Splice Machine Documentation Test

Table of Contents

Splice Machine Developer's Guide

Analytics and Machine Learning

Tuning and Debugging

Splice Machine Developer's Guide

This chapter describes how to do development work with your Splice Machine Database. It is divided into these sections:

Section	Description
Fundamentals	Using the basic developer features of Splice Machine, including: • Running Transactions • Working with Date and Time Values • Using Database Triggers • Using Foreign Keys • Using Window Functions • Using Temporary Tables • Roles and Authorization • Using Spark Libraries • Using the Virtual Table Interface • Using External Tables
Data Ingestion and Streaming	We provide a number of tutorials in the Data Ingestion section of our *Tutorials Guide*, which guide you through specific ingestion and streaming tasks.
Connecting Your Database	The Connecting section of our *Tutorials Guide* includes a number of mini-tutorials that guide you through using our JDBC and ODBC drivers to connect to your Splice Machine database with various programming languages. The Connecting section of our *Tutorials Guide* also includes a number of mini-tutorials that guide you through connecting various Business Intelligence tools to your Splice Machine database.
Functions and Stored Procedures	Provides an overview of writing and using functions and stored procedures in Splice Machine, in these topics: • Using Functions and Stored Procedures • How to Write Functions and Stored Procedures • Storing and Updating Functions and Stored Procedures

Section	Description
[Tuning and Debugging Splice Machine](#)	Describes the tools available to tune and debug your Splice Machine queries: • [Optimizing Queries](#) • [Using Splice Machine Statistics](#) • [Using Explain Plan](#) • [Splice Machine Logging](#) • [Debugging Splice Machine](#)

Splice Machine Fundamentals

This section contains the following fundamental topics for Splice Machine developers:

Topic	Describes
Importing Data Tutorial	How to import data into your Splice Machine database, using built-in procedures
Running Transactions	Introduces you to the basics of running transactions with Splice Machine.
Working with Dates	Provides an overview of working with date and time values in Splice Machine.
Using Database Triggers	Describes database triggers and how you can use them with Splice Machine.
Using Foreign Keys	Describes our implementation of foreign keys and how our implementation ensures referential integrity.
Using Window Functions	A quick summary of window functions, as implemented in Splice Machine SQL.
Using Temporary Tables	Describes how to use temporary tables with Splice Machine.
Roles and Authorization	Describes how Splice Machine authorizes which operations can be performed by which users.
Using Spark Libraries	Describes how to interface with Spark libraries in Splice Machine.
Using the VTI Interface	Allows you to use an SQL interface with data that is external to your database.
Using External Tables	Describes the use of external tables (which are stored as flat files outside of your database) with Splice Machine.
Using HCatalog	How to use Splice Machine with HCatalog.
Using MapReduce	The Splice Machine MapReduce API provides a simple programmatic interface for using MapReduce with HBase and taking advantage of the transactional capabilities that Splice Machine provides.

Topic	Describes
[Working with HBase](#)	Working in HBase with Splice Machine.

Running Transactions In Splice Machine

Splice Machine is a fully transactional database that supports ACID transactions. This allows you to perform actions such as *commit* and *rollback*; in a transactional context, this means that the database does not make changes visible to others until a *commit* has been issued.

This topic includes brief overview information about transaction processing with Splice Machine, in these sections:

- » [Transactions Overview](#)
  - » [ACID Transactions](#) describes what ACID transactions are and why they're important.
  - » [MVCC and Snapshot Isolation](#) describes what snapshot isolation is and how it works in Splice Machine.
- » [Using Transactions](#)
  - » [Committing and Rolling Back Transaction Changes](#) introduces autocommit, commit, and rollback of transactions.
  - » [A Simple Transaction Example](#) presents an example of a transaction using the `splice>` command line interface.
  - » [Using Savepoints](#) describes how to use savepoints within transactions.

Transactions Overview

A transaction is a unit of work performed in a database; to maintain the integrity of the database, each transaction must:

- » complete in its entirety or have no effect on the database
- » be isolated from other transactions that are running concurrently in the database
- » produce results that are consistent with existing constraints in the database
- » write its results to durable storage upon successful completion

ACID Transactions

The properties that describe how transactions must maintain integrity are *A*tomicity, *C*onsistency, *I*solation, and *D*urability. Transactions adhering to these properties are often referred to as *ACID* transactions. Here's a summary of ACID transaction properties:

Property	Description
Atomicity	Requires that each transaction be atomic, i.e. all-or-nothing: if one part of the transaction fails, the entire transaction fails, and the database state is left unchanged. Splice Machine guarantees atomicity in each and every situation, including power failures, errors, and crashes.
Consistency	Ensures that any transaction will bring the database from one valid state to another. Splice Machine makes sure that any data written to the database must be valid according to all defined rules, including constraints, cascades, triggers, and any combination thereof.

Property	Description
Isolation	Ensures that the concurrent execution of transactions results in a system state that would be obtained if transactions were executed serially. Splice Machine implements snapshot isolation using MVCC to guarantee that this is true.
Durability	Ensures that once a transaction has been committed, it will remain so, even in the event of power loss, crashes, or errors. Splice Machine stores changes in durable storage when they are committed.

MVCC and Snapshot Isolation

Splice Machine employs a lockless *snapshot isolation design* that uses *Multiple Version Concurrency Control (MVCC)* to create a new version of the record every time it is updated and enforce consistency. Database systems use concurrency control systems to manage concurrent access. The simplest control method is to use locks that make sure that the writer is finished before any reader can proceed; however, this approach can be very slow. With snapshot isolation, each transaction has its own virtual snapshot of the database, which means that multiple transactions can operate concurrently without creating deadlock conditions.

When Splice Machine needs to update an item in the database, it doesn't actually overwrite the old data value. Instead, it creates a new version with a new timestamp. Which means that readers have access to the data that was available when they began reading, even if that data has been updated by a writer in the meantime. This is referred to as *point-in-time consistency* and ensures that:

» Every transaction runs in its own *transactional context*, which includes a snapshot of the database from when the transaction began.

» Every read made during a transaction will see a consistent snapshot of the database.

» A transaction can only commit its changes if they do not conflict with updates that have been committed while the transaction was running.

Reading and Writing Database Values During a Transaction

When you begin a transaction, you start working within a *transactional context* that includes a snapshot of the database. The operations that read and write database values for your transaction modify your transactional context. When your transaction is complete, you can commit those modifications to the database. The commit of your transaction's changes succeeds unless a *write-write conflict* occurs, which happens when your transaction attempts to commit an update to a value, and another update to that value has already been committed by a transaction that started before your transaction.

This means that the following statements are true with regard to reading values from and writing values to the database during a transaction:

» When you read a value during a transaction, you get the value that was most recently set within your transactional context. If you've not already set the value within your context, then this is the value that had been most recently committed in the database before your transaction began (and before your transactional context was established).

» When you write a value during a transaction, the value is set within your transactional context. It is only written to the database when you commit the transaction; that time is referred to as the *commit timestamp* for your transaction. The value changes that you commit then become visible to transactions that start after your transaction's commit timestamp (until another transaction modifies the value).

» If two parallel transactions attempt to change the same value, then a *write-write conflict* occurs, and the commit of the

transaction that started later fails.

A Snapshot Isolation Example

The following diagram shows an example of snapshot isolation for a set of transactions, some of which are running in parallel:
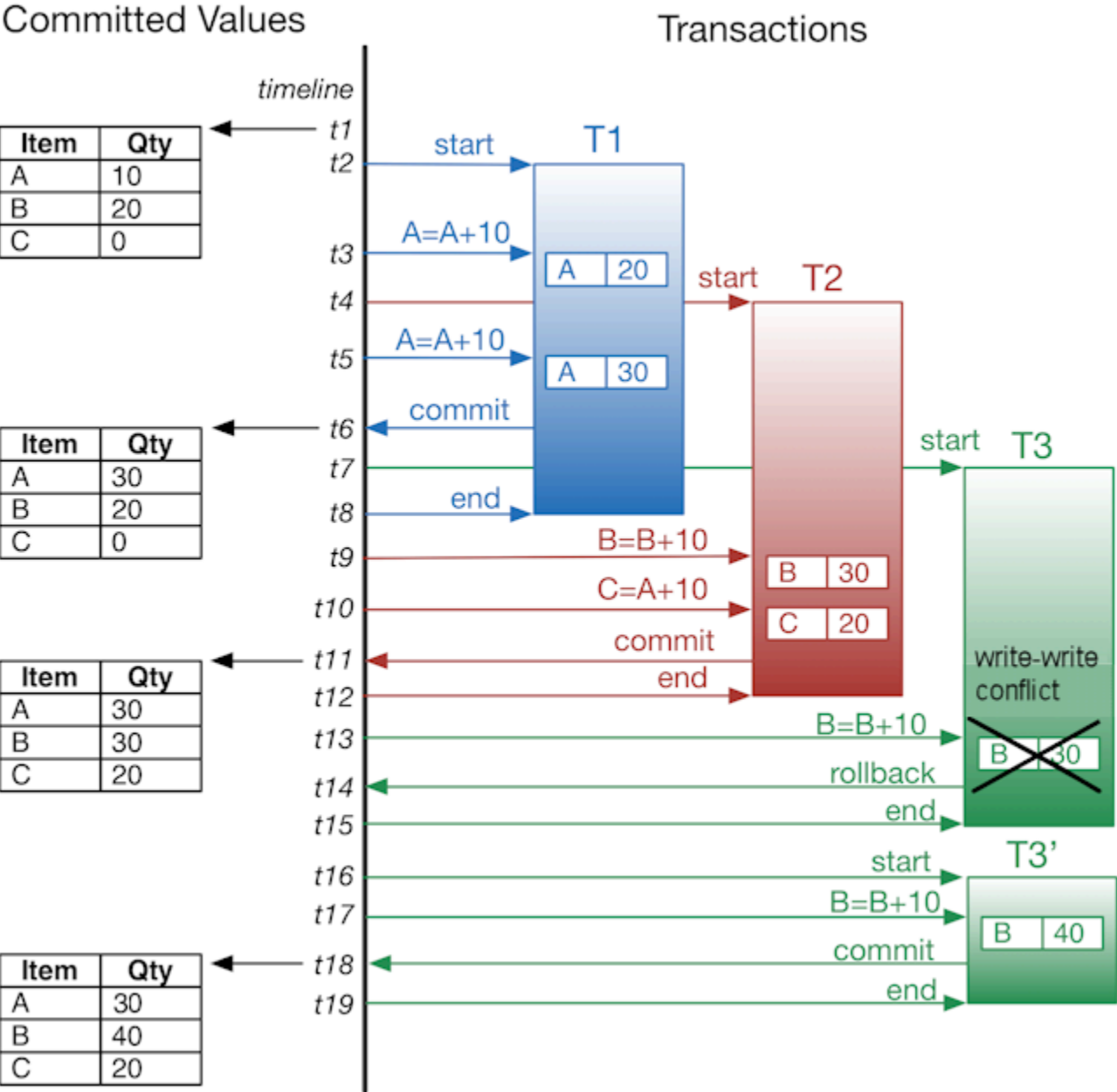
Here's a tabular version of the same transactional timeline, showing the values committed in the database over time, with added commentary:

Time	Committed Values			Transactions				Comments
	A	B	C	T1	T2	T3	T3'	
t1	10	20	0					
t2				**T1 Start**				T1 starts. The starting values within its transactional context are: A=10, B=20, C=0.
t3				**A=A+10 [A=20]**				T1 modifies the value of A within its context.
t4					**T2 Start**			T2 starts. The starting values within its transactional context are the same as for T1: A=10, B=20, C=0.
t5				**A=A+10 [A=30]**				T1 again modifies the value of A within its context
t6	30	20	0	**Commit**				T1 commits its modifications to the database.
t7						**T3 Start**		T3 starts. The starting values within its transactional context include the commits from T1: A=30, B=20, C=0.
t8				**T1 End**				
t9					**B=B+10 [B=30]**			T2 modifies the value of B within its context.
t10					**C=A+10 [C=20]**			T2 modifies the value of C within its context; note that this computation correctly uses the value of A (10) that had been committed prior to the start of T2.
t11	30	30	20		**Commit**			T2 commits its changes.
t12					**T2 End**			
t13						**B=B+10 [B=30]**		T3 modifies B; since its context includes the value of B before T2 committed, it modifies the original value of B [B=20] in its own context.

Time	Committed Values			Transactions				Comments
	A	B	C	T1	T2	T3	T3'	
t14						**Rollback**		T3 attempts to commit its changes, which causes a *write-write conflict*, since T2 already committed an update to value B after T3 started. T3 rolls back and resets.
t15						**T3 End**		
t16							**T3' Start**	T3 reset (T3') starts. The starting values within its transactional context include the commits from T1 and T2: `A=30,  B=30, C=20`.
t17							**B=B+10 [B=40]**	T3 modifies the value of B, which has been updated and committed by T2.
t18	40	40	20				**Commit**	T3 commits its changes.
t19							**T3' End**	

Using Transactions

This section describes using transactions in your database, in these subsections:

» [Committing and Rolling Back Transaction Changes](#) introduces autocommit, commit, and rollback of transactions.

» [A Simple Transaction Example](#) presents an example of a transaction using the splice> command line interface.

» [Using Savepoints](#) describes how to use savepoints within transactions.

» [Using Rollback versus Rollback to Savepoint](#) discusses the differences between rolling back a transaction, and rolling back to a savepoint.

Committing and Rolling Back Transaction Changes

Within a transactional context, how the changes that you make are committed to the database depends on whether `autocommit` is enabled or disabled:

autocommit status	How changes are committed and rolled back
enabled	Changes are automatically committed whenever the operation completes successfully.
	If an operation reports any error, the changes are automatically rolled back.
disabled	Changes are only committed when you explicitly issue a commit command.
	Changes are rolled back when you explicitly issue a rollback command.

> Autocommit is enabled by default. You typically disable autocommit when you want a block of operations to be committed atomically (all at once) instead of committing changes to the database after each operation.

You can turn autocommit on and off by issuing the autocommit on or autocommit off commands at the splice> prompt.

For more information, see these topics in the *Command Line Reference* section of this book:

» autocommit command

» commit command

» rollback command

A Simple Transaction Example

Here is a simple example. Enter the following commands to see *commit* and *rollback* in action:

```
splice> create table myTbl (i int);
splice> autocommit off;                 - commits must be made explicitly
splice> insert into myTbl  values 1,2,3; - inserted but not visible to others
splice> commit;                         - now committed to the database
splice> select * from myTbl;            - verify table contents
splice> insert into myTbl  values 4,5;  - insert more datasplice> select * from myT
bl;            - verify table contents
splice> rollback;                       - roll back latest insertions
splice> select * from myTbl;            - and verify again
...
```

You can turn autocommit back on by issuing the command: autocommit on;

Using Savepoints

Splice Machine supports the JDBC 3.0 Savepoint API, which adds methods for setting, releasing, and rolling back to savepoints within a transaction. Savepoints give you additional control over transactions by allowing you to define logical rollback points within a transaction, which effectively allows you to specify sub-transactions (also known as nested transactions).

You can specify multiple savepoints within a transaction. Savepoints are very useful when processing large transactions: you can implement error recovery schemes that allow you to rollback part of a transaction without having to abort the entire transaction.

You can use these commands to work with Savepoints:

›› create a savepoint with the `savepoint` command

›› release a savepoint with the `release savepoint` command

›› roll a transaction back to an earlier savepoint with the `rollback to savepoint` command

Example

First we'll create a table, turn autocommit off, and insert some data into the table. We then create a savepoint, and verify the contents of our table:

```
splice> CREATE TABLE myTbl(i int);
0 rows inserted/updated/deleted
splice> AUTOCOMMIT OFF;
splice> INSERT INTO myTbl VALUES 1,2,3;
3 rows inserted/updated/deleted
splice> SAVEPOINT savept1;
0 rows inserted/updated/deleted
splice> SELECT * FROM myTbl;
I
-----------
1
2
3

3 rows selected
```

Next we add new values to the table and again verify its contents:

```
splice> INSERT INTO myTbl VALUES 4,5;
2 rows inserted/updated/deleted
splice> SELECT * FROM myTbl;
I
-----------
1
2
3
4
55 rows selected
```

Now we roll back to our savepoint, and verify that the rollback worked:

```
splice> ROLLBACK TO SAVEPOINT savept1;
0 rows inserted/updated/deleted
splice> SELECT * FROM myTbl;
I
-----------
1
2
3

3 rows selected
```

And finally, we commit the transaction:

```
COMMIT;
```

Using Rollback Versus Rollback to Savepoint

There's one important distinction you should be aware of between rolling back to a savepoint versus rolling back the entire transaction:

» When you perform a `rollback`, Splice Machine aborts the entire transaction and creates a new transaction,

» When you perform a `rollback to savepoint`, Splice Machine rolls back part of the changes, but does not create a new transaction.

> Remember that this distinction also holds in a multi-tenant environment. In other words:
>
> » If two users are making modifications to the same table in separate transactions, and one user does a `rollback`, all changes made by that user prior to that rollback are no longer in the database.
>
> » Similarly, if two users are making modifications to the same table in separate transactions, and one user does a `rollback to savepoint`, all changes made by that user since the savepoint was established are no longer in the database.

See Also

» [autocommit](#) command

» [commit](#) command

» [release savepoint](#) command

» [rollback](#) command

» [rollback to savepoint](#) command

» savepoint command

Working With Date and Time Values

This topic provides an overview of working with dates in Splice Machine.

For date and time values to work as expected in your database, you must make sure that all nodes in your cluster are set to the same time zone; otherwise the data you read from your database may differ, when you communicate with different servers!

> Please contact your system administrator if you have any questions about this.

Date and Time Functions

Here is a summary of the `TIMESTAMP` functions included in this release of Splice Machine:

Function	Description
CURRENT_DATE	Returns the current date as a `DATE` value.
DATE	Returns a `DATE` value from a `DATE` value, a `TIMESTAMP` value, a string representation of a date or timestamp value, or a numeric value representing elapsed days since January 1, 1970.
DAY	Returns an integer value between 1 and 31 representing the day portion of a `DATE` value, a `TIMESTAMP` value, or a string representation of a date or timestamp value.
EXTRACT	Extracts various date and time components from a date expression.
LAST_DAY	Returns a `DATE` value representing the date of the last day of the month that contains the input date.
MONTH	Returns an integer value between 1 and 12 representing the month portion of a `DATE` value, a `TIMESTAMP` value, or a string representation of a date or timestamp value.
MONTH_BETWEEN	Returns a decimal number representing the number of months between two dates.
MONTHNAME	Returns the month name from a date expression.
NEXT_DAY	Returns the date of the next specified day of the week after a specified date.
NOW	Returns the current date and time as a `TIMESTAMP` value.
QUARTER	Returns the quarter number (1-4) from a date expression.
TIMESTAMP	Returns a timestamp value from a `TIMESTAMP` value, a string representation of a timestamp value, or a string of digits representing such a value.

Function	Description
TIMESTAMPADD	Adds the value of an interval to a `TIMESTAMP` value and returns the sum as a new timestamp.
TIMESTAMPDIFF	Finds the difference between two timestamps, in terms of the specfied interval.
TO_CHAR	Returns string formed from a DATE value, using a format specification.
TO_DATE	Returns a `DATE` value formed from an input string representation, using a format specification.
WEEK	Returns the week number (1-53) from a date expression.
YEAR	Returns an integer value between 1 and 9999 representing the year portion of a `DATE` value, a `TIMESTAMP` value, or a string representation of a date or timestamp value.

Date Arithmetic

Splice Machine provides simple arithmetic operations addition and subtraction on date and timestamp values. You can:

» find a future date value by adding an integer number of days to a date value

» find a past date value by subtracting an integer number of days from a date value

» subtract two date values to find the difference, in days, between those two values

Here's the syntax for these inline operations:

```
    dateValue { "+" | "-" } numDays
|   numDays   '+' dateValue
|   dateValue '-' dateValue
```

dateValue
> A TIMESTAMP value. This can be a literal date value, a reference to a date value in a table, or the result of a function that produces a date value as its result.

numDays
> An integer value expressing the number of days to add or subtract to a date value.

Result Types

The result type of adding or subtracting a number of days to/from a date value is a date value of the same type (`DATE` or `TIMESTAMP`) as the `dateValue` operand.

The result type of subtracting one date value from another is the number of days between the two dates. This can be a positive or negative integer value.

Notes

A few important notes about these operations:

» Adding a number of days to a date value is commutative, which means that the order of the `dateValue` and `numDays` operands is irrelevant.

» Subtraction of a number of days from a date value is not commutative: the left-side operand must be a date value.

» Attempting to add two date values produces an error, as does attempting to use a date value in a multiplication or division operation.

Examples

This section presents several examples of using date arithmetic. We'll first set up a simple table that stores a string value, a `DATE` value, and a `TIMESTAMP` value, and we'll use those values in our example.

```
splice> CREATE TABLE date_math (s VARCHAR(30), d DATE, t TIMESTAMP);
0 rows inserted/updated/deleted

splice> INSERT INTO date_math values ('2012-05-23 12:24:36', '1988-12-26', '2000-0
6-07 17:12:30');
1 row inserted/updated/deleted
```

Example 1: Add a day to a date column and then to a timestamp column

```
splice> select d + 1 from date_math;
1
----------
1988-12-27
1 row selected

splice> select 1+t from date_math;
1
----------
2000-06-08 17:12:30.0
1 row selected
```

Example 2: Subtract a day from a timestamp column

```
splice> select t - 1 from date_math;
1
----------------------------
2000-06-06 17:12:30.0
1 row selected
```

Example 3: Subtract a date column from the result of the CURRENT_DATE function

```
splice> select current_date - d from date_math;
1
-----------
9551
1 row selected
```

Example 4: Additional examples using literal values

```
splice> values  date('2011-12-26') + 1;
1
----------
2011-12-27
1 row selected

splice> values  date('2011-12-26') - 1;
1
----------
2011-12-25
1 row selected

splice> values  timestamp('2011-12-26', '17:13:30') + 1;
1
---------------------------
2011-12-27 17:13:30.0
1 row selected

splice> values  timestamp('2011-12-26', '17:13:30') - 1;
1
---------------------------
2011-12-25 17:13:30.0
1 row selected

splice> values  date('2011-12-26') - date('2011-06-05');
1
----------
204
1 row selected

splice> values  date('2011-06-05') - date('2011-12-26');
1
----------
-204
1 row selected


splice> values  timestamp('2015-06-07', '05:06:00') - current_date;
1
----------
108
1 row selected

splice> values  timestamp('2011-06-05', '05:06:00') - date('2011-12-26');
1
----------
-203
1 row selected
```

See Also

All of the following are in the *SQL Reference Manual*:

» CURRENT_DATE

» DATE

» DATE

» DAY

» EXTRACT

» LASTDAY

» MONTH

» MONTH_BETWEEN

» MONTHNAME

» NEXTDAY

» NOW

» QUARTER

» TIME

» TIMESTAMP

» TO_CHAR

» TO_DATE

» WEEK

Using Database Triggers

This topic describes database triggers and how you can use them with Splice Machine.

About Database Triggers

A database trigger is procedural code that is automatically executed in response to certain events on a particular table or view in a database. Triggers are mostly used for maintaining the integrity of the information on the database; they are most commonly used to:

>> automatically generate derived column values

>> enforce complex security authorizations

>> enforce referential integrity across nodes in a distributed database

>> enforce complex business rules

>> provide transparent event logging

>> provide sophisticated auditing

>> gather statistics on table access

Components of a Trigger

Each trigger has two required components:

Component	Description
Triggering event (or statement)	The SQL statement that causes a trigger to be fired. This can be one of the following types of statement: >> `INSERT` >> `UPDATE` >> `DELETE`
Trigger action	The procedure that contains the SQL statements to be executed when a triggering statement is issued and any trigger restrictions evaluate to `TRUE`. A trigger action is one of the following: >> arbitrary SQL >> a call to a <u>user-defined stored procedure</u>

When a Trigger Fires

You can define both statement and row triggers as either *before triggers* or *after triggers*:

Trigger Type	Description
Before Triggers	A before trigger fires before the statement's changes are applied and before any constraints have been applied.
After Triggers	An after trigger fires after all constraints have been satisfied and after the changes have been applied to the target table.

How Often a Trigger Fires

You can define triggers as either *statement triggers* or *row triggers*, which defines how often a trigger will fire for a triggering event.

Trigger Type	Description
Statement Triggers	A statement trigger fires once per triggering event, regardless of how many rows (including zero rows) are modified by the event.
Row Triggers	A row trigger fires once for each row that is affected by the triggering event; for example, each row modified by an `UPDATE` statement. If no rows are affected by the event, the trigger does not fire.

> **NOTE:** Triggers are statement triggers by default. You specify a row trigger in the `FOR EACH` clause of the `CREATE TRIGGER` statement.

Examples

This section presents examples of using database triggers.

Example 1: Row Level AFTER Trigger

This example shows a row level trigger that is called after a row is updated in the `employees` table. The action of this trigger is to insert one record into the audit trail table (`employees_log`) for each record that gets updated in the `employees` table.

```
CREATE TRIGGER log_salary_increase
AFTER UPDATE ON employees FOR EACH ROW
INSERT INTO employees_log
    (emp_id, log_date, new_salary, action)
    VALUES (:new.empno, CURRENT_DATE, :new.salary, 'NEW SALARY');
```

If you then issue following statement to update salaries of all employees in the PD department:

```
UPDATE employees
 SET salary = salary + 1000.0
 WHERE department = 'PD';
```

Then the trigger will fire once (and one audit record will be inserted) for each employee in the department named `PD`.

Example 2: Statement Level After Trigger

This example shows a statement level trigger that is called after the `employees` table is updated. The action of this trigger is to insert exactly one record into the change history table (`reviews_history`) whenever the `employee_reviews` table is updated.

This example shows a row level trigger that is called after a row is updated in the `employees` table. The action of this trigger is to insert one record into the audit trail table (`employees_log`) for each record that gets updated in the `employees` table.

```
CREATE TRIGGER log_salary_increase
AFTER UPDATE ON employees referencing NEW as NEW FOR EACH ROW
INSERT INTO employees_log
    (emp_id, log_date, new_salary, action)
    VALUES (NEW.empno, CURRENT_DATE, NEW.salary, 'NEW SALARY');
```

If you then issue the same Update statement as used in the previous example:

```
UPDATE employees SET salary = salary + 1000.0
WHERE department = 'PD';
```

Then the trigger will fire once and exactly one record will be inserted into the `employees_log` table, no matter how many records are updated by the statement.

Example 3: Statement Level Before Trigger

This example shows a row level trigger that is called before a row is inserted into the `employees` table.

```
CREATE TRIGGER empUpdateTrig
BEFORE UPDATE ON employees
    FOR EACH STATEMENT SELECT ID FROM myTbl;
```

See Also

- » [CREATE TRIGGER](#)
- » [DROP TRIGGER](#)
- » [Foreign keys](#)
- » [UPDATE](#)
- » [WHERE](#)

Foreign Keys and Referential Integrity

This topic describes the Splice Machine implementation of *foreign keys* and how our implementation ensures referential integrity.

See our *SQL Reference Manual* for full reference information about defining foreign keys using using <u>constraint clauses</u> when <u>creating </u>a database table.

About Foreign Keys

A foreign key is a column or group of columns in a relational database table that provides a link between data in two tables. A foreign key acts as a cross-reference between tables in that it references the primary key or unique key columns of another table, and thus establishes a link between them.

The purpose of a foreign key is to identify a particular row of the referenced table; as such, the foreign key must be equal to the key in some row of the primary table, or else be `null`. This rule is called a *referential integrity constraint between the two tables*, and is usually abbreviated as just *referential integrity*.

Maintaining Referential Integrity

To maintain referential integrity, Splice Machine ensures that database operations do not violate foreign key constraints, including not allowing any operations that will cause a foreign key to not correspond to a row in the referenced table. This can happen when a row is inserted, updated, or deleted in either table.

For example, suppose you have:

» A table named `Players` with primary key `player_id`. This table is called the *parent table* or *referenced table*.

» A second table named `PlayerStats` has a foreign key, which is also a column named `player_id`. This table is called the *child table* or *referencing table*.

The `player_id` column in the *referencing* table is the foreign key that references the primary key `player_id` in the *referenced* table.

When you insert a new record into the referencing `PlayerStats` table, the insertion must satisfy the foreign key constraint, which means that it must include a `player_id` value that is present in the referenced `Players` table. If this is not so, the insert operation fails in order to maintain the table's referential integrity.

