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 Snaphsot 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 uses the TransactionOracle to cache the rows latest commit Timestamp for in-memory conflict detection. This works well with concurrent transactions but clearly will not work for analytics level table scans where the cache cannot contain all the transactions latest commit timestamps. A simple large table scan would require a transaction lookup for each unique latest commit Timestamp in a region.

Percolator’s Two Phase Commit approach follows a classic two-phase commit protocol. The synchronous, multiple passes over the transactional writes would cause significant performance degradation in the case of large write sets.

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%.

# 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.

|  |  |  |
| --- | --- | --- |
| 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, Complete |

PLAN

Purpose: This table maintains the progression of the statement execution

|  |  |  |
| --- | --- | --- |
| Fields | Data Type | Description |
| rowKey | complex | ActivationID and result set number |
| Attributes:activationID | varchar | Activation ID (Class Name) |
| Attributes:resultSetNumber | int | Result Set Number for the activation. |
| Attributes:status | varchar | Wait, Executing, Complete |

## Client Server Architecture

Derby’s JDBC, ODBC, and IJ clients will need to be able to connect via JDBC to a clustered set of Derby Instances running on HBase Region Servers. The Derby instances will be run via an endpoint co-processor created on the running region server. The endpoint co-processor will be extended in the future to retrieve running information (connections, queries, etc.) from the Derby Instance.

The drivers will determine the servers available by connecting to a zookeeper quorum. If a connection disconnects, zookeeper will have to be checked to determine which Region Servers are still represented as ephemeral nodes connected to zookeeper. This structure is illustrated in Figure X.



## SQL Parsing, Planning, and Execution Durability

Since each region server hosting a derby instance runs on commodity hardware that could fail at any time, the SQL parsing, planning, and execution process needs to be durable. If a failure occurs during the parse and planning phase, the SQL will need to be resent via the JDBC client and the parse and plan process retried. Once the Derby Plan transitions to the Splice Optimization Plan, each Operation Branch (the operations that can be parallelized onto a region) must be serialized into a PLAN table to record the completion of each step. Individual failures in the Branch (via coprocessor or remote lookup RPC) have retry mechanisms that will automatically occur. If the Region Server fails during the Splice Optimization Plan’s Execution, the JDBC client will resend the query to another region server Server to pickup execution at the last completed step. If the step that failed was the scan phase (Transfer to JDBC client), the client will remember the number and the record of the last result set returned. This will allow failures to be handled anywhere in the execution chain.

## 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.

## 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.

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 referring to the state of the row at different times.

RowKey: 1

## Read Operations

All operations will need to start a transaction if it does not exist (3.7) and DDL Operations will need to acquire an exclusive lock.

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. If it does not exist, (2) it will check the Region Server’s commit cache. If the cache does not have the transaction, (3)it will hit the TXN table, place the transaction in the local Region Server’s cache, and then asynchronously roll forward the commit Timestamp.

## Write Operation

All operations will need to start a transaction if it does not exist(3.7) 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)

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

## 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.
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).
4. Table Vacuum will trigger Commit Timestamp roll forwards, abort removals, and extra version removal (Algorithm TBD).

## Deadlock Detection

Deadlocks cannot occur since blocked transactions fail.

## Clustered Indexes and Primary Keys

Tables in HBase can be ordered by one and only one row key mechanism. In other database systems (Greenplum, etc.), clustered indexes / tables have been introduced to order the actual table by the clustering key vs. an extra index of the data. Derby does not currently support the “Clustered Index” concept and handles primary key indexes as a unique index on the key. There are three options to consider with the handling of base tables.

1. Allow one clustering index per table that must be presented during table creation DDL (similar to Greenplum). This index cannot be dropped while the table exists.
2. Allow one primary key index that must be presented during table creation DDL. This index cannot be dropped while that table exists.
3. Keep the current derby logic and support primary keys as unique, indexes.

Option 1 would introduce a partitioning concept to our data. This would have significant performance improvements but it is unclear how this partitioning would correspond to HBase and how to change our execution plan based on it.

