Splice Machine Primary Key and Secondary Indexes

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

# Requirements

# Proposed Solution

## 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

# 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));

}

}

}

}