About Foreign Key Constraints

You can define a foreign key constraint on a table when you create the table with the <u>`CREATE TABLE`</u> statement. Foreign key constraints are always immediate: a violation of a constraint immediately throws an exception.

Here's an example of defining a foreign key, in which we use the `REFERENCES` clause of a column definition in a `CREATE TABLE` statement:

```
CREATE TABLE t1 (c1 NUMERIC PRIMARY KEY);

CREATE TABLE t2 (
    c1 NUMERIC PRIMARY KEY,
    c2 NUMERIC REFERENCES t1(c1) );
```

And here's an example that uses the CONSTRAINT clause to name the foreign key constraint:

```
CREATE TABLE t3 (
    c1 NUMERIC,
    c2 NUMERIC,
    CONSTRAINT t1_fkey FOREIGN KEY (c1) REFERENCES t1);
```

You can also define a foreign key on a combination of columns:

```
CREATE TABLE dept_20
    (employee_id INT, hire_date DATE,
    CONSTRAINT fkey_empid_hiredate
    FOREIGN KEY (employee_id, hire_date)
    REFERENCES dept_21(employee_id, start_date));
```

See Also

- » ALTER TABLE

- » CONSTRAINT

- » CREATE TABLE

- » Using Database Triggers

Splice Machine Window Functions

An SQL *window function* performs a calculation across a set of table rows that are related to the current row, either by proximity in the table, or by the value of a specific column or set of columns; these columns are known as the *partition*.

This topic provides a very quick summary of window functions, as implemented in Splice Machine. For more general information about SQL window functions, we recommending visiting some of the sources listed in the Additional Information section at the end of this topic.

Here's a quick example of using a window function to operate on the following table:

OrderID	CustomerID	Amount
123	1	100
144	1	250
167	1	150
202	1	250
209	1	325
224	1	125
66	2	100
94	2	200
127	2	300
444	2	400

This query will find the first Order ID for each specified Customer ID in the above table:

```
SELECT OrderID, CustomerID,
    FIRST_VALUE(OrderID) OVER (
        PARTITION BY CustomerID
        ORDER BY OrderID
        ROWS BETWEEN UNBOUNDED PRECEDING AND CURRENT ROW )
AS FirstOrderID
FROM ORDERS
WHERE CustomerID IN (1,2);
```

This works by partitioning (grouping) the selected rows by CustomerID, ordering them for purposes of applying the function to the rows in the partition, and then using the `FIRST_VALUE` window function to evaluate the OrderID values in each partition and find the first value in each. The results for our sample table are:

OrderID	CustomerID	FirstOrderID
123	1	123
144	1	123
167	1	123
202	1	123
209	1	123
224	1	123
66	2	66
94	2	66
127	2	66
444	2	66

See the Window Frames section below for a further explanation of this query.

About Window Functions

Window functions:

» Operate on a window, or set of rows. The rows considered by a window function are produced by the query's `FROM` clause as filtered by its `WHERE`, `GROUP BY`, and `HAVING` clauses, if any. This means that any row that doesn't meet the `WHERE` condition is not seen by a window function.

» Are similar to aggregate functions, except that a window function does not group rows into a single output row. Instead, a window function returns a value for every row in the window. This is sometimes referred to as tuple-based aggregation.

» The values are calculated from the set of rows in the window.

» Always contain an `OVER` clause, which determines how the rows of the query are divided and sequenced for processing by the window function.

» The `OVER` clause can contain a `PARTITION` clause that specifies the set of rows in the table that form the window, relative to the current row.

» The `OVER` clause can contain an optional `ORDER BY` clause that specifies in which order rows are processed by the window function. This `ORDER BY` clause is independent of the `ORDER BY` clause that specifies the order in which rows are output.

Note that the ROW NUMBER **must contain** an `ORDER BY` clause.

» The `OVER` clause can also contain an optional *frame clause* that further restricts which of the rows in the partition are sent to the function for evaluation.

About Windows, Partitions, and Frames

Using window functions can seem complicated because they involve a number of overlapping terms, including *window*, *sliding window*, *partition*, *set*, and *window frame*. An additional complication is that window frames can be specified using either *rows* or *ranges*.

Let's start with basic terminology definitions:

Terms	Description
window function	A function that operates on a set of rows and produces output for each row.
window partition	The grouping of rows within a table. Note that window partitions retains the rows, unlike aggregates,
window ordering	The sequence of rows within each partition; this is the order in which the rows are passed to the window function for evaluation.
window frame	A frame of rows within a window partition, relative to the current row. The window frame is used to further restrict the set of rows operated on by a function, and is sometimes referred to as the *row or range* clause.
OVER clause	This is the clause used to define how the rows of the table are divided, or partitioned, for processing by the window function. It also orders the rows within the partition. See the The OVER Clause section below for more information.
partitioning clause	An optional part of an `OVER` clause that divides the rows into partitions, similar to using the `GROUP BY` clause. The default partition is all rows in the table, though window functions are generally calculated over a partition. See the The Partition Clause section below for more information.
ordering clause	Defines the ordering of rows within each partition. See the The Order Clause section below for more information.

Terms	Description
frame clause	Further refines the set of rows when you include an ORDER BY clause in your window function specification, by allowing you to include or exclude rows or values within the ordering. See the The Frame Clause section below for examples and more information.

The OVER Clause

A window function alway contains an OVER clause, which determines how the rows of the query are divided, or partitioned, for processing by the window function.

```
expression OVER(
     [partitionClause]
     [orderClause]
     [frameClause] );
```

expression

Any value expression that does not itself contain window function calls.

> **NOTE:** When you use an aggregate function such as AVG with an OVER clause, the aggregated value is computed per partition.

The Partition Clause

The partition clause, which is optional, specifies how the window function is broken down over groups, in the same way that GROUP BY specifies groupings for regular aggregate functions. Some example partitions are:

» departments within an organization

» regions within a geographic area

» quarters within years for sales

> **NOTE:** If you omit the partition clause, the default partition, which contains all rows in the table, is used.However, since window functions are used to perform calculations over subsets (partitions) of rows in a table, you generally should specify a partition clause.

Syntax

```
PARTITION BY expression [, ...]
```

expression [,…]

A list of expressions that define the partitioning.

If you omit this clause, there is one partition that contains all rows in the entire table.

Here's a simple example of using the partition clause to compute the average order amount per customer:

```
SELECT OrderID, CustomerID, Amount,
    Avg(Amount) OVER (
        PARTITION BY CustomerID
        ORDER BY OrderID
        ROWS BETWEEN UNBOUNDED PRECEDING AND CURRENT ROW )
AS AverageOrderAmt FROM ORDERS
WHERE CustomerID IN (1,2);
```

OrderID	CustomerID	Amount	AverageOrderAmt
123	1	100	200
144	1	250	200
167	1	150	200
202	1	250	200
209	1	325	200
224	1	125	200
66	2	100	250
94	2	200	250
127	2	300	250
444	2	400	250

The Order Clause

You can also control the order in which rows are processed by window functions using `ORDER BY` within your `OVER` clause. This is optional, though it is important for any ranking or cumulative functions.

Syntax

```
ORDER BY expression
    [ ASC | DESC | USING operator ]    [ NULLS FIRST | NULLS LAST ]
    [, ...]
```

Some notes about the `ORDER BY` clause in an `OVER` clause:

>> Ascending order (`ASC`) is the default ordering.

» If you specify `NULLS LAST`, then `NULL` values are returned last; this is the default when you use `ASC` order.

» If you specify `NULLS FIRST`, then `NULL` values are returned first; this is the default when you use `DESC` order.

» The `ORDER BY` clause in your `OVER` clause *does not* have to match the order in which the rows are output.

» You can only specify a *frame clause* if you include an `ORDER BY` clause in your `OVER` clause.

The Frame Clause

The optional frame clause defines which of the rows in the partition (the `frame`) should be evaluated by the window function. You can limit which rows in the partition are passed to the function in two ways:

» Specify a `ROWS` frame to limit the frame to a fixed number of rows from the partition that precede or follow the current row.

» Specify `RANGE` to only include rows in the frame whose evaluated value falls within a certain range of the current row's value. This is the default, and the current default range is `1`, which means that only rows whose value matches that of the current row are passed to the function.

Some sources refer to the frame clause as the *Rows or Ranges* clause. If you omit this clause, the default is to include all rows

> **NOTE:** Window frames can only be used when you include an `ORDER BY` clause within the `OVER` clause.

Syntax

This clause specifies two offsets: one determines the start of the window frame, and the other determines the end of the window frame.

```
[RANGE | ROWS] frameStart |
[RANGE | ROWS] BETWEEN frameStart AND frameEnd
```

RANGE
> The frame includes rows whose values are within a specified range of the current row's value.
>
> The range is determined by the `ORDER BY` column(s). Rows with identical values for their `ORDER BY` columns are referred to as *peer rows*.

ROWS
> The frame includes a fixed number of rows based on their position in the table relative to the current row.

frameStart
> Specifies the start of the frame.
>
> For `ROWS` mode, you can specify:

```
UNBOUNDED PRECEDING
| value PRECEDING
| CURRENT ROW
| value FOLLOWING
```

> *value*
>> A non-negative integer value.

For RANGE mode, you can only specify:

```
CURRENT ROW
| UNBOUNDED FOLLOWING
```

frameEnd
> Specifies the end of the frame. The default value is CURRENT ROW.
>
> For ROWS mode, you can specify:

```
value PRECEDING
| CURRENT ROW
| value FOLLOWING
| UNBOUNDED FOLLOWING
```

> *value*
>> A non-negative integer value.

For RANGE mode, you can only specify:

```
CURRENT ROW
| UNBOUNDED FOLLOWING
```

Ranges and Rows

Probably the easiest way to understand how RANGE and ROWS work is by way of some simple OVER clause examples:

Example 1:

This clause can be used to apply a window function to all rows in the partition from the top of the partition to the current row:

```
OVER (PARTITION BY customerID ORDER BY orderDate)
```

Example 2:

Both of these clauses specify the same set of rows as Example 1:

```
OVER (PARTITION BY customerID ORDER BY orderDate UNBOUNDED PRECEDING preceding)

OVER (PARTITION BY customerID ORDER BY orderDate RANGE BETWEEN UNBOUNDED PRECEDING AND C
URRENT ROW)
```

Example 3:

This clause can be used to apply a window function to the current row and the 3 preceding row's values in the partition:

```
OVER (PARTITION BY customerID ORDER BY orderDate ROWS 3 preceding)
```

FrameStart and FrameEnd

Some important notes about the frame clause:

» `UNBOUNDED PRECEDING` means that the frame starts with the first row of the partition.

» `UNBOUNDED FOLLOWING` means that the frame ends with the last row of the partition.

» You must specify the *frameStart* first and the *frameEnd* last within the frame clause.

» In `ROWS` mode, `CURRENT ROW` means that the frame starts or ends with the current row; in `RANGE` mode, `CURRENT ROW` means that the frame starts or ends with the current row's first or last peer in the `ORDER BY` ordering.

» The default *frameClause* is to include all values from the start of the partition through the current row:

```
RANGE BETWEEN UNBOUNDED PRECEDING AND CURRENT ROW
```

Common Frame Clauses

When learning about window functions, you may find references to these specific frame clause types:

Frame Clause Type	Example
Recycled	`BETWEEN UNBOUNDED PRECEDING AND UNBOUNDED FOLLOWING`
Cumulative	`BETWEEN UNBOUNDED PRECEDING AND CURRENT ROW`
Rolling	`BETWEEN 2 PRECEDING AND 2 FOLLOWING`

Examples

This is a simple example that doesn't use a frame clause:

1. **Rank each year within a player by the number of home runs hit by that player:**

   ```
   RANK() OVER (PARTITION BY playerID ORDER BY H desc);
   ```

Here are some examples of window functions using frame clauses:

1. **Compute the running sum of G for each player:**

   ```
   SUM(G) OVER (PARTITION BY playerID ORDER BY yearID
     RANGE BETWEEN UNBOUNDED PRECEDING AND CURRENT ROW);
   ```

2. **Compute the career year:**

   ```
   YearID - min(YEARID) OVER (PARTITION BY playerID
      RANGE BETWEEN UNBOUNDED PRECEDING AND UNBOUNDED FOLLOWING) + 1;
   ```

3. **Compute a rolling average of games by player:**

   ```
   AVG(G) OVER (PARTITION BY playerID ORDER BY yearID
      ROWS BETWEEN 2 PRECEDING AND 2 FOLLOWING);
   ```

The Ranking Functions

A subset of our window functions are known as *ranking functions*:

» `DENSE_RANK` ranks each row in the result set. If values in the ranking column are the same, they receive the same rank. The next number in the ranking sequence is then used to rank the row or rows that follow, which means that `DENSE_RANK` always returns consecutive numbers.

» `RANK` ranks each row in the result set. If values in the ranking column are the same, they receive the same rank. However, the next number in the ranking sequence is then skipped, which means that `RANK` can return non-consecutive numbers.

» `ROW_NUMBER` assigns a sequential number to each row in the result set.

All ranking functions **must include** an `ORDER BY` clause in the `OVER()` clause, since that is how they compute ranking values.

Window Function Restrictions

Because window functions are only allowed in <u>ORDER BY</u> clauses, and because window functions are computed after both `WHERE` and `HAVING`, you sometimes need to use subqueries with window functions to accomplish what seems like it could be done in a simpler query.

For example, because you cannot use an `OVER` clause in a `WHERE` clause, a query like the following is not possible:

```
SELECT *
FROM Batting
WHERE rank() OVER (PARTITION BY playerID ORDER BY G) = 1;
```

And because `WHERE` and `HAVING` are computed before the windowing functions, this won't work either:

```
SELECT playerID, rank() OVER (PARTITION BY playerID ORDER BY G) as player_rank FRO
M Batting
WHERE player_rank = 1;
```

Instead, you need to use a subquery:

```
SELECT *
FROM (
    SELECT playerID, G, rank() OVER (PARTITION BY playerID ORDER BY G) as "pos"
    FROM Batting
) tmp
WHERE "pos" = 1;
```

And note that the above subquery will add a rank column to the original columns,

Window Functions Included in This Release

Splice Machine is currently expanding the set of SQL functions already able to take advantage of windowing functionality.

The <u>OVER</u> clause topic completes the complete reference information for `OVER`.

Here is a list of the functions that currently support windowing:

- » <u>AVG</u>
- » <u>COUNT</u>
- » <u>DENSE_RANK</u>
- » <u>FIRST_VALUE</u>
- » <u>LAG</u>
- » <u>LAST_VALUE</u>
- » <u>LEAD</u>

» [MAX](#)

» [MIN](#)

» [RANK](#)

» [ROW NUMBER](#)

» [SUM](#)

Additional Information

There are numerous articles about window functions that you can find online. Here are a few you might find valuable:

» The [simple talk articles from Red Gate](#) about window functions are probably the most straightforward and comprehensive descriptions of window functions.

» This [Oracle Technology Network article](#) provides an excellent technical introduction.

» This [PostgreSQL page](#) introduces their version of window functions and links to other pages.

» This [PostgreSQL wiki page](#) about SQL windowing queries page provides a succinct explanation of why windowing functions are used, and includes several useful examples.

» The [Wikipedia SQL SELECT](#) page contains descriptions of specific window functions and links to other pages.

Using Temporary Database Tables

This topic describes how to use temporary tables with Splice Machine.

About Temporary Tables

You can use temporary tables when you want to temporarily save a result set for further processing. One common case for doing so is when you've constructed a result set by running multiple queries. You can also use temporary tables when performing complex queries that require extensive operations such as repeated multiple joins or sub-queries. Storing intermediate results in a temporary table can reduce overall processing time.

An example of using a temporary table to store intermediate results is a web-based application for travel reservations that allows customers to create several alternative itineraries, compare them, and then select one for purchase. Such an app could store each itinerary in a row in a temporary table, using table updates whenever the itinerary changes. When the customer decides upon a final itinerary, that temporary row is copied into a persistent table. And when the customer session ends, the temporary table is automatically dropped.

> **NOTE:** Creating and operating with temporary tables does consume resources, and can affect performance of your queries.

Creating Temporary Tables

Splice Machine provides two statements you can use to create a temporary table; we provide multiple ways to create temporary tables to maintain compatibility with third party Business Intelligence tools.

Splice Machine does not currently support creating temporary tables stored as external tables.

Each of these statements creates the same kind of temporary table, using different syntax

Statement	Syntax
CREATE TEMPORARY TABLE	```CREATE [LOCAL|GLOBAL] TEMPORARY TABLE table-Name { ({column-definition | Table-level constraint} [, {column-definition}] *) (column-name [, column-name] *) } [NOLOGGING	ON COMMIT PRESERVE ROWS];```

Statement	Syntax

<u>DECLARE GLOBAL TEMPORARY TABLE</u>	```DECLARE GLOBAL TEMPORARY TABLE table-Name```
	 ``` { column-definition[, column-definition]```
	 ```* }```
	 ``` [ON COMMIT PRESERVE ROWS]```
	 ``` [NOT LOGGED];```

> **NOTE:** Splice Machine generates a warning if you attempt to specify any other modifiers other than the `NOLOGGING`, `NOT LOGGED`, and `ON COMMIT PRESERVE ROWS` modifiers shown above.

Restrictions on Temporary Tables

You can use temporary tables just like you do permanently defined database tables, with several important exceptions and restrictions that are noted in this section, including these:

- ›› Operational Limitations
- ›› Table Persistence

Operational Limitations

Temporary tables have the following operational limitations; they:

- ›› exist only while a user session is alive
- ›› are visible in system tables, but are otherwise not visible to other sessions or transactions
- ›› cannot be altered using the <u>RENAME COLUMN</u> statements
- ›› do not get backed up
- ›› cannot be used as data providers to views
- ›› cannot be referenced by foreign keys in other tables
- ›› are not displayed by the <u>show tables</u> command

Also note that temporary tables persist across transactions in a session and are automatically dropped when a session terminates.

Table Persistence

Here are two important notes about temporary table persistence. Temporary tables:

- ›› persist across transactions in a session

» are automatically dropped when a session terminates or expires

» can also be dropped with the DROP TABLE statement

Example

```
create local temporary table temp_num_dt (
   smallint_col smallint not null,
   int_col int,
   primary key(smallint_col)) on commit preserve rows;
insert into temp_num_dt values (1,1);
insert into temp_num_dt values (3,2),(4,2),(5,null),(6,4),(7,8);
insert into temp_num_dt values (13,2),(14,2),(15,null),(16,null),(17,8);
select * from temp_num_dt;
```

See Also

» ALTER_TABLE

» CREATE_TEMPORARY_TABLE

» DECLARE_GLOBAL_TEMPORARY_TABLE

» DROP_TABLE

» RENAME_COLUMN

» RENAME_TABLE

Splice Machine Authorization and Roles

This topic describes Splice Machine *user authorization*, which is how Splice Machine authorizes which operations can be performed by which users.

> **NOTE:** The on-premise version of Splice Machine offers several different authentication mechanisms for your database, as described in the Configuring Splice Machine Authentication topic. Native authentication is the default mechanism.

With our built-in native authentication mechanism, the user that requests a connection must provide a valid name and password, which Splice Machine verifies against the repository of users defined for the system. After Splice Machine authenticates the user as valid, user authorization determines what operations the user can perform on the database to which the user is requesting a connection.

Managing Users

Splice manages users with standard system procedures:

>> You can create a user with the SYSCS_UTIL.SYSCS_CREATE_USER procedure:

```
splice> call syscs_util.syscs_create_user('username', 'password');
```

>> You can drop a user with the SYSCS_UTIL.SYSCS_DROP_USER procedure:

```
splice> call syscs_util.syscs_drop_user('username');
```

Managing Roles

When standard authorization mode is enabled, object owners can use roles to administer privileges. Roles are useful for administering privileges when a database has many users. Role-based authorization allows an administrator to grant privileges to anyone holding certain roles, which is less tedious and error-prone than administrating those privileges to a large set of users.

The Database Owner

The *database owner* is `splice`. Only the database owner can create, grant, revoke, and drop roles. However, object owners can grant and revoke privileges for those objects to and from roles, as well as to and from individual users and to `PUBLIC` (all users).

If authentication and SQL authorization are both enabled, only the database owner can perform these actions on the database:

>> start it up

>> shut it down

>> perform a full upgrade

If authentication is not enabled, and no user is supplied, the database owner defaults to `SPLICE`, which is also the name of the default schema.

The database owner log-in information for Splice Machine is configured when your database software is installed. If you're using our database as a service, there is no default userId or password; if you're using our on-premise database, the default userID is `splice`, and the default password is `admin`.

Creating and Using Roles

The database owner can use the <u>GRANT</u> statement to grant a role to one or more users, to `PUBLIC`, or to another role. Roles can be contained within other roles and can inherit privileges from roles that they contain.

Setting Roles

When a user first connects to Splice Machine, no role is set, and the <u>SET ROLE</u> statement to set the current role for that session. The role can be any role that has been granted to the session's current user or to `PUBLIC`.

To unset the current role, you can call `SET ROLE` with an argument of `NONE`. At any time during a session, there is always a current user, but there is a current role only if `SET ROLE` has been called with an argument other than `NONE`. If a current role is not set, the session has only the privileges granted to the user directly or to `PUBLIC`.

Roles in Stored Procedures and Functions

Within stored procedures and functions that contain SQL, the current role depends on whether the routine executes with invoker's rights or with definer's rights, as specified by the `EXTERNAL SECURITY` clause in the<u>CREATE PROCEDURE</u> statements. During execution, the current user and current role are kept on an authorization stack which is pushed during a stored routine call.

» Within routines that execute with invoker's rights, the following applies: initially, inside a nested connection, the current role is set to that of the calling context. So is the current user. Such routines may set any role granted to the invoker or to `PUBLIC`.

» Within routines that execute with definer's rights, the following applies: initially, inside a nested connection, the current role is `NULL`, and the current user is that of the definer. Such routines may set any role granted to the definer or to `PUBLIC`.

Upon return from the stored procedure or function, the authorization stack is popped, so the current role of the calling context is not affected by any setting of the role inside the called procedure or function. If the stored procedure opens more than one nested connection, these all share the same (stacked) current role (and user) state. Any dynamic result set passed out of a stored procedure sees the current role (or user) of the nested context.

Dropping Roles

Only the database owner can drop a role. To drop a role, use the <u>DROP ROLE</u> statement. Dropping a role effectively revokes all grants of this role to users and other roles.

Granting Privileges

Use the <u>GRANT</u> statement to grant privileges on schemas, tables and routines to a role or to a user.

Note that when you grant privileges to a role, you are implicitly granting those same privileges to all roles that contain that role.

Revoking Privileges

Use the `REVOKE` statement to revoke privileges on schemas, tables and routines.

When a privilege is revoked from a user:

>> That session can no longer keep the role, not can it take on that role unless the role is also granted to `PUBLIC`.

>> If that role is the current role of an existing session, the current privileges of the session lose any extra privileges obtained through setting that role.

The default revoke behavior is `CASCADE`, which means that all persistent objects (constraints and views, views and triggers) that rely on a dropped role are dropped. Although there may be other ways of fulfilling that privilege at the time of the revoke, any dependent objects are still dropped. Any prepared statement that is potentially affected will be checked again on the next execute. A result set that depends on a role will remain open even if that role is revoked from a user.

See Also

>> [Configuring Splice Machine Authentication](#)

>> [CREATE_FUNCTION](#)

>> [CREATE_PROCEDURE](#)

>> [CREATE_ROLE](#)

>> [CURRENT_ROLE](#)

>> [DROP_ROLE](#)

>> [GRANT](#)

>> [REVOKE](#)

>> [SET_ROLE](#)

>> [SYSCS_UTIL.SYSCS_CREATE_USER](#)

>> [SYSCS_UTIL.SYSCS_DROP_USER](#)

Using Spark Libraries with Splice Machine

One of the great features of Spark is that a large number of libraries have been and continue to be developed for use with Spark. This topic provides an example of interfacing to the Spark Machine Learning library (MLlib).

You can follow a similar path to interface with other Spark libraries, which involves these steps:

1. Create a class with an API that leverages functionality in the Spark library you want to use.

2. Write a custom procedure in your Splice Machine database that converts a Splice Machine result set into a Spark Resilient Distributed Dataset (RDD).

3. Use the Spark library with the RDD.

Example: Using Spark MLlib with Splice Machine Statistics

This section presents the sample code for interfacing Splice Machine with the Spark Machine Learning Library (MLlib), in these subsections:

» About the Splice Machine SparkMLibUtils Class API describes the SparkMLibUtils class that Splice Machine provides for interfacing with this library.

» Creating our SparkStatistics Example Class summarizes the `SparkStatistics` Java class that we created for this example.

» Run a Sample Program to Use Our Class shows you how to define a custom procedure in your database to interface to the `SparkStatistics` class.

About the Splice Machine SparkMLibUtils Class API

Our example makes use of the Splice Machine `com.splicemachine.example.SparkMLibUtils` class, which you can use to interface between your Splice Machine database and the Spark Machine Learning library.

Here's are the public methods from the `SparkMLibUtils` class:

```
public static JavaRDDLocatedRow> resultSetToRDD(ResultSet rs)
    throws StandardException;

public static JavaRDDVector> locatedRowRDDToVectorRDD(JavaRDDLocatedRow> locatedRowJ
avaRDD, int[] fieldsToConvert)
    throws StandardException;

public static Vector convertExecRowToVector(ExecRow execRow,int[] fieldsToConvert)
    throws StandardException;

public static Vector convertExecRowToVector(ExecRow execRow)
    throws StandardException;
```

resultSetToRDD
> Converts a Splice Machine result set into a Spark Resilient Distributed Dataset (RDD) object.

locatedRowRDDToVectorRDD
> Transforms an RDD into a vector for use with the Machine Learning library. The `fieldsToConvert` parameter specifies which column positions to include in the vector.

convertExecRowToVector
> Converts a Splice Machine `execrow` into a vector. The `fieldsToConvert` parameter specifies which column positions to include in the vector.

Creating our SparkStatistics Example Class

For this example, we define a Java class named SparkStatistics that can query a Splice Machine table, convert that results into a Spark JavaRDD, and then use the Spark MLlib to calculate statistics.

Our class, `SparkStatistics`, defines one public interface:

```
public class SparkStatistics {

    public static void getStatementStatistics(String statement, ResultSet[] resultSe
ts) throws SQLException {
        try {
            // Run sql statement
            Connection con = DriverManager.getConnection("jdbc:default:connection");
            PreparedStatement ps = con.prepareStatement(statement);
            ResultSet rs = ps.executeQuery();

            // Convert result set to Java RDD
            JavaRDDLocatedRow> resultSetRDD = ResultSetToRDD(rs);

            // Collect column statistics
            int[] fieldsToConvert = getFieldsToConvert(ps);
            MultivariateStatisticalSummary summary = getColumnStatisticsSummary(resu
ltSetRDD, fieldsToConvert);

            IteratorNoPutResultSet resultsToWrap = wrapResults((EmbedConnection) co
n,  getColumnStatistics(ps, summary, fieldsToConvert));
            resultSets[0] = new EmbedResultSet40((EmbedConnection)con, resultsToWra
p, false, null, true);
        } catch (StandardException e) {
            throw new SQLException(Throwables.getRootCause(e));
        }
    }
}
```

We call the `getStatementStatistics` from custom procedure in our database, passing it an SQL query . `getStatementStatistics` performs the following operations:

1. **Query your database**
   The first step is to use our JDBC driver to connect to your database and run the query:

   ```
   Connection con = DriverManager.getConnection("jdbc:default:connection");P
   reparedStatement ps = con.prepareStatement(statement);
   ResultSet rs = ps.executeQuery();
   ```

2. **Convert the query results into a Spark RDD**
   Next, we convert the query's result set into a Spark RDD:

   ```
   JavaRDD<LocatedRow> resultSetRDD = ResultSetToRDD(rs);
   ```

3. **Calculate statistics**
   Next, we use Spark to collect statistics for the query, using private methods in our `SparkStatistics` class:

   ```
   int[] fieldsToConvert = getFieldsToConvert(ps);MultivariateStatisticalSum
   mary summary = getColumnStatisticsSummary(resultSetRDD, fieldsToConvert);
   ```

   You can view the implementations of the `getFieldsToConvert` and `getColumnStatisticsSummary`
   methods in the *Appendix* at the end of this topic.

