



---

# SQL Reference Manual

---

*Last generated: March 01, 2018*

© 2018 Splice Machine, Inc. All rights reserved. No part of this publication may be reproduced, distributed, or transmitted in any form or by any means, including photocopying, recording, or other electronic or mechanical methods, without the prior written permission of the publisher, except in the case of brief quotations embodied in critical reviews and certain other noncommercial uses permitted by copyright law.

# Table of Contents

## SQL Reference Manual

### Splice Machine SQL Manual

Introduction .....	1
SQL Summary.....	2
SQL Limitations.....	4
SQL Reserved Words .....	9
SQL Argument Matching.....	19

### Identifiers

Introduction .....	22
Identifier Types.....	24
Identifier Syntax .....	27

### Data Types

Introduction .....	29
BIGINT .....	33
BLOB.....	34
BOOLEAN.....	38
CHAR.....	39
CLOB .....	41
DATE.....	43
DECIMAL .....	45
DOUBLE .....	47
DOUBLE PRECISION.....	49
FLOAT .....	51
INTEGER .....	52
LONG VARCHAR.....	54
NUMERIC .....	55
REAL.....	57
SMALLINT.....	59
TEXT .....	60
TIME.....	62
TIMESTAMP .....	64
VARCHAR.....	67

### Statements

Introduction .....	68
ALTER TABLE .....	72
CALL (Procedure) .....	79
column-definition .....	80
CREATE EXTERNAL TABLE .....	82
CREATE FUNCTION .....	86
CREATE INDEX .....	91
CREATE PROCEDURE .....	95
CREATE ROLE .....	98
CREATE SCHEMA .....	100
CREATE SEQUENCE .....	102
CREATE SYNONYM .....	105
CREATE TABLE .....	107
CREATE TEMPORARY TABLE .....	112
CREATE TRIGGER .....	115
CREATE VIEW .....	119
DECLARE GLOBAL TEMPORARY TABLE .....	122
DELETE .....	124
Dependencies .....	126
DROP FUNCTION .....	128
DROP INDEX .....	129
DROP PROCEDURE .....	130
DROP ROLE .....	132
DROP SCHEMA .....	133
DROP SEQUENCE .....	134
DROP SYNONYM .....	135
DROP TABLE .....	136
DROP TRIGGER .....	137
DROP VIEW .....	138
generated-column-spec .....	139
generation-clause .....	145
GRANT .....	146
INSERT .....	153
PIN TABLE .....	156
RENAME COLUMN .....	159
RENAME INDEX .....	161
RENAME TABLE .....	162
REVOKE .....	163
SELECT .....	171

SET ROLE .....	174
SET SCHEMA.....	176
TRUNCATE TABLE.....	178
UNPIN TABLE.....	179
UPDATE TABLE.....	181

## Clauses

Introduction .....	183
CONSTRAINT.....	185
EXCEPT.....	193
FROM.....	196
GROUP BY .....	198
HAVING.....	200
LIMIT n.....	202
ORDER BY .....	205
OVER .....	208
RESULT OFFSET and FETCH FIRST.....	212
TOP n.....	214
UNION.....	218
USING.....	221
WHERE.....	223
WITH.....	225

## Expressions

Introduction .....	227
About Expressions .....	228
Boolean Expressions .....	232
CASE Expression .....	237
Dynamic Parameters.....	238
Expression Precedence .....	241
NEXT VALUE FOR Expression.....	242
SELECT Expression .....	244
TABLE Expression .....	249
VALUES Expression .....	251

## Join Operations

Introduction .....	254
About Join Operations .....	255
CROSS JOIN .....	257
INNER JOIN.....	259
LEFT OUTER JOIN.....	263

NATURAL JOIN .....	265
RIGHT OUTER JOIN .....	267

## Queries

Introduction .....	269
Query .....	270
Scalar Subquery .....	272
Table Subquery .....	273

## SQL Built-in Functions

Introduction .....	275
ABS .....	282
ACOS .....	283
ADD_MONTHS .....	285
ASIN .....	287
ATAN .....	289
ATAN2 .....	291
AVG .....	293
BIGINT .....	297
CAST .....	299
CEIL .....	304
CHAR .....	306
COALESCE .....	309
Concatenation Operator .....	311
COS .....	313
COSH .....	315
COT .....	317
COUNT .....	319
CURRENT_SCHEMA .....	322
CURRENT_DATE .....	323
CURRENT_ROLE .....	325
CURRENT_TIME .....	326
CURRENT_TIMESTAMP .....	327
CURRENT_USER .....	328
DATE .....	329
DAY .....	332
DEGREES .....	335
DENSE_RANK .....	337
DOUBLE .....	343
EXP .....	345

EXTRACT .....	346
FIRST_VALUE .....	351
FLOOR.....	353
HOUR.....	354
INITCAP .....	355
INSTR .....	357
INTEGER .....	359
LAG .....	361
LAST_DAY .....	364
LAST_VALUE.....	367
LCASE .....	369
LEAD.....	371
LENGTH.....	375
LN (or LOG) .....	378
LOCATE .....	379
LOG10.....	381
LTRIM.....	382
MAX .....	384
MIN.....	387
MINUTE .....	390
MOD .....	391
MONTH .....	395
MONTHNAME.....	397
MONTH_BETWEEN .....	400
NEXT_DAY .....	402
NOW .....	405
NULLIF.....	407
NVL .....	411
PI.....	413
QUARTER.....	414
RADIANS .....	417
RAND .....	419
RANDOM .....	420
RANK.....	421
REGEXP_LIKE .....	427
REPLACE .....	429
ROWID.....	431
ROW_NUMBER.....	433
RTRIM.....	435

SECOND .....	437
SESSION_USER .....	438
SIGN .....	439
SIN .....	440
SINH.....	442
SMALLINT.....	444
SQRT .....	446
STDDEV_POP .....	450
STDDEV_SAMP .....	452
SUBSTR.....	454
SUM .....	456
TAN .....	459
TANH.....	461
TIME.....	463
TIMESTAMP .....	465
TIMESTAMPADD .....	468
TIMESTAMPDIFF .....	470
TO_CHAR.....	472
TO_DATE.....	474
TRIM .....	482
TRUNCATE.....	485
UCASE or UPPER .....	491
USER .....	493
VARCHAR.....	494
WEEK.....	496
YEAR .....	498

## System Procedures

Introduction .....	500
BACKUP_DATABASE.....	507
BULK_IMPORT_HFILE.....	510
CANCEL_BACKUP .....	515
CANCEL_DAILY_BACKUP.....	516
COLLECT_SCHEMA_STATISTICS .....	518
COMPACT_REGION .....	521
COMPUTE_SPLIT_KEY .....	524
CREATE_USER.....	528
DELETE_BACKUP .....	531
DELETE_OLD_BACKUPS .....	533
DELETE_REGION .....	535



DELETE_SNAPSHOT .....	539
DISABLE_COLUMN_STATISTICS .....	540
DROP_SCHEMA_STATISTICS .....	542
DROP_USER .....	544
EMPTY_GLOBAL_STATEMENT_CACHE .....	546
EMPTY_STATEMENT_CACHE .....	548
ENABLE_COLUMN_STATISTICS .....	550
ENABLE_ENTERPRISE .....	552
GET_ACTIVE_SERVERS .....	553
GET_ALL_PROPERTIES .....	554
GET_CURRENT_TRANSACTION .....	558
GET_GLOBAL_DATABASE_PROPERTY Function .....	559
GET_ENCODED_REGION_NAME .....	561
GET_LOGGERS .....	564
GET_LOGGER_LEVEL .....	570
GET_REGIONS .....	572
GET_REGIONS_SERVER_STATS_INFO .....	575
GET_REQUESTS .....	576
GET_RUNNING_OPERATIONS .....	577
GET_SCHEMA_INFO .....	579
GET_SESSION_INFO .....	581
GET_START_KEY .....	583
GET_VERSION_INFO .....	585
GET_WRITE_INTAKE_INFO .....	586
IMPORT_DATA .....	588
INSTALL_JAR .....	595
INVALIDATE_STORED_STATEMENTS .....	597
KILL_OPERATION .....	598
MAJOR_COMPACT_REGION .....	599
MERGE_DATA_FROM_FILE .....	602
MERGE_REGIONS .....	609
MODIFY_PASSWORD .....	611
PEEK_AT_SEQUENCE Function .....	613
PERFORM_MAJOR_COMPACTION_ON_SCHEMA .....	615
PERFORM_MAJOR_COMPACTION_ON_TABLE .....	616
REFRESH_EXTERNAL_TABLE .....	617
REMOVE_JAR .....	618
REPLACE_JAR .....	620
RESET_PASSWORD .....	622

RESTORE_DATABASE .....	624
RESTORE_SNAPSHOT .....	627
SCHEDULE_DAILY_BACKUP .....	628
SET_GLOBAL_DATABASE_PROPERTY .....	630
SET_LOGGER_LEVEL .....	632
SET_PURGE_DELETED_ROWS .....	634
SNAPSHOT_SCHEMA .....	635
SNAPSHOT_TABLE .....	636
SPLIT_TABLE_OR_INDEX .....	637
SPLIT_TABLE_OR_INDEX_AT_POINTS .....	641
UPDATE_ALL_SYSTEM_PROCEDURES .....	643
UPDATE_METADATA_STORED_STATEMENTS .....	645
UPDATE_SCHEMA_OWNER .....	646
UPDATE_SYSTEM_PROCEDURE .....	647
UPSERT_DATA_FROM_FILE .....	649
VACUUM .....	656

## System Tables

Introduction .....	657
SYSALIASES system table .....	660
SYSBACKUP system table .....	661
SYSBACKUPITEMS system table .....	662
SYSBACKUPJOBS system table .....	663
SYSCHECKS system table .....	664
SYSCOLPERMS system table .....	665
SYSCOLUMNS system table .....	667
SYSCOLUMNSTATISTICS system table .....	669
SYSCONGLOMERATES system table .....	671
SYSCONSTRAINTS system table .....	673
SYSDEPENDS system table .....	674
SYSFILES system table .....	675
SYSFOREIGNKEYS system table .....	676
SYSKEYS system table .....	677
SYSPERMS system table .....	678
SYSROLES system table .....	679
SYSROUTINEPERMS system table .....	681
SYSSCHEMAS system table .....	682
SYSSEQUENCES system table .....	683
SYSSNAPSHOTS system table .....	685
SYSSTATEMENTS system table .....	686

SYSTABLEPERMS system table .....	687
SYSTABLES system table .....	689
SYSTABLESTATISTICS system table .....	690
SYSTRIGGERS system table .....	692
SYSUSERS system table .....	694
SYSVIEWS system table .....	695

## Error Codes

Introduction .....	696
Class 01 - Warning.....	699
Class 07 - Dynamic SQL Error.....	701
Class 08 - Connection Exception .....	702
Class 0A - Feature not supported .....	705
Class 0P - Invalid role specification .....	706
Class 21 - Cardinality Violations .....	707
Class 22 - Data Exception .....	708
Class 23 - Constraint Violation.....	711
Class 24 - Invalid Cursor State .....	712
Class 25 - Invalid Transaction State .....	713
Class 28 - Invalid Authorization Specification .....	714
Class 2D - Invalid Transaction Termination.....	715
Class 38 - External Function Exception .....	716
Class 39 - External Routine Invocation Exception .....	717
Class 3B - Invalid SAVEPOINT .....	718
Class 40 - Transaction Rollback .....	719
Class 42 - Syntax Error or Access Rule Violation.....	720
Class 57 - DRDA Network Protocol Execution Failure .....	737
Class 58 - DRDA Network Protocol Protocol Error .....	738
Class XBCA - CacheService.....	740
Class XBCM - ClassManager .....	741
Class XBCX - Cryptography .....	742
Class XBM - Monitor .....	744
Class XCL - Execution exceptions .....	746
Class XCW - Upgrade unsupported.....	749
Class XCX - Internal Utility Errors .....	750
Class XCY - Splice Property Exceptions .....	751
Class X CZ - com.splicemachine.db.database.UserUtility .....	752
Class XD00 - Dependency Manager .....	753
Class XIE - Import/Export Exceptions.....	754
Class XJ - Connectivity Errors .....	756

Class XK - Security Exceptions .....	762
Class XN - Network Client Exceptions .....	763
Class XRE - Replication Exceptions .....	764
Class XSAI - Store access.protocol.interface .....	766
Class XSAM - Store AccessManager .....	767
Class XSAS - Store Sort .....	768
Class XSAX - Store access.protocol.XA statement .....	769
Class XSCB - Store BTree .....	770
Class XSCG0 - Conglomerate .....	771
Class XSCH - Heap .....	772
Class XSDA - RawStore Data.Generic statement .....	773
Class XSDB - RawStore Data.Generic transaction .....	775
Class XSDF - RawStore Data.Filesystem statement .....	776
Class XSDG - RawStore Data.Filesystem database .....	777
Class XSLA - RawStore Log.Generic database exceptions .....	778
Class XSLB - RawStore Log.Generic statement exceptions .....	780
Class XSRS - RawStore protocol.Interface statement .....	781
Class XSTA2 - XACT_TRANSACTION_ACTIVE .....	782
Class XSTB - RawStore Transactions.Basic system .....	783
Class XXXXX - No SQLSTATE .....	784
Class X0 - Execution exceptions .....	785



# SQL Reference Manual

This section contains reference information for Splice Machine SQL. Our implementation includes all of ANSI SQL-99 (SQL3), with added optimizations and features.

Note that this section is modeled on and borrows heavily from the SQL Reference section of the Apache Derby 10.9 documentation, as permitted by the Apache License. This SQL Reference Manual contains the following sections:

Section	Description
<a href="#">Identifiers</a>	Describes the different identifiers used in SQL.
<a href="#">Data Types</a>	Describes the data types used in SQL.
<a href="#">Statements</a>	Reference pages for our implementation of each SQL statement.
<a href="#">Clauses</a>	Reference pages for our implementation of SQL clauses.
<a href="#">Expressions</a>	Describes the expressions you can use in SQL.
<a href="#">Join Operations</a>	Reference pages for our implementation of SQL join operations.
<a href="#">Queries</a>	Reference pages for our implementation of SQL queries.
<a href="#">SQL Built-in Functions</a>	Reference pages for the standard SQL functions featured in our implementation.
<a href="#">Built-in System Procedures and Functions</a>	Reference pages for Splice Machine system procedures and functions.
<a href="#">System Tables</a>	Descriptions of the system tables.
<a href="#">Argument Matching</a>	Describes how Splice Machine matches Java data types and methods with arguments supplied in SQL statements.
<a href="#">SQL Limitations</a>	A summary of various size limitations in Splice Machine.
<a href="#">Reserved Words</a>	A list of the reserved words in Splice Machine.

For a summary of all Splice Machine documentation, see the [Documentation Summary](#) topic.

## Acknowledgment

Since the Apache Derby documentation served as a starting point for this documentation, **Splice Machine would like to acknowledge the contribution of the Apache Derby community** to the Splice Machine product and documentation.

# Splice Machine SQL Summary

This topic summarizes the SQL-99+ features in Splice Machine SQL and some of the [SQL optimizations](#) that our database engine performs.

## SQL Feature Summary

This table summarizes some of the ANSI SQL-99+ features available in Splice Machine:

Feature	Examples
<i>Aggregation functions</i>	AVG, COUNT, MAX, MIN, STDDEV_POP, STDDEV_SAMP, SUM
<i>Conditional functions</i>	CASE, searched CASE
<i>Data Types</i>	INTEGER, REAL, CHARACTER, DATE, BOOLEAN, BIGINT
<i>DDL</i>	CREATE TABLE, CREATE SCHEMA, CREATE INDEX, ALTER TABLE, DELETE, UPDATE
<i>DML</i>	INSERT, DELETE, UPDATE, SELECT
<i>Isolation Levels</i>	Snapshot isolation
<i>Joins</i>	INNER JOIN, LEFT OUTER JOIN, RIGHT OUTER JOIN
<i>Predicates</i>	IN, BETWEEN, LIKE, EXISTS
<i>Privileges</i>	Privileges for SELECT, DELETE, INSERT, EXECUTE
<i>Query Specification</i>	SELECT DISTINCT, GROUP BY, HAVING
<i>SET functions</i>	UNION, ABS, MOD, ALL, CHECK
<i>String functions</i>	CHAR, Concatenation (  ), INSTR, LCASE (LOWER), LENGTH, LTRIM, REGEXP_LIKE, REPLACE, RTRIM, SUBSTR, UCASE (UPPER), VARCHAR
<i>Sub-queries</i>	Yes
<i>Transactions</i>	COMMIT, ROLLBACK

Feature	Examples
<i>Triggers</i>	Yes
<i>User-defined functions (UDFs)</i>	Yes
<i>Views</i>	Including grouped views
<i>Window functions</i>	AVG, COUNT, DENSE_RANK, FIRST_VALUE, LAG, LAST_VALUE, LEAD, MAX, MIN, RANK, ROW_NUMBER, STDDEV_POP, STDDEV_SAMP, SUM

## SQL Optimizations

Splice Machine performs a number of SQL optimizations that enhance the processing speed of your queries:

- » typed columns
- » sparse columns
- » flexible schema
- » secondary indices
- » real-time asynchronous statistics
- » cost-based optimizer



# SQL Limitations

This topic specifies limitations for various values in Splice Machine SQL:

- » [Database Value Limitations](#)
- » [Date, Time, and TimeStamp Limitations](#)
- » [Identifier Length Limitations](#)
- » [Numeric Limitations](#)
- » [String Limitations](#)
- » [XML Limitations](#)

## Database Value Limitations

The following table lists limitations on various database values in Splice Machine.

Value	Limit
Maximum columns in a table	100000
Maximum columns in a view	5000
Maximum number of parameters in a stored procedure	90
Maximum indexes on a table	32767 or storage capacity
Maximum tables referenced in an SQL statement or a view	Storage capacity
Maximum elements in a select list	1012
Maximum predicates in a WHERE or HAVING clause	Storage capacity
Maximum number of columns in a GROUP BY clause	32677
Maximum number of columns in an ORDER BY clause	1012
Maximum number of prepared statements	Storage capacity
Maximum declared cursors in a program	Storage capacity
Maximum number of cursors opened at one time	Storage capacity
Maximum number of constraints on a table	Storage capacity

Value	Limit
Maximum level of subquery nesting	Storage capacity
Maximum number of subqueries in a single statement	Storage capacity
Maximum number of rows changed in a unit of work	Storage capacity
Maximum constants in a statement	Storage capacity
Maximum depth of cascaded triggers	16

## Date, Time, and TimeStamp Limitations

The following table lists limitations on date, time, and timestamp values in Splice Machine.

Value	Limit
Smallest <code>DATE</code> value	0001-01-01
Largest <code>DATE</code> value	9999-12-31
Smallest <code>TIME</code> value	00:00:00
Largest <code>TIME</code> value	24:00:00
Smallest <code>TIMESTAMP</code> value	1677-09-21-00.12.44.000000
Largest <code>TIMESTAMP</code> value	2262-04-11-23.47.16.999999

## Identifier Length Limitations

The following table lists limitations on identifier lengths in Splice Machine.

Identifier	Maximum Number of Characters Allowed
Constraint name	128
Correlation name	128
Cursor name	128

Identifier	Maximum Number of Characters Allowed
Data source column name	128
Data source index name	128
Data source name	128
Savepoint name	128
Schema name	128
Unqualified column name	128
Unqualified function name	128
Unqualified index name	128
Unqualified procedure name	128
Parameter name	128
Unqualified trigger name	128
Unqualified table name, view name, stored procedure name	128

## Numeric Limitations

The following lists limitations on the numeric values in Splice Machine.

Value	Limit
Smallest INTEGER	-2,147,483,648
Largest INTEGER	2,147,483,647
Smallest BIGINT	-9,223,372,036,854,775,808
Largest BIGINT	9,223,372,036,854,775,807
Smallest SMALLINT	-32,768
Largest SMALLINT	32,767
Largest decimal precision	31

Value	Limit
Smallest DOUBLE	-1.79769E+308
Largest DOUBLE	1.79769E+308
Smallest positive DOUBLE	2.225E-307
Largest negative DOUBLE	-2.225E-307
Smallest REAL	-3.402E+38
Largest REAL	3.402E+38
Smallest positive REAL	1.175E-37
Largest negative REAL	-1.175E-37

## String Limitations

The following table lists limitations on string values in Splice Machine.

Value	Maximum Limit
Length of CHAR	254 characters
Length of VARCHAR	32,672 characters
Length of LONG VARCHAR	32,670 characters
Length of CLOB*	2,147,483,647 characters
Length of BLOB*	2,147,483,647 characters
Length of character constant	32,672
Length of concatenated character string	2,147,483,647
Length of concatenated binary string	2,147,483,647
Number of hex constant digits	16,336
Length of DOUBLE value constant	30 characters

Value	Maximum Limit
* If you're using our 32-bit ODBC driver, CLOB and BLOB objects are limited to 512 MB in size, instead of 2 GB , due to address space limitations.	

## Reserved Words

This section lists all of the Splice Machine reserved words, including those in the SQL standard. Splice Machine will return an error if you use any of these keywords as an identifier name unless you surround the identifier name with quotes ("). See [SQL Identifier Syntax](#).

Reserved Word
ADD
ALL
ALLOCATE
ALTER
AND
ANY
ARE
AS
ASC
ASSERTION
AT
AUTHORIZATION
AVG
BEGIN
BETWEEN
BIGINT
BIT
BOOLEAN
BOTH
BY

Reserved Word
CALL
CASCADE
CASCADEED
CASE
CAST
CHAR
CHARACTER
CHECK
CLOSE
COALESCE
COLLATE
COLLATION
COLUMN
COMMIT
CONNECT
CONNECTION
CONSTRAINT
CONSTRAINTS
CONTINUE
CONVERT
CORRESPONDING
CREATE
CROSS
CURRENT

Reserved Word
CURRENT_DATE
CURRENT_ROLE
CURRENT_TIME
CURRENT_TIMESTAMP
CURRENT_USER
CURSOR
DEALLOCATE
DEC
DECIMAL
DECLARE
DEFAULT
DEFERRABLE
DEFERRED
DELETE
DESC
DESCRIBE
DIAGNOSTICS
DISCONNECT
DISTINCT
DOUBLE
DROP
ELSE
END
END-EXEC



Reserved Word
ESCAPE
EXCEPT
EXCEPTION
EXEC
EXECUTE
EXISTS
EXPLAIN
EXTERNAL
FALSE
FETCH
FIRST
FLOAT
FOR
FOREIGN
FOUND
FROM
FULL
FUNCTION
GET
GETCURRENTCONNECTION
GLOBAL
GO
GOTO
GRANT

Reserved Word
GROUP
HAVING
HOUR
IDENTITY
IMMEDIATE
IN
INDICATOR
INITIALLY
INNER
INOUT
INPUT
INSENSITIVE
INSERT
INT
INTEGER
INTERSECT
INTO
IS
ISOLATION
JOIN
KEY
LAST
LEFT
LIKE

Reserved Word
LOWER
LTRIM
MATCH
MAX
MIN
MINUTE
NATIONAL
NATURAL
NCHAR
NVARCHAR
NEXT
NO
NONE
NOT
NULL
NULLIF
NUMERIC
OF
ON
ONLY
OPEN
OPTION
OR
ORDER

Reserved Word
OUTER
OUTPUT
OVER
OVERLAPS
PAD
PARTIAL
PREPARE
PRESERVE
PRIMARY
PRIOR
PRIVILEGES
PROCEDURE
PUBLIC
READ
REAL
REFERENCES
REGEXP_LIKE
RELATIVE
RESTRICT
REVOKE
RIGHT
ROLLBACK
ROWS
ROW_NUMBER

Reserved Word
RTRIM
SCHEMA
SCROLL
SECOND
SELECT
SESSION_USER
SET
SMALLINT
SOME
SPACE
SQL
SQLCODE
SQLERROR
SQLSTATE
SUBSTR
SUBSTRING
SUM
TABLE
TEMPORARY
TEXT
TIMEZONE_HOUR
TIMEZONE_MINUTE
TO
TRANSACTION

Reserved Word
TRANSLATE
TRANSLATION
TRIM
TRUE
UNION
UNIQUE
UNKNOWN
UPDATE
UPPER
USER
USING
VALUES
VARCHAR
VARYING
VIEW
WHenever
WHERE
WITH
WORK
WRITE
XML
XMlexists
XMLPARSE
XMLQUERY

Reserved Word
XMLSERIALIZE
YEAR

## Argument Matching in Splice Machine

When you declare a function or procedure using `CREATE FUNCTION/PROCEDURE`, Splice Machine does not verify whether a matching Java method exists. Instead, Splice Machine looks for a matching method only when you invoke the function or procedure in a later SQL statement.

At that time, Splice Machine searches for a public, static method having the class and method name declared in the `EXTERNAL NAME` clause of the earlier `CREATE FUNCTION/PROCEDURE` statement. Furthermore, the Java types of the method's arguments and return value must match the SQL types declared in the `CREATE FUNCTION/PROCEDURE` statement.

The following may happen:

Result	Description
<i>Success</i>	If exactly one Java method matches, then Splice Machine invokes it.
<i>Ambiguity</i>	If exactly one Java method matches, then Splice Machine invokes it.
<i>Failure</i>	Splice Machine also raises an error if no method matches.

In mapping SQL data types to Java data types, Splice Machine considers the following kinds of matches:

Result	Description	
<i>Primitive Match</i>	Splice Machine looks for a primitive Java type corresponding to the SQL type. For instance, SQL <code>INTEGER</code> matches Java <code>int</code> .	
<i>Wrapper Match</i>	Splice Machine looks for a wrapper class in the <code>java.lang</code> or <code>java.sql</code> packages corresponding to the SQL type. For instance, SQL <code>INTEGER</code> matches <code>java.lang.Integer</code> . For a user-defined type (UDT), Splice Machine looks for the UDT's external name class.	
<i>Array Match</i>	For <code>OUT</code> and <code>INOUT</code> procedure arguments, Splice Machine looks for an array of the corresponding primitive or wrapper type. For example, an <code>OUT</code> procedure argument of type SQL <code>INTEGER</code> matches <code>int[]</code> and <code>Integer[]</code> .	
<i>ResultSet Match</i>	If a procedure is declared to return <i>n</i> <code>RESULT SETS</code> , Splice Machine looks for a method whose last <i>n</i> arguments are of type <code>java.sql.ResultSet[]</code> .	

Splice Machine resolves function and procedure invocations as follows:



Call type	Resolution
<i>Function</i>	Splice Machine looks for a method whose argument and return types are <i>primitive matches</i> or <i>wrapper matches</i> for the function's SQL arguments and return value.
<i>Procedure</i>	<p>Splice Machine looks for a method which returns void and whose argument types match as follows:</p> <ul style="list-style-type: none"> <li>» IN - Method arguments are <i>primitive matches</i> or <i>wrapper matches</i> for the procedure's IN arguments.</li> <li>» OUT and INOUT - Method arguments are <i>array matches</i> for the procedure's OUT and INOUT arguments.</li> </ul> <p>In addition, if the procedure returns <i>n</i> RESULT SETS, then the last <i>n</i> arguments of the Java method must be of type <i>java.sql.ResultSet[]</i></p>

## Example of argument matching

The following function:

```
CREATE FUNCTION TO_DEGREES
  ( RADIANS DOUBLE )
RETURNS DOUBLE
PARAMETER STYLE JAVA
NO SQL LANGUAGE JAVA
EXTERNAL NAME 'example.MathUtils.toDegrees'
;
```

would match all of the following methods:

```
public static double toDegrees( double arg ) {...}
public static Double toDegrees( double arg ) {...}
public static double toDegrees( Double arg ) {...}
public static Double toDegrees( Double arg ) {...}
```

Note that Splice Machine raises an exception if it finds more than one matching method.

## Mapping SQL data types to Java data types

The following table shows how Splice Machine maps specific SQL data types to Java data types.

