Splice Machine Snapshot Isolation

Version 1.0

**Creation Date – January 29th 2013**

**Last Update Date –**

Modification History

|  |  |  |  |
| --- | --- | --- | --- |
| **Revision** | **Date** | **Name** | **Description** |
| 1.0 | 01.29.13 | John Leach | Initial Draft |

**Table of Contents**

1. Overview/Background 4

2. Requirements 5

3. Proposed Solution 6

4. Assumptions/Limitations 15

5. Outstanding Issues **Error! Bookmark not defined.**

6. Other Design Considerations 16

6.1 QA Considerations 16

6.2 Documentation/Help Considerations 16

6.3 Hardware/OS Considerations 16

6.4 I18N/L10N Considerations 16

6.5 Public API Considerations 16

6.6 Operational Considerations 16

6.7 Build/Release Considerations 16

6.8 Upgrade/Migration Considerations 16

6.9 Future Considerations 16

# Overview/Background

The Splice Machine would like to have a durable, highly available, and scalable concurrency model based on Snapshot Isolation (SI) extending HBase’s Multi Version Concurrency Control (MVCC). Several papers have been written recently illustrating techniques for performing Snapshot Isolation style semantics on HBase or BigTable. Ferro and Yabandeh have an exhaustive discussion about Snapshot Isolation comparisons in Section 7 of their Paper *A Critique of Snapshot Isolation*.

Although Write Snapshot Isolation is an interesting concept presented in their paper, at this point, Splice Machine will attempt basic Snapshot Isolation. Both Percolator and Omid contain in memory representations of the write set and follow Snapshot Isolation semantics and structure.

Omid and Percolator share the following design characteristics: (1) Timestamp Oracle for generating increasing timestamps for begin and commit timestamps. (2) Single client with buffered writes in memory.



Figure Transactional Use Case Matrix

Figure 1 displays the transactional use case matrix. Omid focuses on the optimization of write performance for small OLTP transactional sets (OLTP Write) while providing adequate OLTP read at the expense of performance on OLAP style scans. Omid optimizes the write performance by utilizing (1) a Transaction Oracle to handle transactional lookup and write-write conflict detection. (2) All rows in active transactions are stored in memory in the transactional Oracle. (3) No two-phase commit protocol (single pass over data). However, large scans of data would need to perform a lookup for every unique transaction in the data set.

Percolator conceded slower write performance for OLTP transactions while providing superior read performance for OTLP and OLAP reads. Both Omid and Percolator ignore large writes (OLAP Write). Percolator’s optimization for read performance centers on a classic two-phase commit protocol. Writes will require multiple passes over the data to obtain a lock and then to roll forward the commit timestamp. The multiple passes over the data can cause slowness during the commit protocol and also increased random seek access to the underlying data store.

Splice Machine will attempt to merge these two approaches to create a durable, highly available, and scalable concurrency model based on Snapshot Isolation.

# Requirements

The Splice Machine will need to support Snapshot Isolation for large numbers of small transactions (OLTP) and for large analytics queries (OLAP). The addition of Snapshot Isolation cannot increase the write or read latency of HBase by more than 20%. The transactional system needs to be durable in the case of process or server failure. The design should support a maximum of 10K transactions per second while attempting to limit the amount of Network I/O required for those transactions.

# Proposed Solution

## Proposed System Schema

Several new system level tables will be added to the existing Derby System Schemas.

TXN

Purpose: This table will maintain the transactional record through its state changes. There will be a single row per transaction.

|  |  |  |
| --- | --- | --- |
| Fields | Data Type | Description |
| rowKey | long | Transaction Start Timestamp or Transaction ID |
| Attributes:Tsc | long | Transaction Commit Timestamp from Zookeeper |
| Attributes:Status | varchar | Active, Error, Commit, Abort |

TABLE\_LOCK

Purpose: This table will represent the exclusive lock on a table included within.

|  |  |  |
| --- | --- | --- |
| Fields | Data Type | Description |
| rowKey | varchar | TableName |

Conglomerate and Index Table Additions

Purpose: Th

|  |  |  |
| --- | --- | --- |
| Fields | Data Type | Description |
| SI:0 | long | Commit Timestamp |
| SI:1 | byte[] | Row Tombstone – Zero byte[] |

## Acquisition of Shared and Exclusive Table Locks

The *TABLE\_LOCK* table will handle the exclusive locks for tables and indexes. There will not be a concept of a shared table lock. When an exclusive lock is requested, a record will be written into the TABLE\_LOCK table. The exclusive lock co-processor will be called to tell each region to block all transactions on the region greater than the transaction for the TABLE\_LOCK. Once all transactions have completed, the exclusive operation can finish and remove its TABLE\_LOCK and call each region coprocessor to notify to release lock.

If a region fails, it will check the TABLE\_LOCK table before coming back online.

## Null Columns

HBase Tables sparsely populate their columns making the storage of nulls extremely efficient. However, use cases where a value transitions from populated to null must be handled in the case of Snapshot Isolation. A Zero byte[] will be used to tombstone null columns when there was a prior value but not a current value.

## Record Tombstone and Rows with all Nulls

HBase Rows will have a record tombstone in the Percolator family that will mark a row as deleted. The tombstone will have the recorded timestamp of the delete like other updates and be written to the column *tombstone*. All rows will have a column populated in the attributes family (-1) that will be a zero byte[] column representing its creation. This will allow the current derby code to create records with all null columns.