4. **Return the results**
   Finally, we return the results:

   ```
   IteratorNoPutResultSet resultsToWrap = wrapResults((EmbedConnection) co
   n, getColumnStatistics(ps, summary, fieldsToConvert));
   resultSets[0] = new EmbedResultSet40((EmbedConnection)con, resultsToWra
   p, false, null, true);
   ```

Run a Sample Program to Use Our Class

Follow these steps to run a simple example program to use the Spark MLlib library to calculate statistics for an SQL statement.

1. **Create Your API Class**
   The first step is to create a Java class that uses Spark to generate and analyze statistics, as shown in the
   previous section, Creating our SparkStatistics Example Class

2. **Create your custom procedure**
   First we create a procedure in our database that references the `getStatementStatistics` method in our
   API, which takes an SQL query as its input and uses Spark to calculate statistics for the query using MLlib:

```
CREATE PROCEDURE getStatementStatistics(statement varchar(1024))
   PARAMETER STYLE JAVA
   LANGUAGE JAVA
   READS SQL DATA
   DYNAMIC RESULT SETS 1
   EXTERNAL NAME 'com.splicemachine.example.SparkStatistics.getStatementS
tatistics';
```

3. **Create a table to use**

   Let's create a very simple table to illustrate use of our procedure:

   ```
   create table t( col1 int, col2 double);
   insert into t values(1, 10);
   insert into t values(2, 20);
   insert into t values(3, 30);
   insert into t values(4, 40);
   ```

4. **Call your custom procedure to get statistics**

   Now call your custom procedure, which sends an SQL statement to the SparkStatistics class we created to generate a result set:

   ```
   call splice.getStatementStatistics('select * from t');
   ```

Appendix: The SparkStatistics Class

Here's the full code for our SparkStatistics class:

```
package com.splicemachine.example;

import com.google.common.base.Throwables;
import com.google.common.collect.Lists;
import com.splicemachine.db.iapi.error.StandardException;
import com.splicemachine.db.iapi.sql.Activation;
import com.splicemachine.db.iapi.sql.ResultColumnDescriptor;
import com.splicemachine.db.iapi.sql.execute.ExecRow;
import com.splicemachine.db.iapi.types.DataTypeDescriptor;
import com.splicemachine.db.iapi.types.SQLDouble;
import com.splicemachine.db.iapi.types.SQLLongint;
import com.splicemachine.db.iapi.types.SQLVarchar;
import com.splicemachine.db.impl.jdbc.EmbedConnection;
import com.splicemachine.db.impl.jdbc.EmbedResultSet40;
import com.splicemachine.db.impl.sql.GenericColumnDescriptor;
import com.splicemachine.db.impl.sql.execute.IteratorNoPutResultSet;
import com.splicemachine.db.impl.sql.execute.ValueRow;
import com.splicemachine.derby.impl.sql.execute.operations.LocatedRow;
import org.apache.spark.api.java.JavaRDD;
import org.apache.spark.mllib.linalg.Vector;
import org.apache.spark.mllib.stat.MultivariateStatisticalSummary;
import org.apache.spark.mllib.stat.Statistics;
import java.sql.*;
import java.sql.PreparedStatement;
import java.sql.ResultSet;
import java.sql.Types;
import java.util.List;


public class SparkStatistics {

    private static final ResultColumnDescriptor[] STATEMENT_STATS_OUTPUT_COLUMNS = n
ew GenericColumnDescriptor[]{
            new GenericColumnDescriptor("COLUMN_NAME", DataTypeDescriptor.getBuiltIn
DataTypeDescriptor(Types.VARCHAR)),
            new GenericColumnDescriptor("MIN", DataTypeDescriptor.getBuiltInDataType
Descriptor(Types.DOUBLE)),
            new GenericColumnDescriptor("MAX", DataTypeDescriptor.getBuiltInDataType
Descriptor(Types.DOUBLE)),
            new GenericColumnDescriptor("NUM_NONZEROS", DataTypeDescriptor.getBuiltI
nDataTypeDescriptor(Types.DOUBLE)),
            new GenericColumnDescriptor("VARIANCE", DataTypeDescriptor.getBuiltInDat
aTypeDescriptor(Types.DOUBLE)),
            new GenericColumnDescriptor("MEAN", DataTypeDescriptor.getBuiltInDataTyp
eDescriptor(Types.DOUBLE)),
            new GenericColumnDescriptor("NORML1", DataTypeDescriptor.getBuiltInDataT
ypeDescriptor(Types.DOUBLE)),
            new GenericColumnDescriptor("MORML2", DataTypeDescriptor.getBuiltInDataT
ypeDescriptor(Types.DOUBLE)),
            new GenericColumnDescriptor("COUNT", DataTypeDescriptor.getBuiltInDataTy
peDescriptor(Types.BIGINT)),
    };
```

```
    public static void getStatementStatistics(String statement, ResultSet[] resultSe
ts) throws SQLException {
        try {
            // Run sql statement
            Connection con = DriverManager.getConnection("jdbc:default:connection");
            PreparedStatement ps = con.prepareStatement(statement);
            ResultSet rs = ps.executeQuery();

            // Convert result set to Java RDD
            JavaRDDLocatedRow> resultSetRDD = ResultSetToRDD(rs);

            // Collect column statistics
            int[] fieldsToConvert = getFieldsToConvert(ps);
            MultivariateStatisticalSummary summary = getColumnStatisticsSummary(resu
ltSetRDD, fieldsToConvert);

            IteratorNoPutResultSet resultsToWrap = wrapResults((EmbedConnection) co
n, getColumnStatistics(ps, summary, fieldsToConvert));
            resultSets[0] = new EmbedResultSet40((EmbedConnection)con, resultsToWra
p, false, null, true);
        } catch (StandardException e) {
            throw new SQLException(Throwables.getRootCause(e));
        }
    }

    private static MultivariateStatisticalSummary getColumnStatisticsSummary(JavaRDD
LocatedRow> resultSetRDD,
                                                        int[] fieldsToC
onvert) throws StandardException{
        JavaRDDVector> vectorJavaRDD = SparkMLibUtils.locatedRowRDDToVectorRDD(resul
tSetRDD, fieldsToConvert);
        MultivariateStatisticalSummary summary = Statistics.colStats(vectorJavaRDD.r
dd());
        return summary;
    }


    /*
     * Convert a ResultSet to JavaRDD
     */
    private static JavaRDDLocatedRow> ResultSetToRDD (ResultSet resultSet) throws St
andardException{
        EmbedResultSet40 ers = (EmbedResultSet40)resultSet;

        com.splicemachine.db.iapi.sql.ResultSet rs = ers.getUnderlyingResultSet();
        JavaRDDLocatedRow> resultSetRDD = SparkMLibUtils.resultSetToRDD(rs);

        return resultSetRDD;
    }
```

```
    private static int[] getFieldsToConvert(PreparedStatement ps) throws SQLExceptio
n{
        ResultSetMetaData metaData = ps.getMetaData();
        int columnCount = metaData.getColumnCount();
        int[] fieldsToConvert = new int[columnCount];
        for (int i = 0; i  columnCount; ++i) {
            fieldsToConvert[i] = i+1;
        }
        return fieldsToConvert;
    }

    /*
     * Convert column statistics to an iterable row source
     */
    private static IterableExecRow> getColumnStatistics(PreparedStatement ps,
                                                        MultivariateStatisticalSumm
ary summary,
                                                        int[] fieldsToConvert) thro
ws StandardException {
        try {

            ListExecRow> rows = Lists.newArrayList();
            ResultSetMetaData metaData = ps.getMetaData();

            double[] min = summary.min().toArray();
            double[] max = summary.max().toArray();
            double[] mean = summary.mean().toArray();
            double[] nonZeros = summary.numNonzeros().toArray();
            double[] variance = summary.variance().toArray();
            double[] normL1 = summary.normL1().toArray();
            double[] normL2 = summary.normL2().toArray();
            long count = summary.count();

            for (int i= 0; i  fieldsToConvert.length; ++i) {
                int columnPosition = fieldsToConvert[i];
                String columnName = metaData.getColumnName(columnPosition);
                ExecRow row = new ValueRow(9);
                row.setColumn(1, new SQLVarchar(columnName));
                row.setColumn(2, new SQLDouble(min[columnPosition-1]));
                row.setColumn(3, new SQLDouble(max[columnPosition-1]));
                row.setColumn(4, new SQLDouble(nonZeros[columnPosition-1]));
                row.setColumn(5, new SQLDouble(variance[columnPosition-1]));
                row.setColumn(6, new SQLDouble(mean[columnPosition-1]));
                row.setColumn(7, new SQLDouble(normL1[columnPosition-1]));
                row.setColumn(8, new SQLDouble(normL2[columnPosition-1]));
                row.setColumn(9, new SQLLongint(count));
                rows.add(row);
            }
            return rows;
```

```
        }
        catch (Exception e) {
            throw StandardException.newException(e.getLocalizedMessage());
        }
    }

    private static IteratorNoPutResultSet wrapResults(EmbedConnection conn, Iterable
ExecRow> rows) throws
            StandardException {
        Activation lastActivation = conn.getLanguageConnection().getLastActivatio
n();
        IteratorNoPutResultSet resultsToWrap = new IteratorNoPutResultSet(rows, STAT
EMENT_STATS_OUTPUT_COLUMNS,
                lastActivation);
        resultsToWrap.openCore();
        return resultsToWrap;
    }
}
```

See Also

» [Spark Overview](#)

» [Using the Splice Machine Database Console](#)

» You can find the Spark MLlib guide in the Programming Guides section of the Spark documentation
  site: https://spark.apache.org/docs

Using the Splice Machine Virtual Table Interface (VTI)

The Virtual Table Interface (VTI) allows you to use an SQL interface with data that is external to your database. This topic introduces the Splice Machine VTI in these sections:

» About VTI describes the virtual table interface.

» Splice Machine Built-in Virtual Table Interfaces describes the virtual table interfaces built into Splice Machine, and provides examples of using each.

» Creating a Custom Virtual Table Interface walks you through the steps required to create a custom virtual table interface, and demonstrates how to simplify its use with a table function.

» The Splice Machine Built-in VTI Classes provides reference descriptions of the virtual table interface classes built into by Splice Machine.

About VTI

You can use the Splice Machine Virtual Table Interface (*VTI*) to access data in external files, libraries, and databases.

> **NOTE:** A virtual table is a view of data stored elsewhere; the data itself is not duplicated in your Splice Machine database.

The external data source can be any information source, including:

» XML formatted reports and logs.

» Queries that run in external databases that support JDBC, such as Oracle and DB2.

» RSS feeds.

» Flat files in formats such as comma-separated value (csv) format.

About Table Functions

A *table function* returns `ResultSet` values that you can query like you do tables that live in your Splice Machine database. A table function is bound to a constructor for a custom VTI class. Here's an example of a declaration for a table function that is bound to the `PropertiesFileVTI` class, which we walk you through implementing later in this topic:

```
CREATE FUNCTION propertiesFile(propertyFilename VARCHAR(200))
    RETURNS TABLE
        (
          KEY_NAME varchar(100)
          VALUE varchar(200)
        )
    LANGUAGE JAVA
    PARAMETER STYLE SPLICE_JDBC_RESULT_SET
    READS SQL DATA
    EXTERNAL NAME 'com.splicemachine.tutorials.vti.PropertiesFIleVTI.getPropertiesFi
leVTI';
```

Splice Machine Built-in Virtual Table Interfaces

Splice Machine provides two built-in VTI classes that you can use:

Class	Description	Implemented by
SpliceFileVTI	For querying delimited flat files, such as CSV files.	com.splicemachine.derby.vti.SpliceFileVTI
SpliceJDBCVTI	For querying data from external sources that support the JDBC API.	com.splicemachine.derby.vti.SpliceJDBCVTI

Each of these classes implements the `DatasetProvider` interface, which is used by Spark for creating execution trees, and the `VTICosting` interface, which is used by the Splice Machine optimizer.

SpliceFileVTI Example

For example, if we have an input file named `vtiInfile.csv` that contains this information:

```
sculligan,Relief Pitcher,27,08-27-2015,2015-08-27 08:08:08,06:08:08
jpeepers,Catcher,37,08-26-2015,2015-08-21 08:09:08,08:08:08
mbamburger,Manager,47,08-25-2015,2015-08-20 08:10:08,10:08:08
gbrown,Batting Coach,46,08-24-2015,2015-08-21 08:11:08,11:08:08
jardson,Left Fielder,34,08-23-2015,2015-08-22 08:12:08,11:08:08
```

We can use the `SpliceFileVTI` class to select and display the contents of our input file:

```
SELECT * FROM new com.splicemachine.derby.vti.SpliceFileVTI(
  '/<path>/data/vtiInfile.csv','',',') AS b
    (name VARCHAR(10), title VARCHAR(30), age INT, something VARCHAR(12), date_hired
TIMESTAMP, clock TIME);NAME        |TITLE              |AGE   |SOMETHING   |DATE_HIRE
D            |CLOCK
-----------------------------------------------------------------------------------
---
sculligan   |Relief Pitcher   |27    |08X-27-2015 |2015-08-27 08:08:08.0  |06:08:08
jpeepers    |Catcher          |37    |08-26-2015  |2015-08-21 08:09:08.0  |08:08:08
mbamburger  |Manager          |47    |08-25-2015  |2015-08-20 08:10:08.0  |10:08:08
gbrown      |Batting Coach    |46    |08-24-2015  |2015-08-21 08:11:08.0  |11:08:08
jardson     |Left Fielder     |34    |08-23X-2015 |2015-08-22 08:12:08.0  |11:08:08

5 rows selected
```

SpliceJDBCVTI Example

We can use the `SpliceJDBCVTI` class to select and display the contents a table in a JDBC-compliant database. For example, here we query a table stored in a MySQL database:

```
SELECT * FROM new com.splicemachine.derby.vti.SpliceJDBCVTI(
  'jdbc:mysql://localhost/hr?user=root&password=mysql-passwd','mySchema','myTable')
AS b
    (name VARCHAR(10), title VARCHAR(30), age INT, something VARCHAR(12), date_hired
TIMESTAMP, clock TIME);NAME        |TITLE              |AGE   |SOMETHING   |DATE_HIRE
D            |CLOCK
-----------------------------------------------------------------------------------
---
sculligan   |Relief Pitcher    |27    |08X-27-2015 |2015-08-27 08:08:08.0  |06:08:08
jpeepers    |Catcher           |37    |08-26-2015  |2015-08-21 08:09:08.0  |08:08:08
mbamburger  |Manager           |47    |08-25-2015  |2015-08-20 08:10:08.0  |10:08:08
gbrown      |Batting Coach     |46    |08-24-2015  |2015-08-21 08:11:08.0  |11:08:08
jardson     |Left Fielder      |34    |08-23X-2015 |2015-08-22 08:12:08.0  |11:08:08

5 rows selected
```

Creating a Custom Virtual Table Interface

You can create a custom virtual table interface by creating a class that implements the `DatasetProvider` and `VTICosting` interfaces, which are described below.

You can then use your custom VTI within SQL queries by using VTI syntax, as shown in the examples in the previous section. You can also create a table function for your custom VTI, and then call that function in your queries, which simplifies using your interface.

This section walks you through creating a custom virtual table interface that reads and displays the property keys and values in a properties file. This interface can be executed using a table function or by specifying its full method name in SQL statements.

> The full code for this example is in the Splice [Community Sample Code Repository on Github.](#)

The remainder of this section is divided into these subsections:

- [Declare Your Class](#)
- [Implement the Constructors](#)
- [Implement Your Method to Generate Results](#)
- [Implement Costing Methods](#)
- [Implement Other DatasetProvider Methods](#)
- [Use Your Custom Virtual Table Interface](#)

Declare Your Class

The first thing you need to do is to declare your public class; since we're creating an interface to read property files, we'll call our class `PropertiesFileVTI`. To create a custom VTI interface, you need to implement the following classes:

Class	Description
DatasetProvider	Used by Spark to construct execution trees.
VTICosting	Used by the Splice Machine optimizer to estimate the cost of operations.

Here's the declaration:

```
public class PropertiesFileVTI implements DatasetProvider, VTICosting {

    //Used for logging (and optional)
    private static final Logger LOG = Logger.getLogger(PropertiesFileVTI.class);

    //Instance variable that will store the name of the properties file that is bein
g read
    private String fileName;

    //Provide external context which can be carried with the operation
    protected OperationContext operationContext;
```

Implement the Constructors

This section describes the constructors that we implement for our custom class:

- You need to implement an empty constructor if you want to use your class in table functions:

```
public PropertiesFileVTI()
```

» This is the signature used by invoking the VTI using the class name in SQL queries:

```
public PropertiesFileVTI(String pfileName)
```

» This static constructor is called by the VTI - Table Function.

```
public static DatasetProvider getPropertiesFileVTI(String fileName)
```

Here's our implementation of the constructors for the `PropertiesFileVTI` class:

```
public PropertiesFileVTI() {}
public PropertiesFileVTI(String pfileName) {
    this.fileName = pfileName;
}

public static DatasetProvider getPropertiesFileVTI(String fileName) {
    return new PropertiesFileVTI(fileName);
}
```

Implement Your Method to Generate Results

The heart of your virtual table interface is the `DatasetProvider` method `getDataSet`, which you override to generate and return a `DataSet`. It's declaration looks like this:

```
DataSet<LocatedRow> getDataSet(
        SpliceOperation op,        // References the op at the top of the stack
        DataSetProcessor dsp,      // Mechanism for constructing the execution tree
        ExecRow execRow ) throws StandardException;
```

The `VTIOperation` process calls this method to compute the `ResultSet` that it should return. Our `PropertiesFileVTI` implementation is shown here:

```
@Override
public DataSet<LocatedRow> getDataSet(SpliceOperation op, DataSetProcessor dsp, Exec
Row execRow) throws StandardException {
    operationContext = dsp.createOperationContext(op);

        //Create an arraylist to store the key-value pairs
    ArrayList<LocatedRow> items = new ArrayList<LocatedRow>();

    try {
        Properties properties = new Properties();

        //Load the properties file
        properties.load(getClass().getClassLoader().getResourceAsStream(fileName));

        //Loop through the properties and create an array
        for (String key : properties.stringPropertyNames()) {
            String value = properties.getProperty(key);
            ValueRow valueRow = new ValueRow(2);
            valueRow.setColumn(1, new SQLVarchar(key));
            valueRow.setColumn(2, new SQLVarchar(value));
            items.add(new LocatedRow(valueRow));
        }
    } catch (FileNotFoundException e) {
        LOG.error("File not found: " + this.fileName, e);
    } catch (IOException e) {
        LOG.error("Unexpected IO Exception: " + this.fileName, e);
    } finally {
        operationContext.popScope();
    }
    return new ControlDataSet<>(items);
}
```

Implement Costing Methods

The Splice Machine optimizer uses costing estimates to determine the optimal execution plan for each query. You need to implement several costing methods in your VTI class:

» `getEstimatedCostPerInstantiation` returns the estimated cost to instantiate and iterate through your table function. Unless you have an accurate means of estimating this cost, simply return `0` in your implementation.

```
public double getEstimatedCostPerInstantiation( VTIEnvironment vtiEnv) throws S
QLException;
```

» `getEstimatedRowCount` returns the estimated row count for a single scan of your table function. Unless you have an accurate means of estimating this cost, simply return `0` in your implementation.

```
public double getEstimatedCostPerInstantiation( VTIEnvironment vtiEnv) throws S
QLException;
```

» `supportsMultipleInstantiations` returns a Boolean value indicating whether your table function's `ResultSet` can be instantiated multiple times in a single query. For our `PropertiesFileVTI` implementation of this method, we simply return `False,` since there's no reason for our function to be used that way.

```
public double supportsMultipleInstantiations( VTIEnvironment vtiEnv) throws SQL
Exception;
```

> **NOTE:** The VTICosting methods each take a `VTIEnvironment` argument; this is a state variable created by the Splice Machine optimizer, which methods can use to pass information to each other or to learn other details about the operating environment..

Here is the implementation of costing methods for our `PropertiesFileVTI` class:

```
@Override
public double getEstimatedCostPerInstantiation(VTIEnvironment arg0) throws SQLExcept
ion {
    return 0;
}

@Override
public double getEstimatedRowCount(VTIEnvironment arg0) throws SQLException {
    return 0;
}
@Override
public boolean supportsMultipleInstantiations(VTIEnvironment arg0) throws SQLExcepti
on {
    return false;
}
```

Implement Other DatasetProvider Methods

You also need to implement two additional `DatasetProvider` methods:

» `getOperationContext` simply returns the current operation context (this.`operationContext`).

```
OperationContext getOperationContext();
```

» `getMetaData` returns metadata that is used to dynamically bind your table function; this metadata includes a column descriptor for each column in your virtual table, including the name of the column, its type, and its size. In our `PropertiesFileVTI,` we assign the descriptors to a static variable, and our implementation of this method simply returns that value.

```
ResultSetMetaData getMetaData() throws SQLException;
```

Here is the implementation of these methods for our `PropertiesFileVTI` class:

```
@Override
public OperationContext getOperationContext() {
    return this.operationContext;
}@Override
public ResultSetMetaData getMetaData() throws SQLException {
    return metadata;
}

private static final ResultColumnDescriptor[] columnInfo = {
        EmbedResultSetMetaData.getResultColumnDescriptor("KEY1", Types.VARCHAR, fals
e, 200),
        EmbedResultSetMetaData.getResultColumnDescriptor("VALUE", Types.VARCHAR, fal
se, 200),
};

private static final ResultSetMetaData metadata = new EmbedResultSetMetaData(columnI
nfo);
}
```

Use Your Custom Virtual Table Interface

You can create a table function in your Splice Machine database to simplify use of your custom VTI. Here's a table declaration for our custom interface:

```
CREATE FUNCTION propertiesFile(propertyFilename VARCHAR(200))
    RETURNS TABLE
        (
          KEY_NAME varchar(100)
          VALUE varchar(200)
        )
    LANGUAGE JAVA
    PARAMETER STYLE SPLICE_JDBC_RESULT_SET
    READS SQL DATA
    EXTERNAL NAME 'com.splicemachine.tutorials.vti.PropertiesFIleVTI.getPropertiesFi
leVTI';
```

You can now use your interface with table function syntax; for example:

```
select * from table (propertiesFile('sample.properties')) b;
```

You can also use your interface by using VTI syntax in an SQL query; for example:

```
select * from new com.splicemachine.tutorials.vti.PropertiesFileVTI('sample.properti
es')
    as b (KEY_NAME VARCHAR(20), VALUE VARCHAR(100));
```

The Splice Machine Built-in VTI Classes

This section describes the built-in VTI classes:

- ›› The `SpliceFileVTI` class

- ›› The `SpliceJDBCVTI` class

The SpliceFileVTI Class

You can use the `SpliceFileVTI` class to apply SQL queries to a file, such as a csv file, as shown in the examples below.

Constructors

You can use the following constructor methods with the `SpliceFileVTI` class. Each creates a virtual table from a file:

```
public SpliceFileVTI(String fileName)
```

```
public SpliceFileVTI(String fileName,
                     String characterDelimiter,
                     String columnDelimiter)
```

```
public SpliceFileVTI(String fileName,
                     String characterDelimiter,
                     String columnDelimiter,
                     boolean oneLineRecords)
```

fileName
> The name of the file that you are reading.

characterDelimiter
> Specifies which character is used to delimit strings in the imported data. You can specify `null` or the empty string (`' '`) to use the default string delimiter, which is the double-quote (`"`). If your input contains control characters such as newline characters, make sure that those characters are embedded within delimited strings.

columnDelimiter
> The character used to separate columns, Specify `null` if using the comma (`,`) character as your delimiter. Note that the backslash (`\`) character is not allowed as the column delimiter.

oneLineRecords
> A Boolean value that specifies whether each line in the import file contains one complete record; if you specify false, records can span multiple lines in the input file.

The SpliceJDBCVTI Class

You can use the `SpliceJDBCVTI` class to access external databases that provide JDBC connections.

Constructors

You can use the following constructor methods with the `SpliceJDBCVTI` class.

```
public SpliceJDBCVTI(String connectionUrl,
                     String schemaName,
                     String tableName)
```

connectionURL
      The URL of the database connection you are using.

schemaName
      The name of the database schema.

tableName
      The name of the table in the database schema.

```
public SpliceJDBCVTI(String connectionUrl,
                     String sql)
```

connectionURL
      The URL of the database connection you are using.

sql
      The SQL string to execute in that database.

See Also

We recommend visiting the [Derby VTI documentation](#), which provides full reference documentation for the VTI class hierarchy.

Using the Splice Machine External Table Feature

This topic covers the use of external tables in Splice Machine. An external table references a file stored in a flat file format. You can use flat files that are stored in one of these formats:

- » `ORC` is a columnar storage format

- » `PARQUET` is a columnar storage format

- » `Avro` is a data serialization system

- » `TEXTFILE` is a plain text file

You can access `ORC` and `PARQUET` files that have been compressed with either Snappy or ZLIb compression; however, you cannot use a compressed plain text file.

About External Tables

You can use Splice Machine external tables to query the contents of flat files that are stored outside of your database. You query external tables pretty much the same way as you do the tables in your database.

External tables reference files that are stored in a flat file format such as Apache Parquet or Apache Orc, both of which are columnar storage formats that are available in Hadoop. You can use the [CREATE EXTERNAL TABLE](#) statement to create an external table that is connected to a specific flat file.

Using External Tables

This section presents information about importing data into an external table, and includes several examples of using external tables with Splice Machine.

Importing Data Into an External Table

You cannot import data directly into an external table; if you already have an external table in a compatible format, you can use [CREATE EXTERNAL TABLE](#) statement to point at the external file and query against it.

If you want to create an external file from within Splice Machine, follow these steps:

1. Create (or use) a table in your Splice Machine database (your internal table).

2. Use [CREATE EXTERNAL TABLE](#) to create your empty external table, specifying the location where you want that data stored externally.

3. Use [INSERT INTO](#) (your external table) `SELECT` (from your internal table) to populate the external file with your data.

4. You can now query the external table.

Accessing a Parquet File

The following statement creates an external table for querying a `PARQUET` file that is stored on your computer:

```
splice> CREATE EXTERNAL TABLE myExtTbl (
  col1 INT, col2 VARCHAR(24))
    PARTITIONED BY (col1)
    STORED AS PARQUET
    LOCATION '/users/myname/myParquetFile';0 rows inserted/updated/deleted
```

The call to `CREATE EXTERNAL TABLE` associates a Splice Machine external table with the file named `myparquetfile`, and tells Splice Machine that:

» The table should be partitioned based on the values in `col1`.

» The file is stored in PARQUET format.

» The file is located in `/users/myname/myParquetFile`.

After you create the external table, you can query `myExtTbl` just as you would any other table in your database.

Accessing and Updating an ORC File

The following statement creates an external table for creates an external table for an `ORC` file and inserts data into it:

```
splice> CREATE EXTERNAL TABLE myExtTbl2
  (col1 INT, col2 VARCHAR(24))
    PARTITIONED BY (col1)
    STORED AS ORC
    LOCATION '/users/myname/myOrcFile';
0 rows inserted/updated/deleted
splice> INSERT INTO myExtTbl2 VALUES (1, 'One'), (2, 'Two'), (3, 'Three');
3 rows inserted/updated/deletedsplice
> SELECT * FROM myExtTbl2;
COL1          |COL2
---------------------------------
3             |Three
2             |Two
1             |One
```

The call to `CREATE EXTERNAL TABLE` associates a Splice Machine external table with the file named `myOrcFile`, and tells Splice Machine that:

» The table should be partitioned based on the values in `col1`.

» The file is stored in ORC format.

» The file is located in `/users/myname/myOrcFile`.

The call to `INSERT INTO` demonstrates that you can insert values into the external table just as you would with an ordinary table.

Accessing a Plain Text File

You can specify a table constraint on an external table; for example:

```
splice> CREATE EXTERNAL TABLE myTextTable(
      col1 INT, col2 VARCHAR(24))
    PARTITIONED BY (col1)
    ROW FORMAT DELIMITED FIELDS TERMINATED BY ',' ESCAPED BY '\\'
    LINES TERMINATED BY '\\n'
    STORED AS TEXTFILE
    LOCATION '/users/myName/myTextFile';0 rows inserted/updated/deleted
```

The call to `CREATE EXTERNAL TABLE` associates a Splice Machine external table with the file named `myOrcFile`, and tells Splice Machine that:

» The table should be partitioned based on the values in `col1`.

» Each field in each row is terminated by a comma.

» Each line in the file is terminated by a line-end character.

» The file is stored in plain text format.