Option 2 would support a default sort order and uniqueness to our table. The current derby planner would still suggest hitting the index table but our code would need to be modified to the hit the base table. We would need to change the DDL code to handle index creation for primary key in a different way.

Option 3 would keep the current derby logic and planning. This would not be optimal for typical star queries where many to one joins are occurring via foreign keys.

In the short term, Option 3 seems like the easiest for a proof of concept with option 2 occurring before a general release.

## SSTables

SSTables (Sorted String Tables) index structures serializes the index columns as rowkeys into a sorted set of rows. The Sorted string piece now incorporates sortable serialization of most all data types. The index records are not guaranteed to be local to their corresponding base table records. Writes to this index will be sequential writes similar to core HBase tables (fast). The locking and concurrency strategies can mirror the base tables in HBase and be treated as just another table in the transaction.

## SSTables with Locality

TBD

## Lucene / SOLR

Full text indexing rises in important for text-based search but also surprisingly for sparse column population. If you have 1 million possible columns in a table but in general only 16 are populated (Tag Structure), Lucene can be effective in generating that type of index.

<https://github.com/tjake/Solandra>

## Time Series

Most time series have column based storage models coupled with unique handling of dates. This could be implemented with SSTables with a few modifications.

## B-Tree

B-Tree indexes focus on IO reduction via leafs that are generally stored on disk and in-memory for performance. We would need to customize the region server to support such a structure. Generally, these indexes are very effective at reducing I/O for reading rows.

## Bloom Index

Bloom indexes are relatively new on the scene but can be rather effective in quickly discarding blocks of data where specific columns do not exist. Columns and column families can be created atomically on rows to allow values in the base columns to become columns in the index columns. The Bloom filters are loaded into memory on HFile load.

## Replication

<http://hbase.apache.org/replication.html>

## Backups

<https://github.com/oclc/HBase-Backup>

## Recoverability

Rolling back a database to a point in time. The schema changes in Zookeeper would need to be rethought.

# 4 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

[Thoughts about the types of Unit, Functional, Performance, etc. tests that should be written. Also, will these design changes impact/break existing QA test plans?]

## Documentation/Help Considerations

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

## Hardware/OS Considerations

[Are there any hardware-specific or OS-specific issues?]

## I18N/L10N Considerations

[How is this design dealing with Internationalization issues?]

## Public API Considerations

[Are there public API’s? If so add as an additional section]

## Operational Considerations

[Are there any additional considerations for the Operational part of the business?]

## Build/Release Considerations

[Are there any impacts on the build or release processes?]

## Upgrade/Migration Considerations

[When an upgrade or migration occurs, what must be taken into consideration? Are there scripts to accommodate migration? Etc]

## Future Considerations

[What features or design ideas came up that we should consider for the future? List these here to help out Product Management, Development, etc]

# Outstanding Issues

[Remaining issues that need to be resolved still]

# Appendix A

@Override

**public** **void** start(CoprocessorEnvironment e) **throws** IOException {

SpliceLogUtils.*info*(*LOG*, "Starting the coprocessor CoProcessor %s", SpliceDerbyRegionObserver.**class**);

**super**.start(e);

**synchronized** (**this**) {

**if** (*server* == **null**) {

**try** {

*server* = **new** NetworkServerControl();

*server*.start(**new** DerbyOutputLoggerWriter()); // This will log to log4j

SpliceLogUtils.*info*(*LOG*, *server*.getSysinfo());

} **catch** (Exception exception) {

SpliceLogUtils.*logAndThrow*(*LOG*, "Could Not Start Derby - Catastrophic", **new** IOException(exception));

}

}

}

}

/\*\*

\* Logs the stop of the observer.

\*/

@Override

**public** **void** stop(CoprocessorEnvironment e) **throws** IOException {

SpliceLogUtils.*info*(*LOG*, "Stopping the CoProcessor %s",SpliceDerbyRegionObserver.**class**);

**super**.stop(e);

**synchronized** (**this**) {

**if** (*server* != **null**) {

**try** {

*server* = **null**;

} **catch** (Exception exception) {

SpliceLogUtils.*logAndThrow*(*LOG*, "Could Not Start Derby - Catastrophic", **new** IOException(exception));

}

}

}

}