## SQL and Java type correspondence

SQL Type	Primitive Match	Wrapper Match
BOOLEAN	<i>boolean</i>	<i>java.lang.Boolean</i>
SMALLINT	<i>short</i>	<i>java.lang.Integer</i>
INTEGER	<i>int</i>	<i>java.lang.Integer</i>
BIGINT	<i>long</i>	<i>java.lang.Long</i>
DECIMAL	None	<i>java.math.BigDecimal</i>
NUMERIC	None	<i>java.math.BigDecimal</i>
REAL	<i>float</i>	<i>java.lang.Float</i>
DOUBLE	<i>double</i>	<i>java.lang.Double</i>
FLOAT	<i>double</i>	<i>java.lang.Double</i>
CHAR	None	<i>java.lang.String</i>
VARCHAR	None	<i>java.lang.String</i>
LONG VARCHAR	None	<i>java.lang.String</i>
CLOB	None	<i>java.sql.Clob</i>
BLOB	None	<i>java.sql.Blob</i>
DATE	None	<i>java.sql.Date</i>
TIME	None	<i>java.sql.Time</i>
TIMESTAMP	None	<i>java.sql.Timestamp</i>
User-defined type	None	Underlying Java class

## See Also

» [About Data Types](#)

# Identifiers

This section contains the reference documentation for the Splice Machine SQL Identifiers, in these topics:

- » This page provides an introduction to `SQLIdentifiers`.
- » The [SQL Identifier Syntax](#) topic contains additional information about `SQLIdentifier` naming rules, capitalization, and special characters.
- » The [SQL Identifier Types](#) topic provides specific information about the different types of `SQLIdentifiers` that you'll find mentioned in this manual, including:
  - `AuthorizationIdentifier`
  - `column-Name` and `simple-column-Name`
  - `constraint-Name`
  - `correlation-Name`
  - `index-Name`
  - `new-Table-Name`
  - `RoleName`
  - `schemaName`
  - `synonym-Name`
  - `table-Name`
  - `triggerName`
  - `view-Name`

## About SQLIdentifiers

An `SQLIdentifier` is a dictionary object identifier that conforms to the rules of ANSI SQL; identifiers for dictionary objects:

- » are limited to 128 characters
- » are automatically translated into uppercase by the system, making them case-insensitive unless delimited by double quotes
- » cannot be a [Splice Machine SQL keyword](#) unless delimited by double quotes
- » can sometimes be qualified by a schema, table, or correlation name, as described below

## Examples:

Here is an example of a simple, unqualified `SQLIdentifier` used to name a table:

```
CREATE TABLE Coaches ( ID INT NOT NULL );
```

And here's an example of a table name (`Coaches`) qualified by a schema name (`Baseball`):

```
CREATE TABLE Baseball.Coaches ( ID INT NOT NULL );
```

This view name is stored in system catalogs as `PITCHINGCOACHES`, since it is not quoted:

```
CREATE VIEW PitchingCoaches (RECEIVED) AS VALUES 1;
```

Whereas this view name is quoted, and thus is stored as `PitchingCoaches` in the system catalog:

```
CREATE VIEW "PitchingCoaches" (RECEIVED) AS VALUES 1;
```



Complete syntax, including information about case sensitivity and special character usage, in SQL Identifier types is found in the [SQL Identifier Syntax](#) topic in this section.

## Identifier Types

This topic describes the different types of SQLIdentifiers that are used in this manual. .



Complete syntax, including information about case sensitivity and special character usage in SQL Identifier types, is found in the [SQL Identifier Syntax](#) topic in this section.

We use a number of different identifier types in the SQL Reference Manual, all of which are SQLIdentifiers. Some can be qualified with schema, table, or correlation names, as described in the following table:

Topic	Description
Authorization Identifier	<p>An <code>Authorization Identifier</code> is an SQLIdentifier that represents the name of the user when you specify one in a connection request, otherwise known as a user name. When you connect with a user name, that name becomes the default schema name; if you do not specify a user name in the connect request, the default user name and <code>schemaName</code> is <code>SPLICE</code>.</p> <p>User names can be case-sensitive within the authentication system, but they are always case-insensitive within Splice Machine's authorization system unless they are delimited.</p>
column-Name	<p>A <code>column-Name</code> is a SQLIdentifiers that can be unqualified <code>simple-column-Names</code>. or can be qualified with a <code>table-name</code> or <code>correlation-name</code>.</p> <p>See the <a href="#">Column Name Notes</a> section below for information about when a <code>column-Name</code> can or cannot be qualified.</p>
column-Position	<p>A <code>column-Position</code> is an integer value that specifies the ordinal position value of the column. The first column is column 1.</p>
column-Name-or-Position	<p>A <code>column-Name-or-Position</code> is either a <a href="#">column-Name</a> or <a href="#">column-Position</a> value.</p>
constraint-Name	<p>A <code>constraint-Name</code> is a simple SQLIdentifier used to name constraints.</p> <p>You cannot qualify a <code>constraint-Name</code>.</p>

correlation-Name	<p>A <code>correlation-Name</code> is a simple <code>SQLIdentifier</code> used in a <code>FROM</code> clause as a new name or alias for that table.</p> <p>You cannot qualify a <code>correlation-Name</code>, nor can you use it for a column named in the <code>FOR UPDATE</code> clause, as described in the <a href="#">Correlation Name Notes</a> section below</p>
index-Name	<p>An <code>index-Name</code> is an <code>SQLIdentifier</code> that can be qualified with a <code>schemaName</code>.</p> <p>If you do not use a qualifying schema name, the default schema is assumed. Note that system table indexes are qualified with the <code>SYS .</code> schema prefix.</p>
new-Table-Name	<p>A <code>new-Table-Name</code> is a simple <code>SQLIdentifier</code> that is used when renaming a table with the <a href="#">RENAME TABLE</a> statement.</p> <p>You cannot qualify a new table name with a schema name, because the table already exists in a specific schema.</p>
RoleName	<p>A <code>RoleName</code> is a simple <code>SQLIdentifier</code> used to name roles in your database.</p> <p>You cannot qualify a role name.</p>
schemaName	<p>A <code>schemaName</code> is used when qualifying the names of dictionary objects such as tables and indexes.</p> <p>The default user schema is named <code>SPLICE</code> if you do not specify a user name at connection time, <code>SPLICE</code> is assumed as the schema for any unqualified dictionary objects that you reference.</p> <p>Note that you must always qualify references to system tables with the <code>SYS .</code> prefix, e.g. <code>SYS.SYSROLES</code>.</p>
simple-column-Name	<p>A <code>simple-column-Name</code> is used to represent a column that is not qualified by a <code>table-Name</code> or <code>correlation-Name</code>, as described in the <a href="#">Column Name Notes</a> section below.</p>
synonym-Name	<p>A <code>synonym-Name</code> is an <code>SQLIdentifier</code> used for synonyms.</p> <p>You can optionally be qualify a <code>synonym-Name</code> with a <code>schemaName</code>. If you do not use a qualifying schema name, the default schema is assumed.</p>

table-Name	<p>A table-Name is an <code>SQLIdentifier</code> use to name tables.</p> <p>You can optionally qualify a table-Name with a <code>schemaName</code>. If you do not use a qualifying schema name, the default schema is assumed. Note that system table names are qualified with the <code>SYS.</code> schema prefix.</p>
triggerName	<p>A triggerName is an <code>SQLIdentifier</code> used to name user-defined triggers.</p> <p>You can optionally qualify a triggerName with a <code>schemaName</code>. If you do not use a qualifying schema name, the default schema is assumed.</p>
view-Name	<p>A view-Name is an <code>SQLIdentifier</code> used to name views.</p> <p>You can optionally qualify a view-Name with a <code>schemaName</code>. If you do not use a qualifying schema name, the default schema is assumed.</p>

## Column Name Notes {#Note.ColumnName}

Column names are either `simple-column-Name` identifiers, which cannot be qualified, or `column-Name` identifiers that can be qualified with a `table-Name` or `correlation-Name`. Here's the syntax:

```
[ { table-Name | correlation-Name } . ] SQLIdentifier
```

In some circumstances, you must use a `simple-column-Name` and cannot qualify the column name:

- » When creating a table ([CREATE TABLE statement](#)).
- » In a column's `correlation-Name` in a [SELECT expression](#)
- » In a column's `correlation-Name` in a [TABLE expression](#)

## Correlation Name Notes {#Note.CorrelationName}

You cannot use a correlation name for columns that are listed in the `FOR UPDATE` list of a `SELECT`. For example, in the following:

```
SELECT Average AS corrCol1, Homeruns AS corrCol2, Strikeouts FROM Batting FOR U
PDATE of Average, Strikeouts;
```

- » You cannot use `corrCol1` as a correlation name for `Average` because `Average` is listed in the `FOR UPDATE` list.
- » You can use `corrCol2` as a correlation name for `HomeRuns` because the `HomeRuns` column is not in the update list.

## SQL Identifier Syntax

An SQLIdentifier is a dictionary object identifier that conforms to the rules of ANSI SQL; identifiers for dictionary objects:

- » are limited to 128 characters
- » are automatically translated into uppercase by the system, making them case-insensitive unless delimited by double quotes
- » cannot be a [Splice Machine SQL keyword](#) unless delimited by double quotes
- » can sometimes be qualified by a schema, table, or correlation name, as described below

### Examples:

This view name:

```
CREATE VIEW PitchingCoaches(RECEIVED) AS VALUES 1;
```

is stored in system catalogs as PITCHINGCOACHES, since it is not quoted.

Whereas this view name:

```
CREATE VIEW "PitchingCoaches"(RECEIVED) AS VALUES 1;
```

is quoted, and thus is stored as PitchingCoaches in the system catalog

## Qualifying dictionary objects

Since some dictionary objects can be contained within other objects, you can qualify those dictionary object names. Each component is separated from the next by a period (.). You qualify a dictionary object name in order to avoid ambiguity.

### Examples:

Here is an example of a simple, unqualified SQLIdentifier used to name a table:

```
CREATE TABLE Coaches( ID INT NOT NULL );
```

And here's an example of a table name (Coaches) qualified by a schema name (Baseball):

```
CREATE TABLE Baseball.Coaches( ID INT NOT NULL );
```

## Rules for SQL Identifiers

Here are some additional rules that apply to SQLIdentifiers:



» Ordinary identifiers are identifiers not surrounded by double quotation marks:

- An ordinary identifier must begin with a letter and contain only letters, underscore characters (`_`), and digits.
- All Unicode letters and digits are permitted; however, Splice Machine does not attempt to ensure that the characters in identifiers are valid in the database's locale.

» Delimited identifiers are identifiers surrounded by double quotation marks:

- A delimited identifier can contain any characters within the double quotation marks.
- The enclosing double quotation marks are not part of the identifier; they serve only to mark its beginning and end.
- Spaces at the end of a delimited identifier are truncated.
- You can use two consecutive double quotation marks within a delimited identifier to include a double quotation mark within the identifier.

## Capitalization and Special Characters in SQL Statements

You can submit SQL statements to Splice Machine as strings by using JDBC; these strings use the Unicode character set. Within these strings:

- » Double quotation marks delimit special identifiers referred to in ANSI SQL as *delimited identifiers*.
- » Single quotation marks delimit character strings.
- » Within a character string, to represent a single quotation mark or apostrophe, use two single quotation marks. (In other words, a single quotation mark is the escape character for a single quotation mark).
- » SQL keywords are case-insensitive. For example, you can type the keyword `SELECT` as `SELECT`, `Select`, `select`, or `sELECT`.
- » ANSI SQL -style identifiers are case-insensitive unless they are delimited.
- » Java-style identifiers are always case-sensitive.

## Other Special Characters:

- » `*` is a wildcard within a [Select Expression](#). It can also be the multiplication operator. In all other cases, it is a syntactical metasymbol that flags items you can repeat 0 or more times.
- » `%` and `_` are character wildcards when used within character strings following a `LIKE` operator (except when escaped with an escape character). See [Boolean expressions](#).
- » Comments can be either single-line or multi-line, as per the ANSI SQL standard:
  - » Single line comments start with two dashes (`--`) and end with the newline character.
  - » Multi-line comments are bracketed and start with forward slash star (`/*`), and end with star forward slash (`*/`). Note that bracketed comments may be nested. Any text between the starting and ending comment character sequence is ignored.

# Data Types

The SQL type system is used by the language compiler to determine the compile-time type of an expression and by the language execution system to determine the runtime type of an expression, which can be a subtype or implementation of the compile-time type.

Each type has associated with it values of that type. In addition, values in the database or resulting from expressions can be `NULL`, which means the value is missing or unknown. Although there are some places where the keyword `NULL` can be explicitly used, it is not in itself a value, because it needs to have a type associated with it.

This section contains the reference documentation for the Splice Machine SQL Data Types, in the following subsections:

- » [Character String Data Types](#)
- » [Date and Time Data Types](#)
- » [Large Object Binary \(LOB\) Data Types](#)
- » [Numeric Data Types](#)
- » [Other Data Types](#)

## Character String Data Types

These are the [character string data types](#):

Data Type	Description
<a href="#">CHAR</a>	The <code>CHAR</code> data type provides for fixed-length storage of strings.
<a href="#">LONG VARCHAR</a>	The <code>LONG VARCHAR</code> type allows storage of character strings with a maximum length of 32,700 characters. It is identical to <code>VARCHAR</code> , except that you cannot specify a maximum length when creating columns of this type.
<a href="#">VARCHAR</a>	The <code>VARCHAR</code> data type provides for variable-length storage of strings.

## Date and Time Data Types

These are the [date and time data types](#):

Data Type	Description
-----------	-------------

<a href="#">DATE</a>	The <code>DATE</code> data type provides for storage of a year-month-day in the range supported by <i>java.sql.Date</i> .
<a href="#">TIME</a>	The <code>TIME</code> data type provides for storage of a time-of-day value.
<a href="#">TIMESTAMP</a>	The <code>TIMESTAMP</code> data type stores a combined <code>DATE</code> and <code>TIME</code> value, and allows a fractional-seconds value of up to nine digits.

## Large Object Binary (LOB) Data Types

These are the [LOB data types](#):

Data Type	Description
<a href="#">BLOB</a>	The <code>BLOB</code> (binary large object) data type is used for varying-length binary strings that can be up to 2,147,483,647 characters long.
<a href="#">CLOB</a>	The <code>CLOB</code> (character large object) data type is used for varying-length character strings that can be up to 2,147,483,647 characters long.
<a href="#">TEXT</a>	Exactly the same as <code>CLOB</code> .

## Numeric Data Types

These are the [numeric data types](#):

Data Type	Description
<a href="#">BIGINT</a>	The <code>BIGINT</code> data type provides 8 bytes of storage for integer values.
<a href="#">DECIMAL</a>	<p>The <code>DECIMAL</code> data type provides an exact numeric in which the precision and scale can be arbitrarily sized.</p> <p>You can use <code>DECIMAL</code> and <code>NUMERIC</code> interchangeably.</p>

<a href="#">DOUBLE</a>	The <code>DOUBLE</code> data type provides 8-byte storage for numbers using IEEE floating-point notation.  <code>DOUBLE PRECISION</code> can be used synonymously with <code>DOUBLE</code> .
<a href="#">DOUBLE PRECISION</a>	The <code>DOUBLE PRECISION</code> data type provides 8-byte storage for numbers using IEEE floating-point notation.  <code>DOUBLE</code> can be used synonymously with <code>DOUBLE PRECISION</code> .
<a href="#">FLOAT</a>	The <code>FLOAT</code> data type is an alias for either a <code>REAL</code> or <code>DOUBLE PRECISION</code> data type, depending on the precision you specify.
<a href="#">INTEGER</a>	<code>INTEGER</code> provides 4 bytes of storage for integer values.
<a href="#">NUMERIC</a>	The <code>NUMERIC</code> data type provides an exact numeric in which the precision and scale can be arbitrarily sized.  You can use <code>NUMERIC</code> and <code>DECIMAL</code> interchangeably.
<a href="#">REAL</a>	The <code>REAL</code> data type provides 4 bytes of storage for numbers using IEEE floating-point notation.
<a href="#">SMALLINT</a>	The <code>SMALLINT</code> data type provides 2 bytes of storage.

## Other Data Types

These are the [other data types](#):

Data Type	Description
<a href="#">BOOLEAN</a>	Provides 1 byte of storage for logical values.

## See Also

- » [Argument Matching](#)
- » [Assignments](#)

- » [BIGINT](#) data type
- » [BLOB](#) data type
- » [BOOLEAN](#) data type
- » [CHAR](#) data type
- » [CLOB](#) data type
- » [DATE](#) data type
- » [DECIMAL](#) data type
- » [DOUBLE](#) data type
- » [DOUBLE PRECISION](#) data type
- » [FLOAT](#) data type
- » [INTEGER](#) data type
- » [LONG VARCHAR](#) data type
- » [NUMERIC](#) data type
- » [REAL](#) data type
- » [SMALLINT](#) data type
- » [TEXT](#) data type
- » [TIME](#) data type
- » [TIMESTAMP](#) data type
- » [VARCHAR](#) data type

# BIGINT

The `BIGINT` data type provides 8 bytes of storage for integer values.

## Syntax

```
BIGINT
```

## Corresponding Compile-time Java Type

```
java.lang.Long
```

## JDBC Metadata Type (java.sql.Types)

```
BIGINT
```

## Notes

Here are several usage notes for the `BIGINT` data type:

- » The minimum value is `-9223372036854775808` (`java.lang.Long.MIN_VALUE`)
- » The maximum value is `9223372036854775807` (`java.lang.Long.MAX_VALUE`)
- » When mixed with other data types in expressions, the resulting data type follows the rules shown in [Numeric type promotion in expressions](#).
- » An attempt to put an integer value of a larger storage size into a location of a smaller size fails if the value cannot be stored in the smaller-size location. Integer types can always successfully be placed in approximate numeric values, although with the possible loss of some precision. `BIGINT`s can be stored in `DECIMAL`s if the `DECIMAL` precision is large enough for the value.

## Example

```
9223372036854775807
```

# BLOB

A `BLOB` (binary large object) value is a varying-length binary string that can be up to 2GB (2,147,483,647) characters long.

If you're using a `BLOB` with the 32-bit version of our ODBC driver, the size of the `BLOB` is limited to 512 MB, due to address space limitations.

Like other binary types, `BLOB`strings are not associated with a code page. In addition, `BLOB`strings do not hold character data.

## Syntax

```
{BLOB | BINARY LARGE OBJECT} [ ( length [{K | M | G}] ) ]
```

### *length*

An unsigned integer constant that specifies the number of characters in the `BLOB` unless you specify one of the suffixes you see below, which change the meaning of the *length* value. If you do not specify a length value, it defaults to two gigabytes (2,147,483,647).

### *K*

If specified, indicates that the length value is in multiples of 1024 (kilobytes).

### *M*

If specified, indicates that the length value is in multiples of 1024\*1024 (megabytes).

### *G*

If specified, indicates that the length value is in multiples of 1024\*1024\*1024 (gigabytes).

## Corresponding Compile-time Java Type

```
java.sql.Blob
```

## JDBC Metadata Type (java.sql.Types)

```
BLOB
```

## Usage Notes

Use the *getBlob* method on the *java.sql.ResultSet* to retrieve a `BLOB` handle to the underlying data.

There are a number of restrictions on using `BLOB` and `CLOB` / `TEXT` objects, which we refer to as LOB-types:

- » LOB-types cannot be compared for equality (=) and non-equality (!=, <>).

- » LOB-typed values cannot be ordered, so `<`, `<=`, `>`, `>=` tests are not supported.
- » LOB-types cannot be used in indexes or as primary key columns.
- » `DISTINCT`, `GROUP BY`, and `ORDER BY` clauses are also prohibited on LOB-types.
- » LOB-types cannot be involved in implicit casting as other base-types.

## Example

Using an [INSERT](#) statement to put `BLOB` data into a table has some limitations if you need to cast a long string constant to a `BLOB`. You may be better off using a binary stream, as in the following code fragment.



```

package com.splicemachine.tutorials.blob;

import java.io.FileInputStream;
import java.io.FileNotFoundException;
import java.io.InputStream;
import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.PreparedStatement;
import java.sql.ResultSet;
import java.sql.SQLException;
import java.sql.Statement;

public class ExampleInsertBlob {

    /**
     * Example of inserting a blob using JDBC
     *
     * @param args[0] - Image file to insert
     * @param args[1] - JDBC URL - optional - defaults to localhost
     */
    public static void main(String[] args) {
        Connection conn = null;
        Statement statement = null;
        ResultSet rs = null;

        if(args.length == 0) {
            System.out.println("You must pass in an file (like an image) to be loaded");
        }

        try {

            String imageFileToLoad = args[0];

            //Default JDBC Connection String - connects to local database
            String dbUrl = "jdbc:splice://localhost:1527/splicedb;user=splice;password=true";

            //Checks to see if a JDBC URL is passed in
            if(args.length > 1) {
                dbUrl = args[1];
            }

            //For the JDBC Driver - Use the Splice Machine Client Driver
            Class.forName("com.splicemachine.db.jdbc.ClientDriver");

            //Connect to the database
            conn = DriverManager.getConnection(dbUrl);

            //Create a statement
            statement = conn.createStatement();

```

```

//Create a table
statement.execute("CREATE TABLE IMAGES(a INT, test BLOB)");

//Create an input stream
InputStream fin = new FileInputStream(imageFileToLoad);
PreparedStatement ps = conn.prepareStatement("INSERT INTO IMAGES VALUES
(?, ?)");
ps.setInt(1, 1477);

// - set value of input parameter to the input stream
ps.setBinaryStream(2, fin);
ps.execute();

ps.close();

//Lets get the count of records
rs = statement.executeQuery("select count(1) from IMAGES");
if(rs.next()) {
    System.out.println("count=[" + rs.getInt(1) + "]);
}

} catch (ClassNotFoundException cne) {
    cne.printStackTrace();
} catch (SQLException se) {
    se.printStackTrace();
} catch (FileNotFoundException e) {
    // TODO Auto-generated catch block
    e.printStackTrace();
} finally {
    if(rs != null) {
        try { rs.close(); } catch (Exception ignore) { }
    }
    if(statement != null) {
        try { statement.close(); } catch (Exception ignore) { }
    }
    if(conn != null) {
        try { conn.close(); } catch (Exception ignore) { }
    }
}

}

}

```

# BOOLEAN

The `BOOLEAN` data type provides 1 byte of storage for logical values.

## Syntax

```
BOOLEAN
```

## Corresponding Compile-time Java Type

```
java.lang.Boolean
```

## JDBC Metadata Type (java.sql.Types)

```
BOOLEAN
```

## Usage Notes

Here are several usage notes for the `BOOLEAN` data type:

- » The legal values are `true`, `false`, and `null`.
- » `BOOLEAN` values can be cast to and from character type values.
- » For comparisons and ordering operations, `true` sorts higher than `false`.

## Examples

```
values true;  
values false;  
values cast (null as boolean);
```

# CHAR

The `CHAR` data type provides for fixed-length storage of strings.

## Syntax

```
CHAR[ACTER] [(length)]
```

*length*

An unsigned integer literal designating the length in bytes. The default *length* for a `CHAR` is 1; the maximum size of *length* is 254..

## Corresponding Compile-time Java Type

```
java.lang.String
```

## JDBC Metadata Type (java.sql.Types)

```
CHAR
```

## Usage Notes

Here are several usage notes for the `CHAR` data type:

- » Splice Machine inserts spaces to pad a string value shorter than the expected length, and truncates spaces from a string value longer than the expected length. Characters other than spaces cause an exception to be raised. When [comparison boolean operators](#) are applied to `CHARs`, the shorter string is padded with spaces to the length of the longer string.
- » When `CHARs` and `VARCHARs` are mixed in expressions, the shorter value is padded with spaces to the length of the longer value.
- » The type of a string constant is `CHAR`.

## Examples

```
-- within a string constant use two single quotation marks
-- to represent a single quotation mark or apostrophe
VALUES 'hello this is Joe''s string';

-- create a table with a CHAR field
CREATE TABLE STATUS (
  STATUSCODE CHAR(2) NOT NULL
    CONSTRAINT PK_STATUS PRIMARY KEY,
  STATUSDESC VARCHAR(40) NOT NULL
);
```

# CLOB

A CLOB (character large object) value can be up to 2 GB (2,147,483,647 characters) long. A CLOB is used to store unicode character-based data, such as large documents in any character set.

If you're using a CLOB with the 32-bit version of our ODBC driver, the size of the CLOB is limited to 512 MB, due to address space limitations.

Note that, in Splice Machine, TEXT is a synonym for CLOB, and that the documentation for the [TEXT](#) data type functionally matches the documentation for this topic. Splice Machine simply translates TEXT into CLOB.

## Syntax

```
{CLOB | CHARACTER LARGE OBJECT} [ ( length [{K |M |G}] ) ]
```

### *length*

An unsigned integer constant that specifies the number of characters in the CLOB unless you specify one of the suffixes you see below, which change the meaning of the *length* value. If you do not specify a length value, it defaults to two giga-characters (2,147,483,647).

### *K*

If specified, indicates that the length value is in multiples of 1024 (kilo-characters).

### *M*

If specified, indicates that the length value is in multiples of 1024\*1024 (mega-characters).

### *G*

If specified, indicates that the length value is in multiples of 1024\*1024\*1024 (giga-characters).

## Corresponding Compile-time Java Type

```
java.sql.Clob
```

## JDBC Metadata Type (java.sql.Types)

```
CLOB
```

## Usage Notes

Use the *getClob* method on the *java.sql.ResultSet* to retrieve a CLOB handle to the underlying data.

There are a number of restrictions on using BLOB and CLOB / TEXT objects, which we refer to as LOB-types:

- » LOB-types cannot be compared for equality (=) and non-equality (!=, <>).
- » LOB-typed values cannot be ordered, so <, <=, >, >= tests are not supported.
- » LOB-types cannot be used in indexes or as primary key columns.
- » DISTINCT, GROUP BY, and ORDER BY clauses are also prohibited on LOB-types.
- » LOB-types cannot be involved in implicit casting as other base-types.

## Example

```
CREATE TABLE myTable ( largeCol CLOB(65535) );
```

# DATE

The `DATE` data type provides for storage of a year-month-day in the range supported by *java.sql.Date*.

## Syntax

```
DATE
```

## Corresponding Compile-time Java Type

```
java.sql.Date
```

## JDBC Metadata Type (java.sql.Types)

```
DATE
```

## Usage Notes

Here are several notes about using the `DATE` data type:

- » Dates, [timestamps](#) must not be mixed with one another in expressions.
- » Any value that is recognized by the *java.sql.Date* method is permitted in a column of the corresponding SQL date/time data type. Splice Machine supports the following formats for `DATE`:

```
yyyy-mm-dd
mm/dd/yyyy
dd.mm.yyyy
```

- » The first of the three formats above is the *java.sql.Date* format.
- » The year must always be expressed with four digits, while months and days may have either one or two digits.
- » Splice Machine also accepts strings in the locale specific date-time format, using the locale of the database server. If there is an ambiguity, the built-in formats above take precedence.

Please see [Working With Date and Time Values](#) in the for information about using simple arithmetic with `DATE` values.



## Examples

```
VALUES DATE ('1994-02-23');  
VALUES '1993-09-01';
```

## See Also

- » [CURRENT\\_DATE](#) function
- » [DATE](#) function
- » [DAY](#) function
- » [EXTRACT](#) function
- » [LASTDAY](#) function
- » [MONTH](#) function
- » [MONTH\\_BETWEEN](#) function
- » [MONTHNAME](#) function
- » [NEXTDAY](#) function
- » [NOW](#) function
- » [QUARTER](#) function
- » [TIME](#) data type
- » [TIMESTAMP](#) function
- » [TO\\_CHAR](#) function
- » [TO\\_DATE](#) function
- » [WEEK](#) function
- » [\*Working with Dates\*](#) in the *Developer's Guide*

# DECIMAL

The `DECIMAL` data type provides an exact numeric in which the precision and scale can be arbitrarily sized. You can specify the *precision* (the total number of digits, both to the left and the right of the decimal point) and the *scale* (the number of digits of the fractional component). The amount of storage required depends on the precision you specify.

Note that `NUMERIC` is a synonym for `DECIMAL`, and that the documentation for the [NUMERIC](#) data type exactly matches the documentation for this topic.

## Syntax

```
{ DECIMAL | DEC } [(precision [, scale])]
```

### *precision*

Must be between 1 and 31. If not specified, the default precision is 5.

### *scale*

Must be less than or equal to the precision. If not specified, the default scale is 0.

## Usage Notes

Here are several notes about using the `DECIMAL` data type:

- » An attempt to put a numeric value into a `DECIMAL` is allowed as long as any non-fractional precision is not lost. When truncating trailing digits from a `DECIMAL` value, Splice Machine rounds down. For example:

```
-- this cast loses only fractional precision
values cast (1.798765 AS decimal(5,2));
1
-----
1.79

-- this cast does not fit:
values cast (1798765 AS decimal(5,2));
ERROR 22003: The resulting value is outside the range for the data type DECIMAL/NUMERIC (5,2) .
```

- » When mixed with other data types in expressions, the resulting data type follows the rules shown in [Storing values of one numeric data type in columns of another numeric data type](#).
- » When two decimal values are mixed in an expression, the scale and precision of the resulting value follow the rules shown in [Scale for decimal arithmetic](#).
- » Integer constants too big for `BIGINT` are made `DECIMAL` constants.

## Corresponding Compile-time Java Type

```
java.math.BigDecimal
```

## JDBC Metadata Type (java.sql.Types)

```
DECIMAL
```

## Examples

```
VALUES 123.456;  
VALUES 0.001;
```

# DOUBLE

The `DOUBLE` data type provides 8-byte storage for numbers using IEEE floating-point notation. `DOUBLE PRECISION` can be used synonymously with `DOUBLE`, and the documentation for this topic is identical to the documentation for the [DOUBLE PRECISION](#) topic.

## Syntax

```
DOUBLE
```

or, alternately

```
DOUBLE PRECISION
```

## Usage Notes

Here are several usage notes for the `DOUBLE/DOUBLE PRECISION` data type:

» The following range limitations apply:

Limit type	Limitation
Smallest <code>DOUBLE</code> value	<code>-1.79769E+308</code>
Largest <code>DOUBLE</code> value	<code>1.79769E+308</code>
Smallest positive <code>DOUBLE</code> value	<code>2.225E-307</code>
Largest negative <code>DOUBLE</code> value	<code>-2.225E-307</code>

**NOTE:** These limits are different from the `java.lang.Double` Java type limits.

- » An exception is thrown when any double value is calculated or entered that is outside of these value ranges. Arithmetic operations **do not** round their resulting values to zero. If the values are too small, you will receive an exception.
- » Numeric floating point constants are limited to a length of 30 characters.

```
-- this example will fail because the constant is too long:
values 01234567890123456789012345678901e0;
```

- » When mixed with other data types in expressions, the resulting data type follows the rules shown in [Storing values of one numeric data type in columns of another numeric data type](#).