» The file is located in `/users/myName/myTextFile`.

Accessing a Compressed File

This example is exactly the same as our first example, except that the source file has been compressed with Snappy compression:

```
splice> CREATE EXTERNAL TABLE myExtTbl (
      col1 INT, col2 VARCHAR(24))
    COMPRESSED WITH SNAPPY
    PARTITIONED BY (col1)
    STORED AS PARQUET
    LOCATION '/users/myname/myParquetFile';
 0 rows inserted/updated/deleted
```

Manually Refreshing an External Tables

If the schema of the file represented by an external table is updated, Splice Machine needs to refresh its representation. When you use the external table, Spark caches its schema in memory to improve performance; as long as you are using Spark to modify the table, it is smart enough to refresh the cached schema. However, if the table schema is modified outside of Spark, you need to call the <u>SYSCS_UTIL.SYSCS_REFRESH_EXTERNAL_TABLE</u> built-in system procedure. For example:

```
splice> CALL SYSCS_UTIL.SYSCS_REFRESH_EXTERNAL_TABLE('APP', 'myExtTable');
Statement executed.
```

See Also

The <u>CREATE_EXTERNAL_TABLE</u> statement.

The `SYSCS_UTIL.SYSCS_REFRESH_EXTERNAL_TABLE` built-in system procedure.

Using Splice Machine with HCatalog

<div style="border:1px solid orange; border-radius:20px; text-align:center; padding:10px;">

This is an On-Premise-Only topic!   [Learn about our products](#)

</div>

Apache HCatalog is a metadata and table management system for the broader Hadoop platform. HCatalogs table abstraction presents users with a relational view of data in the Hadoop distributed file system (HDFS) and ensures that users need not worry about where or in what format their data is stored. HCatalog supports reading and writing files in any format for which a SerDe (serializer-deserializer) can be written.

Splice Machine integrates with HCatalog, allowing any HiveQL statements to read from (e.g. `SELECT`) and write to (e.g. `INSERT`) Splice Machine tables. You can also use joins and unions to combine native Hive tables with Splice Machine tables.

You can find extensive information about HCatalog on the [Apache Hive web site](#).

Also note that you can also take advantage of our HCatalog integration to access your Splice Machine tables for reading and writing from any database that features HCatalog integration, such as *MongoDB*.

Using HCatalog with a Splice Machine Table

To use HCatalog with Splice Machine, you connect a Hive table with a Splice Machine table using the HiveQL `CREATE EXTERNAL TABLE` statement. In Hive, an external table can point to any HDFS location for its storage; in this case, the table is located in your Splice Machine database.

For example, here's a HiveQL statement that creates a table named `extTest1` that is connected with the `hcattest` table in a Splice Machine database:

```
hive> CREATE EXTERNAL TABLE extTest1(col1 int, col2 varchar(20))
    STORED BY 'com.splicemachine.mrio.api.hive.SMStorageHandler'
    TBLPROPERTIES ("splice.jdbc"="jdbc:splice://localhost:1527/splicedb\;user=splic
e\;password=admin", "splice.tableName"="TEST.hcattest");
```

Your table definition must include:

» The `STORED BY` clause as shown, which allows the table to connect with Splice Machine.

» The `TBLPROPERTIES` clause, which specifies:

  » The JDBC connection you are using. include your access <mark>user</mark> ID and <mark>password</mark>.

  If you are running Splice Machine on a cluster, connect from a machine that is NOT running an HBase RegionServer and specify the IP address of a <mark>regionServer</mark> node, e.g. `10.1.1.110`.

  > **NOTE:** Use `localhost` if you're running the standalone version of Splice Machine.

  » The name of the table in your Splice Machine database with which you are connecting the Hive table.

How Splice Machine Maps Columns

If your Hive table contains a different number of columns than does your Splice Machine database table, Splice Machine maps the columns.

For example, if your Hive table has three columns defined, and your Splice Machine table has four columns defined, then Splice Machine maps the three Hive columns into the first three columns in the Splice Machine table, as shown here:
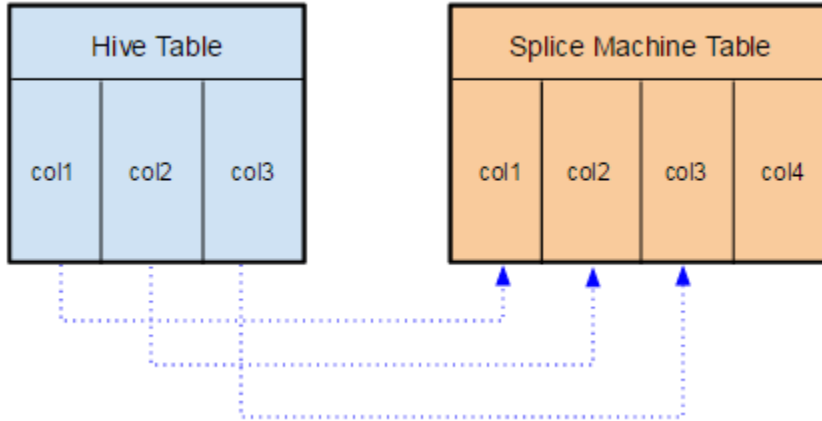
Using HiveQL to Join Tables From Different Data Resources

You can use HiveQL to join tables that point to different data resources. The Java class provided by Splice Machine allows you to include tables from your Splice Machine database in such joins, as illustrated here:
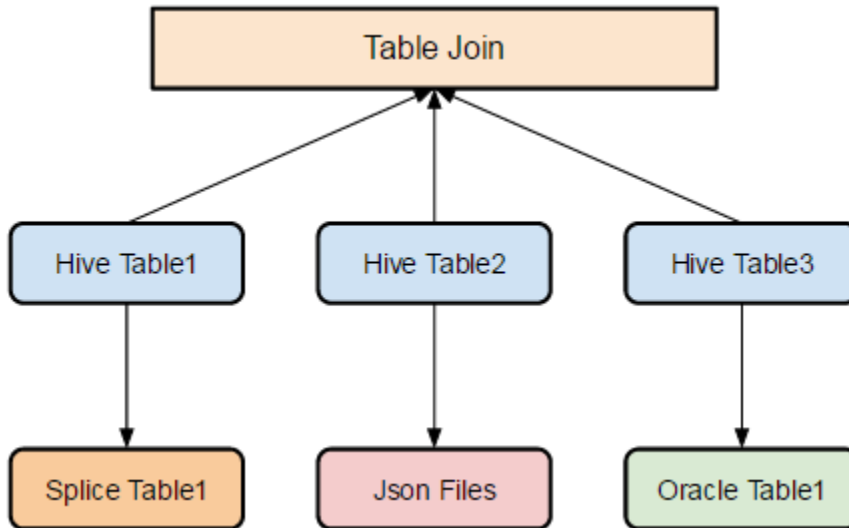
For more information about Hive joins, see the Apache Hive documentation.

Examples

This section contains several simple examples of using HCatalog.

Example 1: Simple Example

This is a simple example of creating an external table in Hive that maps directly to a Splice Machine table.

1. **Create the Splice Machine table**

   This example uses a very simple Splice Machine database table, `hcattest`, which we create with these statements:

   ```
   splice> create table hcattest(col1 int, col2 varchar(20));
   splice> insert into hcattest values(1, 'row1');
   splice> insert into hcattest values(2, 'row2');
   splice> insert into hcattest values(3, 'row3');
   splice> insert into hcattest values(4, 'row4');
   ```

2. **Verify the table**

   Verify that `hcattest` is set up correctly:

   ```
   splice> describe hcattest;
   COLUMN_NAME | TYPE_NAME | DES&|NUM&|COLUMN&|COLUMN_DEF|CHAR_OCTE&|IS_NUL
   L&
   ------------------------------------------------------------------------------
   -
   COL1          |INTEGER    |0     |10  |10        |NULL       |NULL        |YES
   COL2          |VARCHAR    |NULL |NULL|20        |NULL       |40          |YES

   2 rows selected
   splice> select * from hcattest;
   COL1          |COL2
   -------------------------------
   1              row1
   2              row2
   3              row3
   4              row4

   4 rows selected
   ```

3. **Create an external table in Hive**

   You need to create an external table in Hive, and connect that table with the Splice Machine table you just created. Type the following command into the Hive shell, substituting your ==user== ID and ==password==.

   ```
   hive> CREATE EXTERNAL TABLE extTest1(col1 int, col2 varchar(20))
      STORED BY 'com.splicemachine.mrio.api.hive.SMStorageHandler'
      TBLPROPERTIES ("splice.jdbc"="jdbc:splice://localhost:1527/splicedb\;u
   ser=splice\;password=admin", "splice.tableName"="SPLICE.hcattest");
   ```

4. **Use HiveQL to select data in the Splice Machine table**

Once you've created your external table, you can use `SELECT` statements in the Hive shell to retrieve data from Splice Machine table. For example:

```
hive> select * from extTest1;
OK
1row 1
2row 2
3row 3
4row 4
```

If the external table that you created does not have the same number of columns as are in your Splice Machine table, then the Hive table's columns are mapped to columns in the Splice Machine table, as described in How Splice Machine Maps Columns, above.

Example 2: Table with Primary Key

This example uses a Splice Machine table, `tblA`, that has only string types and a primary key.

1. **Create and verify the Splice Machine table**
   This example uses a simple Splice Machine database table that we've created, `tblA`, which we verify:

   ```
   splice> describe tblA;
   COLUMN_NAME | TYPE_NAME | DES&|NUM&|COLUMN&|COLUMN_DEF|CHAR_OCTE&|IS_NUL
   L&
   ---------------------------------------------------------------------------
   -
   COL1         |CHAR       |NULL |NULL|20     |NULL      |40        |YES
   COL2         |VARCHAR    |NULL |NULL|56     |NULL      |112       |YES

   2 rows selected
   splice> select * from tblA;
   COL1         |COL2
   ------------------------------
   char        varchar 2
   char 1      varchar 1

   2 rows selected
   ```

2. **Create an external table in Hive**
   You need to create an external table in Hive, and connect that table with the Splice Machine table you just created. Type the following command into the Hive shell, substituting your `user` ID and `password`.

```
hive> CREATE EXTERNAL TABLE extTest2(col1 String, col2 varchar(56))
    STORED BY 'com.splicemachine.mrio.api.hive.SMStorageHandler'
    TBLPROPERTIES ("splice.jdbc"="jdbc:splice://localhost:1527/splicedb\;u
ser=splice\;password=admin", "splice.tableName"="SPLICE.tblA");
```

3. **Use HiveQL to select data in the Splice Machine table**

   Once you've created your external table, you can use `SELECT` statements in the Hive shell to retrieve data from Splice Machine table. For example:

   ```
   hive> select * from extTest2;
   OK
   char  varchar 2
   char 1varchar 1
   ```

Example 3: Table with No Primary Key

This example uses a Splice Machine table, `tblB`, that has a primary key and uses integer columns.

1. **Create and verify the Splice Machine table**

   This example uses a simple Splice Machine database table that we've created, `tblA`, which we verify:

   ```
   splice> describe tblA;
   COLUMN_NAME | TYPE_NAME | DES&|NUM&|COLUMN&|COLUMN_DEF|CHAR_OCTE&|IS_NUL
   L&
   ----------------------------------------------------------------------
   -
   COL1          |INTEGER    |0     |10  |10     |NULL       |NULL        |NO
   COL2          |INTEGER    |0     |10  |10     |NULL       |NULL        |YES
   COL3          |INTEGER    |0     |10  |10     |NULL       |NULL        |NO

   3 rows selected
   splice> select * from tblB;
   COL1          |COL2          |COL3
   -----------------------------
   1             |1             |1
   2             |2             |2

   2 rows selected
   ```

2. **Create an external table in Hive**

   You need to create an external table in Hive, and connect that table with the Splice Machine table you just created. Type the following command into the Hive shell, substituting your <mark>user</mark> ID and <mark>password</mark>.

```
hive> CREATE EXTERNAL TABLE extTest3(col1 String, col2 varchar(56))
    STORED BY 'com.splicemachine.mrio.api.hive.SMStorageHandler'
    TBLPROPERTIES ("splice.jdbc"="jdbc:splice://localhost:1527/splicedb\;u
ser=splice\;password=admin", "splice.tableName"="SPLICE.tblB");
```

3. **Select data from the Splice Machine input table**

   Once you've created your external table, you can use `SELECT` statements in the Hive shell to retrieve data from Splice Machine input table. table. For example:

```
hive> select * from extTest3;
OK
111
122
```

Splice Machine Map Reduce API

> **This is an On-Premise-Only topic!**   [Learn about our products](#)

The Splice Machine MapReduce API provides a simple programming interface to the Map Reduce Framework that is integrated into Splice Machine. You can use MapReduce to import data, export data, or for purposes such as implementing machine learning algorithms. One likely scenario for using the Splice Machine MapReduce API is for customers who already have a Hadoop cluster, want to use Splice Machine as their transactional database, and need to continue using their batch MapReduce jobs.

This topic includes a summary of the Java classes included in the API, and [presents an example](#) of using the MapReduce API.

Splice Machine MapReduce API Classes

The Splice Machine MapReduce API includes the following key classes:

Class	Description
`SpliceJob`	Creates a transaction for the MapReduce job.
`SMInputFormat`	Creates an object that: » uses Splice Machine to scan the table and decode the data » returns an `ExecRow` (typed data) object
`SMOutputFormat`	Creates an object that: » writes to a buffered cache » dumps the cache into Splice Machine » returns an `ExecRow` (typed data) object.
`SpliceMapReduceUtil`	A Helper class for writing MapReduce jobs in java; this class is used to initiate a mapper job or a reducer job, to set the number of reducers, and to add dependency jars.

> **NOTE:** Each transaction must manage its own commit and rollback operations.

For information about and examples of using Splice Machine with HCatalog, see the <a href="Using Splice Machine with HCatalog topic.

Example of Using the Splice Machine MapReduce API

This topic describes using the Splice Machine MapReduce API, `com.splicemachine.mrio.api`, a simple word count program that retrieves data from an input table, summarizes the count of initial character of each word, and writes the result to an output table.

1. **Define your input and output tables:**
   First, assign the name of the Splice Machine database table from which you want to retrieve data to a variable, and then assign a name for your output table to another variable:

   ```
   String inputTableName  = "WIKIDATA";
   String outputTableName = "USERTEST";
   ```

   You can specify table names using the *<schemaName>.<tableName>* format; if you don't specify a schema name, the default schema is assumed.

2. **Create a new job instance:**
   You need to create a new job instance and assign a name to it:

   ```
   Configuration config = HBaseConfiguration.create();
   Job job = new Job(config, "WordCount");
   ```

3. **Initialize your mapper job:**
   We initialize our sample job using the `initTableMapperJob` utility method:

   ```
   TableMapReduceUtil.initTableMapperJob(
       tableName,                         // input Splice Machine database ta
   ble
       scan,                          // a scan instance to control CF and att
   ribute selection
       MyMapper.class,                  // the mapper
       Text.class,                        // the mapper output key
       InitWritable.class,                // the mapper output value
       job,
       true,
       SpliceInputFormat.class);
   ```

4. **Retrieve values within your map function:**
   Our sample `map` function retrieves and parses a single row with specified columns.

```
public void map(ImmutableBytesWritable row, ExecRow value, Context contex
t)
                 throws InterruptedException, IOException {
    if(value != null) {
        try {
            DataValueDescriptor dataValDesc[]  = value.getRowArray();
            if(dataValDesc[0] != null) {}
            word = dataValDesc[0].getString();
            }
        catch (StandardException e) {
            // TODO Auto-generated catch block
            e.printStackTrace();
        }
        if(word != null) {
            Text key = new Text(word.charAt(0)+"");
            IntWritable val = new IntWritable(1);
            context.write(key, val);
        }
    }
}
```

5. **Manipulate and save the value with reduce function:**

   Our sample `reduce` function manipulates and saves the value by creating an `ExecRow` and filling in the row with the `execRow.setRowArray` method.

```
public void reduce(Text key, IterableIntWritable> values, Context contex
t)
                    throws IOException, InterruptedException {

    IteratorIntWritable> it=values.iterator();
    ExecRow execRow = new ValueRow(2);
    int sum = 0;
    String word = key.toString();
    while (it.hasNext()) {
        sum += it.next().get();
    }
    try{
        DataValueDescriptor []dataValDescs= {new SQLVarchar(word), new SQ
LInteger(sum)};
        execRow.setRowArray(dataValDescs);
        context.write(new ImmutableBytesWritable(Bytes.toBytes(word)), ex
ecRow);
    }
    catch(Exception E) {
        E.printStackTrace();
    }
}
```

6. **Commit or rollback the job:**
   If the job is successful, commit the transaction.

   ```
   job.commit();
   ```

   If the job fails, roll back the transaction.

   ```
   job.rollback();
   ```

Working with HBase

> **This is an On-Premise-Only topic!**   Learn about our products

This topic presents information about working in HBase with Splice Machine, in these sections:

» The Mapping Splice Machine Tables to HBase section shows you how to view your Splice Machine tables and how map the table names that we see in HBase to the more descriptive table names that we see in Splice Machine.

» The Accessing HBase Master section shows you how to access HBase Master so that you can peek under the surface a bit.

Mapping Splice Machine Tables to HBase

If you are looking at tables in the Splice Machine database that do not appear to match what you see in the HBase Master Web user interface, they may not match up. To view your Splice Machine tables in HBase, follow these steps:

1. **Use the Splice interactive command interface to view the tables:**
   You can use the `show tables` command to view the tables in your Splice Machine database:

```
TABLE_SCHEM        |TABLE_NAME          |CONGLOM_ID|REMARKS
-----------------------------------------------------------------
SYS                |SYSALIASES          |368       |
SYS                |SYSBACKUP           |944       |
SYS                |SYSBACKUPFILESET    |976       |
SYS                |SYSBACKUPITEMS      |1056      |
SYS                |SYSBACKUPJOBS       |1184      |
SYS                |SYSCHECKS           |384       |
SYS                |SYSCOLPERMS         |640       |
SYS                |SYSCOLUMNS          |80        |
SYS                |SYSCOLUMNSTATS      |1216      |
SYS                |SYSCONGLOMERATES    |48        |
SYS                |SYSCONSTRAINTS      |336       |
SYS                |SYSDEPENDS          |272       |
SYS                |SYSFILES            |288       |
SYS                |SYSFOREIGNKEYS      |240       |
SYS                |SYSKEYS             |320       |
SYS                |SYSPERMS            |816       |
SYS                |SYSPHYSICALSTATS    |1264      |
SYS                |SYSPRIMARYKEYS      |1424      |
SYS                |SYSROLES            |752       |
SYS                |SYSROUTINEPERMS     |672       |
SYS                |SYSSCHEMAS          |32        |
SYS                |SYSSEQUENCES        |800       |
SYS                |SYSSTATEMENTS       |256       |
SYS                |SYSTABLEPERMS       |656       |
SYS                |SYSTABLES           |64        |
SYS                |SYSTABLESTATS       |1280      |
SYS                |SYSTRIGGERS         |304       |
SYS                |SYSUSERS            |880       |
SYS                |SYSVIEWS            |352       |
SYSIBM             |SYSDUMMY1           |1296      |
```

2. **View the tables in the HBase Master Web Console:**

   To view the HBase tables, use the HBase Master Web Console, at `http://localhost:60010/`.

   Note that all of the user tables in the *Tables* section have numeric names; the numbers match the conglomerate number (`CONGLOM_ID`) values displayed by the `show tables` command.

Tables

User Tables   System Tables   Snapshots

93 table(s) in set. [Details]

Namespace	Table Name	Online Regions	Description
splice	1009	1	'splice:1009', {TABLE_ATTRIBUTES => {METADATA => {'indexDisplayName' => 'SYSCOLPERMS_INDEX3', 'tableDisplayName' => 'SYSCOLPERMS'}}, {NAME => 'V', VERSIONS => '2147483647'}
splice	1025	1	'splice:1025', {TABLE_ATTRIBUTES => {METADATA => {'indexDisplayName' => 'SYSROLES_INDEX2', 'tableDisplayName' => 'SYSROLES'}}, {NAME => 'V', VERSIONS => '2147483647'}
splice	1041	1	'splice:1041', {TABLE_ATTRIBUTES => {METADATA => {'indexDisplayName' => 'SYSUSERS_INDEX1', 'tableDisplayName' => 'SYSUSERS'}}, {NAME => 'V', VERSIONS => '2147483647'}

These numbers are used in HBase as directory names; you can find those directories in your file system and examine the tables directly.

Accessing HBase Master

If you are an HBase veteran or someone interested in seeing a little of what goes on under the surface, you can access HBase Master with the default HBase URL:

```
http://localhost:60010
```

Because Splice Machine encodes and compresses the data for space efficiency, the actual data in your tables is virtually unreadable.

> **NOTE:** In this version, Splice Machine has purposely left ports 60010 and 60030 open for people to see the HBase tables. If security is an issue in your deployment, you can easily block this access.

Using Functions and Stored Procedures

This topic provides an overview of writing and using functions and stored procedures in Splice Machine.

About User-Defined Functions

You can create user-defined database functions that can be evaluated in SQL statements; these functions can be invoked where most other built-in functions are allowed, including within SQL expressions and `SELECT` statement. Functions must be deterministic, and cannot be used to make changes to the database.

You can create two kinds of functions:

» Scalar functions, which always return a single value (or `NULL`),

» Table functions, which return a table.

When you invoke a function within a `SELECT` statement, it is applied to each retrieved row. For example:

```
SELECT ID, Salary, MyAdjustSalaryFcn(Salary) FROM SPLICEBBALL.Salaries;
```

This `SELECT` will execute the `MyAdjustSalaryFcn` to the `Salary` value for each player in the table.

About Stored Procedures

You can group a set of SQL commands together with variable and logic into a stored procedure, which is a subroutine that is stored in your database's data dictionary. Unlike user-defined functions, a stored procedure is not an expression and can only be invoked using the `CALL` statement. Stored procedures allow you to modify the database and return `Result Sets` or nothing at all.

Stored procedures can be used for situations where a complex set of SQL statements are required to process something, and that process is used by various applications; creating a stored procedure increases performance efficiency. They are typically used for:

» checking business rules and validating data before performing actions

» performing significant processing of data with the inputs to the procedure

Comparison of Functions and Stored Procedures

Here's a comparison of Splice Machine functions and stored procedures:

Database Function	Stored Procedure
A Splice Machine database function: » must be written as a public static method in a Java public class » is executed in exactly the same manner as are public static methods in Java » can have multiple input parameters » always returns a single value (which can be null) » cannot modify data in the database	Splice Machine stored procedures can: » return result sets or return nothing at all » issue `update`, `insert`, and `delete` statements » perform DDL statements such as `create` and `drop` » consolidate and centralize code » reduce network traffic and increase execution speed
Can be used in `SELECT` statements.	Cannot be used in in `SELECT` statements. Must be invoked using a `CALL` statement.
Create with the CREATE FUNCTION statement, which is described in our SQL Reference book. You can find an example in the Function and Stored Procedure Examples topic in this section.	Create with the CREATE PROCEDURE statement, which is described in our SQL Reference book. You can find an example in the Function and Stored Procedure Examples topic in this section.

Operations in Which You Can Use Functions and Stored Procedures

The following table provides a list of the differences between functions and stored procedures with regard to when and where they can be used:

Operation	Functions	Stored Procedures
Execute in an SQL Statement	Yes	No
Execute in a Trigger	Yes	Triggers that execute before an operation (*before triggers*) cannot modify SQL data.
Process `OUT / INOUT` Parameters	No	Yes
Return `Resultset(s)`	No	Yes
Execute SQL `Select`	Yes	Yes

Operation	Functions	Stored Procedures
Execute SQL `Update/ Insert/Delete`	No	Yes
Execute DDL (`Create/Drop`)	No	Yes

Viewing Functions and Stored Procedures

You can use the `show functions` and `show procedures` commands in the `splice>` command line interface to display the functions and stored procedures available in your database:

Command	Output
`splice> show functions;`	All functions defined in your database
`splice> show functions in SYSCS_UTIL;`	All functions in the `SYSCS_UTIL` schema in your database
`splice> show procedures;`	All stored procedures defined in your database
`splice> show procedures in SYSCS_UTIL;`	All stored procedures in the `SYSCS_UTIL` schema in your database

Writing and Deploying Functions and Stored Procedures

The remainder of this section presents information about and examples of writing functions and stored procedures for use with Splice Machine, in these topics:

» Writing Functions and Stored Procedures shows you the steps required to write functions stored procedures and add them to your Splice Machine database.

» Storing and Updating Functions and Stored Procedures tells you how to store new JAR files, replace JAR files, and remove JAR files in your Splice Machine database.

» Examples of Splice Machine Functions and Stored Procedures provides you with examples of functions and stored procedures.

See Also

» CREATE_FUNCTION

» CREATE_PROCEDURE

Writing Functions and Stored Procedures

This topic shows you the steps required to write functions and stored procedures for use in your Splice Machine database.

» Refer to the [Introduction to Functions and Stored Procedures](#) topic in this section for an overview and comparison of functions and stored procedures.

» Refer to the [Storing and Updating Functions and Stored Procedures](#) topic in this section for information about storing your compiled code and updating the `CLASSPATH` to ensure that Splice Machine can find your code.

» Refer to the [Functions and Stored Procedure Examples](#) topic in this section for complete sample code for both a function and a stored procedure.

Note that the processes for adding functions and stored procedures to your Splice Machine database are quite similar; however, there are some important differences, so we've separated them into their own sections below.

Writing a Function in Splice Machine

Follow the steps below to write a Splice Machine database function.

1. **Create a Java method**
   Each function maps to a Java method. For example:

   ```
   package com.splicemachine.cs.function;

   public class Functions {
       public static int addNumbers(int val1, int val2) {
           return val1 + val2;
       }
   }
   ```

2. **Create the function in the database**
   You can find the complete syntax for `CREATE FUNCTION` in the *Splice Machine SQL Reference* manual.

   Here's a quick example of creating a function. In this example, `com.splicemachine.cs.function` is the package, `Functions` is the class name, and `addNumbers` is the method name:

   ```
   CREATE FUNCTION add(val1 int, val2 int)
       RETURNS integer
       LANGUAGE JAVA
       PARAMETER STYLE JAVA
       NO SQL
       EXTERNAL NAME 'com.splicemachine.cs.function.Functions.addNumbers';
   ```

3. **Store your compiled Jar file and update your CLASSPATH**
   Follow the instructions in the [Storing and Updating Functions and Stored Procedures](#) topic in this section to:

» store your Jar file

» update the class path so that Splice Machine can find your code when the function is called.

Invoke your function

You can invoke functions just like you would call any built-in database function. For example, if you're using the Splice Machine command line interface (*CLI*), and have created a function named `add`, you could use a statement like the following:

```
SELECT add(1,2) FROM SYS.SYSTABLES;
```

Writing a Stored Procedure in Splice Machine

Follow the steps below to write a Splice Machine database stored procedure.

1. **Write your custom stored procedure:**

   Here is a very simple stored procedure that uses JDBC:

```
package org.splicetest.customprocs;
import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.PreparedStatement;
import java.sql.ResultSet;
import java.sql.SQLException;


/**
 * This class contains custom stored procedures that will be dynamically
loaded into the Splice Machine
 * database with the SQLJ jar file loading system procedures.
 *
 * @author Splice Machine
 */


public class CustomSpliceProcs {
 /**
  * Return the names for all tables in the database.
  *
  * @param rs    result set containing names of all the tables in teh dat
abase
  */

  public static void GET_TABLE_NAMES(ResultSet[] rs)
         throws SQLException
   {
        Connection conn = DriverManager.getConnection("jdbc:default:conne
ction");
        PreparedStatement pstmt = conn.prepareStatement("select * from sy
s.systables");
        rs[0] = pstmt.executeQuery();
        conn.close();
   }
}
```

You can use any Java IDE or text edit to write your code.

You can find additional examples in the [Functions and Stored Procedure Examples](#) topic in this section.

> **NOTE:** See the information about [working with ResultSets](#) in the next section.

2. **Compile your code and build a Jar file**
   You now need to compile your stored procedure and build a jar file for it.

   You can use any Java IDE or build tool, such as *Maven* or *Ant*, to accomplish this. Alternatively, you can use the *javac* Java compiler and the *Java Archive* tool packaged with the JDK.

3. **Copy the Jar file to a cluster node**

   Next, copy your custom Jar file to a region server (any node running an HBase region server) in your Splice Machine cluster. You can copy the file anywhere that allows the splice> interface to access it.