## Transaction Creation

Once the transaction is created, Zookeeper will be called to retrieve a universally increasing timestamp for the start of the Transaction (*Tid or Transaction ID*). The Transaction ID will be written to the *TXN* table with a transaction status of Active. Transactions can have the following statuses (Active, Abort, Commit)

Tid 🡨 Zookeeper Timestamp Oracle

Write Ti 🡪 **TXN** table

## Row Level Data Model

In the example row below, one can see the three different start timestamps (10,18,32) referring to the state of the row at different times. These timestamps (HBase versions) are written during the initial write. The commit timestamps (12,24) will be discussed in detail during the write operations.

The latest operation (top) is an example of an operation that does not currently have its commit timestamp set.

## Read Operations

All operations will need to start a transaction if it does not exist (3.5) and DDL Operations will need to acquire an exclusive lock.

Definitions:

Rr = Set of Reads during a transaction

= Set of Columns for a specific read operation

Ts(txni) = Transaction Begin Timestamp

Tc(txni) = Transaction Commit Timestamp

Read Requests: (Ts(txni), Rr)

The *lastCommit* check will make the following progressions. (1) Check the local record’s *lastCommit* Time if it exists for the Transaction in the *SI:0* column. If it does not exist, (2) it will check the Region Server’s transactional cache (Ts(txni) 🡪 Tc(txni)). If the cache does not have the transaction, (3) it will request the timestamp from the TXN table and then place the transaction in the local Region Server’s cache, and then asynchronously roll forward the commit Timestamp.

## Write Operations

All operations will need to start a transaction if it does not exist (3.5) and DDL Operations will need to acquire an exclusive lock. Writes will need to perform write-write conflict checking detailed below:

Write Requests: (Ts(txni), Rw)

write (Rw )

\*Notice that the whole set is sent for callback (One RPC)

The *lastCommit* check will make the following progressions. (1) Check the local record’s *lastCommit* Time if it exists for the Transaction in the *SI:0* column. If it does not exist, (2) it will check the Region Server’s transactional cache (Ts(txni) 🡪 Tc(txni)). If the cache does not have the transaction, (3) it will request the timestamp from the TXN table and then place the transaction in the local Region Server’s cache, and then asynchronously roll forward the commit Timestamp.

The lastCommit Timestamp check will go against the SI Colum Family. This grouping will have block caching enabled to limit seeks.

## Lazy Last Commit Roll Forward

The Key difference between Percolator and Omid is the commit timestamp identification process. The Splice Machine will attempt to split the difference between the two approaches while adjusting to the reality of not having a single, in memory representation of write operations. Specifically, we will not guarantee LastCommit Timestamp update (Similar to both Percolator and Omid). We will attempt to push forward the commit timestamp in a non-durable way to increase performance. We will do this via the following methods…

1. Asynchronous Write Transaction Callback for write sets smaller than the *TransactionWriteThreshold*. These will perform commit timestamp roll forwards and abort record removal for small transactions via a callback on transactional commit.
2. Asynchronous Read Roll Forward Commit timestamp append and abort removal (Similar to Percolator).
3. Region Compaction will trigger Commit Timestamp roll forwards, abort removals, and extra version removal (Algorithm TBD).

## Promotion

Derby allows for locking rows for updates by utilizing the “*select for update*” syntax. Splice Machine will write a version for that row to perform a write operation during the scan. The write operation will block (fail) others attempting to write to the row creating a serializable isolation level for account management type functions.

## Deadlock Detection

Deadlocks cannot occur since blocked transactions fail.

# Assumptions/Limitations

## 4.1 Serializability

The current approach does not support serializability from a database perspective. It could be added by implementing range locks or by following a different structure outlined in <http://www.cs.umn.edu/tech_reports_upload/tr2012/12-004.pdf>.

## 4.2 Point in Time Recoverability

Our current implementation utilizes Zookeeper for schema definitions (Conglomerate Definitions). We would need to change this to an HBase table if we would like to be able to fully recover to a point in time.

# Other Design Considerations

[Subsections here cover additional design thoughts to the feature]

## QA Considerations

Both OLTP and OLAP style transactions will need to be attempted while failing the region server and splitting the region.

## Documentation/Help Considerations

[Thoughts around the impact of this feature on Documentation and Help are useful here]

## Hardware/OS Considerations

The increase in memory usage on the HBase region server could require larger memory footprints for HBase. Zookeeper will need to retrieve and coordinate increasing timestamps. Solid State Drives and advanced networking would increase transactional speed.

## I18N/L10N Considerations

The Snapshot Isolation fields will be numeric in nature similar to our current underlying derby structure.

## Public API Considerations

The current transaction manager (hbase-trx) will be re-purposed to call zookeeper and persist data in the the **TXN** table.

## Operational Considerations

None.

## Build/Release Considerations

Follows the same coprocessor structure as past implementations.

## Upgrade/Migration Considerations

The Snapshot Isolation piece will be difficult to upgrade now and in the future. Anything that changes the base storage structure will require a significant upgrade.

## Future Considerations

The current design should merge with both Serializability and cross data-center transactions if Splice Machine’s customers request such features.

# Outstanding Issues

## Write Conflict Detection Performance

Omid’s model of a *lastCommitTimestamp* cache at the row level might make sense to decrease seeks for determining write-write conflict.

## Race Conditions

Not Clear how we would respond to massive race conditions.

## Zookeeper vs. HBase for Transaction Table

Unclear whether zookeeper or HBase is a better source of record for the actual transactional information.

## Overall Transactional Intelligence

Recording metrics that provide us with transactional characteristics (rows per write, transactional cardinality, etc.)

# Interesting Articles

Percolator: <http://research.google.com/pubs/pub36726.html>

TODO: Need to add Omid paper, Minnesota, etc.