## Corresponding Compile-time Java Type

```
java.lang.Double
```

## JDBC Metadata Type (java.sql.Types)

```
DOUBLE
```

## Examples

```
3421E+09  
425.43E9  
9E-10  
4356267544.32333E+30
```

# DOUBLE PRECISION

The `DOUBLE PRECISION` data type provides 8-byte storage for numbers using IEEE floating-point notation. `DOUBLE` can be used synonymously with `DOUBLE PRECISION`, and the documentation for this topic is identical to the documentation for the [DOUBLE](#) topic.

## Syntax

```
DOUBLE PRECISION
```

or, alternately

```
DOUBLE
```

## Usage Notes

Here are several usage notes for the `DOUBLE/DOUBLE PRECISION` data type:

- » The following range limitations apply:

Limit type	Limitation
Smallest <code>DOUBLE</code> value	-1.79769E+308
Largest <code>DOUBLE</code> value	1.79769E+308
Smallest positive <code>DOUBLE</code> value	2.225E-307
Largest negative <code>DOUBLE</code> value	-2.225E-307

**NOTE:** These limits are different from the `java.lang.Double` Java type limits

- » An exception is thrown when any double value is calculated or entered that is outside of these value ranges. Arithmetic operations **do not** round their resulting values to zero. If the values are too small, you will receive an exception.
- » Numeric floating point constants are limited to a length of 30 characters.

```
-- this example will fail because the constant is too long:
values 01234567890123456789012345678901e0;
```

- » When mixed with other data types in expressions, the resulting data type follows the rules shown in [Storing values of one numeric data type in columns of another numeric data type](#).

## Corresponding Compile-time Java Type

```
java.lang.Double
```

## JDBC Metadata Type (java.sql.Types)

```
DOUBLE
```

## Examples

```
3421E+09  
425.43E9  
9E-10  
4356267544.32333E+30
```

# FLOAT

The `FLOAT` data type is an alias for either a [DOUBLE PRECISION](#) data type, depending on the precision you specify.

## Syntax

```
Float [ (precision) ]
```

### *precision*

The default precision for `FLOAT` is 52, which is equivalent to `DOUBLE PRECISION`.

A precision of 23 or less makes `FLOAT` equivalent to `REAL`.

A precision of 24 or greater makes `FLOAT` equivalent to `DOUBLE PRECISION`.

If you specify a precision of 0, you get an error. If you specify a negative precision, you get a syntax error.

## JDBC Metadata Type (java.sql.Types)

```
REAL or DOUBLE
```

## Usage Notes

If you are using a precision of 24 or greater, the limits of `FLOAT` are similar to the limits of `DOUBLE`.

If you are using a precision of 23 or less, the limits of `FLOAT` are similar to the limits of `REAL`.

Data defined with type [double](#) at this time. Note that this does not cause a loss of precision, though the data may require slightly more space.



# INTEGER Data Type

The `INTEGER` data type provides 4 bytes of storage for integer values.

## Syntax

```
{ INTEGER | INT }
```

## Corresponding Compile-Time Java Type

```
java.lang.Integer
```

## JDBC Metadata Type (java.sql.Types)

```
INTEGER
```

## Minimum Value

```
-2147483648 (java.lang.Integer.MIN_VALUE)
```

## Maximum Value

```
2147483647 (java.lang.Integer.MAX_VALUE)
```

## Usage Notes

When mixed with other data types in expressions, the resulting data type follows the rules shown in [Numeric type promotion in expressions](#).

See also [Storing values of one numeric data type in columns of another numeric data type](#).

## Examples

```
3453  
425
```



# LONG VARCHAR

The `LONG VARCHAR` type allows storage of character strings with a maximum length of 32,670 characters. It is almost identical to [VARCHAR](#), except that you cannot specify a maximum length when creating columns of this type.

## Syntax

```
LONG VARCHAR
```

## Corresponding Compile-time Java Type

```
java.lang.String
```

## JDBC Metadata Type (java.sql.Types)

```
LONGVARCHAR
```

## Usage Notes

When you are converting from Java values to SQL values, no Java type corresponds to `LONG VARCHAR`.

## NUMERIC Data Type

NUMERIC is a synonym for the [DECIMAL](#) data type and behaves the same way. The documentation below is a mirror of the documentation for the DECIMAL data type.

NUMERIC provides an exact numeric in which the precision and scale can be arbitrarily sized. You can specify the *precision* (the total number of digits, both to the left and the right of the decimal point) and the *scale* (the number of digits of the fractional component). The amount of storage required depends on the precision you specify.

## Syntax

```
NUMERIC [(precision [, scale ])]
```

### *precision*

Must be between 1 and 31. If not specified, the default precision is 5.

### *scale*

Must be less than or equal to the precision. If not specified, the default scale is 0.

## Usage Notes

Here are several notes about using the NUMERIC data type:

- » An attempt to put a numeric value into a NUMERIC is allowed as long as any non-fractional precision is not lost. When truncating trailing digits from a NUMERIC value, Splice Machine rounds down. For example:

```
-- this cast loses only fractional precision
values cast (1.798765 AS numeric(5,2));
1
-----
1.79

-- this cast does not fit:
values cast (1798765 AS numeric(5,2));
ERROR 22003: The resulting value is outside the range for the data type DECIMAL/NUMERIC(5,2).
```

- » When mixed with other data types in expressions, the resulting data type follows the rules shown in [Storing values of one numeric data type in columns of another numeric data type](#).
- » When two numeric values are mixed in an expression, the scale and precision of the resulting value follow the rules shown in [Scale for decimal arithmetic](#).
- » Integer constants too big for BIGINT are made NUMERIC constants.

## Corresponding Compile-time Java Type

```
java.math.BigDecimal
```

## JDBC Metadata Type (java.sql.Types)

```
NUMERIC
```

## Examples

```
VALUES 123.456;  
VALUES 0.001;
```

# REAL

The `REAL` data type provides 4 bytes of storage for numbers using IEEE floating-point notation.

## Syntax

```
REAL
```

## Corresponding Compile-time Java Type

```
java.lang.Float
```

## JDBC Metadata Type (java.sql.Types)

```
REAL
```

## Limitations

`REAL` value ranges:

Limit type	Limit value
<i>Smallest REAL value</i>	-3.402E+38
<i>Largest REAL value</i>	3.402E+38
<i>Smallest positive REAL value</i>	1.175E-37
<i>Largest negative REAL value</i>	-1.175E-37

**NOTE:** These limits are different from the `java.lang.Float` Java type limits.

## Usage Notes

Here are several usage notes for the `REAL` data type:

- » An exception is thrown when any double value is calculated or entered that is outside of these value ranges. Arithmetic operations **do not** round their resulting values to zero. If the values are too small, you will receive an exception. The arithmetic operations take place with double arithmetic in order to detect under flows.
- » Numeric floating point constants are limited to a length of 30 characters.

```
-- this example will fail because the constant is too long:  
values 01234567890123456789012345678901e0;
```

- » When mixed with other data types in expressions, the resulting data type follows the rules shown in [Numeric type promotion in expressions](#).
- » See also [Storing values of one numeric data type in columns of another numeric data type](#).
- » Constants always map to [DOUBLE PRECISION](#); use a `CAST` to convert a constant to a `REAL`.

# SMALLINT

The `SMALLINT` data type provides 2 bytes of storage.

## Syntax

```
SMALLINT
```

## Corresponding Compile-time Java Type

```
java.lang.Short
```

## JDBC Metadata Type (java.sql.Types)

```
SMALLINT
```

## Usage Notes

Here are several usage notes for the `SMALLINT` data type:

- » The minimum value is `-32768` (`java.lang.Short.MIN_VALUE`).
- » The maximum value is `32767` (`java.lang.Short.MAX_VALUE`).
- » When mixed with other data types in expressions, the resulting data type follows the rules shown in [Numeric type promotion in expressions](#).
- » See also [Storing values of one numeric data type in columns of another numeric data type](#).
- » Constants in the appropriate format always map to `INTEGER` or `BIGINT`, depending on their length.



# TEXT

A `TEXT` (character large object) value can be up to 2,147,483,647 characters long. A `TEXT` object is used to store unicode character-based data, such as large documents in any character set.

Note that, in Splice Machine, `TEXT` is a synonym for `CLOB`, and that the documentation for the [CLOB](#) data type functionally matches the documentation for this topic. Splice Machine simply translates `TEXT` into `CLOB`.

## Syntax

```
TEXT [ ( length [{K |M |G}] ) ]
```

### *length*

An unsigned integer constant that specifies the number of characters in the `TEXT` unless you specify one of the suffixes you see below, which change the meaning of the *length* value. If you do not specify a length value, it defaults to two giga-characters (2,147,483,647).

### *K*

If specified, indicates that the length value is in multiples of 1024 (kilo-characters).

### *M*

If specified, indicates that the length value is in multiples of 1024\*1024 (mega-characters).

### *G*

If specified, indicates that the length value is in multiples of 1024\*1024\*1024 (giga-characters).

## Corresponding Compile-time Java Type

```
java.sql.Clob
```

## JDBC Metadata Type (java.sql.Types)

```
CLOB
```

## Usage Notes

Use the *getClob* method on the *java.sql.ResultSet* to retrieve a `CLOB` handle to the underlying data.

There are a number of restrictions on using `BLOB` and `CLOB` / `TEXT` objects, which we refer to as LOB-types:

- » LOB-types cannot be compared for equality (=) and non-equality (!=, <>).

- » LOB-typed values cannot be ordered, so `<`, `<=`, `>`, `>=` tests are not supported.
- » LOB-types cannot be used in indexes or as primary key columns.
- » `DISTINCT`, `GROUP BY`, and `ORDER BY` clauses are also prohibited on LOB-types.
- » LOB-types cannot be involved in implicit casting as other base-types.

## Example

```
CREATE TABLE myTable ( txtCol TEXT(65535) );
```

# TIME

The `TIME` data type provides for storage of a time-of-day value.

## Syntax

```
TIME
```

## Corresponding Compile-time Java Type

```
java.sql.Time
```

## JDBC Metadata Type (java.sql.Types)

```
TIME
```

## Usage Notes

Here are several usage notes for the `TIME` data type:

- » [timestamps](#) cannot be mixed with one another in expressions except with a `CAST`.
- » Any value that is recognized by the *java.sql.Time* method is permitted in a column of the corresponding SQL date/time data type. Splice Machine supports the following formats for `TIME`:

```
hh:mm[:ss]
hh.mm[:ss]
hh[:mm] {AM | PM}
```

The first of the three formats above is the *java.sql.Time* format.

- » Hours may have one or two digits.
- » Minutes and seconds, if present, must have two digits.
- » Splice Machine also accepts strings in the locale specific date-time format, using the locale of the database server. If there is an ambiguity, the built-in formats above take precedence.

Please see [Working With Date and Time Values](#) for information about using simple arithmetic with `TIME` values.

## Examples

```
VALUES TIME ('15:09:02');  
VALUES '15:09:02';
```

## See Also

- » [CURRENT\\_DATE](#) function
- » [DATE](#) data type
- » [DATE](#) function
- » [DAY](#) function
- » [EXTRACT](#) function
- » [LASTDAY](#) function
- » [MONTH](#) function
- » [MONTH\\_BETWEEN](#) function
- » [MONTHNAME](#) function
- » [NEXTDAY](#) function
- » [NOW](#) function
- » [QUARTER](#) function
- » [TIMESTAMP](#) function
- » [TO\\_CHAR](#) function
- » [TO\\_DATE](#) function
- » [WEEK](#) function
- » [\*Working with Dates\*](#) in the *Developer's Guide*

# TIMESTAMP

The `TIMESTAMP` data type stores a combined [TIME](#) value that permits fractional seconds values of up to nine digits.

## Syntax

```
TIMESTAMP
```

## Corresponding Compile-time Java Type

```
java.sql.Timestamp
```

## JDBC Metadata Type (java.sql.Types)

```
TIMESTAMP
```

## About Timestamp Formats

Splice Machine uses the following Java date and time pattern letters to construct timestamps:

Pattern Letter	Description	Format(s)
y	year	yy or yyyy
M	month	MM
d	day in month	dd
h	hour (0-12)	hh
H	hour (0-23)	HH
m	minute in hour	mm
s	seconds	ss
S	tenths of seconds	SSS (up to 6 decimal digits: SSSSSS)
z	time zone text	e.g. Pacific Standard time

Pattern Letter	Description	Format(s)
Z	time zone, time offset	e.g. -0800

The default timestamp format for Splice Machine imports is: `yyyy-MM-dd HH:mm:ss`, which uses a 24-hour clock, does not allow for decimal digits of seconds, and does not allow for time zone specification.

Please see [Working With Date and Time Values](#) for information about using simple arithmetic with `TIMESTAMP` values.

## Examples

The following tables shows valid examples of timestamps and their corresponding format (parsing) patterns:

Timestamp value	Format Pattern	Notes
2013-03-23 09:45:00	yyyy-MM-dd HH:mm:ss	This is the default pattern.
2013-03-23 19:45:00.98-05	yyyy-MM-dd HH:mm:ss.SSZ	This pattern allows up to 2 decimal digits of seconds, and requires a time zone specification.
2013-03-23 09:45:00-07	yyyy-MM-dd HH:mm:ssZ	This patterns requires a time zone specification, but does not allow for decimal digits of seconds.
2013-03-23 19:45:00.98-0530	yyyy-MM-dd HH:mm:ss.SSZ	This pattern allows up to 2 decimal digits of seconds, and requires a time zone specification.
2013-03-23 19:45:00.123  2013-03-23 19:45:00.12	yyyy-MM-dd HH:mm:ss.SSS	This pattern allows up to 3 decimal digits of seconds, but does not allow a time zone specification.  Note that if your data specifies more than 3 decimal digits of seconds, an error occurs.
2013-03-23 19:45:00.1298	yyyy-MM-dd HH:mm:ss.SSSS	This pattern allows up to 4 decimal digits of seconds, but does not allow a time zone specification.

## Usage Notes

Dates, times, and timestamps cannot be mixed with one another in expressions.

Splice Machine also accepts strings in the locale specific datetime format, using the locale of the database server. If there is an ambiguity, the built-in formats shown above take precedence.

At this time, dates in [TimeStamp](#) values only work correctly when limited to this range of date values: 1678-01-01 to 2261-12-31

## See Also

- » [About Data Types](#)
- » [Working with Dates](#) in the *Developer's Guide*

# VARCHAR

The `VARCHAR` data type provides for variable-length storage of strings.

## Syntax

```
{ VARCHAR | CHAR VARYING | CHARACTER VARYING }(length)
```

*length*

An unsigned integer constant. The maximum length for a `VARCHAR` string is 32,672 characters.

## Corresponding Compile-time Java Type

```
java.lang.String
```

## JDBC Metadata Type (java.sql.Types)

```
VARCHAR
```

## Example

```
VARCHAR(2048);
```

## Usage Notes

Here are several notes for the `VARCHAR` data type:

- » Splice Machine does not pad a `VARCHAR` value whose length is less than specified.
- » Splice Machine truncates spaces from a string value when a length greater than the `VARCHAR` expected is provided. Characters other than spaces are not truncated, and instead cause an exception to be raised.
- » When [comparison boolean operators](#) are applied to `VARCHARs`, the lengths of the operands are not altered, and spaces at the end of the values are ignored.
- » When `CHARs` and `VARCHARs` are mixed in expressions, the shorter value is padded with spaces to the length of the longer value.
- » The type of a string constant is `CHAR`, not `VARCHAR`.



# Statements

This section contains the reference documentation for the Splice Machine SQL Statements, in the following subsections:

- » [Data Definition \(DDL\) - General Statements](#)
- » [Data Definition \(DDL\) - Create Statements](#)
- » [Data Definition \(DDL\) - Drop Statements](#)
- » [Data Manipulation \(DML\) Statements](#)
- » [Session Control Statements](#)

## Data Definition - General Statements

These are the [data definition statements](#):

Statement	Description
<a href="#">ALTER TABLE</a>	Add, deletes, or modifies columns in an existing table.
<a href="#">GRANT</a>	Gives privileges to specific user(s) or role(s) to perform actions on database objects.
<a href="#">PIN TABLE</a>	Caches a table in memory for improved performance.
<a href="#">RENAME COLUMN</a>	Renames a column in a table.
<a href="#">RENAME INDEX</a>	Renames an index in the current schema.
<a href="#">RENAME TABLE</a>	Renames an existing table in a schema.
<a href="#">REVOKE</a>	Revokes privileges for specific user(s) or role(s) to perform actions on database objects.
<a href="#">TRUNCATE TABLE</a>	Resets a table to its initial empty state.
<a href="#">UNPIN TABLE</a>	Unpins a pinned (cached) table.

## Data Definition (DDL) - CREATE Statements

These are the [create statements](#):

Statement	Description
<a href="#">CREATE EXTERNAL TABLE</a>	Allows you to query data stored in a flat file as if that data were stored in a Splice Machine table.
<a href="#">CREATE FUNCTION</a>	Creates Java functions that you can then use in expressions.
<a href="#">CREATE INDEX</a>	Creates an index on a table.
<a href="#">CREATE PROCEDURE</a>	Creates Java stored procedures, which you can then call using the <a href="#">Call Procedure</a> statement.
<a href="#">CREATE ROLE</a>	Creates SQL roles.
<a href="#">CREATE SCHEMA</a>	Creates a schema.
<a href="#">CREATE SEQUENCE</a>	Creates a sequence generator, which is a mechanism for generating exact numeric values, one at a time.
<a href="#">CREATE SYNONYM</a>	Creates a synonym, which can provide an alternate name for a table or a view.
<a href="#">CREATE TABLE</a>	Creates a new table.
<a href="#">CREATE TEMPORARY TABLE</a>	Defines a temporary table for the current connection.
<a href="#">CREATE TRIGGER</a>	Creates a trigger, which defines a set of actions that are executed when a database event occurs on a specified table
<a href="#">CREATE VIEW</a>	Creates a view, which is a virtual table formed by a query.
<a href="#">DECLARE GLOBAL TEMPORARY TABLE</a>	Defines a temporary table for the current connection.

## Data Definition (DDL) - DROP Statements

These are the [drop statements](#):

Statement	Description
<a href="#">DROP FUNCTION</a>	Drops a function from a database.
<a href="#">DROP INDEX</a>	Drops an index from a database.
<a href="#">DROP PROCEDURE</a>	Drops a procedure from a database.
<a href="#">DROP ROLE</a>	Drops a role from a database.

Statement	Description
<a href="#">DROP SCHEMA</a>	Drops a schema from a database.
<a href="#">DROP SEQUENCE</a>	Drops a sequence from a database.
<a href="#">DROP SYNONYM</a>	Drops a synonym from a database.
<a href="#">DROP TABLE</a>	Drops a table from a database.
<a href="#">DROP TRIGGER</a>	Drops a trigger from a database.
<a href="#">DROP VIEW</a>	Drops a view from a database.
<a href="#">DROP FUNCTION</a>	Drops a function from a database.

## Data Manipulation (DML) Statements

These are the [data manipulation statements](#):

Statement	Description
<a href="#">CALL PROCEDURE</a>	Calls a stored procedure.
<a href="#">DELETE</a>	Deletes records from a table.
<a href="#">INSERT</a>	Inserts records into a table.
<a href="#">SELECT</a>	Selects records.
<a href="#">UPDATE TABLE</a>	Updates values in a table.

## Session Control Statements

These are the [session control statements](#):

Statement	Description
<a href="#">SET ROLE</a>	Sets the current role for the current SQL context of a session.
<a href="#">SET SCHEMA</a>	Sets the default schema for a connection's session.



For access to the source code for the Community Edition of Splice Machine, visit [our open source GitHub repository](#).

# ALTER TABLE

The `ALTER TABLE` statement allows you to modify a table in a variety of ways, including adding and dropping columns and constraints from the table.



In this release, you **cannot** use `ALTER TABLE` to:

- » add a primary key
- » drop a foreign key constraint

## Syntax

```
ALTER TABLE table-Name
{
  ADD COLUMN column-definition |
  ADD CONSTRAINT clause |
  DROP [ COLUMN ] column-name
  DROP { UNIQUE constraint-name |
        CHECK constraint-name
  }
  ALTER [ COLUMN ] column-alteration
}
```

### *column-definition*

```
Simple-column-name [ DataType ]
[ Column-level-constraint ]*
[ [ WITH ] DEFAULT DefaultConstantExpression
  | generation-clause
]
```

The syntax for the *column-definition* for a new column is a subset of the syntax for a column in a [CREATE TABLE](#) statement.

The *DataType* can be omitted only if you specify a *generation-clause*. If you omit the *DataType*, the type of the generated column is the type of the *generation-clause*. If you specify both a *DataType* and a *generation-clause*, the type of the *generation-clause* must be assignable to *DataType*.

### *column-alteration*

```
column-Name SET DATA TYPE VARCHAR(integer) |
column-name SET INCREMENT BY integer-constant |
column-name RESTART WITH integer-constant |
column-name [ NOT ] NULL |
column-name [ WITH | SET ] DEFAULT default-value |
column-name DROP DEFAULT
```

In the column-alteration, `SET INCREMENT BY integer-constant` specifies the interval between consecutive values of the identity column. The next value to be generated for the identity column will be determined from the last assigned value with the increment applied. The column must already be defined with the `IDENTITY` attribute.

`RESTART WITH integer-constant` specifies the next value to be generated for the identity column. `RESTART WITH` is useful for a table that has an identity column that was defined as `GENERATED BY DEFAULT` and that has a unique key defined on that identity column.

Because `GENERATED BY DEFAULT` allows both manual inserts and system generated values, it is possible that manually inserted values can conflict with system generated values. To work around such conflicts, use the `RESTART WITH` syntax to specify the next value that will be generated for the identity column.

Consider the following example, which involves a combination of automatically generated data and manually inserted data:

```
CREATE TABLE tauto(i INT GENERATED BY DEFAULT AS IDENTITY,
                   k INT)
CREATE UNIQUE INDEX tautoInd ON tauto(i)
INSERT INTO tauto(k) values 1,2;
```

The system will automatically generate values for the identity column. But now you need to manually insert some data into the identity column:

```
INSERT INTO tauto VALUES (3,3);
INSERT INTO tauto VALUES (4,4);
INSERT INTO tauto VALUES (5,5);
```

The identity column has used values 1 through 5 at this point. If you now want the system to generate a value, the system will generate a 3, which will result in a unique key exception because the value 3 has already been manually inserted. To compensate for the manual inserts, issue an `ALTER TABLE` statement for the identity column with `RESTART WITH 6`:

```
ALTER TABLE tauto ALTER COLUMN i RESTART WITH 6;
```

`ALTER TABLE` does not affect any view that references the table being altered. This includes views that have a wildcard asterisk (\*) in their `SELECT` list. You must drop and re-create those views if you wish them to return the new columns.

To change a column constraint to `NOT NULL`, there has to be a valid value for the column.

Splice Machine raises an error if you try to change the *Data Type* of a generated column to a type which is not assignable from the type of the *generation-clause*. Splice Machine also raises an error if you try to add a `DEFAULT` clause to a generated column.

## Usage

The `ALTER TABLE` statement allows you to:

- » add a column to a table
- » add a constraint to a table

- » drop a column from a table
- » drop an existing constraint from a table\*
- » increase the width of a `VARCHAR` column
- » change the increment value and start value of the identity column
- » change the nullability constraint for a column
- » change the default value for a column

## Adding columns

The syntax for the [column-definition](#) for a new column is almost the same as for a column in a `CREATE TABLE` statement. This syntax allows a column constraint to be placed on the new column within the `ALTER TABLE ADD COLUMN` statement. However, a column with a `NOT NULL` constraint can be added to an existing table if you give a default value; otherwise, an exception is thrown when the `ALTER TABLE` statement is executed.

**NOTE:** If a table has an `UPDATE` trigger without an explicit column list, adding a column to that table in effect adds that column to the implicit update column list upon which the trigger is defined, and all references to transition variables are invalidated so that they pick up the new column.

## Adding constraints

`ALTER TABLE ADD CONSTRAINT` adds a table-level constraint to an existing table.



The `ALTER TABLE ADD CONSTRAINT` statement is not currently taking currently running transactions into account, and thus can fail to add the constraint. This issue will be resolved in a future release.

You can reliably add constraints when using the `CREATE TABLE` statement.

The following limitations exist on adding a constraint to an existing table:

- » When adding a check constraint to an existing table, Splice Machine checks the table to make sure existing rows satisfy the constraint. If any row is invalid, Splice Machine throws a statement exception and the constraint is not added.

For information on the syntax of constraints, see [CONSTRAINT](#) clause. Use the syntax for table-level constraint when adding a constraint with the `ADD TABLE ADD CONSTRAINT` syntax.

## Dropping columns

`ALTER TABLE DROP COLUMN` allows you to drop a column from a table.

The keyword `COLUMN` is optional.

You may not drop the last (only) column in a table.

## Modifying columns

The [column-alteration](#) allows you to alter the named column in the following ways:

- » Increasing the width of an existing `VARCHAR` column. `CHARACTER VARYING` or `CHAR VARYING` can be used as synonyms for the `VARCHAR` keyword.

To increase the width of a column of these types, specify the data type and new size after the column name.

You are not allowed to decrease the width or to change the data type. You are not allowed to increase the width of a column that is part of a primary or unique key referenced by a foreign key constraint or that is part of a foreign key constraint.

- » Specifying the interval between consecutive values of the identity column.

To set an interval between consecutive values of the identity column, specify the integer-constant. You must previously define the column with the `IDENTITY` attribute (SQLSTATE 42837). If there are existing rows in the table, the values in the column for which the `SET INCREMENT` default was added do not change.

- » Modifying the nullability constraint of a column.

You can add the `NOT NULL` constraint to an existing column; however, you cannot do so if there are `NULL` values for the column in the table.

You can remove the `NOT NULL` constraint from an existing column; however, you cannot do so if the column is used in a `PRIMARY KEY` constraint.

- » Changing the default value for a column.

You can use `DEFAULT default-value` to change a column default. To disable a previously set default, use `DROP DEFAULT` (alternatively, you can specify `NULL` as the default-value).

## Setting defaults

You can specify a default value for a new column. A default value is the value that is inserted into a column if no other value is specified. If not explicitly specified, the default value of a column is `NULL`. If you add a default to a new column, existing rows in the table gain the default value in the new column.

For more information about defaults, see [CREATE TABLE](#) statement.

An `ALTER TABLE` statement causes all statements that are dependent on the table being altered to be recompiled before their next execution.

## Examples

This section provides examples of using the `ALTER TABLE` statement.