   You can use any remote copying tool, such as scp or ftp. For example:

   ```
   scp custom-splice-procs-1.0.2-SNAPSHOT.jar splice@myServer:myDir
   ```

   See the [Storing and Updating Functions and Stored Procedures](#) topic in this section for more information.

4. **Deploy the Jar file to your cluster**

   Deploying the Jar file requires you to install the file in your database, and to add it to your database's CLASSPATH. You can accomplish both of these steps by calling built-in system procedures from the splice> command line interpreter. For example:

   ```
   CALL SQLJ.INSTALL_JAR(
      '/Users/splice/my-directory-for-jar-files/custom-splice-procs-2.7-SNAPS
   HOT.jar',
      'SPLICE.CUSTOM_SPLICE_PROCS_JAR', 0);

   CALL SYSCS_UTIL.SYSCS_SET_GLOBAL_DATABASE_PROPERTY(
   'derby.database.classpath', 'SPLICE.CUSTOM_SPLICE_PROCS_JAR');
   ```

   The [SQLJ.INSTALL_JAR](#) system procedure uploads the jar file from the local file system where splice> is executing into the HDFS:

   » If you are running a cluster, the Jar files are stored under the /hbase/splicedb/jar directory in HDFS (or MapR-FS).

   » If you are running in standalone mode, the Jar files are stored on the local file system under the splicedb/jar directory in the Splice install directory.

5. **Register your stored procedure with Splice Machine**

   Register your stored procedure with the database by calling the [CREATE PROCEDURE](#) statement. For example:

   ```
   CREATE PROCEDURE SPLICE.GET_TABLE_NAMES()
      PARAMETER STYLE JAVA
      READS SQL DATA
      LANGUAGE JAVA
      DYNAMIC RESULT SETS 1
      EXTERNAL NAME 'org.splicetest.customprocs.CustomSpliceProcs.GET_TABL
   E_NAMES';
   ```

   Note that after running the above CREATE PROCEDURE statement, your procedure will show up in the list of available procedures when you run the Splice Machine show procedures command.

   You can find the complete syntax for [CREATE PROCEDURE](#) in the *Splice Machine SQL Reference* manual.

6. **Run your stored procedure**

You can run your stored procedure by calling it from the `splice>` prompt. For example:

```
splice> call SPLICE.GET_TABLE_NAMES();
```

7. **Updating/Reloading your stored procedure**

   If you make changes to your procedure's code, you need to create a new Jar file and reload that into your databaseby calling the `SQLJ.REPLACE_JAR` system procedure:

   ```
   CALL SQLJ.REPLACE_JAR(
      '/Users/splice/my-directory-for-jar-files/custom-splice-procs-2.7-SNAPS
   HOT.jar',
      'SPLICE.CUSTOM_SPLICE_PROCS_JAR');
   ```

Working with ResultSets

Splice Machine follows the SQL-J part 1 standard for returning `ResultSets` through Java procedures. Each `ResultSet` is returned through one of the parameters passed to the java method. For example, the `resultSet` parameter in the `MY_TEST_PROC` method in our `ExampleStoredProcedure` class:

```
public class ExampleStoredProcedure {
    public static void MY_TEST_PROC(String myInput, ResultSet[] resultSet) throws SQL
Exception {
        ...
    }
}
```

Here are a set of things you should know about `ResultSets[]` in stored procedures:

» The `ResultSets` are returned in the order in which they were created.

» The `ResultSets` must be open and generated from the `jdbc:default:connection` default connection. Any other `ResultSets` are ignored.

» If you close the statement that created the `ResultSet` within the procedure's method, that closes the `ResultSet` you want. Instead, you can close the connection.

» The Splice Machine database engine itself creates the one element `ResultSet` arrays that hold the returned `ResultSets`.

» Although the [CREATE PROCEDURE](#) call allows you to specify the number of `DYNAMIC RESULT SETs`, we currently only support returning a single `ResultSet`.

Storing and Updating Splice Machine Functions and Stored Procedures

This topic describes how to store and update your compiled Java Jar (`.jar`) files when developing stored procedures and functions for Splice Machine.

> **NOTE:** Jar files are not versioned: the `GENERATIONID` is always zero. You can view the metadata for the Jar files in the Splice data dictionary by executing this query: `select * from sys.sysfiles;`

Adding a Jar File

To add a new Jar file to your Splice Machine database, use the `splice>` command line interface to store the Jar and then update your `CLASSPATH` property so that your code can be found:

> **NOTE:** When Splice Machine is searching for a class to load, it first searches the system `CLASSPATH`. If the class is not found in the traditional system class path, Splice Machine then searches the class path set as the value of the `derby.database.classpath` property.

1. **Load your Jar file into the Splice Machine database**

   ```
   splice> CALL SQLJ.INSTALL_JAR(
               '/Users/me/dev/workspace/examples/bin/example.jar',
               'SPLICE.MY_EXAMPLE_APP', 0);
   ```

   Please refer to the [SQLJ.INSTALL_JAR](#) topic for more information about using this system procedure. To summarize:

   >> The first argument is the path on your computer to your Jar file.

   >> The second argument is the name for the stored procedure Jar file in your database, in `schema.name` format.

   >> The third argument is currently unused but required; use `0` as its value.

2. **Update your CLASSPATH**
   You need to update your `CLASSPATH` so that Splice Machine can find your code. You can do this by using the [SYSCS_UTIL.SYSCS_SET_GLOBAL_DATABASE_PROPERTY](#) system procedure to update the `derby.database.classpath` property:

```
splice> CALL SYSCS_UTIL.SYSCS_SET_GLOBAL_DATABASE_PROPERTY(
               'derby.database.classpath',
               'SPLICE.MY_EXAMPLE_APP');
```

Note that if you've developed more than one Jar file, you can update the `derby.database.classpath` property with multiple Jars by separating the Jar file names with colons when you call the SYSCS_UTIL.SYSCS_SET_GLOBAL_DATABASE_PROPERTY system procedure . For example:

```
splice> CALL SYSCS_UTIL.SYSCS_SET_GLOBAL_DATABASE_PROPERTY(
               'derby.database.classpath',
               'SPLICE.MY_EXAMPLE_APP:SPLICE.YOUR_EXAMPLE');
```

Updating a Jar File

You can use the `splice>` command line interface to replace a Jar file:

1. **Replace the stored Jar file**

   ```
   splice> CALL SQLJ.REPLACE_JAR(
                  '/Users/me/dev/workspace/examples/bin/example.jar',
                  'SPLICE.MY_EXAMPLE_APP');
   ```

   Please refer to the SQLJ.REPLACE_JAR topic for more information about using this system procedure. To summarize:

   >> The first argument is the path on your computer to your Jar file.

   >> The second argument is the name for the stored procedure Jar file in your database, in `schema.name` format.

Deleting a Jar File

You can use the `splice>` command line interface to delete a Jar file:

1. **Delete a stored Jar file**

   ```
   splice> CALL SQLJ.REMOVE_JAR('SPLICE.MY_EXAMPLE_APP', 0);
   ```

Please refer to the `SQLJ.REMOVE_JAR` topic for more information about using this system procedure. To summarize:

» The first argument is the name for the stored procedure Jar file in your database, in `schema.name` format.

» The second argument is currently unused but required; use `0` as its value.

NOTE: The Jar file operations (the `SQLJ.REMOVE_JAR` system procedures) are not executed within transactions, which means that committing or rolling back a transaction will not have any impact on these operations.

Examples of Splice Machine Functions and Stored Procedures

This topic walks you through creating, storing, and using a sample database function and a sample database stored procedure, in these sections:

Creating and Using a Sample Function in Splice Machine

This section walks you through creating a sample function named `word_limiter` that limits the number of words in a string; for example, given this sentence:

```
Today is a wonderful day and I am looking forward to going to the beach.
```

If you tell `word_limiter` to return the first five words in the sentence, the returned string would be:

```
Today is a wonderful day
```

Follow these steps to define and use the `word_limiter` function:

1. **Define inputs and outputs**
   We have two inputs:

   » the sentence that we want to limit

   » the number of words to which we want to limit the output

   The output is a string that contains the limited words.

2. **Create the shell of our Java class.**
   We create a class named `ExampleStringFunctions` in the package `com.splicemachine.examples`.

   ```
   package com.splicemachine.example;

   public class ExampleStringFunctions {...
   }
   ```

3. **Create the `wordLimiter` static method**
   This method contains the logic for returning the first n number of words:

```
package com.splicemachine.example;

public class ExampleStringFunctions {
    /**
     * Truncates a string to the number of words specified.  An input of
     * "Today is a wonderful day and I am looking forward to going to th
e beach.", 5
     * will return "Today is a wonderful day".
     *
     * @param inboundSentence
     * @param numberOfWords
     * @return
     */
    public static String wordLimiter(String inboundSentence, int numberOf
Words) {
        String truncatedString = "";
        if(inboundSentence != null) {
            String[] splitBySpace = inboundSentence.split("\\s+");
            if(splitBySpace.length = numberOfWords) {
                truncatedString = inboundSentence;
            } else {
                StringBuilder sb = new StringBuilder();
                for(int i=0; inumberOfWords; i++) {
                    if(i > 0) sb.append(" ");
                    sb.append(splitBySpace[i]);
                }
                truncatedString = sb.toString();
            }
        }
        return truncatedString;
    }
}
```

4. **Compile the class and store the jar file**

   After you compile your class, make sure that the jar file is in a directory in your `classpath`, so that it can be found. You can find more information about this in the [Storing and Updating Functions and Stored Procedures](#) topic in this section.

   If you're using the standalone version of Splice Machine, you can use the following command line interface command:

```
splice> CALL SQLJ.INSTALL_JAR(
            '/Users/me/dev/workspace/examples/bin/example.jar',
            'SPLICE.MY_EXAMPLE_APP', 0);
```

   You must also update the database class path:

```
splice> CALL SYSCS_UTIL.SYSCS_SET_GLOBAL_DATABASE_PROPERTY(
                'derby.database.classpath',
                'SPLICE.MY_EXAMPLE_APP');
```

5. **Define the function in Splice Machine**

   You can find the complete syntax for <u>CREATE FUNCTION</u> in the *Splice Machine SQL Reference* manual. For our example function, we enter the following command at the `splice>` prompt:

```
splice> CREATE FUNCTION WORD_LIMITER(
                    MY_SENTENCE VARCHAR(9999),
                    NUM_WORDS INT) RETURNS VARCHAR(9999)
LANGUAGE JAVA
PARAMETER STYLE JAVA
NO SQL
EXTERNAL NAME 'com.splicemachine.example.ExampleStringFunctions.wordLimit
er';
```

6. **Run the function**

   You can run the function with this syntax:

```
splice> SELECT WORD_LIMITER(
              'Today is a wonderful day and I am looking forward to going t
o the beach.', 5)
    FROM SYSIBM.SYSDUMMY1;
```

Creating and Using a Sample Stored Procedure in Splice Machine

In this section , we create a stored procedure named `GET_INVENTORY_FOR_SKU` that retrieves all of the inventory records for a specific product by using the product's sku code. The input to this procedure is the sku code, and the output is a resultset of records from the inventory table.

Follow these steps to define and use the `GET_INVENTORY_FOR_SKU` function:

1. **Create and populate the inventory table**

   Connect to Splice Machine and create the following table from the command prompt or from an SQL client, using the following statements:

```
CREATE TABLE INVENTORY (
    SKU_CODE VARCHAR(30),
    WAREHOUSE BIGINT,
    QUANTITY BIGINT
);

INSERT INTO INVENTORY VALUES ('ABC123',1,50),('ABC123',2,100),('ABC12
3',3,60),('XYZ987',1,20),('XYZ321',2,0);
```

2. **Create the shell of our Java class.**

   We create a class named `ExampleStringFunctions` in the package `com.splicemachine.examples`.

   ```
   package com.splicemachine.example;

   public class ExampleStoredProcedure{...
   }
   ```

3. **Create the `getSkuInventory` static method**

   This method contains the logic for retrieving inventory records for the specified sku.

   ```
   package com.splicemachine.example;

   import java.sql.Connection;
   import java.sql.DriverManager;
   import java.sql.PreparedStatement;
   import java.sql.ResultSet;
   import java.sql.SQLException;
   public class ExampleStoredProcedure {
       public static void getSkuInventory(String skuCode, ResultSet[] resultS
   et) throws SQLException {
           try {
               Connection con = DriverManager.getConnection("jdbc:default:con
   nection");
               String sql = "select * from INVENTORY " + "where SKU_CODE =
   ?";
               PreparedStatement ps = con.prepareStatement(sql);
               ps.setString(1, skuCode);
               resultSet[0] = ps.executeQuery();;
           } catch (SQLException e) {
               throw e;
           }
       }
   }
   ```

4. **Compile the class and store the jar file**

After you compile your class, make sure that the jar file is in a directory in your `classpath`, so that it can be found. You can find more information about this in the [Storing and Updating Functions and Stored Procedures](#) topic in this section.

If you're using the standalone version of Splice Machine, you can use the following command line interface command:

```
splice> CALL SQLJ.INSTALL_JAR('/Users/me/dev/workspace/examples/bin/examp
le.jar', 'SPLICE.GET_SKU_INVENTORY', 0);
```

You must also update the database class path:

```
splice> CALL SYSCS_UTIL.SYSCS_SET_GLOBAL_DATABASE_PROPERTY('derby.databas
e.classpath', 'SPLICE.GET_SKU_INVENTORY');
```

5. **Define the procedure in Splice Machine**
   For our procedure, we enter the following command at the `splice>` prompt:

```
splice> CREATE PROCEDURE GET_INVENTORY_FOR_SKU(SKU_CODE VARCHAR(30))
LANGUAGE JAVA
PARAMETER STYLE JAVA
READS SQL DATA
EXTERNAL NAME 'com.splicemachine.example.ExampleStoredProcedure.getSkuInv
entory';
```

6. **Run the stored procedure**
   You can run the procedure with this syntax:

```
splice> call GET_INVENTORY_FOR_SKU('ABC123');
```

Ingesting and Streaming Data With Splice Machine

This section provides tutorials to help you to ingest (import) and stream data with Splice Machine:

» Importing Data Tutorial provides an overview of importing data into Splice Machine, including examples and handy tips.

» The Configuring a Kafka Feed shows you how to how to configure a Kafka feed to Splice Machine for data streaming.

» The Create a Kafka Producer shows you how to create a Kafka producer that puts messages on a Kafka queue.

» Streaming MQTT Data walks you through using MQTT Spark streaming with Splice Machine.

» Integrating Apache Storm with Splice Machine walks you through building and running two different examples of integrating Storm with Splice Machine.

Uploading Your Data to an S3 Bucket

You can easily load data into your Splice Machine database from an Amazon Web Services (AWS) S3 bucket. This tutorial walks you through creating an S3 bucket (if you need to) and uploading your data to that bucket for subsequent use with Splice Machine.

> **NOTE:** For more information about S3 buckets, see the AWS documentation.

After completing the configuration steps described here, you'll be able to load data into Splice Machine from an S3 bucket.

Create and Upload Data to an AWS S3 Bucket

Follow these steps to first create a new bucket (if necessary) and upload data to a folder in an AWS S3 bucket:

1. **Log in to the AWS Database Console**
   Your permissions must allow for you to create an S3 bucket.

2. **Select Services at the top of the dashboard**

3. **Select S3 in the Storage section:**
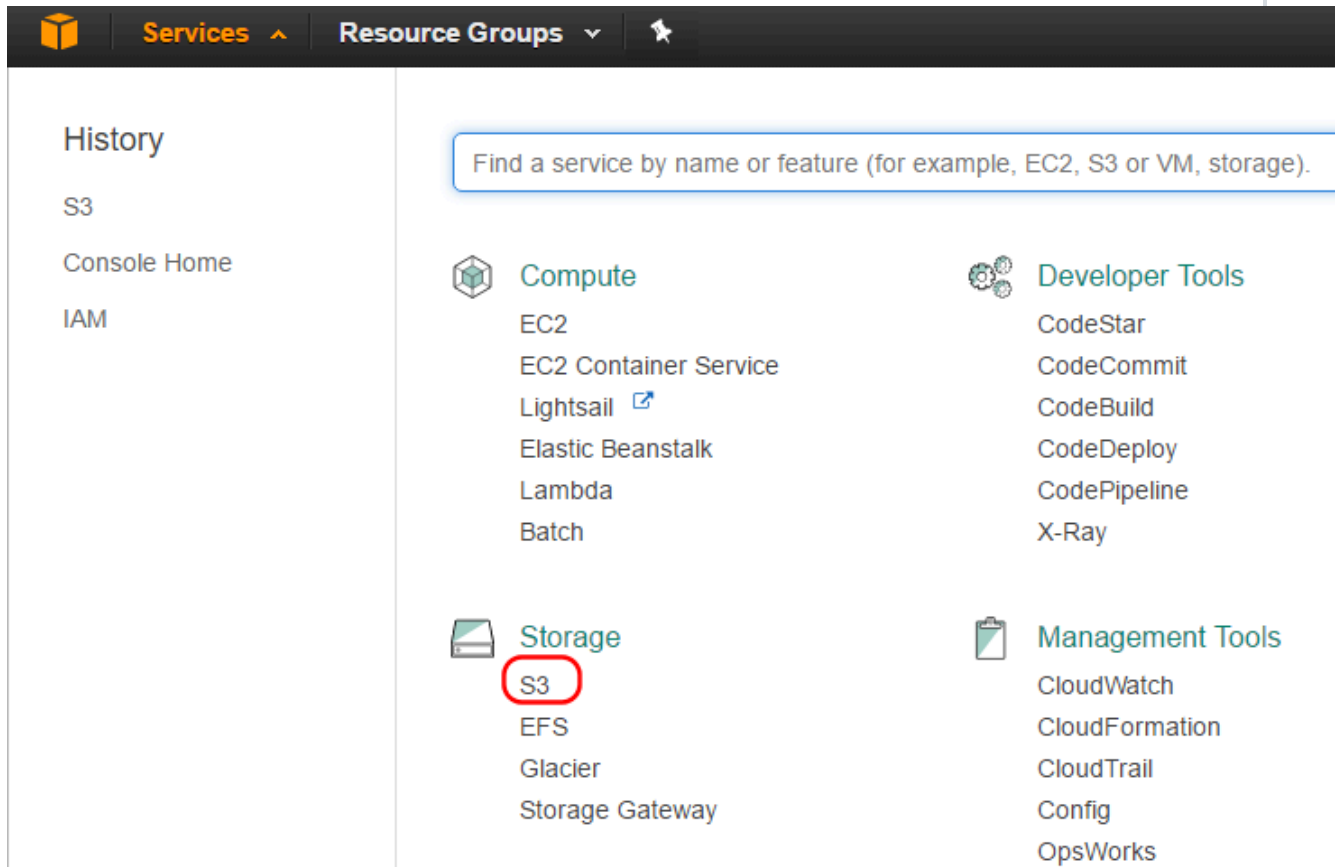
4. **Create a new bucket**

   a. Select **Create Bucket** from the S3 screen
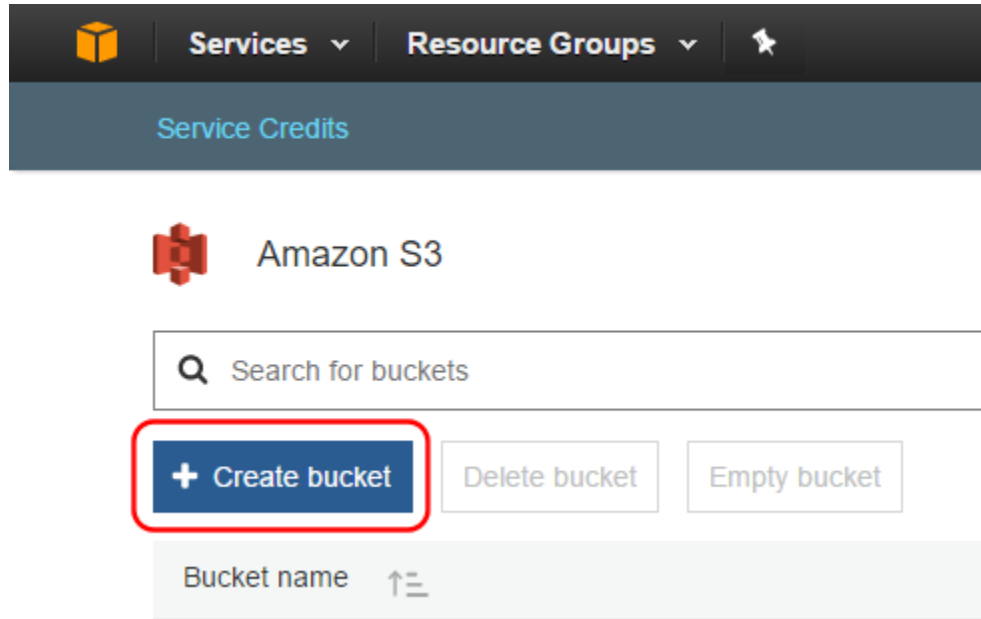
b. Provide a name and select a region for your bucket

The name you select must be unique; AWS will notify you if you attempt to use an already-used name. For optimal performance, choose a region that is close to the physical location of your data; for example:

c. Click the **Next** button to advance to the property settings for your new bucket:

You can click one of the **Learn more** buttons to view or modify details.

d. Click the **Next** button to advance to view or modify permissions settings for your new bucket:

e. Click **Next** to review your settings for the new bucket, and then click the **Create bucket** button to create your new S3 bucket. You'll then land on your S3 Management screen.

5. **Upload data to your bucket**
    After you create the bucket:

    a. Select **Create folder**, enter a name for the new folder, and click the **Save** button.
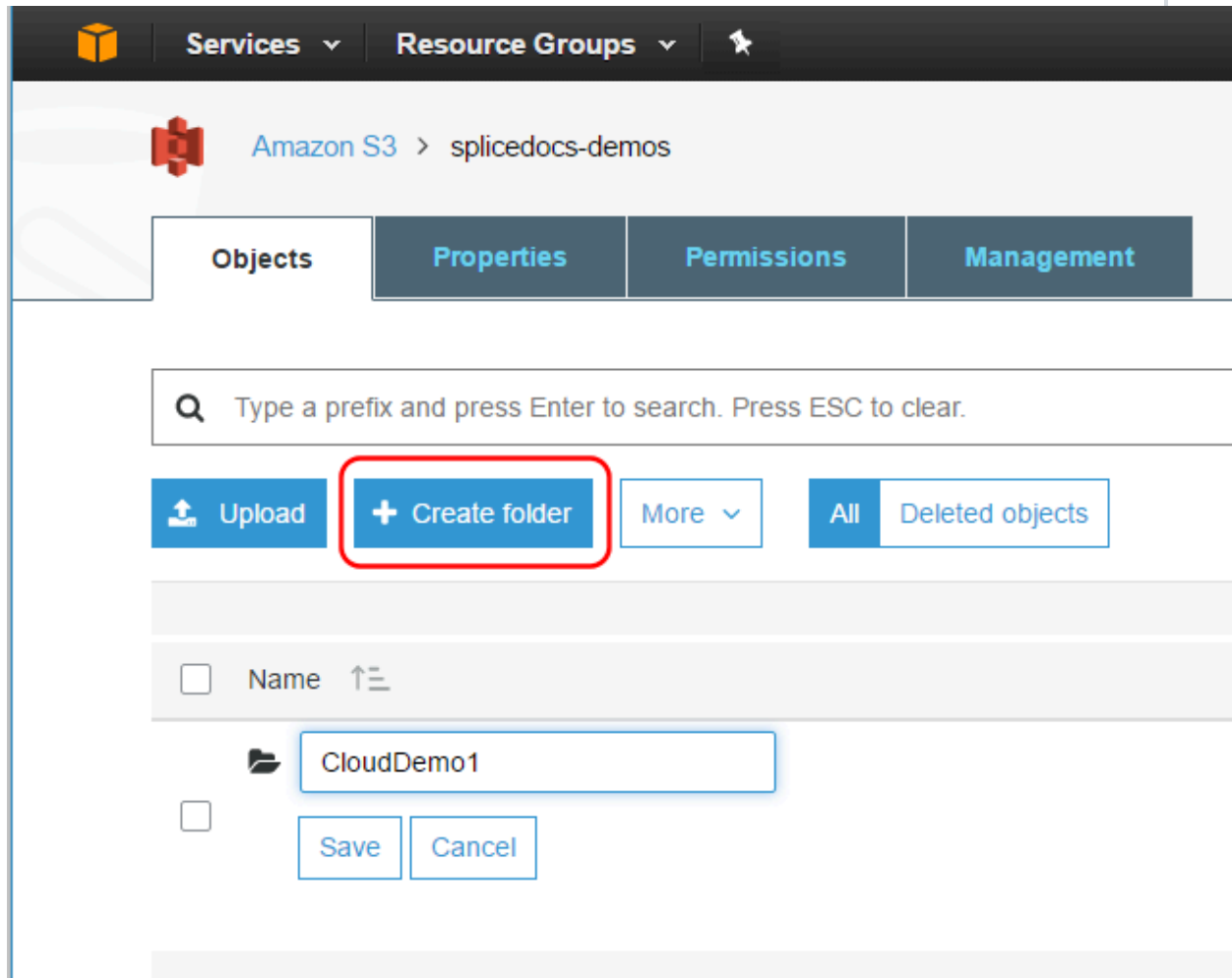
b. Click the **Upload** button to select file(s) to upload to your new bucket folder. You can then drag files into the upload screen, or click **Add Files** and navigate to the files you want to upload to your folder.
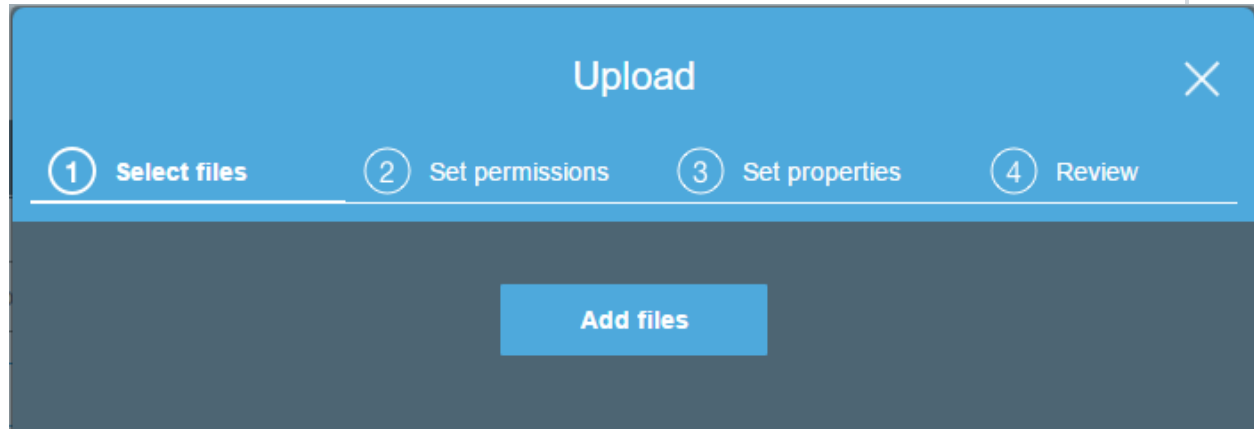
    c. You can then optionally set permissions and properties for the files you are uploading. Once you're done, click the Upload button, and AWS will copy the files into the folder in your S3 bucket.

6. **Make sure Splice Machine can access your bucket:**

    Review the IAM configuration options in our [Configuring an S3 Bucket for Splice Machine Access](#) tutorial to allow Splice Machine to import your data.

Configuring an S3 Bucket for Splice Machine Access

Splice Machine can access S3 buckets, making it easy for you to store and manage your data on AWS. To do so, you need to configure your AWS controls to allow that access. This topic walks you through the required steps.

> **NOTE:** You must have administrative access to AWS to configure your S3 buckets for Splice Machine.

Configure S3 Bucket Access

You can follow these steps to configure access to your S3 bucket(s) for Splice Machine; when you're done, you will have:

» created an IAM policy for an S3 bucket

» created an IAM user

» generated access credential for that user

» attached the security policy to that user

1. **Log in to the AWS Database Console**
   You must have administrative access to configure S3 bucket access.

