordinator because all old transactions are either committed or rolled back during topology changing.

The only thing we have to do – is to recover the active queries list. We need this list to avoid old versions cleanup when there are any old queries are running over this old data because it could lead to query result inconsistency. When all old queries are done we can safely continue cleanup old versions.

To restore active queries at the new coordinator the **MvccQueryTracker** object was introduced. Each tracker is associated with a single query. The purpose of the tracker is:

- To mark each query with an unique id for a solid query tracking.
- To hold a query MVCC snapshot.
- To report to the new MVCC coordinator about the associated active query in case of old coordinator failure.
- To send acks to the new coordinator when the associated query is completed.

Active queries list recovery on the new coordinator looks as follows:

1. When old coordinator fails, an exchange process started and the new coordinator is elected.

2. During this process each node sends a list of active query trackers to the new coordinator.
3. New coordinator combine all those lists to the global one.
4. When an old query finishes, the associated query tracker sends an ack to the new coordinator.
5. Coordinator removes this tracker from the global list when ack is received.
6. When global list becomes empty, this means that all old queries are done and we do not have to hold old date versions in our store – cleanup process begins.

## Read (get and SQL)

Each read operation outside an active transaction or in scope of an optimistic transaction gets or uses a previously received **Query Snapshot** (which considered as read version for optimistic Tx. Note: optimistic transactions cannot be used in scope of DML operations).

All requested snapshots are tracked on **Version Coordinator** to prevent cleaning up the rows are read.

All received snapshots are tracked on local node for **Version Coordinator** recovery needs.

**Query Snapshot** is used for versions filtering (**REPEATABLE_READ** semantics).

Each read operation in scope of active pessimistic Tx uses its (transaction) snapshot for versions filtering (**REPEATABLE_READ** semantics).

On failure the node, which requested a **Query Snapshot** but not sent **QueryDone** message to **Version Coordinator**, such snapshot is removed from active queries map. Rows, which are not visible for all other readers, become available for cleaning up.

The row is considered as visible for read operation when it has visible (COMMITTED and in past) MVCC version (create version) and invisible (ACTIVE or ROLLED_BACK or in future) new MVCC version (update version).

## Update (put and DML)

### Update consist of next steps:

1. obtain a lock (write current version into **lockVer** field)
2. delete aborted versions of row if exist (may be omitted for performance reasons)
3. delete previous committed versions (less or equal to cleanup version of Tx snapshot) if exist (may be omitted for performance reasons)
4. delete previous committed versions (less or equal to cleanup version of Tx snapshot) if exist from all secondary indexes (may be omitted for performance reasons)
5. update **new MVCC version** of previous committed row if exist (**newMvccVer** field)
6. add a row with new version
7. add a row with new version to all secondary indexes.

### Delete consists of next steps:

1. obtain a lock (write current version into **lockVer** field)
2. delete aborted versions of row if exist (may be omitted for performance reasons)
3. delete previous committed versions (less or equal to cleanup version of tx snapshot) if exist (may be omitted for performance reasons)
4. delete previous committed versions (less or equal to cleanup version of tx snapshot) if exist from all secondary indexes (may be omitted for performance reasons)
5. update **new MVCC version** of previous committed row if exist (**newMvccVer** field). The row become invisible after Tx commit

## Cleanup of old versions

Invisible for all readers rows are cleaned up by writers, as was described above, or by **Vacuum** procedure (by analogy with PostgreSQL).

During **Vacuum** all checked rows (which are still visible for at least one reader) are actualized with **TxLog** by setting special hint bits (most significant bits in **MVCC operation counter**) which show the state of Tx that created the row.

After all rows are processed, corresponding TxLog records can be deleted as well.

Related documents:

2017.mvcc.vldb.pdf

concurrency-dis...d-databases.pdf

p209-yu.pdf

rethink-mvcc.pdf

Related threads:

Historical rebalance

Suggestion to improve deadlock detection