## Example 1: Adding Columns to a Table

In this example, we create a new table, and then use `ALTER TABLE` statements to add three columns that we have decided to include:

```
splice> CREATE TABLE PlayerTrades (
  ID INT NOT NULL,
  PlayerName VARCHAR(32),
  Position CHAR(2),
  OldTeam VARCHAR(32),
  NewTeam VARCHAR(32) );
0 rows inserted/updated/deleted

splice> ALTER TABLE PlayerTrades ADD COLUMN Updated TIMESTAMP;
0 rows inserted/updated/deleted

splice> ALTER TABLE PlayerTrades ADD COLUMN TradeDate DATE;
0 rows inserted/updated/deleted

splice> ALTER TABLE PlayerTrades ADD COLUMN Years INT;
0 rows inserted/updated/deleted

splice> INSERT INTO PlayerTrades VALUES( 1, 'Greinke', 'SP', 'Dodgers', 'Giants', CU
RRENT_TIMESTAMP, CURRENT_DATE);
1 row inserted/updated/deleted

splice> DESCRIBE PlayerTrades;
COLUMN_NAME      | TYPE_NAME | DEC& | NUM& | COLUM& | COLUMN_DEF | CHAR_OCTE& | IS_NULL&
-----
ID                | INTEGER  | 0    | 10   | 10     | NULL       | NULL       | NO
PLAYERNAME        | VARCHAR  | NULL | NULL | 32     | NULL       | 64         | YES
POSITION          | CHAR     | NULL | NULL | 2      | NULL       | 4          | YES
OLDTEAM           | VARCHAR  | NULL | NULL | 32     | NULL       | 64         | YES
NEWTEAM           | VARCHAR  | NULL | NULL | 32     | NULL       | 64         | YES
UPDATED           | TIMESTAMP | 9    | 10   | 29     | NULL       | NULL       | YES
TRADEDATE         | DATE     | 0    | 10   | 10     | NULL       | NULL       | YES
YEARS             | INTEGER  | 0    | 10   | 10     | NULL       | NULL       | YES

8 rows selected
```

## Example 2: Altering Columns

In this example, we use `ALTER TABLE` to alter columns in various ways:

- » specify that the `Updated` column cannot be `NULL`
- » set the default value for `Years` to 3
- » set the default value for `NewTeam` to 'Giants'

```
splice> ALTER TABLE PlayerTrades ALTER COLUMN Updated NOT NULL;
0 rows inserted/updated/deleted

splice> ALTER TABLE PlayerTrades ALTER COLUMN Years DEFAULT 3;
0 rows inserted/updated/deleted

splice> ALTER TABLE PlayerTrades ALTER COLUMN NewTeam DEFAULT 'Giants';
0 rows inserted/updated/deleted

splice> DESCRIBE PlayerTrades;
COLUMN_NAME      | TYPE_NAME | DEC | NUM | COLUMN_DEF | COLUMN_DEF | CHAR_OCTET_LENGTH | IS_NULLABLE
-----
ID                | INTEGER  | 0   | 10  | 10         | NULL       | NULL              | NO
PLAYERNAME       | VARCHAR  | NULL | NULL | 32        | NULL       | 64                | YES
POSITION         | CHAR     | NULL | NULL | 2         | NULL       | 4                 | YES
OLDTEAM          | VARCHAR  | NULL | NULL | 32        | NULL       | 64                | YES
NEWTEAM          | VARCHAR  | NULL | NULL | 32        | 'Giants'   | 64                | YES
UPDATED          | TIMESTAMP | 9   | 10  | 29        | NULL       | NULL              | NO
TRADEDATE        | DATE     | 0   | 10  | 10        | NULL       | NULL              | YES
YEARS            | INTEGER  | 0   | 10  | 10        | 3          | NULL              | YES

7 rows selected
```

### Example 3: Dropping a column

This example drops the `Years` column from our table, and then drops the default associated with `NewTeam`:

```
splice> ALTER TABLE PlayerTrades DROP COLUMN Years;
0 rows inserted/updated/deleted

splice> ALTER TABLE PlayerTrades ALTER COLUMN NewTeam DROP DEFAULT;
0 rows inserted/updated/deleted

splice> DESCRIBE PlayerTrades;
COLUMN_NAME      | TYPE_NAME | DEC | NUM | COLUMN_DEF | COLUMN_DEF | CHAR_OCTET_LENGTH | IS_NULLABLE
-----
ID                | INTEGER  | 0   | 10  | 10         | NULL       | NULL              | NO
PLAYERNAME       | VARCHAR  | NULL | NULL | 32        | NULL       | 64                | YES
POSITION         | CHAR     | NULL | NULL | 2         | NULL       | 4                 | YES
OLDTEAM          | VARCHAR  | NULL | NULL | 32        | NULL       | 64                | YES
NEWTEAM          | VARCHAR  | NULL | NULL | 32        | NULL       | 64                | YES
UPDATED          | TIMESTAMP | 9   | 10  | 29        | NULL       | NULL              | NO
TRADEDATE        | DATE     | 0   | 10  | 10        | NULL       | NULL              | YES

7 rows selected
```

### Example 4: Changing Varchar Column Width

This example changes the width of one of our `VARCHAR` columns:

```
splice> ALTER TABLE PlayerTrades ALTER COLUMN PlayerName SET DATA TYPE VARCHAR(40);
0 rows inserted/updated/deleted
```

```
splice> DESCRIBE PlayerTrades;
```

COLUMN_NAME	TYPE_NAME	DEC	NUM	COLUM	COLUMN_DEF	CHAR_OCTE	IS_NULL
ID	INTEGER	0	10	10	NULL	NULL	NO
PLAYERNAME	VARCHAR	NULL	NULL	40	NULL	80	YES
POSITION	CHAR	NULL	NULL	2	NULL	4	YES
OLDTEAM	VARCHAR	NULL	NULL	32	NULL	64	YES
NEWTEAM	VARCHAR	NULL	NULL	32	NULL	64	YES
UPDATED	TIMESTAMP	9	10	29	NULL	NULL	NO
TRADEDATE	DATE	0	10	10	NULL	NULL	YES

```
7 rows selected
```

## Example 5: Changing Increment Value

This example shows creating a table with an identity column, and then changing the increment for that column:

```
splice> CREATE TABLE NewPlayers(
  newID INT NOT NULL GENERATED ALWAYS AS IDENTITY PRIMARY KEY,
  PlayerName);
0 rows inserted/updated/deleted
```

```
splice> ALTER TABLE NewPlayers ALTER COLUMN newID SET INCREMENT BY 10;
0 rows inserted/updated/deleted
```

```
splice> INSERT INTO NewPlayers(PlayerName) ('Greinke'),('Cespedes');
2 rows inserted/updated/deleted
```

```
splice> SELECT * FROM NewPlayers;
NEWID      | PLAYERNAME
```

1	Greinke
11	Cespedes

```
2 rows selected
```

## See Also

- » [CONSTRAINT](#) clause
- » [CREATE TABLE](#) statement
- » [Foreign Keys](#) in the *Developer's Guide*.
- » [Triggers](#) in the *Developer's Guide*.

## CALL (Procedure)

The `CALL (PROCEDURE)` statement is used to call stored procedures.

When you call a stored procedure, the default schema and role are the same as were in effect when the procedure was created.

### Syntax

```
CALL procedure-Name (  
    [ expression [, expression]* ]  
)
```

*procedure-Name*

The name of the procedure that you are calling.

*expression(s)*

Arguments passed to the procedure.

### Example

The following example depends on a fictionalized java class. For functional examples of using `CREATE PROCEDURE`, please see the [Using Functions and Stored Procedures](#) section in our *Developer's Guide*.

```
splice> CREATE PROCEDURE SALES.TOTAL_REVENUE(IN S_MONTH INTEGER,  
    IN S_YEAR INTEGER, OUT TOTAL DECIMAL(10,2))  
    PARAMETER STYLE JAVA  
    READS SQL DATA LANGUAGE JAVA EXTERNAL NAME  
        'com.example.sales.calculateRevenueByMonth';  
splice> CALL SALES.TOTAL_REVENUE(?,?,?);
```

### See Also

» [CREATE PROCEDURE](#) statement

## column-definition

### Simple-column-Name

```
[ DataType ]
[ Column-level-constraint ]*
[ [ WITH ] DEFAULT DefaultConstantExpression
  | generated-column-spec
  | generation-clause
]
[ Column-level-constraint ]*
```

#### *DataType*

Must be specified unless you specify a generation-clause, in which case the type of the generated column is that of the generation-clause.

If you specify both a *DataType* and a *generation-clause*, the type of the *generation-clause* must be assignable to *DataType*

#### *Column-level-constraint*

See the [CONSTRAINT clause](#) documentation.

#### *DefaultConstantExpression*

For the definition of a default value, a DefaultConstantExpression is an expression that does not refer to any table. It can include constants, date-time special registers, current schemas, and null:

```
DefaultConstantExpression:
  NULL
  | CURRENT { SCHEMA | SQLID }
  | DATE
  | TIME
  | TIMESTAMP
  | CURRENT DATE | CURRENT_DATE
  | CURRENT TIME | CURRENT_TIME
  | CURRENT TIMESTAMP | CURRENT_TIMESTAMP
  | literal
```

The values in a DefaultConstantExpression must be compatible in type with the column, but a DefaultConstantExpression has the following additional type restrictions:

- » If you specify `CURRENT SCHEMA` or `CURRENT SQLID`, the column must be a character column whose length is at least 128.
- » If the column is an integer type, the default value must be an integer literal.
- » If the column is a decimal type, the scale and precision of the default value must be within those of the column.

## Example

This example creates a table and uses two column definitions.

```
splice> CREATE TABLE myTable (ID INT NOT NULL, NAME VARCHAR(32) );  
0 rows inserted/updated/deleted
```

## See Also

» [CONSTRAINT](#) clause

# CREATE EXTERNAL TABLE

A `CREATE EXTERNAL TABLE` statement creates a table in Splice Machine that you can use to query data that is stored externally in a flat file, such as a file in Parquet, ORC, or plain text format. External tables are largely used as a convenient means of moving data into and out of your database.

You can query external tables just as you would a regular Splice Machine table; however, you cannot perform any DML operations on an external table, once it has been created. That also means that you cannot create an index on an external table.

If the schema of the external file that you are querying is modified outside of Splice, you need to manually refresh the Splice Machine table by calling the [REFRESH EXTERNAL TABLE](#) built-in system procedure.

If a qualified table name is specified, the schema name cannot begin with `SYS`.

## Syntax

```
CREATE EXTERNAL TABLE table-Name
{
  ( column-definition* )
  [ COMPRESSED WITH compression-format ]
  [ PARTITIONED BY ( column-name ) ] }
  [ ROW FORMAT DELIMITED
    [ FIELDS TERMINATED BY char [ESCAPED BY char] ]
    [ LINES TERMINATED BY char ]
  ]
  STORED AS file-format LOCATION location
}
```

### *table-Name*

The name to assign to the new table.

### *compression-format*

The compression algorithm used to compress the flat file source of this external table. You can specify one of the following values:

» ZLIB

» SNAPPY

If you don't specify a compression format, the default is uncompressed. You cannot specify a *compression-format* when using the `TEXTFILE` *file-format*; doing so generates an error.

### *column-definition*

A column definition.

The maximum number of columns allowed in a table is 100000.

### *column-name*

The name of a column.

#### *char*

A single character used as a delimiter or escape character. Enclose this character in single quotes; for example, `' '`.

To specify a special character that includes the backslash character, you must escape the backslash character itself. For example:

- » `\\` to indicate a backslash character
- » `\n` to indicate a newline character
- » `\t` to indicate a tab character

#### *file-format*

The format of the flat file source of this external table. This is currently one of these values:

- » `ORC` is a columnar storage format
- » `PARQUET` is a columnar storage format
- » `Avro` is a data serialization system
- » `TEXTFILE` is a plain text file

#### *location*

The location at which the file is stored.

## Usage Notes

Here are some notes about using external tables:

- » If the data types in the table schema you specify do not match the schema of the external file, an error occurs and the table is not created.
- » You cannot define indexes or constraints on external tables
- » The `ROW FORMAT` parameter is only applicable to plain text (`TextFile`) not supported for columnar storage format files (`ORC` or `PARQUET` files)
- » If you specify the location of a non-existent file when you create an external table, Splice Machine automatically creates an external file at that location.
- » `AVRO` external tables do not currently work with compressed files; any compression format you specify will be ignored.
- » Splice Machine isn't able to know when the schema of the file represented by an external table is updated; when this occurs, you need to update the external table in Splice Machine by calling the [`SYSCS\_UTIL.SYSCS\_REFRESH\_EXTERNAL\_TABLE`](#) built-in system procedure.
- » You cannot specify a *compression-format* when using the `TEXTFILE file-format`; doing so generates an error.



## Examples

This section presents examples of the `CREATE EXTERNAL TABLE` statement.

This example creates an external table for a `PARQUET` file:

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

This example creates an external table for an `AVRO` file:

```
splice> CREATE EXTERNAL TABLE myAvroTable(
        col1 INT, col2 VARCHAR(24))
        PARTITIONED BY (col1)
        STORED AS AVRO
        LOCATION '/users/myName/myAvroFile'
    );
0 rows inserted/updated/deleted
```

This example creates an external table for an `ORC` file and inserts data into it:

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

This example creates an external table for a plain text file:

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

This example creates an external table for a PARQUET file that was compressed with Snappy compression:

```
splice> CREATE EXTERNAL TABLE mySnappyParquetTable(
    coll INT, col2 VARCHAR(24))
    COMPRESSED WITH SNAPPY
    PARTITIONED BY (coll)
    STORED AS PARQUET
    LOCATION '/users/myName/mySnappyParquetFile'
);
0 rows inserted/updated/deleted
```

## See Also

- » [CREATE TABLE](#)
- » [PIN TABLE](#)
- » [DROP TABLE](#)
- » [REFRESH EXTERNAL TABLE](#)
- » [Foreign Keys](#)
- » [Triggers](#)

# CREATE FUNCTION

The `CREATE FUNCTION` statement allows you to create Java functions, which you can then use in expressions.

For details on how Splice Machine matches functions to Java methods, see [Argument matching](#).

## Syntax

```
CREATE FUNCTION functionName (
  [ functionName ]
  [, functionName ] *
)
RETURNS returnType [ functionElement ] *
```

*functionName*

[SQL Identifier](#)

If `schemaName` is not provided, then the current schema is the default schema. If a qualified procedure name is specified, the schema name cannot begin with `SYS`.

*functionParameter*

[ parameterName ] [DataType](#)

`parameterName` is an identifier that must be unique within the function's parameter names.

**NOTE:** Data-types such as `BLOB`, `CLOB`, `LONG VARCHAR` are not allowed as parameters in a `CREATE FUNCTION` statement.

*returnDataType*

[DataType](#) |  
[TableType](#)

*functionElement*

See the description of [Function Elements](#) in the next section.

## TableType

```
TABLE ( ColumnElement [,ColumnElement]* )
```

*ColumnElement*

A [SQL Identifier](#).

Table functions return `TableType` results. Currently, only Splice Machine-style table functions are supported, which are functions that return JDBC *ResultSet*s.

When values are extracted from a *ResultSet*, the data types of the values are coerced to match the data types declared in the `CREATE FUNCTION` statement. Here are a few coercion rules you should know about:

- » values that are too long are truncated to the maximum declared length
- » if a string value is returned in the *ResultSet* for a column of type `CHAR`, and the string is shorter than the column length, the string is padded with spaces

## Function Elements

```
{
  LANGUAGE { JAVA }
| DeterministicCharacteristic
| EXTERNAL NAME javaMethodName
| PARAMETER STYLE parameterStyle
| sqlStatementType
| nullInputAction
}
```

The function elements may appear in any order, but each type of element can only appear once.

These function elements are required:

- » *LANGUAGE*
- » *EXTERNAL NAME*
- » *PARAMETER STYLE*

### *LANGUAGE*

Only `JAVA` is accepted at this time. Splice Machine will call the function as a public static method in a Java class.

### *DeterministicCharacteristic*

```
DETERMINISTIC | NOT DETERMINISTIC
```

The default value is `NOT DETERMINISTIC`.

Specifying `DETERMINISTIC` indicates that the function always returns the same result, given the same input values. This allows Splice Machine to call the function with greater efficiency; however, specifying this for a function that is actually non-deterministic will have the opposite effect – efficiency of calls to the function will be reduced.

### *javaMethodName*

```
class_name.method_name
```

This is the name of the Java method to call when this function executes.

*parameterStyle*

```
JAVA | DERBY_JDBC_RESULT_SET
```

Only use `DERBY_JDBC_RESULT_SET` if this is a Splice Machine-style table function that returns a [TableType](#) result, and is mapped to a Java method that returns a JDBC *ResultSet*.

Otherwise, use `JAVA`-style parameters, which means that a parameter-passing convention is used that conforms to the Java language and SQL Routines specification. `INOUT` and `OUT` parameters are passed as single entry arrays to facilitate returning values. Result sets can be returned through additional parameters to the Java method of type `java.sql.ResultSet[]` that are passed single entry arrays.

Splice Machine does not support long column types such as `LONG VARCHAR` or `BLOB`; an error will occur if you try to use one of these long column types.

*sqlStatementType**CONTAINS SQL*

Indicates that SQL statements that neither read nor modify SQL data can be executed by the function. Statements that are not supported in any function return a different error.

*NO SQL*

Indicates that the function cannot execute any SQL statements

*READS SQL DATA*

Indicates that some SQL statements that do not modify SQL data can be included in the function. Statements that are not supported in any stored function return a different error. This is the default value.

*nullInputAction**RETURNS NULL ON NULL INPUT*

If any input argument is null, the function is not invoked, and the result is null.

*CALLED ON NULL INPUT*

This is the default value.

The function is invoked even if all input arguments are null, which means that the invoked function must test for null argument values. The result may be null or not null.

## Example of declaring a scalar function

For more complete examples of using `CREATE FUNCTION`, please see the [Using Functions and Stored Procedures](#) section of our *Developer's Guide*.

```
splice> CREATE FUNCTION TO_DEGREES( RADIANS DOUBLE )
  RETURNS DOUBLE
  PARAMETER STYLE JAVA
  NO SQL LANGUAGE JAVA
  EXTERNAL NAME 'java.lang.Math.toDegrees';

0 rows inserted/updated/deleted
```

## Example of declaring a table function

This example reads data from a mySql database and inserts it into a Splice Machine database.

We first implement a class that contains a public static method that connects to an external (foreign) database, uses a prepared statement to pull results from it, and returns those results as a JDBC `ResultSet`:

```
package splicemachine.example.vti;
import java.sql.*;
public class EmployeeTable{
    public static ResultSet read() throws SQLException {
        Connection conn DriverManager.getConnection(
            "jdbc:mysql://localhost/hr?user=myName&password=myPswd" );
        PreparedStatement ps = conn.prepareStatement(
            "SELECT * FROM hrSchema.EmployeeTable" );
        return ps.executeQuery();
    }
}
```

Next we use the [CREATE FUNCTION](#) statement to declare a table function to read data from our external database and insert it into our Splice Machine database:

```
CREATE FUNCTION externalEmployees()
  RETURNS TABLE
  (
    employeeId      INT,
    lastName        VARCHAR( 50 ),
    firstName       VARCHAR( 50 ),
    birthday        DATE
  )
  LANGUAGE JAVA
  PARAMETER STYLE SPLICE_JDBC_RESULT_SET READS SQL DATA EXTERNAL NAME 'com.splicemachine.example.vti.readEmployees';
```

Now we're ready to invoke our table function to read data from the external database and insert it into a table in our Splice Machine database.

To invoke a table function, you must wrap it in a `TABLE` constructor in the `FROM` list of a query. For example, we could insert employee data from that database into a table named `employees` in our Splice Machine database:

```
INSERT INTO employees
  SELECT myExtTbl.*
  FROM TABLE (externalEmployees() ) myExtTbl;
```

**NOTE:** You **MUST** specify the table alias when using a virtual table; for example, myExtTbl in the above example.

## See Also

- » [CREATE\\_\\_PROCEDURE](#) statement
- » [CURRENT\\_\\_USER](#) function
- » [Data Types](#)
- » [DROP FUNCTION](#) statement
- » [Schema Name](#)
- » [SQL Identifier](#)
- » [SESSION\\_\\_USER](#) function
- » [USER](#) function

# CREATE INDEX

A `CREATE INDEX` statement creates an index on a table. Indexes can be on one or more columns in the table.

## Syntax

```
CREATE [UNIQUE] INDEX indexName
ON tableName (
  simpleColumnName
  [ ASC | DESC ]
  [ , simpleColumnName [ ASC | DESC ] ] *
)
[ SPLITKEYS splitKeyInfo HFILE hfileLocation ]
```

### *indexName*

An identifier, the length of which cannot exceed 128 characters.

### *tableName*

A table name, which can optionally be qualified by a schema name.

### *simpleColumnName*

A simple column name.

You cannot use the same column name more than once in a single `CREATE INDEX` statement. Note, however, that a single column name can be used in multiple indexes.

### *splitKeyInfo*

```
AUTO |
{ LOCATION filePath
  [ colDelimiter ]
  [ charDelimiter ]
  [ timestampFormat ]
  [ dateFormat ]
  [ timeFormat ]
}
```

Use the optional `SPLITKEYS` section to create indexes using HFile Bulk Loading, which is described in the [Using Bulk Hfile Indexing](#) section, below. Using bulk HFiles improves performance for large datasets, and is related to our [Bulk HFile Import procedure](#).

You can specify `AUTO` to have Splice Machine scan the data and determine the splits automatically. Or you can specify your own split keys in a CSV file; if you're using a CSV file, you can optionally include delimiter and format specifications, as described in the following parameter definitions. Each parameter name links to a fuller description of the possible parameter values, which are the similar to those used in our [Import Parameters Tutorial](#).

### *colDelimiter*

The character used to separate columns. You don't need to specify this if using the comma (,) character as your delimiter.



*charDelimiter*

The character is used to delimit strings in the imported data. You don't need to specify this if using the double-quote (") character as your delimiter.

*timeStampFormat*

The format of timestamps stored in the file. You don't need to specify this if no time columns in the file, or if the format of any timestamps in the file match the Java.sql.Timestamp default format, which is: "yyyy-MM-dd HH:mm:ss".

*dateFormat*

The format of datestamps stored in the file. You don't need to specify this if there are no date columns in the file, or if the format of any dates in the file match the pattern: "yyyy-MM-dd".

*timeFormat*

The format of time values stored in the file. You can set this to null if there are no time columns in the file, or if the format of any times in the file match pattern: "HH:mm:ss".

*hFileLocation*

The location (full path) in which the temporary HFiles will be created. These files will automatically be deleted after the index creation process completes. This parameter is required when specifying split keys.

## Usage

Splice Machine can use indexes to improve the performance of data manipulation statements. In addition, `UNIQUE` indexes provide a form of data integrity checking.

**Index names are unique within a schema.** (Some database systems allow different tables in a single schema to have indexes of the same name, but Splice Machine does not.) Both index and table are assumed to be in the same schema if a schema name is specified for one of the names, but not the other. If schema names are specified for both index and table, an exception will be thrown if the schema names are not the same. If no schema name is specified for either table or index, the current schema is used.

You cannot create an index that has the same index columns as an existing index; if you attempt to do so, Splice Machine issues a warning and does not create the index, as you can see in this example:

```
splice> CREATE INDEX idx1 ON myTable(id, eventType);
0 rows inserted/updated/deleted
splice> CREATE INDEX idx2 ON myTable(id, eventType);
WARNING 01504: The new index is a duplicate of an existing index: idx1.
splice> DROP INDEX idx2;
ERROR 42X65: Index 'idx2' does not exist.
```

By default, Splice Machine uses the ascending order of each column to create the index. Specifying `ASC` after the column name does not alter the default behavior. The `DESC` keyword after the column name causes Splice Machine to use descending order for the column to create the index. Using the descending order for a column can help improve the performance of queries that require the results in mixed sort order or descending order and for queries that select the minimum or maximum value of an indexed column.

If a qualified index name is specified, the schema name cannot begin with `SYS`.

## Using Bulk HFiles to Create an Index

Bulk HFile indexing improves performance when indexing very large datasets. The table you're indexing is temporarily converted into HFiles to take advantage of HBase bulk loading; once the indexing operation is complete, the temporary HFiles are automatically deleted. This is very similar to using HFile Bulk Loading for importing large datasets, which is described in our [Bulk HFile Import Tutorial](#).

You can have Splice Machine automatically determine the splits by scanning the data, or you can define the split keys in a CSV file. In the following example, we use our understanding of the `Orders` table to first create a CSV file named `ordersKey.csv` that contains the split keys we want, and then use the following `CREATE INDEX` statement to create the index:

```
CREATE INDEX o_Cust_Idx on Orders (
  o_custKey,
  o_orderKey
)
SPLITKEYS LOCATION '/tmp/ordersKey.csv'
COLDELIMITER '|'
HFILE LOCATION '/tmp/HFiles';
```

The `/tmp/ordersKey.csv` file specifies the index keys; it uses the `|` character as a column delimiter. The temporary HFiles are created in the `/tmp/HFiles` directory.

## Indexes and constraints

Unique and primary key constraints generate indexes that enforce or “back” the constraint (and are thus sometimes called *backing indexes*). If a column or set of columns has a `UNIQUE` or `PRIMARY KEY` constraint on it, you can not create an index on those columns.

Splice Machine has already created it for you with a system-generated name. System-generated names for indexes that back up constraints are easy to find by querying the system tables if you name your constraint. Adding a `PRIMARY KEY` or `UNIQUE` constraint when an existing `UNIQUE` index exists on the same set of columns will result in two physical indexes on the table for the same set of columns. One index is the original `UNIQUE` index and one is the backing index for the new constraint.

## Statement Dependency System

Prepared statements that involve `SELECT`, `INSERT`, `UPDATE`, and `DELETE` on the table referenced by the `CREATE INDEX` statement are invalidated when the index is created.

## Example

```
splice> CREATE TABLE myTable (ID INT NOT NULL, NAME VARCHAR(32) NOT NULL );
0 rows inserted/updated/deleted

splice> CREATE INDEX myIdx ON myTable(ID);
0 rows inserted/updated/deleted
```

## See Also

- » [DELETE](#) statement
- » [DROP INDEX](#) statement
- » [INSERT](#) statement
- » [SELECT](#) statement
- » [UPDATE](#) statement

# CREATE PROCEDURE

The `CREATE PROCEDURE` statement allows you to create Java procedures, which you can then call using the `CALL PROCEDURE` statement.

For details on how Splice Machine matches procedures to Java methods, see [Argument matching](#).

## Syntax

```
CREATE PROCEDURE procedureName (
  [ procedureParameter
  [, procedureParameter] ] *
)
  [ ProcedureElement ] *
```

*procedureName*

```
[ SQL Identifier ]
```

If `schemaName` is not provided, then the current schema is the default schema. If a qualified procedure name is specified, the schema name cannot begin with `SYS`.

*procedureParameter*

```
[ { IN | OUT | INOUT } ] [ parameterName ] DataType
```

`parameterName` is an identifier that must be unique within the procedure's parameter names.

By default, parameters are `IN` parameters unless you specify otherwise.

Data-types such as `BLOB`, `CLOB`, `LONG VARCHAR` are not allowed as parameters in a `CREATE PROCEDURE` statement.

**NOTE:** Also: At this time, Splice Machine will return only one `ResultSet` from a stored procedure.

*procedureElement*

See the description of [procedure Elements](#) in the next section.

## Procedure Elements

```
{
  LANGUAGE { JAVA }
| DeterministicCharacteristic
| EXTERNAL NAME javaMethodName
| PARAMETER STYLE parameterStyle
| sqlStatementType
}
```

The procedure elements may appear in any order, but each type of element can only appear once. These procedure elements are required:

- » *LANGUAGE*
- » *EXTERNAL NAME*
- » *PARAMETER STYLE*

### *LANGUAGE*

Only **JAVA** is accepted at this time. Splice Machine will call the procedure as a public static method in a Java class.

### *DeterministicCharacteristic*

```
DETERMINISTIC | NOT DETERMINISTIC
```

The default value is **NOT DETERMINISTIC**.

Specifying **DETERMINISTIC** indicates that the procedure always returns the same result, given the same input values. This allows Splice Machine to call the procedure with greater efficiency; however, specifying this for a procedure that is actually non-deterministic will have the opposite effect – efficiency of calls to the procedure will be reduced.

### *javaMethodName*

```
class_name.method_name
```

This is the name of the Java method to call when this procedure executes.

### *parameterStyle*

```
JAVA
```

Stored procedures use a parameter-passing convention is used that conforms to the Java language and SQL Routines specification. **INOUT** and **OUT** parameters are passed as single entry arrays to facilitate returning values. Result sets can be returned through additional parameters to the Java method of type `java.sql.ResultSet[]` that are passed single entry arrays.

Splice Machine does not support long column types such as **LONG VARCHAR** or **BLOB**; an error will occur if you try to use one of these long column types.

### *sqlStatementType*

**CONTAINS SQL**

Indicates that SQL statements that neither read nor modify SQL data can be executed by the procedure.

**NO SQL**

Indicates that the procedure cannot execute any SQL statements

**READS SQL DATA**

Indicates that some SQL statements that do not modify SQL data can be included in the procedure. This is the default value.

**MODIFIES SQL DATA**

Indicates that the procedure can execute any SQL statement.

## Example

The following example depends on a fictionalized java class. For functional examples of using `CREATE PROCEDURE`, please see the [Using Functions and Stored Procedures](#) section of our *Developer's Guide*.

```
splice> CREATE PROCEDURE SALES.TOTAL_REVENUE (
  IN S_MONTH INTEGER,
  IN S_YEAR INTEGER, OUT TOTAL DECIMAL(10,2))
  PARAMETER STYLE JAVA
  READS SQL DATA LANGUAGE
  JAVA EXTERNAL NAME 'com.example.sales.calculateRevenueByMonth';
0 rows inserted/updated/deleted
```

## See Also

- » [Argument matching](#)
- » [CREATE FUNCTION](#) statement
- » [CURRENT\\_USER](#) function
- » [Data Types](#)
- » [Schema Name](#)
- » [SQL Identifier](#)
- » [SESSION\\_USER](#) function
- » [USER](#) function

# CREATE ROLE

The `CREATE ROLE` statement allows you to create an SQL role. Only the database owner can create a role.

## Syntax

```
CREATE ROLE roleName
```

*roleName*

The name of an SQL role.

## Using

Before you issue a `CREATE ROLE` statement, verify that the `derby.database.sqlAuthorization` property is set to `TRUE`. The `derby.database.sqlAuthorization` property enables SQL authorization mode.

You cannot create a role name if there is already a user by that name. An attempt to create a role name that conflicts with an existing user name raises the *SQLException* X0Y68. If user names are not controlled by the database owner (or administrator), it may be a good idea to use a naming convention for roles to reduce the possibility of collision with user names.

Splice Machine tries to avoid name collision between user names and role names, but this is not always possible, because Splice Machine has a pluggable authorization architecture. For example, an externally defined user may exist who has never yet connected to the database, created any schema objects, or been granted any privileges. If Splice Machine knows about a user name, it will forbid creating a role with that name. Correspondingly, a user who has the same name as a role will not be allowed to connect. Splice Machine built-in users are checked for collision when a role is created.

A role name cannot start with the prefix `SYS` (after case normalization). The purpose of this restriction is to reserve a name space for system-defined roles at a later point. Use of the prefix `SYS` raises the *SQLException* 4293A.

You cannot create a role with the name `PUBLIC` (after case normalization). `PUBLIC` is a reserved authorization identifier. An attempt to create a role with the name `PUBLIC` raises *SQLException* 4251B.

## Examples

### Creating a Role

Here's a simple example of creating a role:

```
splice> CREATE ROLE statsEditor_role;
0 rows inserted/updated/deleted
```

## Examples of Invalid Role Names

Here are several examples of attempts to create a role using names that are reserved and cannot be used as role names. Each of these generates an error:

```
splice> CREATE ROLE public;  
splice> CREATE ROLE "PUBLIC";  
splice> CREATE ROLE sysrole;
```

## See Also

- » [DROP\\_ROLE](#) statement
- » [GRANT](#) statement
- » [REVOKE](#) statement
- » [RoleName](#)
- » [SET\\_ROLE](#) statement
- » [SYSROLES](#) system table



# CREATE SCHEMA

The `CREATE SCHEMA` statement allows you to create a database schema, which is a way to logically group objects in a single collection and provide a unique name-space for those objects.