2. **Select Services at the top of the dashboard**

3. **Access the IAM (Identify and Access Management) service:**
   Select **IAM** in the **Security, Identity & Compliance** section:
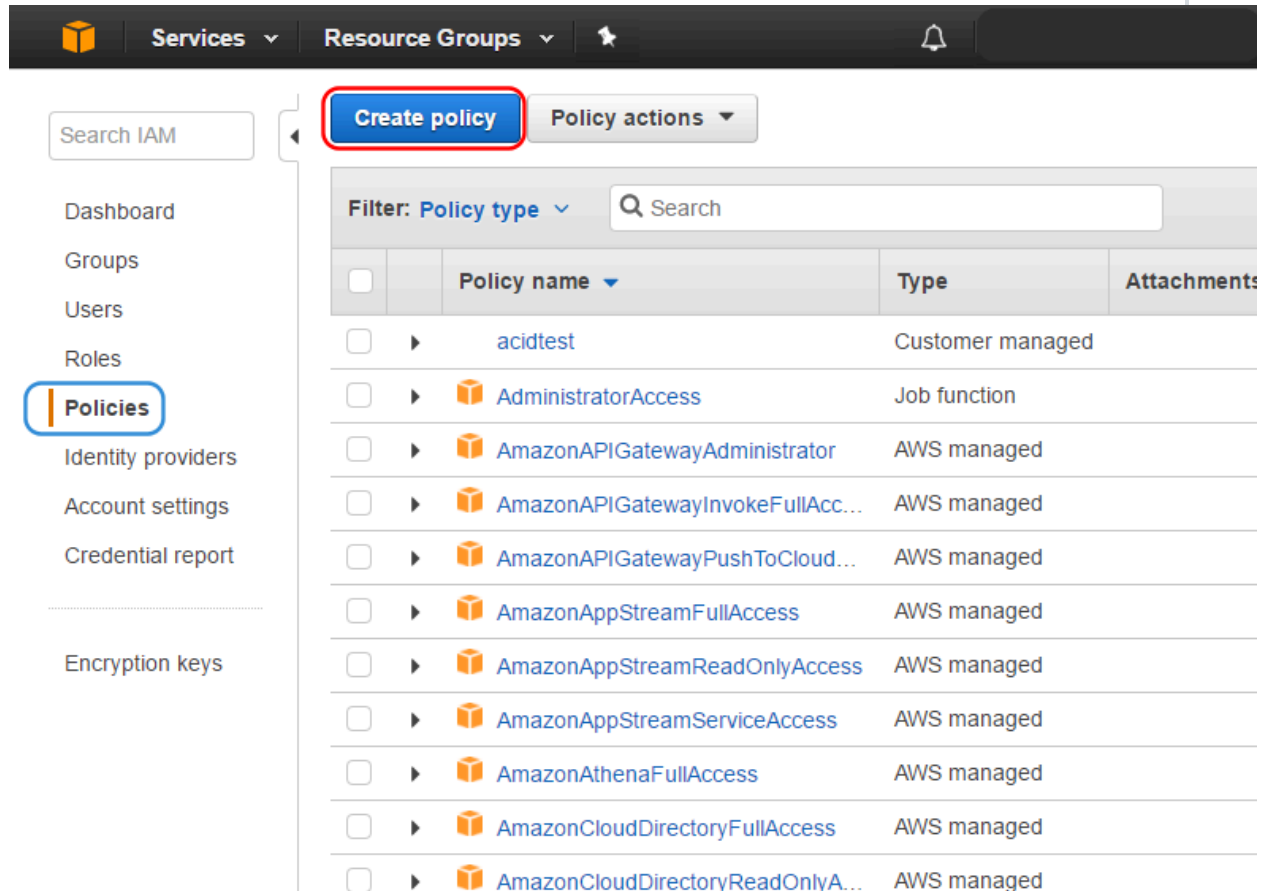
4. **Create a new policy:**

    a.   Select **Policies** from the IAM screen, then select **Create Policy:**

    b.   Select **Create Your Own Policy** to enter your own policy:

c. In the **Review Policy** section, which should be pre-selected, specify a name for this policy (we call it `splice_access`):

d. Paste the following JSON object specification into the **Policy Document** field and then modify the highlighted values to specify your bucket name and folder path.

```
        {
            "Version": "2017-04-17",
            "Statement": [
                {
                    "Effect": "Allow",
                    "Action": [
                      "s3:PutObject",
                      "s3:GetObject",
                      "s3:GetObjectVersion",
                      "s3:DeleteObject",
                      "s3:DeleteObjectVersion"
                    ],
                    "Resource": "arn:aws:s3:::<bucket_name>/<prefix>/*"
                },
                {
                    "Effect": "Allow",
                    "Action": "s3:ListBucket",
                    "Resource": "arn:aws:s3:::<bucket_name>",
                    "Condition": {
                        "StringLike": {
                            "s3:prefix": [
                                "<prefix>/*"
                            ]
                        }
                    }
                },
                {
                    "Effect": "Allow",
                    "Action": "s3:GetAccelerateConfiguration",
                    "Resource": "arn:aws:s3:::<bucket_name>"
                }
            ]
        }
```

   e. Click **Validate Policy** to verify that your policy settings are valid.

f. Click **Create Policy** to create and save the policy.

5. **Add Splice Machine as a user:**
   After you create the policy:

   a. Select **Users** from the left-hand navigation pane.

   b. Click **Add User**.

   c. Enter a **User name** (we've used *SpliceMachine*) and select **Programmatic access** as the access type:

Add user

1 Details   2 Permissions   3 Review   4 Complete

Set user details

You can add multiple users at once with the same access type and permissions. Learn more

User name*   [SpliceMachine]

➕ Add another user

Select AWS access type

Select how these users will access AWS. Access keys and autogenerated passwords are provided in the last step. Learn more

Access type*   ☑ **Programmatic access**
Enables an **access key ID** and **secret access key** for the AWS API, CLI, SDK, and other development tools.

☐ **AWS Management Console access**
Enables a **password** that allows users to sign-in to the AWS Management Console.

* Required   Cancel   **Next: Permissions**

d. Click **Attach existing policies directly**.

e. Select the policy you just created and click **Next**:

Add user

1 Details   2 Permissions   3 Review   4 Complete

Set permissions for SpliceMachine

Add user to group   Copy permissions from existing user   Attach existing policies directly

Attach one or more existing policies directly to the user or create a new policy. Learn more

Create policy   🔄 Refresh

Filter: Policy type ▾   🔍 Search   Showing 250 results

		Policy name ▾	Type	Attachments ▾	Description
☐	▸	SelfPasswordChange	Customer managed	1	
☑	▸	splice_access	Customer managed	0	

f. Review your settings, then click **Create User**.

6. **Save your access credentials**
You **must** write down your Access key ID and secret access key; you will be unable to recover the secret access key.

Add user

1 Details 2 Permissions 3 Review **4 Complete**

✅ Success
You successfully created the users shown below. You can view and download user security credentials. You can also email users instructions for signing in to the AWS Management Console. This is the last time these credentials will be available to download. However, you can create new credentials at any time.

Users with AWS Management Console access can sign-in at: https://sfc-samples.signin.aws.amazon.com/console

⬇ Download .csv

		User	Access key ID	Secret access key
▸	✅	**SpliceMachine**	ALJBK7XFF4QKZCCAOPPR	********* Show

Close

Splice Machine strongly recommends that you click the **Download .csv** button and save your credentials in a file for future reference. Once you close this screen, you'll be unable to display your secret access key.

Importing Data Into Your Splice Machine Database

This tutorial guides you through importing (loading) data into your Splice Machine database. It contains these topics:

Tutorial Topic	Description
1: Tutorial Overview	*This topic*. Introduces the import options that are available to you and helps you determine which option best meets your needs.
2: Parameter Usage	Provides detailed specifications of the parameter values you must supply to the import procedures.
3: Input Data Handling	Provides detailed information and tips about input data handling during ingestion.
4: Error Handling	Helps you to understand and use logging to discover and repair any input data problems that occur during an ingestion process.
5: Usage Examples	Walks you through examples of importing data with the `SYSCS_UTIL.IMPORT_DATA`, `SYSCS_UTIL.UPSERT_DATA_FROM_FILE`, and `SYSCS_UTIL.MERGE_DATA_FROM_FILE` system procedures.
6: Bulk HFile Examples	Walks you through examples of using the `SYSCS_UTIL.BULK_IMPORT_HFILE` system procedure.
7: Importing TPCH Data	Walks you through importing TPCH sample data into your database.

Overview of Importing Data Into Your Database

The remainder of this topic introduces you to importing (loading) data into your Splice Machine database. It summarizes the different import procedures we provide, presents a quick look at the procedure declarations, and helps you to decide which one matches your conditions. It contains the following sections:

» Data Import Procedures summarizes the built-in system procedures that you can use to import your data.

» Which Procedure Should I Use to Import My Data? presents a decision tree that makes it easy to decide which procedure to use for your circumstances.

» Import Procedures Syntax shows the syntax for our import procedures.

Data Import Procedures

Splice Machine includes four different procedures for importing data into your database, three of which use identical syntax; the fourth provides a more behind-the-scenes method that is quicker when loading large data sets, but requires more work and care on your part. The table below summarizes these import procedures:

System Procedure	Description
`SYSCS_UTIL.IMPORT_DATA`	Imports data into your database, creating a new record in your table for each record in the imported data. `SYSCS_UTIL.IMPORT_DATA` inserts the default value of each column that is not specified in the input.
`SYSCS_UTIL.UPSERT_DATA_FROM_FILE`	Imports data into your database, creating new records and *updating existing records* in the table. Identical to `SYSCS_UTIL.IMPORT_DATA` except that will update matching records. `SYSCS_UTIL.UPSERT_DATA_FROM_FILE` also inserts or updates the value in the table of each column that is not specified in the input; inserting the default value (or NULL if there is no default) for that column.
`SYSCS_UTIL.MERGE_DATA_FROM_FILE`	Imports data into your database, creating new records and *updating existing records* in the table. Identical to `SYSCS_UTIL.UPSERT_DATA_FROM_FILE` except that it does not replace values in the table for unspecified columns when updating an existing record in the table.
`SYSCS_UTIL.BULK_IMPORT_HFILE`	Takes advantage of HBase bulk loading to import table data into your database by temporarily converting the table file that you're importing into HFiles, importing those directly into your database, and then removing the temporary HFiles. This procedure uses syntax very similar to the other import procedures and has improved performance for large tables; however, the bulk HFile import requires extra work on your part and lacks constraint checking.

Which Procedure Should I Use to Import My Data?

The following diagram helps you decide which of our data importation procedures best fits your needs:
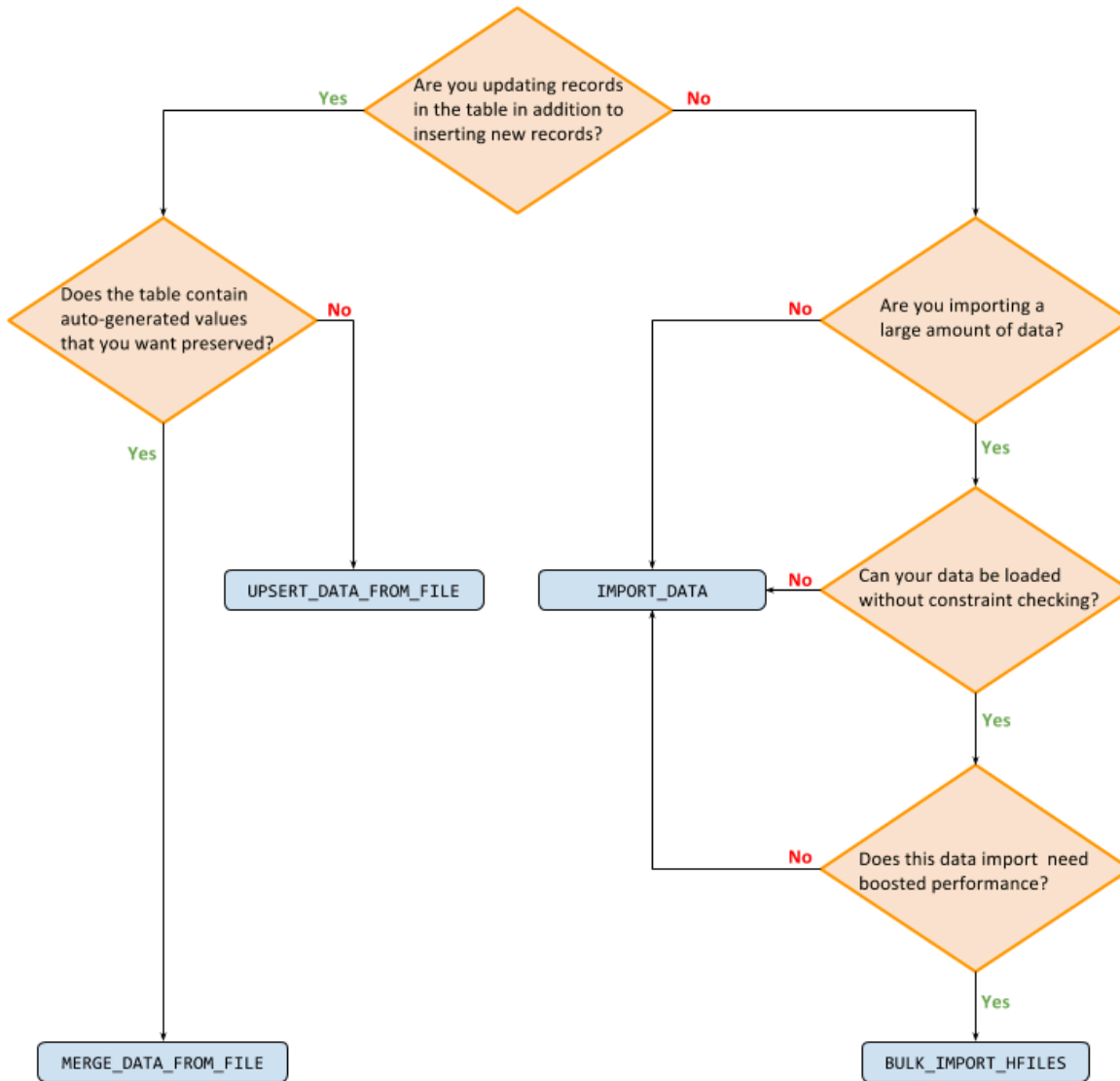
Notes

>> The `IMPORT_DATA` procedure imports new records into a database. The `UPSERT_DATA_FROM_FILE` and `MERGE_DATA_FROM_FILE` procedures import new records and update existing records. Importing all new records is faster because the database doesn't need to check if the record already exists in the database.

>> If your table contains auto-generated column values and you don't want those values overwritten when a record gets updated, use the `MERGE_DATA_FROM_FILE` procedure (`UPSERT_DATA_FROM_FILE` will overwrite).

>> The `BULK_IMPORT_HFILE` procedure is great when you're importing a very large dataset and need extra performance. However, it does not perform constraint checking.

Import Procedures Syntax

Here are the declarations of our four data import procedures; as you can see, three of our four import procedures use identical parameters, and the fourth (SYSCS_UTIL.BULK_IMPORT_HFILE) adds a couple extra parameters at the end:

```
call SYSCS_UTIL.IMPORT_DATA (
        schemaName,
        tableName,
        insertColumnList | null,
        fileOrDirectoryName,
        columnDelimiter | null,
        characterDelimiter | null,
        timestampFormat | null,
        dateFormat | null,
        timeFormat | null,
        badRecordsAllowed,
        badRecordDirectory | null,
        oneLineRecords | null,
        charset | null
        );
```

```
call SYSCS_UTIL.UPSERT_DATA_FROM_FILE (
        schemaName,
        tableName,
        insertColumnList | null,
        fileOrDirectoryName,
        columnDelimiter | null,
        characterDelimiter | null,
        timestampFormat | null,
        dateFormat | null,
        timeFormat | null,
        badRecordsAllowed,
        badRecordDirectory | null,
        oneLineRecords | null,
        charset | null
);
```

```
call SYSCS_UTIL.MERGE_DATA_FROM_FILE (
        schemaName,
        tableName,
        insertColumnList | null,
        fileOrDirectoryName,
        columnDelimiter | null,
        characterDelimiter | null,
        timestampFormat | null,
        dateFormat | null,
        timeFormat | null,
        badRecordsAllowed,
        badRecordDirectory | null,
        oneLineRecords | null,
        charset | null
);
```

```
call SYSCS_UTIL.BULK_IMPORT_HFILE (
    schemaName,
    tableName,
    insertColumnList | null,
    fileName,
    columnDelimiter | null,
    characterDelimiter | null,
    timestampFormat | null,
    dateFormat | null,
    timeFormat | null,
    maxBadRecords,
    badRecordDirectory | null,
    oneLineRecords | null,
    charset | null,
    bulkImportDirectory,
    skipSampling
);
```

You'll find descriptions and detailed reference information for all of these parameters in the Import Parameters topic of this tutorial.

And you'll find detailed reference descriptions of all four procedures in our SQL Reference Manual.

See Also

- ›› Importing Data: Input Parameters
- ›› Importing Data: Input Data Handling
- ›› Importing Data: Error Handling
- ›› Importing Data: Usage Examples
- ›› Importing Data: Bulk HFile Examples

» [Importing Data: Importing TPCH Data](#)

» [SYSCS_UTIL.IMPORT_DATA](#)

» [SYSCS_UTIL.UPSERT_DATA_FROM_FILE](#)

» [SYSCS_UTIL.MERGE_DATA_FROM_FILE](#)

» [SYSCS_UTIL.BULK_IMPORT_HFILE](#)

Streaming Data with Kafka: Creating a Producer

This topic demonstrates how to create a Kafka Producer to feed data into Splice Machine; we'll subsequently use this producer in other tutorials.

Watch the Video:

The following video shows you how to create a Kafka producer to feed data into Splice Machine.

Streaming Data with Kafka: Configuring a Feed

This topic demonstrates how to create a Kafka Feed , which puts messages on a Kafka Queue. We'll make use of this class in other tutorials.

Watch the Video:

The following video shows you how to configure a Kafka feed to Splice Machine.

Streaming MQTT Spark Data

This topic walks you through using MQTT Spark streaming with Splice Machine. MQTT is a lightweight, publish-subscribe messaging protocol designed for connecting remotely when a small footprint is required. MQTT is frequently used for data collection with the Internet of Things (IoT).

The example code in this tutorial uses Mosquitto, which is an open source message broker that implements the MQTT. This tutorial uses a cluster managed by MapR; if you're using different platform management software, you'll need to make a few adjustments in how the code is deployed on your cluster.

> **NOTE:** All of the code used in this tutorial is available in our GitHub community repository.

You can complete this tutorial by watching a short video or by following the written directions below.

Watch the Video

The following video shows you how to:

» put messages on an MQTT queue

» consume those messages using Spark streaming

» save those messages to Splice Machine with a virtual table (VTI)

Written Walk Through

This section walks you through the same sequence of steps as the video, in these sections:

» Deploying the Tutorial Code walks you through downloading and deploying the sample code.

» About the Sample Code describes the high-level methods in each class.

» About the Sample Code Scripts describes the scripts used to deploy and execute the sample code.

Deploying the Tutorial Code

Follow these steps to deploy the tutorial code:

> 1. **Download the code from our GitHub community repository.**
>   Pull the code from our git repository:

```
https://github.com/splicemachine/splice-community-sample-code/tree/maste
r/tutorial-mqtt-spark-streaming
```

2. **Compile and package the code:**

```
mvn clean compile package
```

3. **Copy three JAR files to each server:**
   Copy these three files:

```
./target/splice-tutorial-mqtt-2.0.jarspark-streaming-mqtt_2.10-1.6.1.jaro
rg.eclipse.paho.client.mqttv3-1.1.0.jar
```

   to this directory on each server:

```
/opt/splice/default/lib
```

4. **Restart Hbase**

5. **Create the target table in splice machine:**
   Run this script to create the table:

```
create-tables.sql
```

6. **Start Mosquitto:**

```
sudo su /usr/sbin/mosquitto -d -c /etc/mosquitto/mosquitto.conf > /var/lo
g/mosquitto.log 2>&1
```

7. **Start the Spark streaming script:**

```
sudo -su mapr ./run-mqtt-spark-streaming.sh tcp://srv61:1883 /testing 10
```

   The first parameter (`tcp://srv61:1883`) is the MQTT broker, the second (`/testing`) is the topic name, and the third (`10`) is the number of seconds each stream should run.

8. **Start putting messages on the queue:**
   Here's a java program that is set up to put messages on the queue:

```
java -cp /opt/splice/default/lib/splice-tutorial-mqtt-2.0-SNAPSHOT.jar:/o
pt/splice/default/lib/org.eclipse.paho.client.mqttv3-1.1.0.jar com.splice
machine.tutorials.sparkstreaming.mqtt.MQTTPublisher tcp://localhost:1883
/testing 1000 R1
```

The first parameter (`tcp://localhost:1883`) is the MQTT broker, the second (`/testing`) is the topic name, the third (`1000`) is the number of iterations to execute, and the fourth parameter (`R1`) is a prefix for this run.

> **NOTE:** The source code for this utility program is in a different GitHub project than the rest of this code. You'll find it in the `tutorial-kafka-producer` Github project.

About the Sample Code

This section describes the main class methods used in this MQTT example code; here's a summary of the classes:

Java Class	Description
`MQTTPublisher`	Puts csv messages on an MQTT queue.
`SparkStreamingMQTT`	The Spark streaming job that reads messages from the MQTT queue.
`SaveRDD`	Inserts the data into Splice Machine using the `RFIDMessageVTI` class.
`RFIDMessageVTI`	A virtual table interface for parsing an `RFIDMessage`.
`RFIDMessage`	Java object (a POJO) for converting from a csv string to an object to a database entry.

MQTTPublisher

This class puts CSV messages on an MQTT queue. The function of most interest in MQTTPublisher.java is `DoDemo`, which controls our sample program:

```
public void doDemo() {
    try {
        long startTime = System.currentTimeMillis();
        client = new MqttClient(broker, clientId);
        client.connect();
        MqttMessage message = new MqttMessage();
        for (int i=0; inumMessages; i++) {
            // Build a csv string
            message.setPayload( prefix + "Asset" + i ", Location" + i + "," + new Ti
mestamp((new Date()).getTime())).getBytes());
            client.publish(topicName, message);
            if (i % 1000 == 0) {
                System.out.println("records:" + i + " duration=" + (System.currentTi
meMillis() - startTime));
                startTime = System.currentTimeMillis();
            }
        client.disconnect();
    } catch (MqttException e) {
        e.printStackTrace();
    }
}
```

DoDemo does a little initialization, then starts putting messages out on the queue. Our sample program is set up to loop until it creates `numMessages` messages; after every 1000 messages, it displays a status message that helps us determine how much time is going to put messages on the queue, and how much to take them off the queue.

DoDemo builds a csv record (line) for each message, setting an asset ID, a location ID, and a timestamp in the `payload` of the message. It them publishes that message to the topic `topicName`.

SparkStreamingMQTT

Once the messages are on the queue, our `SparkStreamingMQTT` class object reads them from the queue and inserts them into our database. The main method in this class is `processMQTT`:

```
public void processMQTT(final String broker, final String topic, final int numSecond
s) {

    LOG.info("************ SparkStreamingMQTTOutside.processMQTT start");

    // Create the spark application and set the name to MQTT
    SparkConf sparkConf = new SparkConf().setAppName("MQTT");

    // Create the spark streaming context with a 'numSeconds' second batch size
    jssc = new JavaStreamingContext(sparkConf, Durations.seconds(numSeconds));
    jssc.checkpoint(checkpointDirectory);

    LOG.info("************ SparkStreamingMQTTOutside.processMQTT about to read the M
QTTUtils.createStream");
    //2. MQTTUtils to collect MQTT messages
    JavaReceiverInputDStreamString> messages = MQTTUtils.createStream(jssc, broker,
topic);

    LOG.info("************ SparkStreamingMQTTOutside.processMQTT about to do foreach
RDD");
    //process the messages on the queue and save them to the database
    messages.foreachRDD(new SaveRDD());

    LOG.info("************ SparkStreamingMQTTOutside.processMQTT prior to context.st
rt");
    // Start the context
    jssc.start();
    jssc.awaitTermination();
}
```

The `processMQTT` method takes three parameters:

broker
> The URL of the MQTT broker.

topic
> The MQTT topic name.

numSeconds
> The number of seconds at which streaming data will be divided into batches.

The `processMQTT` method processes the messages on the queue and saves them by calling the `SaveMDD` class.

SaveRDD

The `SaveRDD` class is an example of a Spark streaming function that uses our virtual table interface (VTI) to insert data into your Splice Machine database. This function checks for messages in the stream, and if there any, it creates a connection your database and uses a prepared statement to insert the messages into the database.

```
/**
 * This is an example of spark streaming function that
 * inserts data into Splice Machine using a VTI.
 *
 * @author Erin Driggers
 */

public class SaveRDD implements FunctionJavaRDDString>, Void>, Externalizable {

    private static final Logger LOG = Logger.getLogger(SaveRDD.class);

    @Override
    public Void call(JavaRDDString> rddRFIDMessages) throws Exception {
        LOG.debug("About to read results:");
        if (rddRFIDMessages != null '& rddRFIDMessages.count() > 0) {
            LOG.debug("Data to process:");
            //Convert to list
            ListString> rfidMessages = rddRFIDMessages.collect();
            int numRcds = rfidMessages.size();

            if (numRcds > 0) {
                try {
                    Connection con = DriverManager.getConnection("jdbc:splice://loca
lhost:1527/splicedb;user=splice;password=admin");

                    //Syntax for using a class instance in a VTI, this could also b
e a table function
                    String vtiStatement = "INSERT INTO IOT.RFID "
                            + "select s.* from new com.splicemachine.tutorials.spark
streaming.mqtt.RFIDMessageVTI(?) s ("
                            + RFIDMessage.getTableDefinition() + ")";
                    PreparedStatement ps = con.prepareStatement(vtiStatement);
                    ps.setObject(1, rfidMessages);
                    ps.execute();
                } catch (Exception e) {
                    //It is important to catch the exceptions as log messages becaus
e it is difficult
                    //to trace what is happening otherwise
                    LOG.error("Exception saving MQTT records to the database" + e.ge
tMessage(), e);
                } finally {
                    LOG.info("Complete insert into IOT.RFID");
                }
            }
        }
        return null;
    }
```

The heart of this function is the statement that creates the prepared statement, using a VTI class instance:

```
String vtiStatement = "INSERT INTO IOT.RFID "    + "select s.* from new     com.spl
icemachine.tutorials.sparkstreaming.mqtt.RFIDMessageVTI(?) s ("
    + RFIDMessage.getTableDefinition() + ")";
PreparedStatement ps = con.prepareStatement(vtiStatement);
```

Note that the statement references both our `RFIDMessage` and `RFIDMessageVTI` classes, which are described below.

RFIDMessageVTI

The `RFIDMessageVTI` class implements an example of a virtual table interface that reads in a list of strings that are in CSV format, converts that into an `RFIDMessage` object, and returns the resultant list in a format that is compatible with Splice Machine.

This class features an override of the `getDataSet` method, which loops through each CSV record from the input stream and converts it into an `RFIDMessage` object that is added onto a list of message items:

```
@Override
public DataSetLocatedRow> getDataSet(SpliceOperation op, DataSetProcessor dsp, ExecR
ow execRow) throws StandardException {
    operationContext = dsp.createOperationContext(op);

    //Create an arraylist to store the key / value pairs
    ArrayListLocatedRow> items = new ArrayListLocatedRow>();

    try {

        int numRcds = this.records == null ? 0 : this.records.size();

        if (numRcds > 0) {

            LOG.info("Records to process:" + numRcds);
            //Loop through each record convert to a SensorObject
            //and then set the values
            for (String csvString : records) {
                CsvBeanReader beanReader = new CsvBeanReader(new StringReader(csvStr
ing), CsvPreference.STANDARD_PREFERENCE);
                RFIDMessage msg = beanReader.read(RFIDMessage.class, header, process
ors);
                items.add(new LocatedRow(msg.getRow()));
            }
        }
    } catch (Exception e) {
        LOG.error("Exception processing RFIDMessageVTI", e);
    } finally {
        operationContext.popScope();
    }
    return new ControlDataSet>(items);
}
```

> For more information about using our virtual table interface, see Using the Splice Machine Virtual Table Interface.

RFIDMessage

The `RFIDMessage` class creates a simple Java object (a POJO) that represents an RFID message; we use this to convert an incoming CSV-formatted message into an object. This class includes getters and setters for each of the object properties, plus the `getTableDefinition` and `getRow` methods:

```
/**
 * Used by the VTI to build a Splice Machine compatible resultset
 *
 * @return
 * @throws SQLException
 * @throws StandardException
 */
public ValueRow getRow() throws SQLException, StandardException {
    ValueRow valueRow = new ValueRow(5);
    valueRow.setColumn(1, new SQLVarchar(this.getAssetNumber()));
    valueRow.setColumn(2, new SQLVarchar(this.getAssetDescription()));
    valueRow.setColumn(3, new SQLTimestamp(this.getRecordedTime()));
    valueRow.setColumn(4, new SQLVarchar(this.getAssetType()));
    valueRow.setColumn(5, new SQLVarchar(this.getAssetLocation()));
    return valueRow;
}

/**
 * Table definition to use when using a VTI that is an instance of a class
 *
 * @return
 */
public static String getTableDefinition() {
    return "ASSET_NUMBER varchar(50), "
    + "ASSET_DESCRIPTION varchar(100), "
    + "RECORDED_TIME TIMESTAMP, "
    + "ASSET_TYPE VARCHAR(50), "
    + "ASSET_LOCATION VARCHAR(50) ";
}
```

The `getTableDefinition` method is a string description of the table into which you're inserting records; this pretty much replicates the specification you would use in an SQL `CREATE_TABLE` statement.

The `getRow` method creates a data row with the appropriate number of columns, uses property getters to set the value of each column, and returns the row as a `resultset` that is compatible with Splice Machine.

About the Sample Code Scripts

These are also two scripts that we use with this tutorial:

Class	Description
`/ddl/create-tables.sql`	A simple SQL script that you can use to have Splice Machine create the table into which RFID messages are stored.
`/scripts/run-mqtt-spark-streaming.sh`	Starts the Spark streaming job.

Integrating Apache Storm with Splice Machine

This topic walks you through building and running two different examples of integrating Storm with Splice Machine:

» [Inserting Random Values in Splice Machine with Storm](#)

» [Inserting Data into Splice Machine from MySQL](#)

Inserting Random Values in Splice Machine with Storm

This example iterates through a list of random words and numbers, inserting the values into a Splice Machine database; follow along in these steps:

1. **Download the Sample Code:**
   Pull the code from our git repository:

   ```
   https://github.com/splicemachine/splice-community-sample-code/tree/master/tutorial-storm
   ```

2. **Check the prerequisites:**
   You must be running Splice Machine 2.0 or later on the same machine on which you will run this example code. If Splice Machine is running on a different machine, you'll need to modify the `server` variable in the `SpliceDumperTopology.java` file; change it to the name of the server that is running Splice Machine.

   You also must have `maven v.3.3.9` or later installed.

3. **Create the test table in Splice Machine:**
   This example inserts data into a Splice Machine table named `testTable`. You need to create that table by entering this command at the `splice>` prompt:

   ```
   splice> CREATE TABLE testTable( word VARCHAR(100), number INT );
   ```

4. **Compile the sample code:**
   Compile the sample code with this command

   ```
   % mvn clean compile dependency:copy-dependencies
   ```

5. **Run the sample code:**
   Follow these steps:

   a. **Make sure that Splice Machine is running and that you have created the `testTable` table.**

   b. Execute this script to run the program:

   ```
   % run-storm.sh
   ```

<blockquote>

c. Query `testTable` in Splice Machine to verify that it has been populated with random words and numbers:

```
splice> select * from testTable;
```

</blockquote>

About the Sample Code Classes

The random insertion example contains the following java classes, each of which is described below:

Class	Description
`SpliceCommunicator.java`	Contains methods for communicating with Splice Machine.
`SpliceConnector.java`	Establishes a JDBC connection with Splice Machine.
`SpliceDumperBolt.java`	Dumps data into Splice Machine.
`SpliceDumperTopology.java`	Defines the Storm topology for this example.
`SpliceIntegerSpout.java`	Emits tuples that are inserted into the Splice Machine table.

Inserting Data into Splice Machine from MySQL

This example uses Storm to read data from a MySQL database, and insert that data into a table in Splice Machine.

1. **Download the Sample Code:**
   Pull the code from our git repository:

   ```
   https://github.com/splicemachine/splice-community-sample-code/tree/master/tutorial-storm
   ```

2. **Check the prerequisites:**
   You must be running Splice Machine 2.0 or later on the same machine on which you will run this example code. If Splice Machine is running on a different machine, you'll need to modify the `server` variable in the `MySqlToSpliceTopology`.java file; change it to the name of the server that is running Splice Machine.

   You also must have `maven v.3.3.9` or later installed.

This example assumes that your MySQL database instance is running on the same machine on which you're running Splice Machine, and that the root user does not have a password. If either of these is not true, then you need to modify the call to the `seedBufferQueue` method in the `MySqlSpout.java` file. This method takes four parameters that you may need to change:

```
seedBufferQueue( MySqlServer, MySqlDatabase, mySqlUserName, mySqlPasswor
d );
```

The default settings used in this example are:

```
seedBufferQueue( "localhost", "test", "root", "" );
```

3. **Create the students table in Splice Machine:**
   This example inserts data into a Splice Machine table named `students`. You need to create that table by entering this command at the `splice>` prompt:

   ```
   splice> CREATE TABLE students( name VARCHAR(100) );
   ```

4. **Create the students table in your MySQL database:**
   This example read data from a MySQL table named `students`. You need to create that table in MySQL:

   ```
   $$ CREATE TABLE students( id INTEGER, name VARCHAR(100) );
   ```

   If your MySQL instance is on a different machine

5. **Compile the sample code:**
   Compile the sample code with this command

   ```
   % mvn clean compile dependency:copy-dependencies
   ```

6. **Run the sample code:**
   Follow these steps:

   a. **Make sure that Splice Machine is running and that you have created the `testTable` table.**

   b. Execute this script to run the program:

   ```
   % run-mysql-storm.sh
   ```

   c. Query the `students` table in Splice Machine to verify that it has been populated with data from the MySQL table:

   ```
   splice> select * from students;
   ```

About the Sample Code Classes

This example contains the following java classes:

Class	Description
`MySqlCommunicator.java`	Contains methods for communicating with MySQL.
`MySqlConnector.java`	Establishes a JDBC connection with MySQL.
`MySqlSpliceBolt.java`	Dumps data from MySQL into Splice Machine.
`MySqlSpout.java`	Emits tuples from MySQL that are inserted into the Splice Machine table.
`MySqlToSpliceTopology.java`	Defines the Storm topology for this example.
`SpliceCommunicator.java`	Contains methods for communicating with Splice Machine.
`SpliceConnector.java`	Establishes a JDBC connection with Splice Machine.

Analytics and Machine Learning With Splice Machine

This section provides tutorials to help you to use analytics and machine learning tools with Splice Machine:

>> [Connecting with Apache Zeppelin](#) shows you how to use Zeppelin with Splice Machine.

Connecting with Apache Zeppelin

> **This is an On-Premise-Only topic!**   [Learn about our products](#)

This tutorial walks you through connecting your on-premise Splice Machine database with Apache Zeppelin, which is a web-based notebook project currently in incubation at Apache. In this tutorial, you'll learn how to use SQL to query your Splice Machine database from Zeppelin.

> **NOTE:**
>
> Zeppelin is already integrated into the Splice Machine Database-as-Service product; please see our *Using Zeppelin* documentation for more information.
>
> See https://zeppelin.apache.org/ to learn more about Apache Zeppelin.

You can complete this tutorial by [watching a short video](#), or by [following the written directions](#) below.

Watch the Video

The following video shows you how to connect Splice Machine with Apache Zeppelin..

Written Walk Through

This section walks you through using SQL to query a Splice Machine database with Apache Zeppelin..

1. **Install Zeppelin:**
   If you're running on AWS, you can install the Zeppelin sandbox application; if you're using an on-premise database, we recommend following the [instructions in this video.](#)

2. **Create a new interpreter to run with Splice:**

   a. **Select the Interpreter tab in Zeppelin.**

   b. Click the **Create** button (in the upper right of the Zeppelin window) to create a new interpreter. Fill in the property fields as follows:

Name	Whatever name you like; we're using `SpliceMachine`
Interpreter	Select `jdbc` from the drop-down list of interpreter types.
default.url	`jdbc:splice:/`myServer`:1527/splicedb` (replace myServer with the name of the server that you're using)
default password	`admin`
default userId	`splice`
common.max_count	`1000`
default.driver	`com.splicemachine.db.jdbc.ClientDriver`
Artifacts	Insert the path to the Splice Machine jar file; for example: `/tmp/db-client-2.5.0.1708-SNAPSHOT.jar`

    c. Click the Save button to save your interpreter definition.

3. Create a note:
Select the **Notebook** tab in Zeppelin, and then click **+ Create new note**.

    a. Specify a name and click the Create Note button.

    b. Enable interpreters for the note. In this case, we move the Splice Machine interpreter to the top of the list, then click the Save button to make it the default interpreter:

c. Create a Zeppelin paragraph (a jdbc action) that calls a stored procedure. The procedure we're calling in this tutorial is named MOVIELENS; it is used to analyze data in a table. In this case, we're using this procedure to report statistics on the Age column in our movie watchers database. This Zeppelin paragraph looks like this:

```
%jdbccall MOVIELENS.ContinuousFeatureReport('movielens.user_demographic
s');
```

The %jdbc specifies that we're creating a paragraph that uses a JDBC interpreter; since we've made the SpliceMachine driver our default JDBC connector, it will be used.

d. The results of this call look like this:

e. We can also create a new paragraph that performs additional analysis; you'll see that whenever you run a paragraph in Zeppelin, it automatically leaves room at the bottom to create another paragraph.

```
%jdbcselect count(1) num_age, age from MOVIELENS.USER_DEMOGRAPHICS grou
p by age;
```

The results of this paragraph:

```
%jdbc
select count(1) num_age, age from MOVIELENS.USER_DEMOGRAPHICS group by age
```

NUM_AGE	AGE
39	30
23	44
21	40
6	52
3	65
17	34
9	57
3	61
5	13

Took 0 seconds. Last updated by anonymous at time Jul 14, 2016 7:13:56 PM.

4. **Change how you view your data**
   To get a better sense of what you can do with Zeppelin, we'll modify how we visualize this data:

   a. Click the rightmost settings icon, then click **settings**.

   b. **Move age to the xAxis, and the number of people of that age to the yAxis.**

   c. You'll now see the distribution of ages:
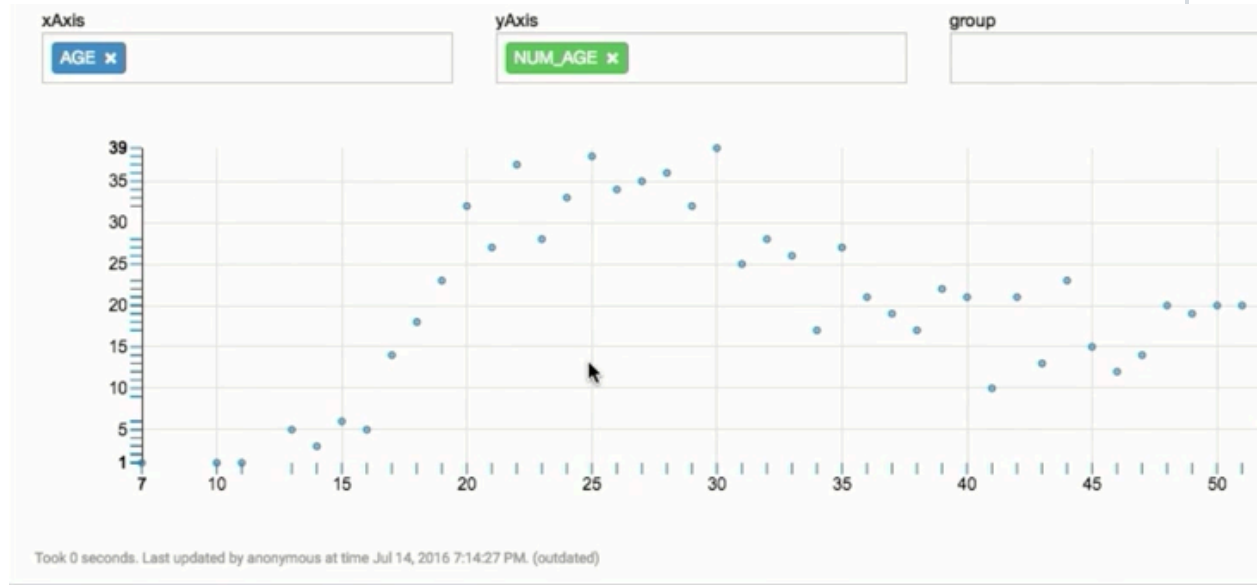
d. Click the graphs button to select other data visualizations:

Tuning and Debugging Query Performance

This section contains the following topics to help you optimize your Splice Machine database queries:

Topic	Describes
Optimizing Queries	Gets you started with optimizing your queries for Splice Machine.
Using Statistics	Using our statistics gathering facility to help optimize queries.
Explain Plan	The Explain Plan feature, which allows you to examine the execution plan for a query without executing the query.
Logging	The Splice Machine logging facility.
Debugging	Describes the parameter values to use when using debugger software with Splice Machine.

For access to the source code for the Community Edition of Splice Machine, visit our open source GitHub repository.

Optimizing Splice Machine Queries

This topic introduces you to Splice Machine query optimization techniques, including information about executing SQL expressions without a table context, and using optimization hints.

Introduction to Query Optimization

Here are a few mechanisms you can use to optimize your Splice Machine queries:

Optimization Mechanism	Description
Use Explain Plan	You can use the Splice Machine Explain Plan facility to display the execution plan for a statement without actually executing the statement. You can use Explain Plan to help determine options such as which join strategy to use or which index to select. See the About Explain Plan topic.
Use Statistics	Your database administrator can refine which statistics are collected on your database, which in turn enhances the operation of the query optimizer. See the Using Statistics topic.
Use `WHERE` clauses	Use a `WHERE` clause in your queries to restrict how much data is scanned.
Use indexes	You can speed up a query by having an index on the criteria used in the `WHERE` clause, or if the `WHERE` clause is using a primary key. Composite indexes work well for optimizing queries.
Use Splice Machine *query hints*	The Splice Machine query optimizer allows you to provide hints to help in the optimization process.

Using Select Without a Table

Sometimes you want to execute SQL scalar expressions without having a table context. For example, you might want to create a query that evaluates an expression and returns a table with a single row and one column. Or you might want to evaluate a list of comma-separated expressions and return a table with a single row and multiple columns one for each expression.

In Splice Machine, you can execute queries without a table by using the `sysibm.sysdummy1` dummy table. Here's the syntax:

```
select expression FROM sysibm.sysdummy1
```

And here's an example:

```
splice> select 1+ 1 from sysibm.sysdummy1;
```

Using Splice Machine Query Hints

You can use *hints* to help the Splice Machine query interface optimize your database queries.

> **NOTE:** The Splice Machine optimizer is constantly being improved, and new hint types sometimes get added. One recent addition is the ability to specify that a query should be run on (or not on) Spark, if possible.

Types of Hints

There are different kinds of hints you can supply, each of which is described in a section below; here's a summary:

Hint Type	Examples	Used to Specify
Index	`--splice-properties index=my_index`	Which index to use or not use
Join Order	`--splice-properties joinOrder=fixed`	Which join order to use for two tables
Join Strategy	`--splice-properties joinStrategy=sortmerge`	How a join is processed (in conjunction with the Join Order hint)
Pinned Table	`--splice-properties pin=true`	That you want the pinned (cached in memory) version of a table used in a query
Spark	`--splice-properties useSpark=true`	That you want a query to run (or not run) on Spark
Delete	`--splice-properties bulkDeleteDirectory='/path'`	That you are deleting a large amount of data and want to bypass the normal write pipeline to speed up the deletion.

Including Hints in Your Queries

Hints MUST ALWAYS be at the end of a line, meaning that you must always terminate hints with a newline character.

You cannot add the semicolon that terminates the command immediately after a hint; the semicolon must go on the next line, as shown in the examples in this topic.

> Many of the examples in this section show usage of hints on the `splice>` command line. Follow the same rules when using hints programmatically.

Hints can be used in two locations: after a table identifier or after a `FROM` clause. Some hint types can be use after a table identifier, and some can be used after a `FROM` clause:

Hint after a:	Hint types	Example

Table identifier	`index`
	
	`joinStrategy`
	
	`pin`
	
	`useSpark`
	
	`bulkDeleteDirectory`
		SELECT * FROM
		  member_info m, rewards r,
		  points p --SPLICE-PROPERTIES index=ie_point WHERE...

A `FROM` clause	`joinOrder`	SELECT * FROM --SPLICE-PROPERTIES joinOrder=fixed
		  mytable1 e, mytable2 t
		  WHERE e.id = t.parent_id;

This example shows proper placement of the hint and semicolon when the hint is at the end of statement:

```
SELECT * FROM my_table --splice-properties index=my_index;
```

If your command is broken into multiple lines, you still must add the hints at the end of the line, and you can add hints at the ends of multiple lines; for example:

```
SELECT * FROM my_table_1 --splice-properties index=my_index
, my_table_2 --splice-properties index=my_index_2
WHERE my_table_1.id = my_table_2.parent_id;
```

> **NOTE:** In the above query, the first command line ends with the first index hint, because hints must always be the last thing on a command line. That's why the comma separating the table specifications appears at the beginning of the next line.

Index Hints

Use *index hints* to tell the query interface how to use certain indexes for an operation.

To force the use of a particular index, you can specify the index name; for example:

```
splice> SELECT * FROM my_table --splice-properties index=my_index
> ;
```

To tell the query interface to not use an index for an operation, specify the null index. For example:

```
splice> SELECT * FROM my_table --splice-properties index=null
> ;
```

And to tell the query interface to use specific indexes on specific tables for an operation, you can add multiple hints. For example:

```
splice> SELECT * FROM my_table_1    --splice-properties index=my_index
> , my_table_2                --splice-properties index=my_index_2
> WHERE my_table_1.id = my_table_2.parent_id;
```

Important Note About Placement of Index Hints

Each `index` hint in a query **MUST** be specified alongside the table containing the index, or an error will occur.

For example, if we have a table named `points` with an index named `ie_point` and another table named `rewards` with an index named `ie_rewards`, then this hint works as expected:

```
SELECT * FROM   member_info m,
    rewards r,
    points p    --SPLICE-PROPERTIES index=ie_point
WHERE...
```

But the following hint will generate an error because `ie_rewards` is not an index on the points table.

```
SELECT * FROM
    member_info m,
    rewards r,
    points p    --SPLICE-PROPERTIES index=ie_rewards
WHERE...
```

JoinOrder Hints

Use `JoinOrder` hints to tell the query interface in which order to join two tables. You can specify these values for a `JoinOrder` hint:

» Use `joinOrder=FIXED` to tell the query optimizer to order the table join according to how where they are named in the `FROM` clause.

» Use `joinOrder=UNFIXED` to specify that the query optimizer can rearrange the table order.

> **NOTE:** `joinOrder=UNFIXED` is the default, which means that you don't need to specify this hint to allow the optimizer to rearrange the table order.

Here are examples:

Hint	Example
`joinOrder=FIXED`	`splice> SELECT * FROM --SPLICE-PROPERTIES joinOrder=fixed>` `mytable1 e, mytable2 t> WHERE e.id = t.parent_id;`
`joinOrder=UNFIXED`	`splice> SELECT * from --SPLICE-PROPERTIES joinOrder=unfixed>` `mytable1 e, mytable2 t WHERE e.id = t.parent_id;`

JoinStrategy Hints

You can use a `JoinStrategy` hint in conjunction with a `joinOrder` hint to tell the query interface how to process a join. For example, this query specifies that the `SORTMERGE` join strategy should be used:

```
SELECT * FROM      --SPLICE-PROPERTIES joinOrder=fixed
   mytable1 e, mytable2 t --SPLICE-PROPERTIES joinStrategy=SORTMERGE
   WHERE e.id = t.parent_id;
```

And this uses a `joinOrder` hint along with two `joinStrategy` hints:

```
SELECT *
  FROM --SPLICE-PROPERTIES joinOrder=fixed
  keyword k
  JOIN campaign c  --SPLICE-PROPERTIES joinStrategy=NESTEDLOOP
    ON k.campaignid = c.campaignid
  JOIN adgroup g  --SPLICE-PROPERTIES joinStrategy=NESTEDLOOP
    ON k.adgroupid = g.adgroupid
  WHERE adid LIKE '%us_gse%'
```

You can specify these join strategies:

JoinStrategy Value	Strategy Description
`BROADCAST`	Read the results of the Right Result Set (*RHS*) into memory, then for each row in the left result set (*LHS*), perform a local lookup to determine the right side of the join. `BROADCAST` will only work on equijoin (=) predicates that do not include a function call.

MERGE	Read the Right and Left result sets simultaneously in order and join them together as they are read.
	`MERGE` joins require that both the left and right result sets be sorted according to the join keys. `MERGE` requires an equijoin predicate that does not include a function call.
NESTEDLOOP	For each row on the left, fetch the values on the right that match the join.
	`NESTEDLOOP` is the only join that can work with any join predicate of any type; however this type of join is generally very slow.
SORTMERGE	Re-sort both the left and right sides according to the join keys, then perform a `MERGE` join on the results.
	`SORTMERGE` requires an equijoin predicate with no function calls.

Pinned Table Hint

You can use the `pin` hint to specify to specify that you want a query to run against a pinned version of a table.

```
splice> PIN TABLE myTable;splice> SELECT COUNT(*) FROM my_table --splice-properties
pin=true> ;
```

You can read more about pinning tables in the [PIN TABLE](#) statement topic.

Spark Hints

You can use the `useSpark` hint to specify to the optimizer that you want a query to run on (or not on) Spark. The Splice Machine query optimizer automatically determines whether to run a query through our Spark engine or our HBase engine, based on the type of query; you can override this by using a hint:

> **NOTE:** The Splice Machine optimizer uses its estimated cost for a query to decide whether to use spark. If your statistics are out of date, the optimizer may end up choosing the wrong engine for the query.

```
splice> SELECT COUNT(*) FROM my_table --splice-properties useSpark=true> ;
```

You can also specify that you want the query to run on HBase and not on Spark. For example:

```
splice> SELECT COUNT(*) FROM your_table --splice-properties useSpark=false> ;
```

Delete Hints

You can use the `bulkDeleteDirectory` hint to specify that you want to use our bulk delete feature to optimize the deletion of a large amount of data. Similar to our [bulk import feature](#), bulk delete generates HFiles, which allows us to bypass the Splice Machine write pipeline and HBase write path when performing the deletion. This can significantly speed up the deletion process.

You need to specify the directory to which you want the temporary HFiles written; you must have write permissions on this directory to use this feature. If you're specifying an S3 bucket on AWS, please review our [Configuring an S3 Bucket for Splice Machine Access](#) tutorial before proceeding.

```
splice> DELETE FROM my_table --splice-properties bulkDeleteDirectory='/bulkFilesPat
h'
;
```

> **NOTE:** We recommend performing a major compaction on your database after deleting a large amount of data; you should also be aware of our new [SYSCS_UTIL.SET_PURGE_DELETED_ROWS](#) system procedure, which you can call before a compaction to specify that you want the data physically (not just logically) deleted during compaction.

Using Statistics with Splice Machine

This topic introduces how to use database statistics with Splice Machine. Database statistics are a form of metadata (data about data) that assists the Splice Machine query optimizer; the statistics help the optimizer select the most efficient approach to running a query, based on information that has been gathered about the tables involved in the query.

This topic describes how to:

- [Collecting statistics](#) for a schema or table in your database

- [Dropping statistics](#) for a schema

- [Enable and disable collection of statistics](#) on specific columns in tables in your database

- [View collected statistics](#)

Statistics are inexact; in fact, some statistics like table cardinality are estimated using advanced algorithms, due to the resources required to compute these values. It's important to keep this in mind when basing design decisions on values in database statistics tables.

It's also important to note that the statistics for your database are not automatically refreshed when the data in your database changes, which means that when you query a statistical table or view, the results you see may not exactly match the data in the actual tables.

Collecting Statistics

You can collect statistics on a schema or table using the `splice> Analyze` command.

You can also use the `SYSCS_UTIL.COLLECT_SCHEMA_STATISTICS` procedure to collect statistics on an entire schema, including every table in the schema. For example:

```
splice> CALL SYSCS_UTIL.COLLECT_SCHEMA_STATISTICS( 'SPLICEBBALL', false );
```

During statistical collection:

- Statistics are automatically collected on columns in a primary key, and on columns that are indexed. These are called *keyed columns*.

- Statistics are also collected on columns for which you have enabled statistics collection, as described in the next section, [Enabling and Disabling Statistics.](#)

Once collection of statistics has completed, the Splice Machine query optimizer will automatically begin using the updated statistics to optimize query execution plans.

When Should You Collect Statistics?

We advise that you collect statistics after you have:

- Created an index on a table.

- Modified a significant number of rows in a table with update, insert, or delete operations.

> **NOTE:** A general rule-of-thumb is that you should collect statistics after modifying more than 10% of data.

On Which Columns Should You Collect Statistics?

By default, Splice Machine collects statistics on all columns in a table.

To reduce the operational cost of analyzing large tables (such as fact tables), you can tell Splice Machine to not collect statistics on certain columns by running the SYSCS_UTIL.DISABLE_COLUMN_STATISTICS built-in system procedure:

```
SYSCS_UTIL.DISABLE_COLUMN_STATISTICS( schema, table, column);
```

Splice Machine strongly recommends that you *always* collect statistics on small tables, such as a table that has hundreds of rows on each region server.

How to Determine if You Should Collect or Drop Statistics

You can use Explain Plan in your development or test environment to determine how dropping or collecting statistics changes the execution plan for a query.

Dropping Statistics

If you subsequently wish to drop statistics for a schema, you can use the SYSCS_UTIL.DROP_SCHEMA_STATISTICS procedure to drop statistics for an entire schema. For example:

```
splice> CALL SYSCS_UTIL.DROP_SCHEMA_STATISTICS('SPLICEBBALL');
```

Enabling and Disabling Statistics on Specific Columns

When you collect statistics, Splice Machine automatically collects statistics on keyed columns, which are columns in a primary key and columns that are indexed.

Please review the recommendations and restrictions regarding which columns should or should not have statistics collection enabled, as noted below in the Selecting Columns for Statistics Collection subsection.

You can also explicitly enable statistics collection on specific columns in tables using the SYSCS_UTIL.ENABLE_COLUMN_STATISTICS procedure. For example:

```
CALL SYSCS_UTIL.ENABLE_COLUMN_STATISTICS('SPLICEBBALL', 'Players', 'Birthdate');
```

Disabling Statistics Collection on a Column

If you subsequently wish to disable collection of statistics on a specific column in a table, use the SYSCS_UTIL.DISABLE_COLUMN_STATISTICS procedure:

```
splice> CALL SYSCS_UTIL.DISABLE_COLUMN_STATISTICS('SPLICEBBALL', 'Players', 'Birthda
te');
```

Once you've enabled or disabled statistics collection for one or more table columns, you should update the query optimizer by
collecting statistics on the table or schema.

Selecting Columns for Statistics Collection

You can only collect statistics on columns containing data that can be ordered. This includes all numeric types, Boolean values,
some `CHAR` and `BIT` data types, and date and timestamp values.

When selecting columns on which statistics should be collected, keep these ideas in mind:

» The process of collecting statistics requires both memory and compute time to complete; the more statistics you collect,
the longer it takes and the more of your computing resources that it uses.

» You *should collect statistics* for any column that is used as a predicate in a query.

» You *should collect statistics* for any column that is used in a `select distinct`, `Group by`, `order by`, or `join`
clause.

» You *do not need to enable statistics* for columns that are merely carried through the computation; however, doing so may
improve heap size estimations, which in turn can make broadcast joins more likely to be chosen.

As you can see, selecting columns for statistics is a tradeoff between the resources required to collect the statistics, and the
improvements in optimization that result from having the statistics collected.

Viewing Collected Statistics

Splice Machine provides two system tables you can query to view the statistics that have been collected for your database:

» Query the `SYSTABLESTATISTICS System Table` to view statistics for specific tables

» Query the `SYSCOLUMNSTATISTICS System Table` to view statistics for specific table columns

See Also

» Splice Machine Data Assignments and Comparisons

» `SYSCS_UTIL.COLLECT_SCHEMA_STATISTICS`

» `SYSCS_UTIL.DISABLE_COLUMN_STATISTICS`

» `SYSCS_UTIL.DROP_SCHEMA_STATISTICS`

» `SYSCS_UTIL.ENABLE_COLUMN_STATISTICS`

» `SYSCOLUMNSTATISTICS System Table`

» `SYSTABLESTATISTICS System Table`

About Explain Plan

You can use the `explain` command to display what the execution plan will be for a statement without executing the statement. This topic presents and describes several examples.