## Syntax

```
CREATE SCHEMA {  
    [ schemaName ]  
}
```

The `CREATE SCHEMA` statement is used to create a schema. A schema name cannot exceed 128 characters. Schema names must be unique within the database.

A schema name cannot start with the prefix `SYS` (after case normalization). Use of the prefix `SYS` raises a *SQLException*.

## CREATE SCHEMA examples

To create a schema for airline-related tables, use the following syntax:

```
splice> CREATE SCHEMA FLIGHTS;  
0 rows inserted/updated/deleted
```

To create a schema employee-related tables, use the following syntax:

```
splice> CREATE SCHEMA EMP;  
0 rows inserted/updated/deleted
```

To create a table called `availability` in the `EMP` and `FLIGHTS` schemas, use the following syntax:

```

splice> CREATE TABLE Flights.Availability(
    Flight_ID CHAR(6) NOT NULL,
    Segment_Number INT NOT NULL,
    Flight_Date DATE NOT NULL,
    Economy_Seats_Taken INT,
    Business_Seats_Taken INT,
    FirstClass_Seats_Taken INT,
    CONSTRAINT Flt_Avail_PK
    PRIMARY KEY (Flight_ID, Segment_Number, Flight_Date)
);
0 rows inserted/updated/deleted

splice> CREATE TABLE EMP.AVAILABILITY(
    Hotel_ID INT NOT NULL,
    Booking_Date DATE NOT NULL,
    Rooms_Taken INT,
    CONSTRAINT HotelAvail_PK PRIMARY KEY (Hotel_ID, Booking_Date)
);
0 rows inserted/updated/deleted

```

## See Also

- » [DROP SCHEMA](#) statement
- » [Schema Name](#)
- » [SET SCHEMA](#) statement

# CREATE SEQUENCE

The `CREATE SEQUENCE` statement creates a sequence generator, which is a mechanism for generating exact numeric values, one at a time.

## Syntax

```
CREATE SEQUENCE
  [ SQL Identifier ]
  [ sequenceElement ]*
```

The sequence name is composed of an optional *schemaName* and a *SQL Identifier*. If a *schemaName* is not provided, the current schema is the default schema. If a qualified sequence name is specified, the schema name cannot begin with `SYS.`

### *schemaName*

The name of the schema to which this sequence belongs. If you do not specify a schema name, the current schema is assumed.

You cannot use a schema name that begins with the `SYS.` prefix.

### *SQL Identifier*

The name of the sequence

### *sequenceElement*

```
{
  AS dataType
  | START WITH startValue
  | INCREMENT BY incrementValue
  | MAXVALUE maxValu | NO MAXVALUE
  | MINVALUE minValu | NO MINVALUE
  | CYCLE | NO CYCLE
}
```

### *dataType*

If specified, the *dataType* must be an integer type (`SMALLINT`, `INT`, or `BIGINT`). If not specified, the default data type is `INT`.

### *startValue*

If specified, this is a signed integer representing the first value returned by the sequence object. The `START` value must be a value less than or equal to the maximum and greater than or equal to the minimum value of the sequence object.

The default start value for a new ascending sequence object is the minimum value. The default start value for a descending sequence object is the maximum value.

### *incrementValue*

If specified, the `incrementValue` is a non-zero signed integer value that fits in a *DataType* value.

If this is not specified, the `INCREMENT` defaults to 1. If `incrementValue` is positive, the sequence numbers get larger over time; if it is negative, the sequence numbers get smaller over time.

#### *minValue*

If specified, `minValue` must be a signed integer that fits in a *Data Type* value.

If `minValue` is not specified, or if `NO MINVALUE` is specified, then `minValue` defaults to the smallest negative number that fits in a *Data Type* value.

#### *maxValue*

If specified, `maxValue` must be a signed integer that fits in a *Data Type* value.

If `maxValue` is not specified, or if `NO MAXVALUE` is specified, then `maxValue` defaults to the largest positive number that fits in a *Data Type* value.

Note that the `maxValue` must be greater than the `minValue`.

#### *CYCLE*

The `CYCLE` clause controls what happens when the sequence generator exhausts its range and wraps around.

If `CYCLE` is specified, the wraparound behavior is to reinitialize the sequence generator to its `START` value.

If `NO CYCLE` is specified, Splice Machine throws an exception when the generator wraps around. The default behavior is `NO CYCLE`.

To retrieve the next value from a sequence generator, use a [NEXT VALUE FOR expression](#).

## Usage Privileges

The owner of the schema where the sequence generator lives automatically gains the `USAGE` privilege on the sequence generator, and can grant this privilege to other users and roles. Only the database owner and the owner of the sequence generator can grant these `USAGE` privileges. The `USAGE` privilege cannot be revoked from the schema owner. See [GRANT statement](#) and [REVOKE statement](#) for more information.

## Performance

To boost performance and concurrency, Splice Machine pre-allocates ranges of upcoming values for sequences. The lengths of these ranges can be configured by adjusting the value of the `derby.language.sequence.preallocator` property.

## Examples

The following statement creates a sequence generator of type `INT`, with a start value of `-2147483648` (the smallest `INT` value). The value increases by 1, and the last legal value is the largest possible `INT`. If `NEXT VALUE FOR` is invoked on the generator after it reaches its maximum value, Splice Machine throws an exception.

```
splice> CREATE SEQUENCE order_id;  
0 rows inserted/updated/deleted
```

This example creates a player ID sequence that starts with the integer value 100:

```
splice> CREATE SEQUENCE PlayerID_seq  
  START WITH 100;  
0 rows inserted/updated/deleted
```

The following statement creates a sequence of type `BIGINT` with a start value of 3,000,000,000. The value increases by 1, and the last legal value is the largest possible `BIGINT`. If `NEXT VALUE FOR` is invoked on the generator after it reaches its maximum value, Splice Machine throws an exception.

```
splice> CREATE SEQUENCE order_entry_id  
  AS BIGINT  
  START WITH 3000000000;  
0 rows inserted/updated/deleted
```

## See Also

- » [DROP SEQUENCE](#) statement
- » [GRANT](#) statement
- » [Next Value For](#) expression
- » [REVOKE](#) statement
- » [Schema Name](#)
- » [SQL Identifier](#)

# CREATE SYNONYM

The `CREATE SYNONYM` statement allows you to create an alternate name for a table or a view.

## Syntax

```
CREATE SYNONYM ( tableName } );
```

### *synonymName*

An [SQLIdentifier](#), which can optionally include a schema name. This is the new name you want to create for the view or table.

### *viewName*

An [SQLIdentifier](#) that identifies the view for which you are creating a synonym.

### *tableName*

An [SQLIdentifier](#) that identifies the table for which you are creating a synonym.

## Usage

**NOTE:** Currently, you can only use a synonym instead of the original qualified table or view name in these statements: [DELETE](#).

Here are a few other important notes about using synonyms:

- » Synonyms share the same name space as tables or views. You cannot create a synonym with the same name as a table that already exists in the same schema. Similarly, you cannot create a table or view with a name that matches a synonym already present.
- » You can create a synonym for a table or view that does not yet exist; however, you can only use the synonym if the table or view is present in your database.
- » You can create synonyms for other synonyms (nested synonyms); however, an error will occur if you attempt to create a synonym that results in a circular reference.
- » You cannot create synonyms in system schemas. Any schema that starts with `SYS` is a system schema.
- » You cannot define a synonym for a temporary table.

## Example

```
splice> CREATE SYNONYM Hitting FOR Batting;
0 rows inserted/updated/deleted

splice> SELECT ID, Games FROM Batting WHERE ID < 11;
ID      |GAMES
-----|-----
1       |150
2       |137
3       |100
4       |143
5       |149
6       |93
7       |133
8       |52
9       |115
10      |100

0 rows inserted/updated/deleted

splice> SELECT ID, Games FROM Hitting WHERE ID < 11;
ID      |GAMES
-----|-----
1       |150
2       |137
3       |100
4       |143
5       |149
6       |93
7       |133
8       |52
9       |115
10      |100

0 rows inserted/updated/deleted
```

## See Also

- » [DROP\\_SYNONYM](#) statement
- » [SHOW\\_SYNONYMS](#) command in our *Developer's Guide*.

# CREATE TABLE

A `CREATE TABLE` statement creates a table. Tables contain columns and constraints, rules to which data must conform. Table-level constraints specify a column or columns. Columns have a data type and can specify column constraints (column-level constraints).

The table owner and the database owner automatically gain the following privileges on the table and are able to grant these privileges to other users:

- » INSERT
- » SELECT
- » TRIGGER
- » UPDATE

These privileges cannot be revoked from the table and database owners.



Only database and schema owners can use the `CREATE TABLE` statement, which means that table creation privileges cannot be granted to others.

For information about constraints, see [CONSTRAINT clause](#).

You can specify a default value for a column. A default value is the value to be inserted into a column if no other value is specified. If not explicitly specified, the default value of a column is `NULL`.

If a qualified table name is specified, the schema name cannot begin with `SYS`.

**NOTE:** The [PIN TABLE](#) statements are documented separately in this section.

## Syntax

There are two different variants of the `CREATE TABLE` statement, depending on whether you are specifying the column definitions and constraints, or whether you are modeling the columns after the results of a query expression with the `CREATE TABLE AS` form:



```
CREATE TABLE table-Name
{
    ( { column-definition |
        Table-level constraint }
      [ , { column-definition } ] *
    )
|
  [ ( column-name ]* ) ]
AS query-expression [AS <name>]
WITH NO DATA
}
```

***table-Name***

The name to assign to the new table.

***column-definition***

A column definition.

The maximum number of columns allowed in a table is 100000.

***Table-level constraint***

A constraint that applies to the table.

***column-name***

A column definition.

***AS query-expression***

See the [CREATE TABLE AS](#) section below.

If this select list contains an expression, you must name the result of the expression. Refer to the final example at the bottom of this topic page.

***WITH NO DATA***

See the [CREATE TABLE AS](#) section below.

## CREATE TABLE ... AS ...

With this alternate form of the `CREATE TABLE` statement, the column names and/or the column data types can be specified by providing a query. The columns in the query result are used as a model for creating the columns in the new table.

**You cannot include** an `ORDER BY` clause in the query expression you use in the `CREATE TABLE AS` statement.

**NOTE:** If the select list contains an expression, **you must name the result of the expression.** Refer to the final example at the bottom of this topic page.

If no column names are specified for the new table, then all the columns in the result of the query expression are used to create same-named columns in the new table, of the corresponding data type(s). If one or more column names are specified for the new table, then the same number of columns must be present in the result of the query expression; the data types of those columns are used for the corresponding columns of the new table.

The `WITH NO DATA` clause specifies that the data rows which result from evaluating the query expression are not used; only the names and data types of the columns in the query result are used.

There is currently a known problem using the `CREATE TABLE AS` form of the `CREATE TABLE` statement .when the data to be inserted into the new table results from a `RIGHT OUTER JOIN` operation. For example, the following statement currently produces a table with all `NULL` values:

```
splice> CREATE TABLE t3 AS
  SELECT t1.a,t1.b,t2.c,t2.d
  FROM t1 RIGHT OUTER JOIN t2 ON t1.b = t2.c
  WITH DATA;
0 rows inserted/updated/deleted
```

There's a simple workaround for now: create the table without inserting the data, and then insert the data; for example:

```
splice> CREATE TABLE t3 AS
  SELECT t1.a,t1.b,t2.c,t2.d
  FROM t1 RIGHT OUTER JOIN t2 ON t1.b = t2.c
  WITH NO DATA;
0 rows inserted/updated/deleted

splice> INSERT INTO t3
  SELECT t1.a,t1.b,t2.c,t2.d
  FROM t1 RIGHT OUTER JOIN t2 ON t1.b = t2.c;
0 rows inserted/updated/deleted
```

## Examples

This section presents examples of both forms of the `CREATE TABLE` statement.

### CREATE TABLE

This example creates our `Players` table:

```
splice> CREATE TABLE Players(
  ID          SMALLINT NOT NULL PRIMARY KEY,
  Team        VARCHAR(64) NOT NULL,
  Name        VARCHAR(64) NOT NULL,
  Position    CHAR(2),
  DisplayName  VARCHAR(24),
  BirthDate   DATE
);
0 rows inserted/updated/deleted
```

This example includes a table-level primary key definition that includes two columns:

```
splice> CREATE TABLE HOTELAVAILABILITY (
    Hotel_ID INT NOT NULL,
    Booking_Date DATE NOT NULL,
    Rooms_Taken INT DEFAULT 0,
    PRIMARY KEY (Hotel_ID, Booking_Date ));
0 rows inserted/updated/deleted
```

This example assigns an identity column attribute with an initial value of 5 that increments by 5, and also includes a primary key constraint:

```
splice> CREATE TABLE PEOPLE (
    Person_ID INT NOT NULL GENERATED ALWAYS AS IDENTITY (START WITH 5, INCREMEN
T BY 5)
    CONSTRAINT People_PK PRIMARY KEY,
    Person VARCHAR(26) );
0 rows inserted/updated/deleted
```

**NOTE:** For more examples of CREATE TABLE statements using the various constraints, see [CONSTRAINT clause](#)

## CREATE TABLE AS

This example creates a new table that uses all of the columns (and their data types) from an existing table, but does not duplicate the data:

```
splice> CREATE TABLE NewPlayers
    AS SELECT *
        FROM Players WITH NO DATA;
0 rows inserted/updated/deleted
```

This example creates a new table that includes the data and uses only some of the columns from an existing table, and assigns new names for the columns:

```
splice> CREATE TABLE MorePlayers (ID, PlayerName, Born)
    AS SELECT ID, DisplayName, Birthdate
        FROM Players WITH DATA;
94 rows inserted/updated/deleted
```

This example creates a new table using unnamed expressions in the query and shows that the data types are the same for the corresponding columns in the newly created table:

```
splice> CREATE TABLE T3 (X,Y)
    AS SELECT 2*I AS COL1, 2.0*F AS COL2
        FROM T1 WITH NO DATA;
0 rows inserted/updated/deleted
```

## See Also

- » [ALTER TABLE](#) statement
- » [CREATE EXTERNAL TABLE](#) statement
- » [PIN TABLE](#) statement
- » [CONSTRAINT](#) clause
- » [DROP TABLE](#) statement
- » [UNPIN TABLE](#) statement
- » [Foreign Keys](#) in the *Developer's Guide*.
- » [Triggers](#) in the *Developer's Guide*.

# CREATE TEMPORARY TABLE

The `CREATE TEMPORARY TABLE` statement defines a temporary table for the current connection.

This statement is similar to the [DECLARE GLOBAL TEMPORARY TABLE](#) statements, but uses different syntax to provide compatibility with external business intelligence tools.

For general information and notes about using temporary tables, see the [Using Temporary Tables](#) topic in our *Developer's Guide*.

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

## Syntax

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

**NOTE:** Splice Machine generates a warning if you attempt to specify any other modifiers other than the `NOLOGGING` and `ON COMMIT PRESERVE ROWS` modifiers shown above.

### *LOCAL | GLOBAL*

These values are ignored by Splice Machine, and are in place simply to provide compatibility with external tools that use this syntax.

### *table-Name*

Names the temporary table.

### *Table-level constraint*

A constraint that is applied to this table, as described in the [Constraints](#) clause topic.

### *column-definition*

Specifies a column definition. See [column-definition](#) for more information.

**NOTE:** You cannot use `generated-column-spec` in column-definitions for temporary tables.

### *column-name*

A [SQL Identifier](#) that names a column in the table.

### *NOLOGGING*

If you specify this, operations against the temporary table will not be logged; otherwise, logging will take place as usual.

**ON COMMIT PRESERVE ROWS**

Specifies that the data in the temporary table is to be preserved until the session terminates.

## 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.

### Operational Limitations

Temporary tables have the following operational limitations:

- » exist only while a user session is alive
- » are not visible to other sessions or transactions
- » cannot be altered using the [RENAME COLUMN](#) 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 [SHOW TABLES](#) 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

## Examples

```
splice> CREATE GLOBAL TEMPORARY TABLE FirstAndLast (
    id INT NOT NULL PRIMARY KEY,
    firstName VARCHAR(8) NOT NULL,
    lastName VARCHAR(10) NOT NULL )
    ON COMMIT PRESERVE ROWS;
0 rows inserted/updated/deleted
```

## See Also

- » [DECLARE GLOBAL TEMPORARY TABLE](#) statement
- » [Using Temporary Tables](#) in the *Developer's Guide*.

# CREATE TRIGGER

A `CREATE TRIGGER` statement creates a trigger, which defines a set of actions that are executed when a database event known as the `triggering` event occurs on a specified table. The event can be a `INSERT`, `UPDATE`, or `DELETE` statement. When a trigger fires, the set of SQL statements that constitute the action are executed.

You can define any number of triggers for a single table, including multiple triggers on the same table for the same event. To define a trigger on a table, you must be the owner of the database, the owner of the table's schema, or have `TRIGGER` privileges on the table. You cannot define a trigger for any schema whose name begins with `SYS`.

The [Database Triggers](#) topic in our *Developer's Guide* provides additional information about database triggers.

## Syntax

```
CREATE TRIGGER TriggerName
{ AFTER | BEFORE }
{ INSERT | DELETE | UPDATE [ OF column-Name [, column-Name]* ] }
ON table-Name
[ ReferencingClause ]
[ FOR EACH { ROW | STATEMENT } ]
Triggered-SQL-statement
```

### *TriggerName*

The name to associate with the trigger.

### *AFTER | BEFORE*

Triggers are defined as either `Before` or `After` triggers.

`BEFORE` triggers fire before the statement's changes are applied and before any constraints have been applied.

`AFTER` triggers fire after all constraints have been satisfied and after the changes have been applied to the target table.

When a database event occurs that fires a trigger, Splice Machine performs actions in this order:

- » It fires `BEFORE` triggers.
- » It performs constraint checking (primary key, unique key, foreign key, check).
- » It performs the `INSERT`, `UPDATE`, `SELECT`, or `DELETE` operations.
- » It fires `AFTER` triggers.

When multiple triggers are defined for the same database event for the same table for the same trigger time (before or after), triggers are fired in the order in which they were created.

### *INSERT | DELETE | SELECT | UPDATE*

Defines which database event causes the trigger to fire. If you specify `UPDATE`, you can specify which column(s) cause the triggering event.

### *table-Name*



The name of the table for which the trigger is being defined.

### *ReferencingClause*

A means of referring to old/new data that is currently being changed by the database event that caused the trigger to fire. See the [Referencing Clause](#) section below.

### *FOR EACH {ROW | STATEMENT}*

A `FOR EACH ROW` triggered action executes once for each row that the triggering statement affects.

A `FOR EACH STATEMENT` trigger fires once per triggering event and regardless of whether any rows are modified by the insert, update, or delete event.

### *Triggered-SQL-Statement*

The statement that is executed when the trigger fires. The statement has the following restrictions:

- » It must not contain any dynamic (?) parameters.
- » It cannot create, alter, or drop any table.
- » It cannot add an index to or remove an index from any table.
- » It cannot add a trigger to or drop a trigger from any table.
- » It must not commit or roll back the current transaction or change the isolation level.
- » Before triggers cannot have `INSERT`, `UPDATE`, `SELECT`, or `DELETE` statements as their action.
- » Before triggers cannot call procedures that modify SQL data as their action.
- » The `NEW` variable of a `BEFORE` trigger cannot reference a generated column.

The statement can reference database objects other than the table upon which the trigger is declared. If any of these database objects is dropped, the trigger is invalidated. If the trigger cannot be successfully recompiled upon the next execution, the invocation throws an exception and the statement that caused it to fire will be rolled back.

## The Referencing Clause

Many triggered-SQL-statements need to refer to data that is currently being changed by the database event that caused them to fire. The triggered-SQL-statement might need to refer to the old (pre-change or *before*) values or to the new (post-change or *after*) values. You can refer to the data that is currently being changed by the database event that caused the trigger to fire.

Note that the referencing clause can designate only one new correlation or identifier and only one old correlation or identifier.

## Transition Variables in Row Triggers

Use the transition variables `OLD` and `NEW` with row triggers to refer to a single row before (`OLD`) or after (`NEW`) modification. For example:

```
REFERENCING OLD AS DELETEDROW;
```

You can then refer to this correlation name in the triggered-SQL-statement:

```
splice> DELETE FROM HotelAvailability WHERE hotel_id = DELETEDROW.hotel_id;
```

The OLD and NEW transition variables map to a *java.sql.ResultSet* with a single row.

INSERT row triggers cannot reference an OLD row.

**NOTE:** DELETE row triggers cannot reference a NEW row.

## Trigger Recursion

The maximum trigger recursion depth is 16.

## Examples

This section presents examples of creating triggers:

### A statement trigger:

```
splice> CREATE TRIGGER triggerName
  AFTER UPDATE
  ON TARGET_TABLE
  FOR EACH STATEMENT
    INSERT INTO AUDIT_TABLE VALUES (CURRENT_TIMESTAMP, 'TARGET_TABLE was update
d');
0 rows inserted/updated/deleted
```

### A statement trigger calling a custom stored procedure:

```
splice> CREATE TRIGGER triggerName
  AFTER UPDATE
  ON TARGET_TABLE
  FOR EACH STATEMENT
    CALL my_custom_stored_procedure('arg1', 'arg2');
0 rows inserted/updated/deleted
```

### A simple row trigger:

```
splice> CREATE TRIGGER triggerName
  AFTER UPDATE
  ON TARGET_TABLE
  FOR EACH ROW
    INSERT INTO AUDIT_TABLE VALUES (CURRENT_TIMESTAMP, 'TARGET_TABLE row was updat
ed');
0 rows inserted/updated/deleted
```

**A row trigger defined on a subset of columns:**

```
splice> CREATE TRIGGER triggerName
  AFTER UPDATE OF col1, col2
  ON TARGET_TABLE
  FOR EACH ROW
    INSERT INTO AUDIT_TABLE VALUES (CURRENT_TIMESTAMP, 'TARGET_TABLE col1 or col2
of row was updated');
0 rows inserted/updated/deleted
```

```
splice> CREATE TRIGGER UpdateSingles
  AFTER UPDATE OF Hits, Doubles, Triples, Homeruns
  ON Batting
  FOR EACH ROW
    UPDATE Batting Set Singles=(Hits-(Doubles+Triples+Homeruns));
0 rows insert/updated/deleted
```

**A row trigger defined on a subset of columns, referencing new and old values:**

```
splice> CREATE TRIGGER triggerName
  AFTER UPDATE OF col1, col2
  ON T
  REFERENCING OLD AS OLD_ROW NEW AS NEW_ROW
  FOR EACH ROW
    INSERT INTO AUDIT_TABLE VALUES (CURRENT_TIMESTAMP, 'TARGET_TABLE row was updat
ed', OLD_ROW.col1, NEW_ROW.col1);
0 rows insert/updated/deleted
```

**A row trigger defined on a subset of columns, referencing new and old values, calling custom stored procedure:**

```
splice> CREATE TRIGGER triggerName
  AFTER UPDATE OF col1, col2
  ON T
  REFERENCING OLD AS OLD_ROW NEW AS NEW_ROW
  FOR EACH ROW
    CALL my_custom_stored_procedure('arg1', 'arg2', OLD_ROW.col1, NEW_ROW.col1);
0 rows insert/updated/deleted
```

**See Also**

- » [Database Triggers](#) in the *Developer's Guide*
- » [DROP TRIGGER](#) statement
- » [WHERE](#) clause

# CREATE VIEW

Views are virtual tables formed by a query. A view is a dictionary object that you can use until you drop it. Views are not updatable.

If a qualified view name is specified, the schema name cannot begin with SYS.

## Syntax

```
CREATE VIEW view-Name
  [ ( Simple-column-Name ] * ) ]
AS ORDER BY clause ]
  [ RESULT OFFSET clause ]
  [ FETCH FIRST clause ]
```

A view definition can contain an optional view column list to explicitly name the columns in the view. If there is no column list, the view inherits the column names from the underlying query. All columns in a view must be uniquely named.

### *view-Name*

The name to assign to the view.

### *Simple-column-Name\**

An optional list of names to be used for columns of the view. If not given, the column names are deduced from the query.

The maximum number of columns in a view is 5000.

### *AS Query [ORDER BY clause]*

A SELECT or VALUES command that provides the columns and rows of the view.

### *result offset and fetch first clauses*

The [FETCH FIRST clause](#), which can be combined with the RESULT OFFSET clause, limits the number of rows added to the view.

## Examples

This example creates a view that shows the age of each player in our database:

```
splice> CREATE VIEW PlayerAges (Player, Team, Age)
  AS SELECT DisplayName, Team,
    INT( (Now - Birthdate) / 365.25) AS Age
  FROM Players;
```

0 rows inserted/updated/deleted

```
splice> SELECT * FROM PlayerAges WHERE Age > 30 ORDER BY Team, Age DESC;
```

PLAYER	TEAM	AGE
Robert Cohen	Cards	40
Jason Larrimore	Cards	37
David Janssen	Cards	36
Mitch Hassleman	Cards	35
Mitch Brandon	Cards	35
Tam Croonster	Cards	34
Alex Wister	Cards	34
Yuri Milleton	Cards	33
Jonathan Pearlman	Cards	33
Michael Rastono	Cards	32
Barry Morse	Cards	32
Carl Vanamos	Cards	32
Jan Bromley	Cards	31
Thomas Hillman	Giants	40
Mark Briste	Giants	38
Randy Varner	Giants	38
Jason Lilliput	Giants	38
Jalen Ardson	Giants	36
Sam Castleman	Giants	35
Alex Paramour	Giants	34
Jack Peepers	Giants	34
Norman Aikman	Giants	33
Craig McGawn	Giants	33
Kameron Fannais	Giants	33
Jason Martell	Giants	33
Harry Pennello	Giants	32
Jason Minman	Giants	32
Trevor Imhof	Giants	32
Steve Raster	Giants	32
Greg Brown	Giants	31
Alex Darba	Giants	31
Joseph Arkman	Giants	31
Tam Lassiter	Giants	31
Martin Cassman	Giants	31
Yuri Piamam	Giants	31

35 rows selected

## Statement Dependency System

View definitions are dependent on the tables and views referenced within the view definition. DML (data manipulation language) statements that contain view references depend on those views, as well as the objects in the view definitions that the views are dependent on. Statements that reference the view depend on indexes the view uses; which index a view uses can change from statement to statement based on how the query is optimized. For example, given:

```
splice> CREATE TABLE T1 (C1 DOUBLE PRECISION);
0 rows inserted/updated/deleted

splice>CREATE FUNCTION SIN (DATA DOUBLE)
  RETURNS DOUBLE
  EXTERNAL NAME 'java.lang.Math.sin'
  LANGUAGE JAVA PARAMETER STYLE JAVA;
0 rows inserted/updated/deleted

splice> CREATE VIEW V1 (C1) AS SELECT SIN(C1) FROM T1;
0 rows inserted/updated/deleted
```

The following `SELECT`:

```
SELECT * FROM V1;
```

Is dependent on view *V1*, table *T1*, and external scalar function *SIN*.

## See Also

- » [DROP VIEW](#) statement
- » [ORDER BY](#) clause

# DECLARE GLOBAL TEMPORARY TABLE

The `DECLARE GLOBAL TEMPORARY TABLE` statement defines a temporary table for the current connection.

This statement is similar to the [CREATE GLOBAL TEMPORARY TABLE](#) and `CREATE LOCAL TEMPORARY TABLE` statements, but uses different syntax to provide compatibility with external business intelligence tools.

For general information and notes about using temporary tables, see the [Using Temporary Tables](#) topic in our *Developer's Guide*.