Using Explain Plan to See the Execution Plan for a Statement

To display the execution plan for a statement without actually executing the statement, use the `explain` command on the statement:

```
splice> explain Statement;
```

Statement
      An SQL statement.

Explain Plan and DDL Statements

SQL Data Definition Language (DDL) statements have no known cost, and thus do not require optimization. Because of this, the `explain` command does not work with DDL statements; attempting to `explain` a DDL statement such as `CREATE TABLE` will generate a syntax error. You **cannot** use `explain` with any of the following SQL statements:

- » `ALTER`
- » `CREATE` … (any statement that starts with `CREATE`)
- » `DROP` … (any statement that starts with `DROP`)
- » `GRANT`
- » `RENAME` … (any statement that starts with `RENAME`)
- » `REVOKE`
- » `TRUNCATE TABLE`

Explain Plan Output

When you run the `explain` command, it displays output in a tree-structured format:

- » The first row in the output summarizes the plan
- » Each row in the output represents a node in the tree.
- » For join nodes, the right side of the join is displayed first, followed by the left side.
- » Child node rows are indented and prefixed with '`->`'.

The first node in the plan output contains these fields:

Field	Description
n=*number*	The number of nodes.
rows=*number*	The number of output rows.
updateMode=[mode]	The update mode of the statement.
engine=[engineType]	engineType=Spark means this query will be executed by our OLAP engine. engineType=control means this query will be executed by our OLTP engine.

Each node row in the output contains the following fields:

Field	Description
[*node label*]	The name of the node
n=*number*	The result set number. This is primarily used internally, and can also be used to determine the relative ordering of optimization.
totalCost=*number*	The total cost to perform this operation. This is computed *as if the operation is at the top of the operation tree*.
processingCost=*number*	The cost to process all data in this node. Processing includes everything except for reading the final results over the network.
transferCost=*number*	The cost to send the final results over the network to the control node.
outputRows=*number*	The total number of rows that are output.

Field	Description
`outputHeapSize=number unit`	The total size of the output result set, and the unit in which that size is expressed, which is one of: » B » KB » MB » GB » TB
`partitions==number`	The number of partitions involved. *Partition* is currently equivalent to *Region*; however, this will not necessarily remain true in future releases.

For example:

```
splice> explain select * from sys.systables t, sys.sysschemas s
        where t.schemaid =s.schemaid;

Plan
-----------------------------------------------------------------------------
-------------Cursor(n=5,rows=20,updateMode=READ_ONLY (1),engine=control)
  -> ScrollInsensitive(n=4,totalCost=21.728,outputRows=20,outputHeapSize=6.641 KB,p
artitions=1)
    -> BroadcastJoin(n=3,totalCost=12.648,outputRows=20,outputHeapSize=6.641 KB,par
titions=1,preds=[(T.SCHEMAID[4:4] = S.SCHEMAID[4:8])])
      -> TableScan[SYSSCHEMAS(32)](n=2,totalCost=4.054,outputRows=20,outputHeapSiz
e=6.641 KB,partitions=1)
      -> TableScan[SYSTABLES(48)](n=1,totalCost=4.054,outputRows=20,outputHeapSiz
e=3.32 KB,partitions=1)

5 rows selected
```

The next topic, Explain Plan Examples, contains a number of examples, annotated with notes to help you understand the output of each.

See Also

» Explain Plan Examples

Explain Plan Examples

This topic contains the following examples of using the `explain` command to display the execution plan for a statement, including the following examples:

- » [TableScan Examples](#)
- » [IndexScan Examples](#)
- » [Projection and Restriction Examples](#)
- » [Index Lookup](#)
- » [Join Example](#)
- » [Union Example](#)
- » [Order By Example](#)
- » [Aggregation Operation Examples](#)
- » [Subquery Example](#)

> The format and meaning of the common fields in the output plans shown here is found in the previous topic, [About Explain Plan](#).

TableScan Examples

This example show a plan for a `TableScan` operation that has no qualifiers, known as a *raw scan*:

```
splice> explain select * from sys.systables;Plan
--------------------------------------------------------------------------------
-------------
Cursor(n=3,rows=20,updateMode=READ_ONLY (1),engine=control)
  -> ScrollInsensitive(n=2,totalCost=8.594,outputRows=20,outputHeapSize=3.32 KB,par
titions=1)
    -> TableScan[SYSTABLES(48)](n=1,totalCost=4.054,outputRows=20,outputHeapSiz
e=3.32 KB,partitions=1)

3 rows selected
```

This example show a plan for a `TableScan` operation that does have qualifiers::

```
splice> explain select * from sys.systables --SPLICE-PROPERTIES index=NULL    where t
ablename='SYSTABLES';Plan
--------------------------------------------------------------------------------
-------------Cursor(n=3,rows=18,updateMode=READ_ONLY (1),engine=control)
  -> ScrollInsensitive(n=2,totalCost=8.54,outputRows=18,outputHeapSize=2.988 KB,par
titions=1)
    -> TableScan[SYSTABLES(48)](n=1,totalCost=4.054,outputRows=18,outputHeapSiz
e=2.988 KB,partitions=1,preds=[(TABLENAME[0:2] = SYSTABLES)])

3 rows selected
```

Nodes

» The plan labels this operation as `TableScan[`*tableId*(*conglomerateId*)`]`:

> » *tableId* is the name of the table, in the form `schemaName '.' tableName`.

> » *conglomerateId* is an ID that is unique to every HBase table; this value is used internally, and can be used for certain administrative tasks

» The `preds` field includes qualifiers that were pushed down to the base table.

IndexScan Examples

This example show a plan for an `IndexScan` operation that has no predicates:

```
splice> explain select tablename from sys.systables; --covering index

Plan
--------------------------------------------------------------------------------
-------------Cursor(n=3,rows=20,updateMode=READ_ONLY (1),engine=control)
  -> ScrollInsensitive(n=2,totalCost=8.31,outputRows=20,outputHeapSize=560 B,partit
ions=1)
    -> IndexScan[SYSTABLES_INDEX1(145)](n=1,totalCost=4.054,outputRows=20,outputHea
pSize=560 B,partitions=1,baseTable=SYSTABLES(32))

3 rows selected
```

This example shows a plan for an `IndexScan` operation that contains predicates:

```
splice> explain select tablename from sys.systables --SPLICE-PROPERTIES index=SYSTAB
LES_INDEX1
        where tablename = 'SYSTABLES';

Plan
--------------------------------------------------------------------------------
-------------Cursor(n=3,rows=18,updateMode=READ_ONLY (1),engine=control)
  -> ScrollInsensitive(n=2,totalCost=8.272,outputRows=18,outputHeapSize=432 B,parti
tions=1)
    -> IndexScan[SYSTABLES_INDEX1(145)](n=1,totalCost=4.049,outputRows=18,outputHea
pSize=432 B,partitions=1,baseTable=SYSTABLES(48),preds=[(TABLENAME[0:1] = SYSTABLE
S)])

3 rows selected
```

Nodes

» The plan labels this operation as `IndexScan[`*indexId(conglomerateId)*`]`:

  » *indexId* is the name of the index

  » *conglomerateId* is an ID that is unique to every HBase table; this value is used internally, and can be used for certain administrative tasks

» The `preds` field includes qualifiers that were pushed down to the base table.

Projection and Restriction Examples

This example show a plan for a `Projection` operation:

```
splice> explain select tablename || 'hello' from sys.systables;

Plan
--------------------------------------------------------------------------------
-------------Cursor(n=4,rows=20,updateMode=READ_ONLY (1),engine=control)
  -> ScrollInsensitive(n=3,totalCost=8.302,outputRows=20,outputHeapSize=480 B,parti
tions=1)
    -> ProjectRestrict(n=2,totalCost=4.054,outputRows=20,outputHeapSize=480 B,parti
tions=1)
      -> IndexScan[SYSTABLES_INDEX1(145)](n=1,totalCost=4.054,outputRows=20,outputH
eapSize=480 B,partitions=1,baseTable=SYSTABLES(48))

4 rows selected
```

This example shows a plan for a `Restriction` operation:

```
splice> explain select tablename from sys.systables
        where tablename like '%SYS%';

Plan
--------------------------------------------------------------------------------
------------Cursor(n=4,rows=10,updateMode=READ_ONLY (1),engine=control)
  -> ScrollInsensitive(n=3,totalCost=8.178,outputRows=10,outputHeapSize=240 B,parti
tions=1)
    -> ProjectRestrict(n=2,totalCost=4.054,outputRows=10,outputHeapSize=240 B,parti
tions=1,preds=[like(TABLENAME[0:1], %SYS%)])
      -> IndexScan[SYSTABLES_INDEX1(145)](n=1,totalCost=4.054,outputRows=20,outputH
eapSize=240 B,partitions=1,baseTable=SYSTABLES(48))


4 rows selected
```

Nodes

» The plan labels both projection and restriction operations as `ProjectRestrict`. which can contain both *projections* and *non-qualifier restrictions*. A *non-qualifier restriction* is a predicate that cannot be pushed to the underlying table scan.

Index Lookup

This example shows a plan for an `IndexLookup` operation:

```
splice> explain select * from SYS.SYSTABLES --SPLICE-PROPERTIES index=SYSTABLES_INDE
X1
where tablename = 'SYSTABLES';;

Plan
--------------------------------------------------------------------------------
------------Cursor(n=4,rows=18,updateMode=READ_ONLY (1),engine=control)
  -> ScrollInsensitive(n=3,totalCost=177.265,outputRows=18,outputHeapSize=921.586 K
B,partitions=1)
    -> IndexLookup(n=2,totalCost=78.715,outputRows=18,outputHeapSize=921.586 KB,par
titions=1)
      -> IndexScan[SYSTABLES_INDEX1(145)](n=1,totalCost=6.715,outputRows=18,outputH
eapSize=921.586 KB,partitions=1,baseTable=SYSTABLES(48),preds=[(TABLENAME[1:2] = SYS
TABLES)])
```

Nodes

» The plan labels the operation as `IndexLookup`; you may see this labeled as an `IndexToBaseRow` operation elsewhere.

» Plans for `IndexLookup` operations do not contain any special fields.

Join Example

This example shows a plan for a `Join` operation:

```
splice> explain select * from sys.systables t, sys.sysschemas s
        where t.schemaid =s.schemaid;

Plan
-------------------------------------------------------------------------------
-------------Cursor(n=5,rows=20,updateMode=READ_ONLY (1),engine=control)
  -> ScrollInsensitive(n=4,totalCost=21.728,outputRows=20,outputHeapSize=6.641 KB,p
artitions=1)
    -> BroadcastJoin(n=3,totalCost=12.648,outputRows=20,outputHeapSize=6.641 KB,par
titions=1,preds=[(T.SCHEMAID[4:4] = S.SCHEMAID[4:8])])
      -> TableScan[SYSSCHEMAS(32)](n=2,totalCost=4.054,outputRows=20,outputHeapSiz
e=6.641 KB,partitions=1)
      -> TableScan[SYSTABLES(48)](n=1,totalCost=4.054,outputRows=20,outputHeapSiz
e=3.32 KB,partitions=1)

5 rows selected
```

Nodes

» The plan labels the operation using the *join type* followed by `Join`; the possible values are:

  » `BroadcastJoin`

  » `MergeJoin`

  » `MergeSortJoin`

  » `NestedLoopJoin`

  » `OuterJoin`

» The plan may include a `preds` field, which lists the join predicates.

» `NestedLoopJoin` operations do not include a `preds` field; instead, the predicates are listed in either a `ProjectRestrict` or in the underlying scan.

» The right side of the *Join* operation is listed first, followed by the left side of the join.

Outer Joins

An *outer join* does not display it as a separate strategy in the plan; instead, it is treated a *postfix* for the strategy that's used. For example, if you are using a Broadcast join, and it's a left outer join, then you'll see `BroadcastLeftOuterJoin`. Here's an example:

```
explain select s.schemaname,t.tablename from sys.sysschemas s left outer join sys.sy
stables t
> on s.schemaid = t.schemaid;
Plan
-------------------------------------------------------------------------------
-------------
Cursor(n=6,rows=20,updateMode=READ_ONLY (1),engine=control)
  -> ScrollInsensitive(n=5,totalCost=348.691,outputRows=20,outputHeapSize=2 MB,part
itions=1)
    -> ProjectRestrict(n=4,totalCost=130.579,outputRows=20,outputHeapSize=2 MB,part
itions=1)
      -> BroadcastLeftOuterJoin(n=3,totalCost=130.579,outputRows=20,outputHeapSiz
e=2 MB,partitions=1,preds=[(S.SCHEMAID[4:1] = T.SCHEMAID[4:4])])
        -> IndexScan[SYSTABLES_INDEX1(145)](n=2,totalCost=7.017,outputRows=20,outpu
tHeapSize=2 MB,partitions=1,baseTable=SYSTABLES(48))
        -> TableScan[SYSSCHEMAS(32)](n=1,totalCost=7.516,outputRows=20,outputHeapSi
ze=1023.984 KB,partitions=1)
```

Union Example

This example shows a plan for a `Union` operation:

```
splice> explain select tablename from sys.systables t union all select schemaname fr
om sys.sysschemas;

Plan
-------------------------------------------------------------------------------
-------------
Cursor(n=5,rows=40,updateMode=READ_ONLY (1),engine=control)
  -> ScrollInsensitive(n=4,totalCost=16.668,outputRows=40,outputHeapSize=1.094 KB,p
artitions=1)
    -> Union(n=3,totalCost=12.356,outputRows=40,outputHeapSize=1.094 KB,partition
s=1)
      -> IndexScan[SYSSCHEMAS_INDEX1(209)](n=2,totalCost=4.054,outputRows=20,output
HeapSize=1.094 KB,partitions=1,baseTable=SYSSCHEMAS(32))
      -> IndexScan[SYSTABLES_INDEX1(145)](n=1,totalCost=4.054,outputRows=20,outputH
eapSize=480 B,partitions=1,baseTable=SYSTABLES(48))

5 rows selected
```

Nodes

» The right side of the `Union` is listed first, followed by the left side of the union,

Order By Example

This example shows a plan for an order by operation:

```
splice> explain select tablename from sys.systables order by tablename desc;

Plan
--------------------------------------------------------------------------------
-------------Cursor(n=4,rows=20,updateMode=READ_ONLY (1),engine=control)
  -> ScrollInsensitive(n=3,totalCost=16.604,outputRows=20,outputHeapSize=480 B,part
itions=1)
    -> OrderBy(n=2,totalCost=12.356,outputRows=20,outputHeapSize=480 B,partition
s=1)
      -> IndexScan[SYSTABLES_INDEX1(145)](n=1,totalCost=4.054,outputRows=20,outputH
eapSize=480 B,partitions=1,baseTable=SYSTABLES(48))

4 rows selected
```

Nodes

» The plan labels this operation as `OrderBy`.

Aggregation Operation Examples

This example show a plan for a grouped aggregate operation:

```
splice> explain select tablename, count(*) from sys.systables group by tablename;

Plan
--------------------------------------------------------------------------------
-------------Cursor(n=6,rows=20,updateMode=READ_ONLY (1),engine=control)
  -> ScrollInsensitive(n=5,totalCost=12.568,outputRows=20,outputHeapSize=480 B,part
itions=16)
    -> ProjectRestrict(n=4,totalCost=8.32,outputRows=20,outputHeapSize=480 B,partit
ions=16)
      -> GroupBy(n=3,totalCost=4.054,outputRows=20,outputHeapSize=480 B,partition
s=1)
        -> ProjectRestrict(n=2,totalCost=4.054,outputRows=20,outputHeapSize=480 B,p
artitions=1)
          -> IndexScan[SYSTABLES_INDEX1(145)](n=1,totalCost=4.054,outputRows=20,out
putHeapSize=480 B,partitions=1,baseTable=SYSTABLES(48))

6 rows selected)
```

This example shows a plan for a scalar aggregate operation:

```
splice> explain select count(*) from sys.systables;

Plan
--------------------------------------------------------------------------------
-------------Cursor(n=6,rows=1,updateMode=READ_ONLY (1),engine=control)
  -> ScrollInsensitive(n=5,totalCost=8.797,outputRows=1,outputHeapSize=0 B,partitio
ns=1)
    -> ProjectRestrict(n=4,totalCost=4.257,outputRows=1,outputHeapSize=0 B,partitio
ns=1)
      -> GroupBy(n=3,totalCost=4.054,outputRows=20,outputHeapSize=3.32 KB,partition
s=1)
        -> ProjectRestrict(n=2,totalCost=4.054,outputRows=20,outputHeapSize=3.32 K
B,partitions=1)
          -> IndexScan[SYSTABLES_INDEX1(145)](n=1,totalCost=4.054,outputRows=20,out
putHeapSize=3.32 KB,partitions=1,baseTable=SYSTABLES(48))

6 rows selected
```

Nodes

» The plan labels both grouped and scaled aggregate operations as `GroupBy`.

Subquery Example

This example shows a plan for a `SubQuery` operation:

```
splice> explain select tablename, (select tablename from sys.systables t2 where t2.t
ablename = t.tablename)from sys.systables t;

Plan
--------------------------------------------------------------------------------
-------------Cursor(n=6,rows=20,updateMode=READ_ONLY (1),engine=control)
  -> ScrollInsensitive(n=5,totalCost=8.302,outputRows=20,outputHeapSize=480 B,parti
tions=1)
    -> ProjectRestrict(n=4,totalCost=4.054,outputRows=20,outputHeapSize=480 B,parti
tions=1)
      -> Subquery(n=3,totalCost=12.55,outputRows=20,outputHeapSize=480 B,partition
s=1,correlated=true,expression=true,invariant=true)
        -> IndexScan[SYSTABLES_INDEX1(145)](n=2,totalCost=4.054,outputRows=20,outpu
tHeapSize=480 B,partitions=1,baseTable=SYSTABLES(48),preds=[(T2.TABLENAME[0:1] = T.T
ABLENAME[4:1])])
      -> IndexScan[SYSTABLES_INDEX1(145)](n=1,totalCost=4.054,outputRows=20,outputH
eapSize=480 B,partitions=1,baseTable=SYSTABLES(48))

6 rows selected
```

Nodes

»   Subqueries are listed as a second query tree, whose starting indentation level is the same as the `ProjectRestrict` operation that *owns* the subquery.

»   Includes a *correlated* field, which specifies whether or not the query is treated as correlated or uncorrelated.

»   Includes an *expression* field, which specifies whether or not the subquery is an expression.

»   Includes an *invariant* field, which indicates whether the subquery is invariant.

See Also

»   [About Explain Plan](#)

Splice Machine Logging

This topic describes the logging facility used in Splice Machine. Splice Machine also allows you to exercise direct control over logging in your database. You can:

- [Manually disable logging](#) of all SQL statements.

- Configure individual logger objects to log [specific message levels.](#)

- Call the `log` function for a specific logger object, passing it a logging level value and a message. If the logger is currently configured to record messages at the specified level, the message is added to the log; otherwise, the logger ignores the message. Logger levels are ordered hierarchically; any message with a level equal to or greater than the hierarchical level for which logging is enabled is recorded into the log.

Splice Machine uses the open source [Apache log4j Logging API](#), which allows you to associate a logger object with any java class, among other features. Loggers can be set to capture different levels of information.

> Splice Machine enables statement logging by default.

Logging too much information can slow your system down, but not logging enough information makes it difficult to debug certain issues. The [Splice Machine Loggers](#) section below summarizes the loggers used by Splice Machine and the system components that they use. You can use the SQL logging functions described in the [SQL Logger Functions](#) section below to retrieve or modify the logging level of any logger used in Splice Machine.

Manually Disabling Logging

Logging of SQL statements is automatically enabled in Splice Machine; to disable logging of statements, you can do so in either of these ways:

- You can pass an argument to your Splice Machine JVM startup script, as follows:

    ```
    -Dderby.language.logStatementText=false
    ```

- You can add the following property definition to your `hbase-default.xml`, `hbase-site.xml`, or `splice-site.xml` file:

    ```
    <property>
    <name>splice.debug.logStatementContext</name>
    <value>false</value>
    <description>Property to enable logging of all statements.</description>
    </property>
    ```

You can examine the logged data in your region server's logs; if you want to change the location where events are logged, see the instructions in the Installation Guide for your platform (Cloudera, Hortonworks, or MapR).

Logger Levels

The `log4j` API defines six logger levels. The following table displays them from the lowest level to the highest:

Logger Level	What gets logged for a logger object set to this level
TRACE	Captures all messages.
DEBUG	Captures any message whose level is `DEBUG`, `INFO`, `WARN`, `ERROR`, or `FATAL`.
INFO	Captures any message whose level is `INFO`, `WARN`, `ERROR`, or `FATAL`.
WARN	Captures any message whose level is `WARN`, `ERROR`, or `FATAL`.
ERROR	Captures any message whose level is `ERROR` or `FATAL`.
FATAL	Captures only messages whose level is `FATAL`.

Splice Machine Loggers

The following table summarizes the loggers used in the Splice Machine environment that might interest you if you're trying to debug performance issues in your database:

Logger Name	Default Level	Description
`org.apache`	ERROR	Logs all Apache software messages
`com.splicemachine.db`	WARN	Logs all Derby software messages
`com.splicemachine.db.shared.common.sanity`	ERROR	Logs all Derby Sanity Manager messages
`com.splicemachine.derby.impl.sql.catalog`	WARN	Logs Derby SQL catalog/dictionary messages
`com.splicemachine.db.impl.sql.execute.operations`	WARN	Logs Derby SQL operation messages
`org.apache.zookeeper.server.ZooKeeperServer`	INFO	Used to determine when Zookeeper is started
`org.apache.zookeeper.server.persistence.FileTxnSnapLog`	INFO	Logs Zookeeper transactions

Logger Name	Default Level	Description
`com.splicemachine`	`WARN`	By default, controls all Splice Machine logging
`com.splicemachine.derby.hbase.SpliceDriver`	`INFO`	Prints start-up and shutdown messages to the log and to the console
`com.splicemachine.derby.management.StatementManager`	`ERROR`	Set the level of this logger to `TRACE` to record execution time for SQL statements

SQL Logger Functions

Splice Machine SQL includes the following built-in system procedures, all documented in our *SQL Reference Manual*, for interacting with the Splice Machine logs:

System Procedure	Description
`SYSCS_UTIL.SYSCS_GET_LOGGERS`	Displays a list of the active loggers.
`SYSCS_UTIL.SYSCS_GET_LOGGER_LEVEL`	Displays the current logging level of the specified logger.
`SYSCS_UTIL.SYSCS_SET_LOGGER_LEVEL`	Sets the current logging level of the specified logger.

See Also

» `SYSCS_UTIL.SYSCS_GET_LOGGERS`

» `SYSCS_UTIL.SYSCS_GET_LOGGER_LEVEL`

» `SYSCS_UTIL.SYSCS_SET_LOGGER_LEVEL`

Debugging Splice Machine

This topic describes the parameter values you need to know for debugging Splice Machine with a software tool:

» Create a configuration in your software to remotely attach to Splice Machine

» Connect to port `4000`.

> **NOTE:** If you're debugging code that is to be run in a Spark worker, connect to port `4020` instead.

Example

Here's an example of an *IntelliJ IDEA* debugging configuration using port `4000`:
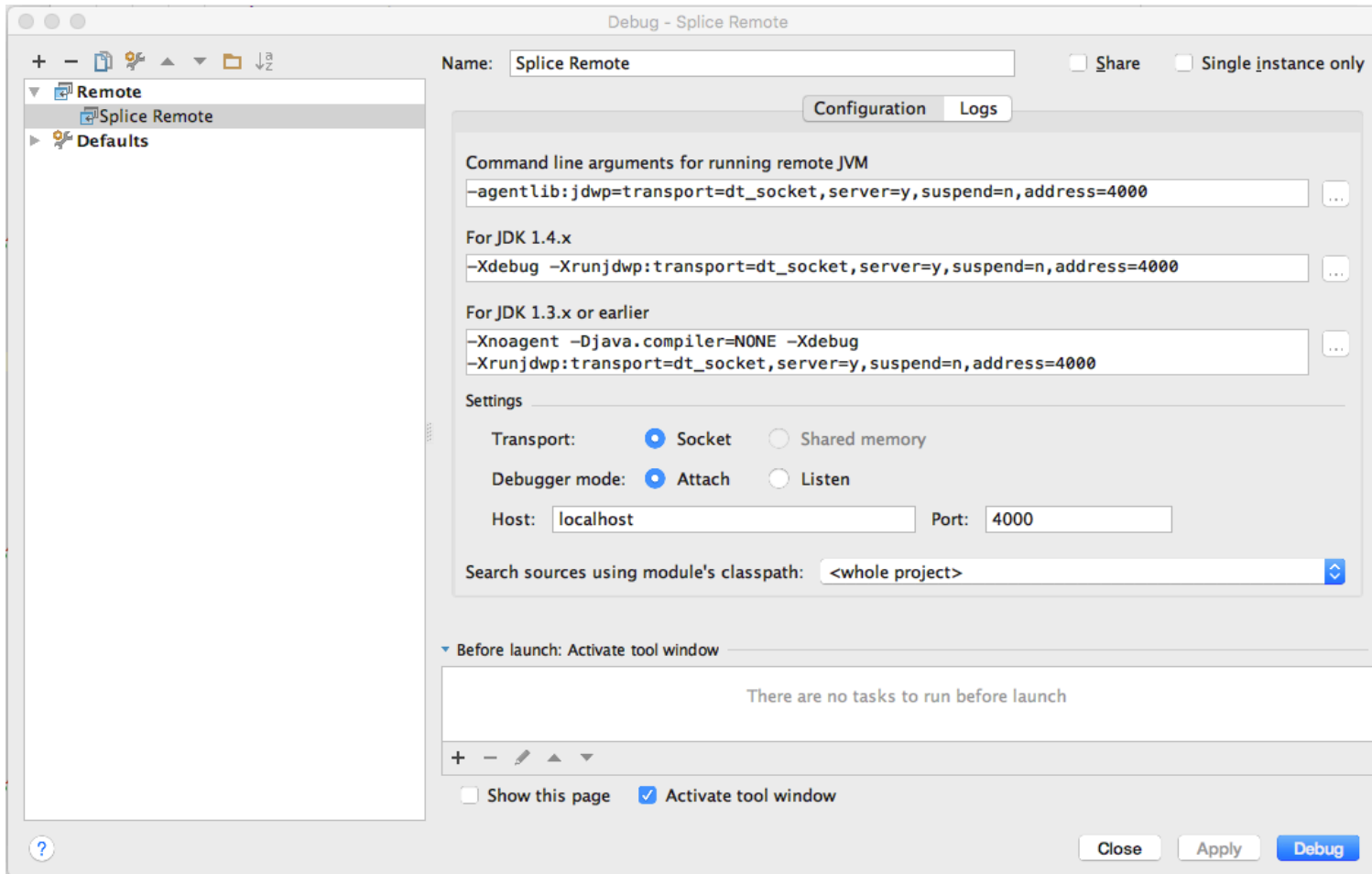
For access to the source code for the Community Edition of Splice Machine, visit our open source GitHub repository.

Using Splice Machine Snapshots

This topic describes how to use the Splice Machine snapshot feature to create a restorable snapshot of a table or schema; this is commonly used when importing or deleting a significant amount of data from a database.

Overview

Snapshots allow you to create point-in-time backups of tables (or an entire schema) without actually cloning the data.

> **NOTE:** Snapshots include both the data and indexes for tables.

You use these system procedures and tables to work with snapshots:

- ›› Use the `SYSCS_UTIL.SNAPSHOT_TABLE` system procedure to create a named snapshot for a table.

- ›› Use the `SYSCS_UTIL.SNAPSHOT_SCHEMA` system procedure to create a named snapshot for a schema.

- ›› Use the `SYSCS_UTIL.RESTORE_SNAPSHOT` system procedure to restore a table or schema from a named snapshot.

- ›› Use the `SYSCS_UTIL.DELETE_SNAPSHOT` system procedure to delete a named snapshot.

- ›› Information about stored snapshots, including their names, is found in the `SYS.SYSSNAPSHOTS` system table.