## Syntax

```
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 `NOT LOGGED` and `ON COMMIT PRESERVE ROWS` modifiers shown above.

*table-Name*

Names the temporary table.

*column-definition*

Specifies a column definition. See [column-definition](#) for more information.

**NOTE:** You cannot use `generated-column-spec` in column-definitions for temporary tables.

*ON COMMIT PRESERVE ROWS*

Specifies that the data in the temporary table is to be preserved until the session terminates.

*NOT LOGGED*

If you specify this, operations against the temporary table will not be logged; otherwise, logging will take place as usual.

## 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.

## Operational Limitations

Temporary tables have the following operational limitations:

- » exist only while a user session is alive
- » are not visible to other sessions or transactions
- » cannot be altered using the [RENAME COLUMN](#) 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 [SHOW TABLES](#) 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

## Examples

```
splice> DECLARE GLOBAL TEMPORARY TABLE FirstAndLast (
    id INT NOT NULL PRIMARY KEY,
    firstName VARCHAR(8) NOT NULL,
    lastName VARCHAR(10) NOT NULL )
ON COMMIT PRESERVE ROWS
NOT LOGGED;
0 rows inserted/updated/deleted
```

## See Also

- » [CREATE TEMPORARY TABLE](#) statement
- » [Using Temporary Tables](#) in the *Developer's Guide*.



# DELETE

The `DELETE` statement deletes records from a table.

Our [Bulk HFile Delete](#) feature can be used to optimize deletion of large amounts of data.

## Syntax

```
{
  DELETE FROM correlation-Name
  [WHERE clause]
}
```

*table-Name*

The name of the table from which you want to delete records.

*correlation-Name*

The optional alias (alternate name) for the table.

*WHERE clause*

The clause that specifies which record(s) to select for deletion.

## Usage

The `DELETE` statement removes all rows identified by the table name and [WHERE](#) clause.

## Examples

```
splice> DELETE FROM Players WHERE Year(Birthdate) > 1990;
8 rows inserted/updated/deleted
```

## Using our Bulk HFile Delete Feature

Our Bulk Delete feature leverages HFile bulk deletion to significantly speed things up when you are deleting a lot of data; it does so by generating HFiles for the deletion and then bypasses the Splice Machine write pipeline and HBase write path when deleting the data.

You simply add a [splice-properties](#) hint that specifies where to generate the HFiles. 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.

## Statement Dependency System

A searched delete statement depends on the table being updated, all of its conglomerates (units of storage such as heaps or indexes), and any other table named in the `WHERE` clause. A [DROP INDEX](#) statement for the target table of a prepared searched delete statement invalidates the prepared searched delete statement.

A `CREATE INDEX` or `DROP INDEX` statement for the target table of a prepared positioned delete invalidates the prepared positioned delete statement.

## See Also

- » [CREATE INDEX](#) statement
- » [DROP INDEX](#) statement
- » [SELECT](#) statement
- » [WHERE](#) clause

## Interaction with the Dependency System

Splice Machine internally tracks the dependencies of prepared statements, which are SQL statements that are precompiled before being executed. Typically they are prepared (precompiled) once and executed multiple times.

Prepared statements depend on the dictionary objects and statements they reference. (Dictionary objects include tables, columns, constraints, indexes, and views, and triggers. Removing or modifying the dictionary objects or statements on which they depend invalidates them internally, which means that Splice Machine will automatically try to recompile the statement when you execute it. If the statement fails to recompile, the execution request fails. However, if you take some action to restore the broken dependency (such as restoring the missing table), you can execute the same prepared statement, because Splice Machine will recompile it automatically at the next execute request.

Statements depend on one another—an `UPDATE WHERE CURRENT` statement depends on the statement it references. Removing the statement on which it depends invalidates the `UPDATE WHERE CURRENT` statement.

In addition, prepared statements prevent execution of certain DDL statements if there are open results sets on them.

Manual pages for each statement detail what actions would invalidate that statement, if prepared. Here is an example using The Splice Machine command line interface:

```

splice> CREATE TABLE mytable (mycol INT);
0 rows inserted/updated/deleted
splice> INSERT INTO mytable VALUES (1), (2), (3);
3 rows inserted/updated/deleted -- this example uses the
ij command prepare, which prepares a statement
splice> prepare p1 AS 'INSERT INTO MyTable VALUES (4)';
-- p1 depends on mytable;
splice> execute p1;
1 row inserted/updated/deleted
-- Splice Machine executes it without recompiling
splice> CREATE INDEX i1 ON mytable(mycol);
0 rows inserted/updated/deleted
-- p1 is temporarily invalidated because of new index
splice> execute p1;
1 row inserted/updated/deleted
-- Splice Machine automatically recompiles and executes p1
splice> DROP TABLE mytable;
0 rows inserted/updated/deleted
-- Splice Machine permits you to drop table
-- because result set of p1 is closed
-- however, the statement p1 is temporarily invalidated
splice> CREATE TABLE mytable (mycol INT);
0 rows inserted/updated/deleted
splice> INSERT INTO mytable VALUES (1), (2), (3);
3 rows inserted/updated/deleted
splice> execute p1;
1 row inserted/updated/deleted
-- p1 is invalid, so Splice Machine tries to recompile it
-- before executing.
-- It is successful and executes.
splice> DROP TABLE mytable;
0 rows inserted/updated/deleted
-- statement p1 is now invalid
-- and this time the attempt to recompile it
-- upon execution will fail
splice> execute p1;
ERROR 42X05: Table/View 'MYTABLE' does not exist.

```

## See Also

- » [CREATE INDEX](#) statement
- » [CREATE TABLE](#) statement
- » [DROP TABLE](#) statement
- » [INSERT](#) statement
- » [Using the splice> prompt](#)

# DROP FUNCTION

The `DROP FUNCTION` statement drops a function from your database. Functions are added to the database with the [CREATE FUNCTION](#) statement.

## Syntax

```
DROP FUNCTION function-name
```

### *function-Name*

The name of the function that you want to drop from your database.

## Usage

Use this statement to drop a function from your database. It is valid only if there is exactly one function instance with the *function-name* in the schema. The specified function can have any number of parameters defined for it.

An error will occur in any of the following circumstances:

- » If no function with the indicated name exists in the named or implied schema (the error is SQLSTATE 42704)
- » If there is more than one specific instance of the function in the named or implied schema
- » If you try to drop a user-defined function that is invoked in the [generation-clause](#) of a generated column
- » If you try to drop a user-defined function that is invoked in a view

## Example

```
splice> DROP FUNCTION TO_DEGREES;  
0 rows inserted/updated/deleted
```

## See Also

- » [CREATE FUNCTION](#) statement

# DROP INDEX

The `DROP INDEX` statement removes the specified index.

## Syntax

```
DROP INDEX index-Name
```

*index-Name*

The name of the index that you want to drop from your database.

## Examples

```
splice> DROP INDEX myIdx;  
0 rows inserted/updated/deleted
```

## See Also

- » [CREATE INDEX](#) statement
- » [DELETE](#) statement
- » [INSERT](#) statement
- » [SELECT](#) statement
- » [UPDATE](#) statement

# DROP PROCEDURE

The `DROP PROCEDURE` statement drops a procedure from your database. Procedures are added to the database with the [CREATE PROCEDURE](#) statement.

## Syntax

```
DROP PROCEDURE procedure-name
```

*procedure-Name*

The name of the procedure that you want to drop from your database.

## Usage

Use this statement to drop a statement from your database. It is valid only if there is exactly one procedure instance with the *procedure-name* in the schema. The specified procedure can have any number of parameters defined for it.

An error will occur in any of the following circumstances:

- » If no procedure with the indicated name exists in the named or implied schema (the error is SQLSTATE 42704)
- » If there is more than one specific instance of the procedure in the named or implied schema
- » If you try to drop a user-defined procedure that is invoked in the [generation-clause](#) of a generated column
- » If you try to drop a user-defined procedure that is invoked in a view

## Example

```
splice> DROP PROCEDURE SALES.TOTAL_REVENUE;  
0 rows inserted/updated/deleted
```

## See Also

- » [Argument matching](#)
- » [CREATE PROCEDURE](#) statement
- » [CURRENT\\_USER](#) function
- » [Data Types](#)
- » [Schema Name](#)
- » [SQL Identifier](#)

» [SESSION\\_USER](#) function

» [USER](#) function



# DROP ROLE

The `DROP ROLE` statement allows you to drop a role from your database.

## Syntax

```
DROP ROLE roleName
```

*roleName*

The name of the role that you want to drop from your database.

## Usage

Dropping a role has the effect of removing the role from the database dictionary. This means that no session user can henceforth set that role (see [CURRENT\\_ROLE function](#)) will now have a `NULL CURRENT_ROLE`.

Dropping a role also has the effect of revoking that role from any user and role it has been granted to. See the [REVOKE statement](#) for information on how revoking a role may impact any dependent objects.

## Example

```
splice> DROP ROLE statsEditor_role;  
0 rows inserted/updated/deleted
```

## See Also

- » [CREATE\\_ROLE](#) statement
- » [GRANT](#) statement
- » [REVOKE](#) statement
- » [SET\\_ROLE](#) statement
- » [SYSROLES](#) system table

# DROP SCHEMA

The `DROP SCHEMA` statement drops a schema. The target schema must be empty for the drop to succeed.

Neither the *SPLICE* schema (the default user schema) nor the *SYS* schema can be dropped.

## Syntax

```
DROP SCHEMA schemaName RESTRICT
```

*schema*

The name of the schema that you want to drop from your database.

*RESTRICT*

This is **required**. It enforces the rule that the schema cannot be deleted from the database if there are any objects defined in the schema.

## Example

```
splice> DROP SCHEMA Baseball_Stats RESTRICT;  
0 rows inserted/updated/deleted
```

## See Also

- » [CREATE SCHEMA](#) statement
- » [Schema Name](#)
- » [SET SCHEMA](#) statement

# DROP SEQUENCE

The `DROP SEQUENCE` statement removes a sequence generator that was created using a [CREATE SEQUENCE statement](#).

## Syntax

```
DROP SEQUENCE [ schemaName "." ] SQL Identifier RESTRICT
```

### *schemaName*

The name of the schema to which this sequence belongs. If you do not specify a schema name, the current schema is assumed.

You cannot use a schema name that begins with the `SYS .` prefix.

### *SQL Identifier*

The name of the sequence.

### *RESTRICT*

This is **required**. It specifies that if a trigger or view references the sequence generator, Splice Machine will throw an exception.

## Usage

Dropping a sequence generator implicitly drops all `USAGE` privileges that reference it.

## Example

```
splice> DROP SEQUENCE PLAYERID_SEQ RESTRICT;  
0 rows inserted/updated/deleted
```

## See Also

» [CREATE SEQUENCE](#) statement

» [Schema Name](#)

# DROP SYNONYM

The `DROP SYNONYM` statement drops a synonym that was previously defined for a table or view.

## Syntax

```
DROP SYNONYM synonymName
```

*synonymName*

The name of the synonym that you want to drop from your database.

## Example

```
splice> DROP SYNONYM Hitting;  
0 rows inserted/updated/deleted
```

## See Also

- » [CREATE SYNONYM](#) statement
- » [SHOW SYNONYMS](#) command in our *Developer's Guide*.

# DROP TABLE

The `DROP TABLE` statement removes the specified table.

## Syntax

```
DROP TABLE [IF EXISTS] table-Name
```

*table-Name*

The name of the schema that you want to drop from your database.

## Statement dependency system

Indexes and constraints, constraints (primary, unique, check and references from the table being dropped) and triggers on the table are silently dropped.

Dropping a table invalidates statements that depend on the table. (Invalidating a statement causes it to be recompiled upon the next execution. See [Interaction with the dependency system](#).)

## Example

```
splice> DROP TABLE Salaries;  
0 rows inserted/updated/deleted
```

## See Also

- » [ALTER TABLE](#) statement
- » [CREATE TABLE](#) statement
- » [CONSTRAINT](#) clause

# DROP TRIGGER

The `DROP TRIGGER` statement removes the specified trigger.

## Syntax

```
DROP TRIGGER TriggerName
```

*TriggerName*

The name of the trigger that you want to drop from your database.

## Example

```
splice> DROP TRIGGER UpdateSingles;  
0 rows inserted/updated/deleted
```

## Statement dependency system

When a table is dropped, all triggers on that table are automatically dropped; this means that do not have to drop a table's triggers before dropping the table.

## See Also

- » [Database Triggers](#) in the *Developer's Guide*
- » [CREATE TRIGGER](#) statement

# DROP VIEW

The `DROP VIEW` statement drops the specified view.

## Syntax

```
DROP VIEW view-Name
```

*view-Name*

The name of the view that you want to drop from your database.

## Example

```
splice> DROP VIEW PlayerAges;  
0 rows inserted/updated/deleted
```

## Statement dependency system

Any statements referencing the view are invalidated on a `DROP VIEW` statement.

## See Also

- » [CREATE VIEW](#) statement
- » [ORDER BY](#) clause

## generated-column-spec

A generated column is one whose value is defined by an expression, typically involving values from other columns in the same table. The value of a generated column is automatically updated whenever there's a change in the value of any column upon which the expression depends.

```
[ GENERATED { ALWAYS | BY DEFAULT } AS IDENTITY
[ ( START WITH IntegerConstant
[ , INCREMENT BY IntegerConstant ] ) ] ] ]
```

### {ALWAYS | BY DEFAULT} AS IDENTITY

A table can have at most one identity column. See the [Identity Column Attributes](#) section below for more information about identity columns. Splice Machine supports two kinds of identity columns:

#### GENERATED ALWAYS

An identity column that is `GENERATED ALWAYS` will increment the default value on every insertion and will store the incremented value into the column. Unlike other defaults, you cannot insert a value directly into or update an identity column that is `GENERATED ALWAYS`. Instead, either specify the `DEFAULT` keyword when inserting into the identity column, or leave the identity column out of the insertion column list altogether. For example:

```
create table greetings
  (i int generated always as identity, ch char(50));
insert into greetings values (DEFAULT, 'hello');
insert into greetings(ch) values ('bonjour');
```

Automatically generated values in a `GENERATED ALWAYS` identity column are unique. Creating an identity column does not create an index on the column.

#### GENERATED BY DEFAULT

An identity column that is `GENERATED BY DEFAULT` will only increment and use the default value on insertions when no explicit value is given. Unlike `GENERATED ALWAYS` columns, you can specify a particular value in an insertion statement to be used instead of the generated default value.

To use the generated default, either specify the `DEFAULT` keyword when inserting into the identity column, or just leave the identity column out of the insertion column list. To specify a value, included it in the insertion statement. For example:

```
create table greetings
  (i int generated by default as identity, ch char(50));
  -- specify value "1":
insert into greetings values (1, 'hi');
  -- use generated default
insert into greetings values (DEFAULT, 'salut');
  -- use generated default
insert into greetings(ch) values ('bonjour');
```



Note that unlike a `GENERATED ALWAYS` column, a `GENERATED BY DEFAULT` column does not guarantee uniqueness. Thus, in the above example, the `hi` and `salut` rows will both have an identity value of “1”, because the generated column starts at 1 and the user-specified value was also 1. You can prevent duplication by specifying a `START WITH` value, and using a primary key or unique constraint on the identity column

#### *START WITH IntegerConstant*

The first identity value that Splice Machine should assign.

#### *INCREMENT BY IntegerConstant*

The amount by which to increment the identity value each time one is assigned.

## Identity Column Attributes

A table can have at most one identity column.

For `SMALLINT`, `INT`, and `BIGINT` columns with identity attributes, Splice Machine automatically assigns increasing integer values to the column. Identity column attributes behave like other defaults in that when an insert statement does not specify a value for the column, Splice Machine automatically provides the value. However, the value is not a constant; Splice Machine automatically increments the default value at insertion time.

The `IDENTITY` keyword can only be specified if the data type associated with the column is one of the following exact integer types.

» [`SMALLINT`](#)

» [`INT`](#)

» [`BIGINT`](#)

By default, the initial value of an identity column is 1, and the amount of the increment is 1. You can specify any positive integer value for both the initial value and the interval amount when you define the column with the key words `START WITH` and `INCREMENT BY`. Splice Machine increments the value with each insert. A value of 0 raises a statement exception.

The maximum and minimum values allowed in identity columns are determined by the data type of the column. Attempting to insert a value outside the range of values supported by the data type raises an exception. The following table shows the supported ranges.

Data Type	Maximum Value	Minimum Value
<code>SMALLINT</code>	32767 ( <i>java.lang.Short.MAX_VALUE</i> )	-32768 ( <i>java.lang.Short.MIN_VALUE</i> )
<code>INT</code>	2147483647 ( <i>java.lang.Integer.MAX_VALUE</i> )	-2147483648 ( <i>java.lang.Integer.MIN_VALUE</i> )
<code>BIGINT</code>	9223372036854775807 ( <i>java.lang.Long.MAX_VALUE</i> )	-9223372036854775808 ( <i>java.lang.Long.MIN_VALUE</i> )

Automatically generated values in an identity column are unique. Use a primary key or unique constraint on a column to guarantee uniqueness. Creating an identity column *does not* create an index on the column.

**NOTE:** Specify the schema, table, and column name using the same case as those names are stored in the system tables—that is, all upper case unless you used delimited identifiers when creating those database objects.

## Using Generated Columns

Splice Machine keeps track of the last increment value for a column in a cache. It also stores the value of what the next increment value will be for the column on disk in the `AUTOINCREMENTVALUE` column of the `SYS.SYSCOLUMNS` system table. Rolling back a transaction does not undo this value, and thus rolled-back transactions can leave “gaps” in the values automatically inserted into an identity column. Splice Machine behaves this way to avoid locking a row in `SYS.SYSCOLUMNS` for the duration of a transaction and keeping concurrency high.

When an insert happens within a triggered-SQL-statement, the value inserted by the triggered-SQL-statement into the identity column is available from *ConnectionInfo* only within the trigger code. The trigger code is also able to see the value inserted by the statement that caused the trigger to fire. However, the statement that caused the trigger to fire is not able to see the value inserted by the triggered-SQL-statement into the identity column. Likewise, triggers can be nested (or recursive).

An SQL statement can cause trigger T1 to fire. T1 in turn executes an SQL statement that causes trigger T2 to fire. If both T1 and T2 insert rows into a table that cause Splice Machine to insert into an identity column, trigger T1 cannot see the value caused by T2's insert, but T2 can see the value caused by T1's insert. Each nesting level can see increment values generated by itself and previous nesting levels, all the way to the top-level SQL statement that initiated the recursive triggers. You can only have 16 levels of trigger recursion.

## Examples

```

create table greetings
  (i int generated by default
   as identity (START WITH 2, INCREMENT BY 1),
  ch char(50));
-- specify value "1":
insert into greetings values (1, 'hi');
-- use generated default
insert into greetings values (DEFAULT, 'salut');
-- use generated default
insert into greetings(ch) values ('bonjour');
drop table if exists words;

splice> CREATE TABLE WORDS(WORD VARCHAR(20), UWORD GENERATED ALWAYS AS (UPPER(WOR
D)));
0 rows inserted/updated/deleted
splice> CREATE INDEX IDX_UWORD ON WORDS(UWORD);
0 rows inserted/updated/deleted
splice> INSERT INTO WORDS(WORD) VALUES 'chocolate', 'Coca-Cola', 'hamburger', 'carro
t';
4 rows inserted/updated/deleted
splice> select * from words;
WORD                |UWORD
-----
chocolate           |CHOCOLATE
Coca-Cola           |COCA-COLA
hamburger           |HAMBURGER
carrot              |CARROT

4 rows selected
splice> select upper(word) from words;
1
-----
CHOCOLATE
COCA-COLA
HAMBURGER
CARROT

4 rows selected
splice> drop table if exists t;
0 rows inserted/updated/deleted
WARNING 42Y55: 'DROP TABLE' cannot be performed on 'T' because it does not exist.
splice> CREATE TABLE T(COL1 INT, COL2 INT, COL3 GENERATED ALWAYS AS (COL1+COL2));
0 rows inserted/updated/deleted
splice> INSERT INTO T (COL1, COL2) VALUES (1,2), (3,4), (5,6);
3 rows inserted/updated/deleted
splice> select * from t;
COL1      |COL2      |COL3
-----
1          |2          |3
3          |4          |7
5          |6          |11

```

```
3 rows selected
splice> UPDATE T SET COL2 = 100 WHERE COL1 = 1;
1 row inserted/updated/deleted
splice> select * from t;
COL1      |COL2      |COL3
-----
```

```
1          |100        |101
3          |4          |7
5          |6          |11
```

```
3 rows selected
```

# generation-clause

## Syntax

```
GENERATED ALWAYS AS ( value-expression )
```

### *value-expression*

An [Expression](#) that resolves to a single value, with some limitations:

- » The *generation-clause* may reference other non-generated columns in the table, but it must not reference any generated column. The *generation-clause* must not reference a column in another table.
- » The *generation-clause* must not include subqueries.
- » The *generation-clause* may invoke user-coded functions, if the functions meet the requirements in the [User Function Restrictions](#) section below.

## User Function Restrictions

The *generation-clause* may invoke user-coded functions, if the functions meet the following requirements:

- » The functions must not read or write SQL data.
- » The functions must have been declared DETERMINISTIC.
- » The functions must not invoke any of the following possibly non-deterministic system functions:
  - » [SESSION\\_USER](#)

## Example

```
CREATE TABLE employee (
  employeeID      int,
  name            varchar( 50 ),
  caseInsensitiveName  GENERATED ALWAYS AS( UPPER( name ) )
);

CREATE INDEX caseInsensitiveEmployeeName
ON employee( caseInsensitiveName );
```

# GRANT

Use the `GRANT` statement to give privileges to a specific user or role, or to all users, to perform actions on database objects. You can also use the `GRANT` statement to grant a role to a user, to `PUBLIC`, or to another role.

The syntax that you use for the `GRANT` statement depends on whether you are granting privileges to a schema object or granting a role.



Only database and schema owners can use the `CREATE TABLE` statement, which means that table creation privileges cannot be granted to others, even with `GRANT ALL PRIVILEGES`.

## Syntax for Schemas

```
GRANT ALL PRIVILEGES | schema-privilege {, schema-privilege }*
ON [SCHEMA] schema-Name
TO grantees
```

### *schema-privilege*

```
DELETE
| INSERT
| REFERENCES [( column-identifier {, column-identifier}* )]
| SELECT [( column-identifier {, column-identifier}* )]
| TRIGGER
| UPDATE [( column-identifier {, column-identifier}* )]
```

See the [Privilege Types](#) section below for more information.



Column-level privileges are available only with a Splice Machine Enterprise license.

You cannot grant or revoke privileges at the `column-identifier` level with the Community version of Splice Machine.

To obtain a license for the Splice Machine Enterprise Edition, please [Contact Splice Machine Sales](#) today.

### *schema-Name*

The name of the schema to which you are granting access.

### *grantees*

The user(s) or role(s) to whom you are granting access. See the [About Grantees](#) section below for more information.

**NOTES:**

- » When you drop a schema from your database, all privileges associated with the schema are removed.
- » Table-level privileges override schema-level privileges.

## Syntax for Tables

```
GRANT ALL PRIVILEGES | table-privilege {, table-privilege }*    ON [TABLE] { tableName
| viewName }
    TO grantees
```

*table-privilege*

```
DELETE
| INSERT
| REFERENCES [( column-identifier {, column-identifier}* )]
| SELECT [( column-identifier {, column-identifier}* )]
| TRIGGER
| UPDATE [( column-identifier {, column-identifier}* )]
```

See the [Privilege Types](#) section below for more information.



Column-level privileges are available only with a Splice Machine Enterprise license.

You cannot grant or revoke privileges at the `column-identifier` level with the Community version of Splice Machine.

To obtain a license for the Splice Machine Enterprise Edition, please [Contact Splice Machine Sales](#) today.

*table-Name*

The name of the table to which you are granting access.

*view-Name*

The name of the view to which you are granting access.

*schema-Name*

The name of the schema to which you are granting access.

*grantees*

The user(s) or role(s) to whom you are granting access. See the [About Grantees](#) section below for more information.

**NOTES:**

- » When you drop a table from your database, all privileges associated with the table are removed.



» Table-level privileges override schema-level privileges.

## Syntax for Routines

```
GRANT EXECUTE
  ON { FUNCTION | PROCEDURE } {function-name | procedure-name}
  TO grantees
```

*function-name | procedure-name*

The name of the function or procedure to which you are granting access.

*grantees*

The user(s) or role(s) to whom you are granting access. See the [About Grantees](#) section below for more information.

## Syntax for User-defined Types

```
GRANT USAGE
  SQL Identifier
  TO grantees
```

*[schema-name.] SQL Identifier*

The type name is composed of an optional *schemaName* and a *SQL Identifier*. If a *schemaName* is not provided, the current schema is the default schema. If a qualified UDT name is specified, the schema name cannot begin with SYS.

*grantees*

The user(s) or role(s) to whom you are granting access. See the [About Grantees](#) section below for more information.

## Syntax for Roles

```
GRANT roleName
  { roleName } *
  TO grantees
```

*roleName*

The name to the role(s) to which you are granting access.

*grantees*

The user(s) or role(s) to whom you are granting access. See the [About Grantees](#) section below for more information.

Before you can grant a role to a user or to another role, you must create the role using the [CREATE ROLE statement](#). Only the database owner can grant a role.

A role A *contains* another role B if role B is granted to role A, or is contained in a role C granted to role A. Privileges granted to a contained role are inherited by the containing roles. So the set of privileges identified by role A is the union of the privileges granted to role A and the privileges granted to any contained roles of role A.

## About Grantees

A grantee can be one or more specific users, one or more specific roles, or all users (`PUBLIC`). Either the object owner or the database owner can grant privileges to a user or to a role. Only the database owner can grant a role to a user or to another role.

Here's the syntax:

```
{
  roleName | PUBLIC }
[, { roleName | PUBLIC } ] *
```

### *AuthorizationIdentifier*

An expression.

### *roleName*

The name of the role.

Either the object owner or the database owner can grant privileges to a user or to a role. Only the database owner can grant a role to a user or to another role.


### `PUBLIC`

Use the keyword `PUBLIC` to specify all users.

When `PUBLIC` is specified, the privileges or roles affect all current and future users.

The privileges granted to `PUBLIC` and to individual users or roles are independent privileges. For example, a `SELECT` privilege on table `t` is granted to both `PUBLIC` and to the authorization ID `harry`. If the `SELECT` privilege is later revoked from the authorization ID `harry`, Harry will still be able to access the table `t` through the `PUBLIC` privilege.

## Privilege Types

Privilege Type	Usage
ALL PRIVILEGES	<p>To grant all of the privileges to the user or role for the specified table. You can also grant one or more table privileges by specifying a privilege-list.</p> <div>  <p>Only database and schema owners can use the <code>CREATE TABLE</code> statement, which means that table creation privileges cannot be granted to others, even with <code>GRANT ALL PRIVILEGES</code>.</p> </div>
DELETE	To grant permission to delete rows from the specified table.
INSERT	To grant permission to insert rows into the specified table.
REFERENCES	To grant permission to create a foreign key reference to the specified table. If a column list is specified with the <code>REFERENCES</code> privilege, the permission is valid on only the foreign key reference to the specified columns.
SELECT	<p>To grant permission to perform <a href="#">SelectExpressions</a> on a table or view. If a column list is specified with the <code>SELECT</code> privilege, the permission is valid on only those columns. If no column list is specified, then the privilege is valid on all of the columns in the table.</p> <p>For queries that do not select a specific column from the tables involved in a <code>SELECT</code> statement or <i>SelectExpression</i> (for example, queries that use <code>COUNT ( * )</code>), the user must have at least one column-level <code>SELECT</code> privilege or table-level <code>SELECT</code> privilege.</p>
TRIGGER	To grant permission to create a trigger on the specified table.
UPDATE	To grant permission to use the <a href="#">WHERE</a> clause, you must have the <code>SELECT</code> privilege on the columns in the row that you want to update.

## Usage Notes

The following types of privileges can be granted:

- » Delete data from a specific table.
- » Insert data into a specific table.
- » Create a foreign key reference to the named table or to a subset of columns from a table.
- » Select data from a table, view, or a subset of columns in a table.

- » Create a trigger on a table.
- » Update data in a table or in a subset of columns in a table.
- » Run a specified function or procedure.
- » Use a user-defined type.

Before you issue a `GRANT` statement, check that the `derby.database.sqlAuthorization` property is set to `true`. The `derby.database.sqlAuthorization` property enables the SQL Authorization mode.

You can grant privileges on an object if you are the owner of the object or the database owner. See documentation for the [CREATE](#) statements for more information.

## Examples

### Granting Privileges to Users

To grant the `SELECT` privilege on the schema `SpliceBBase` to the authorization IDs `Bill` and `Joan`, use the following syntax:

```
splice> GRANT SELECT ON SCHEMA SpliceBBase TO Bill, Joan;
0 rows inserted/updated/deleted
```

To grant the `SELECT` privilege on table `Salaries` to the authorization IDs `Bill` and `Joan`, use the following syntax:

```
splice> GRANT SELECT ON TABLE Salaries TO Bill, Joan;
0 rows inserted/updated/deleted
```

To grant the `UPDATE` and `TRIGGER` privileges on table `Salaries` to the authorization IDs `Joe` and `Anita`, use the following syntax:

```
splice> GRANT UPDATE, TRIGGER ON TABLE Salaries TO Joe, Anita;
0 rows inserted/updated/deleted
```

To grant the `SELECT` privilege on table `Hitting` in the `Baseball_stats` schema to all users, use the following syntax:

```
splice> GRANT SELECT ON TABLE Baseball_stats.Hitting to PUBLIC;
0 rows inserted/updated/deleted
```

To grant the `EXECUTE` privilege on procedure `ComputeValue` to the authorization ID `george`, use the following syntax:

```
splice> GRANT EXECUTE ON PROCEDURE ComputeValue TO george;
0 rows inserted/updated/deleted
```

## Granting Roles to Users

To grant the role `purchases_reader_role` to the authorization IDs `george` and `maria`, use the following syntax:

```
splice> GRANT purchases_reader_role TO george,maria;  
0 rows inserted/updated/deleted
```

## Granting Privileges to Roles

To grant the `SELECT` privilege on schema `SpliceBBall` to the role `purchases_reader_role`, use the following syntax:

```
splice> GRANT SELECT ON SCHEMA SpliceBBall TO purchases_reader_role;  
0 rows inserted/updated/deleted
```

To grant the `SELECT` privilege on table `t` to the role `purchases_reader_role`, use the following syntax:

```
splice> GRANT SELECT ON TABLE t TO purchases_reader_role;  
0 rows inserted/updated/deleted
```

## See Also

- » [CREATE ROLE](#) statement
- » [CREATE TRIGGER](#) statement
- » [DROP ROLE](#) statement
- » [REVOKE](#) statement
- » [RoleName](#)
- » [SET ROLE](#) statement
- » [SELECT](#) expression
- » [SELECT](#) statement
- » [SYSROLES](#) system table
- » [UPDATE](#) statement
- » [WHERE](#) clause

# INSERT

An `INSERT` statement creates rows or columns and stores them in the named table. The number of values assigned in an `INSERT` statement must be the same as the number of specified or implied columns.

Whenever you insert into a table which has generated columns, Splice Machine calculates the values of those columns.

## Syntax

```
INSERT INTO table-Name
  [ (Simple-column-Name)* ) ]
  Query [ ORDER BY clause ]
  [ result offset clause ]
  [ fetch first clause ];
```

### *table-Name*

The table into which you are inserting data.

### *Simple-column-Name*\*

An optional list of names of the columns to populate with data.

### *Query* [*ORDER BY clause*]

A `SELECT` or `VALUES` command that provides the columns and rows of data to insert. The query can also be a `UNION` expression.

See the [Using the ORDER BY Clause](#) section below for information about using the `ORDER BY` clause.

Single-row and multiple-row `VALUES` expressions can include the keyword `DEFAULT`. Specifying `DEFAULT` for a column inserts the column's default value into the column. Another way to insert the default value into the column is to omit the column from the column list and only insert values into other columns in the table. For more information, see [VALUES expression](#).

### *result offset and fetch first clauses*

The [fetch first clause](#), which can be combined with the `result offset` clause, limits the number of rows added to the table.

## Using the ORDER BY Clause

When you want insertion to happen with a specific ordering (for example, in conjunction with auto-generated keys), it can be useful to specify an `ORDER BY` clause on the result set to be inserted.

If the Query is a `VALUES` expression, it cannot contain or be followed by an `ORDER BY`, `result offset`, or `fetch first` clause. However, if the `VALUES` expression does not contain the `DEFAULT` keyword, the `VALUES` clause can be put in a subquery and ordered, as in the following statement:

```
INSERT INTO t SELECT * FROM (VALUES 'a','c','b') t ORDER BY 1;
```

For more information about queries, see [Query](#).

## Examples

These examples insert records with literal values:

```
splice> INSERT INTO Players
VALUES( 99, 'Giants', 'Joe Bojangles', 'C', 'Little Joey', '07/11/1991');
1 row inserted/updated/deleted

splice> INSERT INTO Players
VALUES (99, 'Giants', 'Joe Bojangles', 'C', 'Little Joey', '07/11/1991'),
      (73, 'Giants', 'Lester Johns', 'P', 'Big John', '06/09/1984'),
      (27, 'Cards', 'Earl Hastings', 'OF', 'Speedy Earl', '04/22/1982');
3 rows inserted/updated/deleted
```

This example creates a table name OldGuys that has the same columns as our Players table, and then loads that table with the data from Players for all players born before 1980:

```
splice> CREATE TABLE OldGuys (
  ID          SMALLINT NOT NULL PRIMARY KEY,
  Team        VARCHAR(64) NOT NULL,
  Name        VARCHAR(64) NOT NULL,
  Position    CHAR(2),
  DisplayName VARCHAR(24),
  BirthDate   DATE
);

splice> INSERT INTO OldGuys
SELECT * FROM Players
WHERE BirthDate < '01/01/1980';
```

## Statement dependency system

The `INSERT` statement depends on the table being inserted into, all of the conglomerates (units of storage such as heaps or indexes) for that table, and any other table named in the statement. Any statement that creates or drops an index or a constraint for the target table of a prepared `INSERT` statement invalidates the prepared `INSERT` statement.

## See Also

- » [FETCH FIRST](#) clause
- » [ORDER BY](#) clause
- » [Queries](#)

» [RESULT OFFSET](#) clause



# PIN TABLE

The `PIN TABLE` statement allows you to pin (cache) a table in memory, which can improve performance for tables that are being used frequently in analytic operations.

The pinned version of a table is a static version of that table; updates to the underlying table are not automatically reflected in the pinned version. To refresh the pinned version, you need to unpin and then repin the table, as described in the [Usage Notes](#) section below.

## Syntax

```
PIN TABLE tableName;
```

*tableName*

A string that specifies the name of the table that you want to pin in memory.

An error occurs if the named table does not exist.

## Usage Notes

Here are a few important notes about using pinned tables:

- » [Pinned and Unpinned Table Versions](#)
- » [Refreshing the Pinned Version of a Table](#)
- » [Unpinning and Dropping Pinned Tables](#)

## Pinned and Unpinned Table Versions

Once you pin a table, you effectively have two versions of it to work with:

- » The original table continues to work just as usual
- » The pinned version is a static version of the table at the time you pinned it. To access the pinned version of a table, you must specify the Splice `pin=true` property. If you do not specify this property in your query, the query will operate on the unpinned version of the table.

The pinned version (version) of a table is statically cached in memory; this means that:

- » Updates to the table (unpinned version) are not automatically reflected in the pinned version of the table.
- » Updates to the pinned version of the table are not permitted: you cannot insert into, delete from, or update the pinned version of a table.

Here's a simple example that illustrates these qualities:

```

splice> CREATE TABLE myTbl (col1 int, col2 varchar(10));
0 rows inserted/updated/deleted
splice> INSERT INTO myTbl VALUES (1, 'One'), (2, 'Two');
2 rows inserted/updated/deleted
splice> PIN TABLE myTbl;
0 rows inserted/updated/deleted
splice> INSERT INTO myTbl VALUES (3, 'Three'), (4, 'Four');
2 rows inserted/updated/deleted
splice> SELECT * FROM myTbl;
COL1      |COL2
-----
1          One
2          Two
3          Three
4          Four

4 rows selected
splice> SELECT * FROM myTbl --splice-properties pin=true
> ;
COL1      |COL2
-----
1          One
2          Two2 rows selected
splice> UPDATE myTbl SET col1=11 WHERE col1=1;1 row in
serted/updated/deleted
splice> UPDATE myTbl --splice-properties pin=trueSET col1=21 W
HERE col1=2;ERROR: Pinned Table read failed with exception Table or view not found i
n database.

```

## Refreshing the Pinned Version of a Table

If you update the table and want the pinned version to reflect those updates, you need to refresh your pinned table version. You can simply unpin the table from memory, and then repin it into memory:

```

splice> UNPIN TABLE Players;0 rows inserted/updated/deleted
splice> PIN TABLE Player
s;0 rows inserted/updated/deleted

```

Now the pinned version of the table matches the original version.

## Unpinning and Dropping Pinned Tables

When you drop a table (with the [DROP TABLE](#) statement), the pinned version is automatically deleted and can no longer be used.

To delete just the pinned version of a table, use the [UNPIN TABLE](#) statement.

## See Also

» [CREATE EXTERNAL TABLE](#) statement

- » [CREATE TABLE](#) statement
- » [UNPIN TABLE](#) statement
- » [Query Optimizations](#) in the *Splice Machine Developer's Guide*

# RENAME COLUMN

Use the `RENAME COLUMN` statement to rename a column in a table.

The `RENAME COLUMN` statement allows you to rename an existing column in an existing table in any schema (except the schema `SYS`).

## Syntax

```
RENAME COLUMN simple-Column-Name
TO simple-Column-Name
```

*table-Name*

The name of the table containing the column to rename.

*simple-Column-Name*

The name of the column to be renamed.

*simple-Column-Name*

The new name for the column.

## Usage Notes

To rename a column, you must either be the database owner or the table owner.

To perform other table alterations, see the [ALTER TABLE statement](#).

If a view, trigger, check constraint, or [generation-clause](#) of a generated column references the column, an attempt to rename it will generate an error.

**NOTE:** The `RENAME COLUMN` statement is not allowed if there are any open cursors that reference the column that is being altered.

**NOTE:** If there is an index defined on the column, the column can still be renamed; the index is automatically updated to refer to the column by its new name.

## Examples

To rename the `Birthdate` column in table `Players` to `BornDate`, use the following syntax:

```
splice> RENAME COLUMN Players.Birthdate TO BornDate;
0 rows inserted/updated/deleted
```

If you want to modify a column's data type, you can combine `ALTER TABLE`, `UPDATE`, and `RENAME COLUMN` using these steps, as show in the example below:

1. Add a new column to the table with the new data type
2. Copy the values from the “old” column to the new column with an `UPDATE` statement.
3. Drop the “old” column.
4. Rename the new column with the old column's name.

```
splice> ALTER TABLE Players ADD COLUMN NewPosition VARCHAR(8);
0 rows inserted/updated/deleted

splice> UPDATE Players SET NewPosition = Position;
0 rows inserted/updated/deleted

splice> ALTER TABLE Players DROP COLUMN Position;
0 rows inserted/updated/deleted

splice> RENAME COLUMN Players.NewPosition TO Position;
0 rows inserted/updated/deleted
```

## See Also

» [ALTER](#) statement

# RENAME INDEX

The `RENAME INDEX` statement allows you to rename an index in the current schema. Users cannot rename indexes in the `SYS` schema.

## Syntax

```
RENAME INDEX index-Name TO new-index-Name
```

*index-Name*

The name of the index to be renamed.

*new-Index-Name*

The new name for the index.

## Example

```
splice> RENAME INDEX myIdx TO Player_index;  
0 rows inserted/updated/deleted
```

## See Also

» [ALTER](#) statement

# RENAME TABLE

The `RENAME TABLE` statement allows you to rename an existing table in any schema (except the schema `SYS`).

To rename a table, you must either be the database owner or the table owner.

## Syntax

```
RENAME TABLE table-Name TO new-Table-Name
```

*table-Name*

The name of the table to be renamed.

*new-Table-Name*

The new name for the table.

## Usage Notes

Attempting to rename a table generates an error if:

- » there is a view or a foreign key that references the table
- » there are any check constraints or triggers on the table

## Example

```
splice> RENAME TABLE MorePlayers to PlayersTest;  
0 rows inserted/updated/deleted
```

See the [ALTER TABLE statement](#) for more information.

# REVOKE

Use the `REVOKE` statement to remove privileges from a specific user or role, or from all users, to perform actions on database objects. You can also use the `REVOKE` statement to revoke a role from a user, from `PUBLIC`, or from another role.

The syntax that you use for the `REVOKE` statement depends on whether you are revoking privileges to a schema object or revoking a role.

## Syntax for SCHEMA

```
REVOKE privilege-type
ON [ SCHEMA ] schema
FROM grantees
```

### *privilege-type*

```
DELETE
| INSERT
| REFERENCES [( column-identifier {, column-identifier}* )]
| SELECT [( column-identifier {, column-identifier}* )]
| TRIGGER
| UPDATE [( column-identifier {, column-identifier}* )]
```

See the [Privilege Types](#) section below for more information.



Column-level privileges are available only with a Splice Machine Enterprise license.

You cannot grant or revoke privileges at the `column-identifier` level with the Community version of Splice Machine.

To obtain a license for the Splice Machine Enterprise Edition, please [Contact Splice Machine Sales](#) today.

### *schema-Name*

The name of the schema for which you are revoking access.

### *grantees*

The user(s) or role(s) for whom you are revoking access. See the [About Grantees](#) section below for more information.



## Syntax for Tables

```
REVOKE privilege-type
ON [ TABLE ] table-Name
FROM grantees
```

### *privilege-type*

```
DELETE
| INSERT
| REFERENCES [( column-identifier {, column-identifier}* )]
| SELECT [( column-identifier {, column-identifier}* )]
| TRIGGER
| UPDATE [( column-identifier {, column-identifier}* )]
```

See the [Privilege Types](#) section below for more information.



Column-level privileges are available only with a Splice Machine Enterprise license.

You cannot grant or revoke privileges at the `column-identifier` level with the Community version of Splice Machine.

To obtain a license for the Splice Machine Enterprise Edition, please [Contact Splice Machine Sales](#) today.

### *table-Name*

The name of the table for which you are revoking access.

### *view-Name*

The name of the view for which you are revoking access.

### *grantees*

The user(s) or role(s) for whom you are revoking access. See the [About Grantees](#) section below for more information.

## Syntax for Routines

```
REVOKE EXECUTE ON { FUNCTION | PROCEDURE }
{function-name | procedure-name}
TO grantees RESTRICT
```

### *function-name | procedure-name*

The name of the function or procedure for which you are revoking access.

### *grantees*

The user(s) or role(s) for whom you are revoking access. See the [About Grantees](#) section below for more information.

**RESTRICT**

You **must** use this clause when revoking access for routines.

The `RESTRICT` clause specifies that the `EXECUTE` privilege cannot be revoked if the specified routine is used in a view, trigger, or constraint, and the privilege is being revoked from the owner of the view, trigger, or constraint.

## Syntax for User-defined types

```
REVOKE USAGE
ON TYPE SQL Identifier
FROM grantees RESTRICT
```

*[schema-name.] SQL Identifier*

The user-defined type (UDT) name is composed of an optional *schemaName* and a *SQL Identifier*. If a *schemaName* is not provided, the current schema is the default schema. If a qualified UDT name is specified, the schema name cannot begin with `SYS`.

*grantees*

The user(s) or role(s) for whom you are revoking access. See the [About Grantees](#) section below for more information.

**RESTRICT**

You **must** use this clause when revoking access for user-defined types.

The `RESTRICT` clause specifies that the `EXECUTE` privilege cannot be revoked if the specified UDT is used in a view, trigger, or constraint, and the privilege is being revoked from the owner of the view, trigger, or constraint.

## Syntax for Roles

```
REVOKE roleName
{ roleName } *
FROM grantees
```

*roleName*

The name to the role(s) for which you are revoking access.

*grantees*

The user(s) or role(s) for whom you are revoking access. See the [About Grantees](#) section below for more information.

Only the database owner can revoke a role.

## About Grantees

A grantee can be one or more specific users, one or more specific roles, or all users (`PUBLIC`). Either the object owner or the database owner can grant privileges to a user or to a role. Only the database owner can grant a role to a user or to another role.

Here's the syntax:

```
{      roleName | PUBLIC }
[, { roleName | PUBLIC } ] *
```

### *AuthorizationIdentifier*

An expression.

### *roleName*

The name of the role.

Either the object owner or the database owner can grant privileges to a user or to a role. Only the database owner can grant a role to a user or to another role.

### *PUBLIC*

Use the keyword `PUBLIC` to specify all users.

When `PUBLIC` is specified, the privileges or roles affect all current and future users.

The privileges granted to `PUBLIC` and to individual users or roles are independent privileges. For example, a `SELECT` privilege on table `t` is granted to both `PUBLIC` and to the authorization ID `harry`. If the `SELECT` privilege is later revoked from the authorization ID `harry`, Harry will still be able to access the table `t` through the `PUBLIC` privilege..

## Privilege Types

Privilege Type	Usage
ALL PRIVILEGES	To revoke all of the privileges to the user or role for the specified table. You can also grant one or more table privileges by specifying a privilege-list.
DELETE	To revoke permission to delete rows from the specified table.
INSERT	To revoke permission to insert rows into the specified table.
REFERENCES	To revoke permission to create a foreign key reference to the specified table. If a column list is pecified with the <code>REFERENCES</code> privilege, the permission is valid on only the foreign key reference to the specified columns.

Privilege Type	Usage
SELECT	<p>To revoke permission to perform <a href="#">SelectExpressions</a> on a table or view. If a column list is specified with the <code>SELECT</code> privilege, the permission is valid on only those columns. If no column list is specified, then the privilege is valid on all of the columns in the table.</p> <p>For queries that do not select a specific column from the tables involved in a <code>SELECT</code> statement or <i>SelectExpression</i> (for example, queries that use <code>COUNT ( * )</code>), the user must have at least one column-level <code>SELECT</code> privilege or table-level <code>SELECT</code> privilege.</p>
TRIGGER	To revoke permission to create a trigger on the specified table.
UPDATE	To revoke permission to use the <a href="#">WHERE</a> clause, you must have the <code>SELECT</code> privilege on the columns in the row that you want to update.

## Usage Notes

The following types of privileges can be revoked:

- » Delete data from a specific table.
- » Insert data into a specific table.
- » Create a foreign key reference to the named table or to a subset of columns from a table.
- » Select data from a table, view, or a subset of columns in a table.
- » Create a trigger on a table.
- » Update data in a table or in a subset of columns in a table.
- » Run a specified routine (function or procedure).
- » Use a user-defined type.

Before you issue a `REVOKE` statement, check that the `derby.database.sqlAuthorization` property is set to `true`. The `derby.database.sqlAuthorization` property enables the SQL Authorization mode.

You can revoke privileges on an object if you are the owner of the object or the database owner. See the `CREATE` statement for the database object that you want To revoke privileges on for more information.

You can revoke privileges for an object if you are the owner of the object or the database owner.

## Prepared statements and open result sets

Checking for privileges happens at statement execution time, so prepared statements are still usable after a revoke action. If sufficient privileges are still available for the session, prepared statements will be executed, and for queries, a result set will be returned.

Once a result set has been returned to the application (by executing a prepared statement or by direct execution), it will remain accessible even if privileges or roles are revoked in a way that would cause another execution of the same statement to fail.

## Cascading object dependencies

For views, triggers, and constraints, if the privilege on which the object depends on is revoked, the object is automatically dropped. Splice Machine does not try to determine if you have other privileges that can replace the privileges that are being revoked.

## Limitations

The following limitations apply to the REVOKE statement:

### *Table-level privileges*

All of the table-level privilege types for a specified grantee and table ID are stored in one row in the SYSTABLEPERMS system table. For example, when `user2` is granted the `SELECT` and `DELETE` privileges on table `user1.t1`, a row is added to the SYSTABLEPERMS table. The `GRANTEE` field contains `user2` and the `TABLEID` contains `user1.t1`. The `SELECTPRIV` and `DELETEPRIV` fields are set to `Y`. The remaining privilege type fields are set to `N`.

When a grantee creates an object that relies on one of the privilege types, Splice Machine engine tracks the dependency of the object on the specific row in the SYSTABLEPERMS table. For example, `user2` creates the view `v1` by using the statement `SELECT * FROM user1.t1`, the dependency manager tracks the dependency of view `v1` on the row in SYSTABLEPERMS for `GRANTEE(user2), TABLEID(user1.t1)`.

The dependency manager knows only that the view is dependent on a privilege type in that specific row, but does not track exactly which privilege type the view is dependent on.

When a `REVOKE` statement for a table-level privilege is issued for a grantee and table ID, all of the objects that are dependent on the grantee and table ID are dropped. For example, if `user1` revokes the `DELETE` privilege on table `t1` from `user2`, the row in SYSTABLEPERMS for `GRANTEE(user2), TABLEID(user1.t1)` is modified by the `REVOKE` statement. The dependency manager sends a revoke invalidation message to the view `user2.v1` and the view is dropped even though the view is not dependent on the `DELETE` privilege for `GRANTEE(user2), TABLEID(user1.t1)`.

### *Column-level privileges*

Only one type of privilege for a specified grantee and table ID are stored in one row in the SYSCOLPERMS system table. For example, when `user2` is granted the `SELECT` privilege on table `user1.t1` for columns `c12` and `c13`, a row is added to the SYSCOLPERMS. The `GRANTEE` field contains `user2`, the `TABLEID` contains `user1.t1`, the `TYPE` field contains `S`, and the `COLUMNS` field contains `c12, c13`.

When a grantee creates an object that relies on the privilege type and the subset of columns in a table ID, Splice Machine engine tracks the dependency of the object on the specific row in the SYSCOLPERMS table. For example, `user2` creates the view `v1` by using the statement `SELECT c11 FROM user1.t1`, the dependency manager tracks the dependency of

view `v1` on the row in `SYSCOLPERMS` for `GRANTEE(user2)`, `TABLEID(user1.t1)`, `TYPE(s)`. The dependency manager knows that the view is dependent on the `SELECT` privilege type, but does not track exactly which columns the view is dependent on.

When a `REVOKE` statement for a column-level privilege is issued for a grantee, table ID, and type, all of the objects that are dependent on the grantee, table ID, and type are dropped. For example, if `user1` revokes the `SELECT` privilege on column `c12` on table `user1.t1` from `user2`, the row in `SYSCOLPERMS` for `GRANTEE(user2)`, `TABLEID(ser1.t1)`, `TYPE(s)` is modified by the `REVOKE` statement. The dependency manager sends a revoke invalidation message to the view `user2.v1` and the view is dropped even though the view is not dependent on the column `c12` for `GRANTEE(user2)`, `TABLEID(user1.t1)`, `TYPE(s)`.

### Roles

Splice Machine tracks any dependencies on the definer's current role for views and constraints, constraints, and triggers. If privileges were obtainable only via the current role when the object in question was defined, that object depends on the current role. The object will be dropped if the role is revoked from the defining user or from `PUBLIC`, as the case may be.

Also, if a contained role of the current role in such cases is revoked, dependent objects will be dropped. Note that dropping may be too pessimistic. This is because Splice Machine does not currently make an attempt to recheck if the necessary privileges are still available in such cases.

## Revoke Examples

### Revoking User Privileges

To revoke the `SELECT` privilege on schema `SpliceBBall` from the authorization IDs `Bill` and `Joan`, use the following syntax:

```
splice> REVOKE SELECT ON SCHEMA SpliceBBall FROM Bill, Joan;
0 rows inserted/updated/deleted
```

To revoke the `SELECT` privilege on table `Salaries` from the authorization IDs `Bill` and `Joan`, use the following syntax:

```
splice> REVOKE SELECT ON TABLE Salaries FROM Bill, Joan;
0 rows inserted/updated/deleted
```

To revoke the `UPDATE` and `TRIGGER` privileges on table `Salaries` from the authorization IDs `Joe` and `Anita`, use the following syntax:

```
splice> REVOKE UPDATE, TRIGGER ON TABLE Salaries FROM Joe, Anita;
0 rows inserted/updated/deleted
```

To revoke the `SELECT` privilege on table `Hitting` in the `Baseball_stats` schema from all users, use the following syntax:

```
splice> REVOKE SELECT ON TABLE Baseball_Stats.Hitting FROM PUBLIC;
0 rows inserted/updated/deleted
```

To revoke the `EXECUTE` privilege on procedure `ComputeValue` from the authorization ID `george`, use the following syntax:

```
splice> REVOKE EXECUTE ON PROCEDURE ComputeValue FROM george;  
0 rows inserted/updated/deleted
```

## Revoking User Roles

To revoke the role ``purchases_reader_role`` from the authorization IDs `george` and `maria``, use the following syntax:

```
splice> REVOKE purchases_reader_role FROM george,maria;  
0 rows inserted/updated/deleted
```

## Revoking Role Privileges

To revoke the `SELECT` privilege on schema `SpliceBBall` from the role `purchases_reader_role`, use the following syntax:

```
splice> REVOKE SELECT ON SCHEMA SpliceBBall FROM purchases_reader_role;  
0 rows inserted/updated/deleted
```

To revoke the `SELECT` privilege on table `t` to the role `purchases_reader_role`, use the following syntax:

```
splice> REVOKE SELECT ON TABLE t FROM purchases_reader_role;  
0 rows inserted/updated/deleted
```

## See Also

- » [CREATE ROLE](#) statement
- » [DROP ROLE](#) statement
- » [GRANT](#) statement
- » [RoleName](#)
- » [SET ROLE](#) statement
- » [SELECT](#) expression
- » [SELECT](#) statement
- » [SYSROLES](#) system table
- » [UPDATE](#) statement
- » [WHERE](#) clause

# SELECT

Use the `SELECT` statement to query a database and receive back results.

## Syntax

```
SELECT Query
  [ORDER BY clause]
  [result offset clause]
  [fetch first clause]
```

### Query

The `SELECT` statement is so named because the typical first word of the query construct is `SELECT`. (*Query* includes the `SELECT` expressions).

### ORDER BY clause

The `ORDER BY` clause allows you to order the results of the `SELECT`. Without the `ORDER BY` clause, the results are returned in random order.

### result offset and fetch first clauses

The `fetch first clause`, which can be combined with the `result offset clause`, limits the number of rows fetched.

## Example

This example selectss all records in the `Players` table:

```
splice> SELECT * FROM Players WHERE ID < 11;
ID      |TEAM      |POS&|DISPLAYNAME      |BIRTHDATE
-----|-----|-----|-----|-----
1       |Giants    |C    |Buddy Painter    |1987-03-27
2       |Giants    |1B   |Billy Bopper     |1988-04-20
3       |Giants    |2B   |John Purser      |1990-10-30
4       |Giants    |SS   |Bob Cranker      |1987-01-21
5       |Giants    |3B   |Mitch Duffer     |1991-01-15
6       |Giants    |LF   |Norman Aikman    |1982-01-05
7       |Giants    |CF   |Alex Paramour    |1981-07-02
8       |Giants    |RF   |Harry Pennello   |1983-04-13
9       |Giants    |OF   |Greg Brown       |1983-12-24
10      |Giants    |RF   |Jason Minman     |1983-11-06

10 rows selected
```

This example selects the Birthdate of all players born in November or December:



```
splice> SELECT BirthDate
        FROM Players
        WHERE MONTH(BirthDate) > 10
        ORDER BY BIRTHDATE;
BIRTHDATE
```

```
-----
1980-12-19
1983-11-06
1983-11-28
1983-12-24
1984-11-22
1985-11-07
1985-11-26
1985-12-21
1986-11-13
1986-11-24
1986-12-16
1987-11-12
1987-11-16
1987-12-17
1988-12-21
1989-11-17
1991-11-15
```

17 rows selected

This example selects the name, team, and birth date of all players born in 1985 and 1989:

```
splice> SELECT DisplayName, Team, BirthDate
        FROM Players
        WHERE YEAR(BirthDate) IN (1985, 1989)
        ORDER BY BirthDate;
```

DISPLAYNAME	TEAM	BIRTHDATE
-----		
Jeremy Johnson	Cards	1985-03-15
Gary Kosovo	Giants	1985-06-12
Michael Hillson	Cards	1985-11-07
Mitch Canepa	Cards	1985-11-26
Edward Erdman	Cards	1985-12-21
Jeremy Packman	Giants	1989-01-01
Nathan Nickels	Giants	1989-05-04
Ken Straiter	Cards	1989-07-20
Marcus Bamburger	Giants	1989-08-01
George Goomba	Cards	1989-08-08
Jack Hellman	Cards	1989-08-09
Elliot Andrews	Giants	1989-08-21
Henry Socomy	Giants	1989-11-17

13 rows selected

## Statement dependency system

The `SELECT` statement depends on all the tables and views named in the query and the conglomerates (units of storage such as heaps and indexes) chosen for access paths on those tables.

The `SELECT` statement depends on all aliases used in the query. Dropping an alias invalidates any prepared `SELECT` statement that uses the alias.

## See Also

- » [CREATE INDEX](#) statement
- » [CREATE VIEW](#) statement
- » [DROP INDEX](#) statement
- » [DROP VIEW](#) statement
- » [GRANT](#) statement
- » [ORDER BY](#) clause
- » [FETCH FIRST](#) clause
- » [RESULT OFFSET](#) clause

# SET ROLE

The `SET ROLE` statement allows you to set the current role for the current SQL context of a session.

You can set a role only if the current user has been granted the role, or if the role has been granted to `PUBLIC`.

**NOTE:** The `SET ROLE` statement is not transactional; a rollback does not undo the effect of setting a role. If a transaction is in progress, an attempt to set a role results in an error.

## Syntax

```
SET ROLE { roleName | 'string-constant' | ? | NONE }
```

### *roleName*

The role you want set as the current role.

You can specify a *roleName* of `NONE` to unset the current role.

If you specify the role as a string constant or as a dynamic parameter specification (?), any leading and trailing blanks are trimmed from the string before attempting to use the remaining (sub)string as a *roleName*. The dynamic parameter specification can be used in prepared statements, so the `SET ROLE` statement can be prepared once and then executed with different role values. You cannot specify `NONE` as a dynamic parameter.

## Usage Notes

Setting a role identifies a set of privileges that is a union of the following:

- » The privileges granted to that role
- » The union of privileges of roles contained in that role (for a definition of role containment, see “Syntax for roles” in [GRANT statement](#))

In a session, the *current privileges* define what the session is allowed to access. The *current privileges* are the union of the following:

- » The privileges granted to the current user
- » The privileges granted to `PUBLIC`
- » The privileges identified by the current role, if set

**NOTE:** You can find the available role names in the [SYS.SYSROLES](#) system table.

## SQL Example

This examples set the role of the current user to `reader_role`:

```
splice> SET ROLE reader_role;
0 rows inserted/updated/deleted
```

## JDBC Example

This examples set the role of the current user to `reader_role`:

```
stmt.execute("SET ROLE admin");           -- case normal form: ADMIN
stmt.execute("SET ROLE \"admin\"");       -- case normal form: admin
stmt.execute("SET ROLE none");            -- special case

PreparedStatement ps = conn.prepareStatement("SET ROLE ?");
ps.setString(1, "  admin ");              -- on execute: case normal form: ADMIN
ps.setString(1, "\"admin\"");             -- on execute: case normal form: admin
ps.setString(1, "none");                  -- on execute: syntax error
ps.setString(1, "\"none\"");              -- on execute: case normal form: none
```

## See Also

- » [CREATE\\_ROLE](#) statement
- » [DROP\\_ROLE](#) statement
- » [GRANT](#) statement
- » [REVOKE](#) statement
- » [RoleName](#)
- » [SET\\_ROLE](#) statement
- » [SELECT](#) expression
- » [SELECT](#) statement
- » [SYSROLES](#) system table
- » [UPDATE](#) statement
- » [WHERE](#) clause

# SET SCHEMA

The `SET SCHEMA` statement sets the default schema for a connection's session to the designated schema. The default schema is used as the target schema for all statements issued from the connection that do not explicitly specify a schema name.

The target schema must exist for the `SET SCHEMA` statement to succeed. If the schema doesn't exist an error is returned.

**NOTE:** The `SET SCHEMA` statement is not transactional: if the `SET SCHEMA` statement is part of a transaction that is rolled back, the schema change remains in effect.

## Syntax

```
SET [CURRENT] SCHEMA [=] schemaName
```

*schemaName*

The name of the schema; this name is not case sensitive.

## Examples

These examples are equivalent:

```
splice> SET SCHEMA BASEBALL;
0 rows inserted/updated/deleted

splice> SET SCHEMA Baseball;
0 rows inserted/updated/deleted

splice> SET CURRENT SCHEMA BaseBall;
0 rows inserted/updated/deleted

splice> SET CURRENT SQLID BASEBALL;
0 rows inserted/updated/deleted

splice> SET SCHEMA "BASEBALL";
0 rows inserted/updated/deleted

splice> SET SCHEMA 'BASEBALL'
0 rows inserted/updated/deleted
```

These fail because of case sensitivity:

```
splice> SET SCHEMA "Baseball";
ERROR 42Y07: Schema 'Baseball' does not exist

splice> SET SCHEMA 'BaseBall';
ERROR 42Y07: Schema 'BaseBall' does not exist
```

Here's an example using a prepared statement:

```
PreparedStatement ps = conn.prepareStatement("set schema ?");
ps.setString(1,"HOTEL");
ps.executeUpdate();
... these work:
ps.setString(1,"SPLICE");

ps.executeUpdate();
ps.setString(1,"splice");           //error - string is case sensitive
// no app will be found
ps.setNull(1, Types.VARCHAR); //error - null is not allowed
```

## See Also

- » [CREATE SCHEMA](#) statement
- » [DROP SCHEMA](#) statement
- » [Schema Name](#)

# TRUNCATE TABLE

The `TRUNCATE TABLE` statement allows you to quickly remove all content from the specified table and return it to its initial empty state.

To truncate a table, you must either be the database owner or the table owner.

You cannot truncate system tables or global temporary tables with this statement.

## Syntax

```
TRUNCATE TABLE table-Name
```

*table-Name*

The name of the table to truncate.

## Examples

To truncate the entire `Players_Test` table, use the following statement:

```
splice> TRUNCATE TABLE Players_Test;  
0 rows inserted/updated/deleted
```

# UNPIN TABLE

The `UNPIN TABLE` statement unpins a table, which means that the pinned (previously cached) version of the table no longer exists.

## Syntax

```
UNPIN TABLE table-Name
```

*table-Name*

The name of the pinned table that you want to unpin.

## Example

```
splice> CREATE TABLE myTbl (col1 int, col2 varchar(10));
0 rows inserted/updated/deleted
splice> INSERT INTO myTbl VALUES (1, 'One'), (2, 'Two');
2 rows inserted/updated/deleted
COL 1      |COL2
-----
1          One
2          Two
2 rows selected
splice> PIN TABLE myTbl;
0 rows inserted/updated/deleted
splice> SELECT * FROM myTbl --splice-properties pin=true
> ;
COL 1      |COL2
-----
1          One
2          Two
2 rows selected
splice> UNPIN TABLE myTbl;splice> SELECT * FROM myTbl;
COL 1      |COL2
-----
1          One
2          Two
2 rows selected
splice> SELECT * FROM myTbl --splice-properties pin=true
> ERROR: Pinned table read failed with exception 'Table or view not found in database'
```



## See Also

- » [CREATE EXTERNAL TABLE](#) statement
- » [CREATE TABLE](#) statement
- » [PIN TABLE](#) statement
- » [Query Optimizations](#) in the *Splice Machine Developer's Guide*

# UPDATE

Use the `UPDATE` statement to update existing records in a table.

## Syntax

```
{
  UPDATE table-Name
  [[AS] correlation-Name]
  SET column-Name = Value
    [ , column-Name = Value} ]*
  [WHERE clause]
}
```

### *table-Name*

The name of the table to update.

### *correlation-Name*

An optional correlation name for the update.

### *column-Name = Value*

Sets the value of the named column to the named value in any records .

*Value* is either an [Expression](#) or the literal `DEFAULT`. If you specify `DEFAULT` for a column's value, the value is set to the default defined for the column in the table.

The `DEFAULT` literal is the only value that you can directly assign to a generated column. Whenever you alter the value of a column referenced by the [generation-clause](#) of a generated column, Splice Machine recalculates the value of the generated column.

### *WHERE clause*

Specifies the records to be updated.

## Example

This example updates the Birthdate value for a specific player:

```
splice> UPDATE Players
  SET Birthdate='03/27/1987'
  WHERE DisplayName='Buddy Painter';
1 row inserted/updated/deleted
```

This example updates the team name associated with all players on the `Giants` team:

```
splice> UPDATE Players  
      SET Team='SFGiants'  
      WHERE Team='Giants';  
48 rows inserted/updated/deleted
```

## Statement dependency system

A searched update statement depends on the table being updated, all of its conglomerates (units of storage such as heaps or indexes), all of its constraints, and any other table named in the `DROP INDEX` statement or an [ALTER TABLE](#) statement for the target table of a prepared searched update statement invalidates the prepared searched update statement.

A `CREATE` or `DROP INDEX` statement or an `ALTER TABLE` statement for the target table of a prepared positioned update invalidates the prepared positioned update statement.

Dropping an alias invalidates a prepared update statement if the latter statement uses the alias.

Dropping or adding triggers on the target table of the update invalidates the update statement.

## See Also

- » [ALTER TABLE](#) statement
- » [CONSTRAINT](#) clause
- » [CREATE TABLE](#) statement
- » [CREATE TRIGGER](#)
- » [DROP INDEX](#) statement
- » [DROP TRIGGER](#)
- » [WHERE](#) clause

# Clauses

This section contains the reference documentation for the Splice Machine SQL Clauses, in the following topics:

Clause	Description
<a href="#">CONSTRAINT</a>	Optional clause in <a href="#">ALTER TABLE</a> statements that specifies a rule to which the data must conform.
<a href="#">EXCEPT</a>	Takes the distinct rows in the results from one a <a href="#">SELECT</a> statement.
<a href="#">FROM</a>	A clause in a <a href="#">SelectExpression</a> that specifies the tables from which the other clauses of the query can access columns for use in expressions.
<a href="#">GROUP BY</a>	Part of a <a href="#">SelectExpression</a> that groups a result into subsets that have matching values for one or more columns.
<a href="#">HAVING</a>	Restricts the results of a <code>GROUP BY</code> clause in a <a href="#">SelectExpression</a> .
<a href="#">LIMIT n</a>	Limits the number of results returned by a query.
<a href="#">OVER</a>	Used in window functions to define the window on which the function operates.
<a href="#">ORDER BY</a>	Allows you to specify the order in which rows appear in the result set.
<a href="#">RESULT OFFSET and FETCH FIRST</a>	Provide a way to skip the N first rows in a result set before starting to return any rows and/or to limit the number of rows returned in the result set.
<a href="#">TOP n</a>	Limits the number of results returned by a query.
<a href="#">UNION</a>	Combines the result sets from two queries into a single table that contains all matching rows.
<a href="#">USING</a>	Specifies which columns to test for equality when two tables are joined.
<a href="#">WHERE</a>	An optional part of a <a href="#">UPDATE statement</a> that lets you select rows based on a Boolean expression.
<a href="#">WITH</a>	Allows you to name subqueries to make your queries more readable and/or to improve efficiency.



For access to the source code for the Community Edition of Splice Machine, visit [our open source GitHub repository](#).

# CONSTRAINT

A `CONSTRAINT` clause is a rule to which data must conform, and is an optional part of [ALTER TABLE](#) statements. Constraints can optionally be named.

There are two types of constraints:

## *column-level constraints*

A column-level constraint refers to a single column in a table (the column that it follows syntactically) in the table. Column constraints, other than `CHECK` constraints, do not specify a column name.

## *table-level constraints*

A table-level constraints refers to one or more columns in a table by specifying the names of those columns. Table-level `CHECK` constraints can refer to 0 or more columns in the table.

Column constraints and table constraints have the same function; the difference is in where you specify them.

- » Table constraints allow you to specify more than one column in a `PRIMARY KEY` or `CHECK`, `UNIQUE` or `FOREIGN KEY` constraint definition.
- » Column-level constraints (except for check constraints) refer to only one column.

## Column Constraints

```
{
  NOT NULL |
  [ [CONSTRAINT constraint-Name] {PRIMARY KEY} ]
}
```

```
{
  NOT NULL |
  [ [CONSTRAINT constraint-Name]
  {
    CHECK (searchCondition) |
    {
      PRIMARY KEY |
      UNIQUE |
      REFERENCES clause
    }
  }
}
```

### **NOT NULL**

Specifies that this column cannot hold `NULL` values (constraints of this type are not nameable).

### **PRIMARY KEY**

Specifies the column that uniquely identifies a row in the table. The identified columns must be defined as NOT NULL.

**NOTE:** At this time, you **cannot** add a primary key using ALTER TABLE.

#### UNIQUE

Specifies that values in the column must be unique.

#### FOREIGN KEY

Specifies that the values in the column must correspond to values in a referenced primary key or unique key column or that they are NULL.

#### CHECK

Specifies rules for values in the column.

## Table Constraints

```
[CONSTRAINT constraint-Name]
{
  PRIMARY KEY ( Simple-column-Name
    [ , Simple-column-Name ]* )
}
```

```
[CONSTRAINT constraint-Name]
{
  CHECK (searchCondition) |
  {
    PRIMARY KEY ( Simple-column-Name [ , Simple-column-Name ]* ) |
    UNIQUE ( Simple-column-Name [ , Simple-column-Name ]* ) |
    FOREIGN KEY ( Simple-column-Name [ , Simple-column-Name ]* )
    REFERENCES clause
  }
}
```

#### PRIMARY KEY

Specifies the column or columns that uniquely identify a row in the table. NULL values are not allowed.

**NOTE:** At this time, you **cannot** add a primary key using ALTER TABLE.

#### UNIQUE

Specifies that values in the columns must be unique.

#### FOREIGN KEY

Specifies that the values in the columns must correspond to values in referenced primary key or unique columns or that they are NULL.

**NOTE:** If the foreign key consists of multiple columns, and *any* column is `NULL`, the whole key is considered `NULL`. The insert is permitted no matter what is on the non-null columns.

### CHECK

Specifies a wide range of rules for values in the table.

## Primary Key Constraints

**NOTE:** At this time, you **cannot** alter primary keys using `ALTER TABLE`.

Primary keys are constrained as follows:

- » A primary key defines the set of columns that uniquely identifies rows in a table.
- » When you create a primary key constraint, none of the columns included in the primary key can have `NULL` constraints; that is, they must not permit `NULL` values.
- » A table can have at most one `PRIMARY KEY` constraint.

## Unique constraints

A `UNIQUE` constraint defines a set of columns that uniquely identify rows in a table only if all the key values are not `NULL`. If one or more key parts are `NULL`, duplicate keys are allowed.

For example, if there is a `UNIQUE` constraint on `col1` and `col2` of a table, the combination of the values held by `col1` and `col2` will be unique as long as these values are not `NULL`. If one of `col1` and `col2` holds a `NULL` value, there can be another identical row in the table.

A table can have multiple `UNIQUE` constraints.

## Foreign key constraints

Foreign keys provide a way to enforce the referential integrity of a database. A foreign key is a column or group of columns within a table that references a key in some other table (or sometimes the same table). The foreign key must always include the columns of which the types exactly match those in the referenced primary key or unique constraint.

For a table-level foreign key constraint in which you specify the columns in the table that make up the constraint, you cannot use the same column more than once.

If there is a column list in the *ReferencesSpecification* (a list of columns in the referenced table), it must correspond either to a unique constraint or to a primary key constraint in the referenced table. The *ReferencesSpecification* can omit the column list for the referenced table if that table has a declared primary key.



If there is no column list in the *ReferencesSpecification* and the referenced table has no primary key, a statement exception is thrown. (This means that if the referenced table has only unique keys, you must include a column list in the *ReferencesSpecification*.)

A foreign key constraint is satisfied if there is a matching value in the referenced unique or primary key column. If the foreign key consists of multiple columns, the foreign key value is considered `NULL` if any of its columns contains a `NULL`.

It is possible for a foreign key consisting of multiple columns to allow one of the columns to contain a value for which there is no matching value in the referenced columns, per the ANSI SQL standard. To avoid this situation, create `NOT NULL` constraints on all of the foreign key's columns.

## Foreign key constraints and DML

When you insert into or update a table with an enabled foreign key constraint, Splice Machine checks that the row does not violate the foreign key constraint by looking up the corresponding referenced key in the referenced table. If the constraint is not satisfied, Splice Machine rejects the insert or update with a statement exception.

When you update or delete a row in a table with a referenced key (a primary or unique constraint referenced by a foreign key), Splice Machine checks every foreign key constraint that references the key to make sure that the removal or modification of the row does not cause a constraint violation.

If removal or modification of the row would cause a constraint violation, the update or delete is not permitted and Splice Machine throws a statement exception.

Splice Machine performs constraint checks at the time the statement is executed, not when the transaction commits.

`PRIMARY KEY` constraints generate unique indexes. `FOREIGN KEY` constraints generate non-unique indexes.

`UNIQUE` constraints generate unique indexes if all the columns are non-nullable, and they generate non-unique indexes if one or more columns are nullable.

Therefore, if a column or set of columns has a `UNIQUE`, `PRIMARY KEY`, or `FOREIGN KEY` constraint on it, you do not need to create an index on those columns for performance. Splice Machine has already created it for you.

## Check constraints

You can use check constraints to limit which values are accepted by one or more columns in a table. You specify the constraint with a Boolean expression; if the expression evaluates to `true`, the value is allowed; if the expression evaluates to `false`, the constraint prevents the value from being entered into the database. The search condition is applied to each row that is modified on an `INSERT` or `UPDATE` at the time of the row modification. When a constraint is violated, the entire statement is aborted. You can apply check constraints at the column level or table level.

For example, you could specify that values in the salary column for the players on your team must be between \$250,000 and \$30,000,000 with this expression:

```
salary >= 250000 AND salary <= 30000000.
```

Any attempt to insert or update a record with a salary value out of that range would fail.

## Search Condition

A *searchCondition* is any [Boolean expression](#) that meets the requirements specified below. If a *constraint-Name* is not specified, Splice Machine generates a unique constraint name (for either column or table constraints).

### Requirements for search condition

If a check constraint is specified as part of a column-definition, a column reference can only be made to the same column. Check constraints specified as part of a table definition can have column references identifying columns previously defined in the [CREATE TABLE](#) statement.

The search condition must always return the same value if applied to the same values. Thus, it cannot contain any of the following:

- » Dynamic parameters
- » Date/Time Functions ([CURRENT\\_TIMESTAMP](#))
- » Subqueries
- » User Functions (such as [CURRENT\\_USER](#))

## Examples

```

-- column-level primary key constraint named OUT_TRAY_PK:
CREATE TABLE SAMP.OUT_TRAY
(
  SENT TIMESTAMP,
  DESTINATION CHAR(8),
  SUBJECT CHAR(64) NOT NULL CONSTRAINT
  OUT_TRAY_PK PRIMARY KEY,
  NOTE_TEXT VARCHAR(3000)
);

-- the table-level primary key definition allows you to
-- include two columns in the primary key definition:
CREATE TABLE SAMP.SCHED
(
  CLASS_CODE CHAR(7) NOT NULL,
  DAY SMALLINT NOT NULL,
  STARTING TIME,
  ENDING TIME,
  PRIMARY KEY (CLASS_CODE, DAY)
);

-- Use a column-level constraint for an arithmetic check
-- Use a table-level constraint
-- to make sure that a employee's taxes does not
-- exceed the bonus
CREATE TABLE SAMP.EMP
(
  EMPNO CHAR(6) NOT NULL CONSTRAINT EMP_PK PRIMARY KEY,
  FIRSTNME CHAR(12) NOT NULL,
  MIDINIT VARCHAR(12) NOT NULL,
  LASTNAME VARCHAR(15) NOT NULL,
  SALARY DECIMAL(9,2) CONSTRAINT SAL_CHK CHECK (SALARY >= 10000),
  BONUS DECIMAL(9,2),
  TAX DECIMAL(9,2),
  CONSTRAINT BONUS_CHK CHECK (BONUS > TAX)
);

-- use a check constraint to allow only appropriate
-- abbreviations for the meals
CREATE TABLE FLIGHTS
(
  FLIGHT_ID CHAR(6) NOT NULL ,
  SEGMENT_NUMBER INTEGER NOT NULL ,
  ORIG_AIRPORT CHAR(3),
  DEPART_TIME TIME,
  DEST_AIRPORT CHAR(3),
  ARRIVE_TIME TIME,
  MEAL CHAR(1) CONSTRAINT MEAL_CONSTRAINT
  CHECK (MEAL IN ('B', 'L', 'D', 'S')),
  PRIMARY KEY (FLIGHT_ID, SEGMENT_NUMBER)
);

```

## Statement dependency system

[INSERT](#) and [UPDATE](#) statements depend on all constraints on the target table.

[DELETE](#) statements depend on unique, primary key, and foreign key constraints.

These statements are invalidated if a constraint is added to or dropped from the target table.

## See Also

- » [ALTER TABLE](#) statement
- » [CREATE TABLE](#) statement
- » [INSERT](#) statement
- » [DELETE](#) statement
- » [Foreign Keys](#) in the *Developer's Guide*.
- » [Triggers](#) in the *Developer's Guide*.
- » [UPDATE](#) statement

# EXCEPT

The `EXCEPT` operator combines the result set of two or more similar `SELECT` queries, returning the results from the first query that do not appear in the results of the second query.

## Syntax

```
EXCEPT [ SELECT expression ] *
```

### *SELECT expression*

A `SELECT` expression that does not include an `ORDER BY` clause.

If you include an `ORDER BY` clause, that clause applies to the intersection operation.

### *DISTINCT*

(Optional). Indicates that only distinct (non-duplicate) rows from the queries are included. This is the default.

### *ALL*

(Optional). Indicates that all rows from the queries are included, including duplicates. With `ALL`, a row that has  $m$  duplicates in the left table and  $n$  duplicates in the right table will appear  $\max(m-n, 0)$  times in the result set.

## Usage

Each `SELECT` statement in the operation must contain the same number of columns, with similar data types, in the same order. Although the number, data types, and order of the fields in the select queries that you combine in an `EXCEPT` clause must correspond, you can use expressions, such as calculations or subqueries, to make them correspond.

When comparing column values for determining `DISTINCT` rows, two `NULL` values are considered equal.

## Results

A result set.

## Examples

```
CREATE TABLE t1( id INTEGER NOT NULL PRIMARY KEY,
                  i1 INTEGER, i2 INTEGER,
                  c10 char(10), c30 char(30), tm time);
```

```
CREATE TABLE t2( id INTEGER NOT NULL PRIMARY KEY,
                  i1 INTEGER, i2 INTEGER,
                  vc20 varchar(20), d double, dt date);
```

```
INSERT INTO t1(id,i1,i2,c10,c30) VALUES
(1,1,1,'a','123456789012345678901234567890'),
(2,1,2,'a','bb'),
(3,1,3,'b','bb'),
(4,1,3,'zz','5'),
(5,NULL,NULL,NULL,'1.0'),
(6,NULL,NULL,NULL,'a');
```

```
INSERT INTO t2(id,i1,i2,vc20,d) VALUES
(1,1,1,'a',1.0),
(2,1,2,'a',1.1),
(5,NULL,NULL,'12345678901234567890',3),
(100,1,3,'zz',3),
(101,1,2,'bb',NULL),
(102,5,5,'',NULL),
(103,1,3,' a',NULL),
(104,1,3,'NULL',7.4);
```

```
splice> SELECT id,i1,i2 FROM t1 EXCEPT SELECT id,i1,i2 FROM t2 ORDER BY id,i1,i2;
```

ID	I1	I2
4	1	3
3	1	3
6	NULL	NULL

3 rows selected

```
splice> SELECT i1,i2 FROM t1 EXCEPT SELECT i1,i2 FROM t2 where id = -1 ORDER BY 1,2;
```

I1	I2
NULL	NULL
1	3
1	2
1	1

4 rows selected

```
splice> SELECT i1,i2 FROM t1 where id = -1 EXCEPT SELECT i1,i2 FROM t2 ORDER BY 1,2;  
I1      |I2  
-----  
  
0 rows selected
```

## See Also

» [Union clause](#)



# FROM

The `FROM` clause is a mandatory clause in a [SelectExpression](#). It specifies the tables ([TableExpression](#)) from which the other clauses of the query can access columns for use in expressions.

## Syntax

```
FROM TableExpression [ , TableExpression ]*
```

### *TableExpression*

Specifies a table, view, or function; it is the source from which a [TableExpression](#) selects a result.

## Examples

```
SELECT Cities.city_id
FROM Cities
WHERE city_id < 5;

-- other types of TableExpressions
SELECT TABLENAME, ISINDEX
FROM SYS.SYSTABLES T, SYS.SYSCONGLOMERATES C
WHERE T.TABLEID = C.TABLEID
ORDER BY TABLENAME, ISINDEX;

-- force the join order
SELECT *
FROM Flights, FlightAvailability
WHERE FlightAvailability.flight_id = Flights.flight_id
AND FlightAvailability.segment_number = Flights.segment_number
AND Flights.flight_id < 'AA1115';

-- a TableExpression can be a joinOperation. Therefore
-- you can have multiple join operations in a FROM clause
SELECT COUNTRIES.COUNTRY, CITIES.CITY_NAME,
       FLIGHTS.DEST_AIRPORT
FROM COUNTRIES LEFT OUTER JOIN CITIES
ON COUNTRIES.COUNTRY_ISO_CODE = CITIES.COUNTRY_ISO_CODE
LEFT OUTER JOIN FLIGHTS
ON Cities.AIRPORT = FLIGHTS.DEST_AIRPORT;
```

## See Also

» [SELECT](#) expression

» [TABLE](#) expression

# GROUP BY

A `GROUP BY` clause is part of a [SelectExpression](#), that groups a result into subsets that have matching values for one or more columns. In each group, no two rows have the same value for the grouping column or columns. NULLs are considered equivalent for grouping purposes.

You typically use a `GROUP BY` clause in conjunction with an aggregate expression.

Using the `ROLLUP` syntax, you can specify that multiple levels of grouping should be computed at once.

## Syntax

```
GROUP BY
{
  column-Name-or-Position ]* |
  ROLLUP ( column-Name-or-Position
           [ , column-Name-or-Position ]* )
}
```

### *column-Name-or-Position*

Must be either the name or position of a column from the current scope of the query; there can be no columns from a query block outside the current scope. For example, if a `GROUP BY` clause is in a subquery, it cannot refer to columns in the outer query.

## Usage Notes

*SelectItems* in the [SelectExpression](#) with a `GROUP BY` clause must contain only aggregates or grouping columns.

## Examples

### Create our Test Table:

```
CREATE TABLE Test1
(
  TRACK_SEQ VARCHAR(40),
  TRACK_CD VARCHAR(18),
  REC_SEQ_NBR BIGINT,
  INDIV_ID BIGINT,
  BIZ_ID BIGINT,
  ADDR_ID BIGINT,
  HH_ID BIGINT,
  TRIAD_CB_DT DATE
);
```

Populate our Test Table:

```
CREATE TABLE Test1
INSERT INTO Test1 VALUES
  ('1','A',1,1,1,1,1,'2017-07-01'),
  ('1','A',1,1,2,2,2,'2017-07-02'),
  ('3','C',3,1,3,3,3,'2017-07-03'),
  ('1','A',1,2,1,1,1,'2017-07-01'),
  ('1','A',1,2,2,2,2,'2017-07-02'),
  ('3','C',3,2,3,3,3,'2017-07-03');
```

Example: Query Using Column Names:

```
SELECT indiv_id, track_seq, rec_seq_nbr, triad_cb_dt, ROW_NUMBER()
OVER (PARTITION BY indiv_id ORDER BY triad_cb_dt desc,rec_seq_nbr desc) AS ranking
FROM Test1
GROUP BY indiv_id,track_seq,rec_seq_nbr,triad_cb_dt;
INDIV_ID |TRACK_SEQ |REC_SEQ_NBR |TRIAD_CB_DT|RANKING
-----|-----|-----|-----|-----
1        |1         |1          |2017-07-01|3
1        |1         |1          |2017-07-02|2
1        |3         |3          |2017-07-03|1
2        |1         |1          |2017-07-01|3
2        |1         |1          |2017-07-02|2
2        |3         |3          |2017-07-03|1
6 rows selected
```

Example: Query Using Column Positions:

```
SELECT indiv_id, track_seq, rec_seq_nbr, triad_cb_dt, ROW_NUMBER()
OVER (PARTITION BY indiv_id ORDER BY triad_cb_dt desc,rec_seq_nbr desc) AS ranking
FROM Test1
GROUP BY 1,2,3,4;
INDIV_ID |TRACK_SEQ |REC_SEQ_NBR |TRIAD_CB_DT|RANKING
-----|-----|-----|-----|-----
1        |1         |1          |2017-07-01|3
1        |1         |1          |2017-07-02|2
1        |3         |3          |2017-07-03|1
2        |1         |1          |2017-07-01|3
2        |1         |1          |2017-07-02|2
2        |3         |3          |2017-07-03|1
6 rows selected
```

See Also

>> [SELECT](#) expression

# HAVING

A **HAVING** clause restricts the results of a `SelectExpression`.

The **HAVING** clause is applied to each group of the grouped table, similarly to how a [WHERE](#) clause is applied to a select list.

If there is no `GROUP BY` clause, the **HAVING** clause is applied to the entire result as a single group. The [SELECT](#) expression cannot refer directly to any column that does not have a `GROUP BY` clause. It can, however, refer to constants, aggregates, and special registers.

## Syntax

```
HAVING searchCondition
```

*searchCondition*

A specialized Boolean expression, as described in the next section.

## Using

The *searchCondition*, is a specialized [booleanExpression](#) that can contain only;

- » grouping columns (see [GROUP BY](#) clause)
- » columns that are part of aggregate expressions
- » columns that are part of a subquery

For example, the following query is illegal, because the column `SALARY` is not a grouping column, it does not appear within an aggregate, and it is not within a subquery:

```
SELECT COUNT(*)  
FROM SAMP.STAFF  
GROUP BY ID  
HAVING SALARY > 15000;
```

Aggregates in the **HAVING** clause do not need to appear in the `SELECT` list. If the **HAVING** clause contains a subquery, the subquery can refer to the outer query block if and only if it refers to a grouping column.

## Example

```
-- Find the total number of economy seats taken on a flight,  
-- grouped by airline,  
-- only when the group has at least 2 records.  
SELECT SUM(ECONOMY_SEATS_TAKEN), AIRLINE_FULL  
FROM FLIGHTAVAILABILITY, AIRLINES  
WHERE SUBSTR(FLIGHTAVAILABILITY.FLIGHT_ID, 1, 2) = AIRLINE  
GROUP BY AIRLINE_FULL  
HAVING COUNT(*) > 1;
```

## See Also

- » [SELECT](#) expression
- » [GROUP BY](#) clause
- » [WHERE](#) clause

## LIMIT n

A `LIMIT n` clause, limits the results of a query to a specified number of records.

### Syntax

```
'{ ' LIMIT {count} ' } '
```

**NOTE:** You must surround the `LIMIT` clause with left and right curly brackets (`{` and `}`).

*count*

An integer value specifying the maximum number of rows to return from the query.

## Examples

```
splice> select * from limittest order by a;
A |B |C |D
```

```
-----
a1 |b1 |c1 |d1
a2 |b2 |c2 |d2
a3 |b3 |c3 |d3
a4 |b4 |c4 |d4
a5 |b5 |c5 |d5
a6 |b6 |c6 |d6
a7 |b7 |c7 |d7
a8 |b8 |c8 |d8
8 rows selected
```

```
splice> select * from limittest order by a {LIMIT 1};
A |B |C |D
```

```
-----
a1 |b1 |c1 |d1
1 row selected
```

```
splice> select * from limittest order by a {LIMIT 3};
A |B |C |D
```

```
-----
a1 |b1 |c1 |d1
a2 |b2 |c2 |d2
a3 |b3 |c3 |d3
3 rows selected
```

```
splice> select * from limittest order by a {LIMIT 10};
A |B |C |D
```

```
-----
a1 |b1 |c1 |d1
a2 |b2 |c2 |d2
a3 |b3 |c3 |d3
a4 |b4 |c4 |d4
a5 |b5 |c5 |d5
a6 |b6 |c6 |d6
a7 |b7 |c7 |d7
a8 |b8 |c8 |d8
8 rows selected
```

## See Also

- » [RESULT OFFSET](#) clause
- » [SELECT](#) expression



» [TOP n](#) clause

# ORDER BY

The `ORDER BY` clause is an optional element of the following:

- » A [SELECT statement](#)
- » A [SelectExpression](#)
- » A [VALUES expression](#)
- » A [ScalarSubquery](#)
- » A [TableSubquery](#)

It can also be used in an [CREATE VIEW](#) statement.

An `ORDER BY` clause allows you to specify the order in which rows appear in the result set. In subqueries, the `ORDER BY` clause is meaningless unless it is accompanied by one or both of the `result offset` and `fetch first` clauses or in conjunction with the `ROW_NUMBER` function, since there is no guarantee that the order is retained in the outer result set. It is permissible to combine `ORDER BY` on the outer query with `ORDER BY` in subqueries.

## Syntax

```
ORDER BY { column-Name |
           ColumnPosition |
           Expression }
[ ASC | DESC ]
[ , column-Name | ColumnPosition | Expression
  [ ASC | DESC ]
  [ NULLS FIRST | NULLS LAST ]
] *
```

### *column-Name*

A column name, as described in the [SELECT](#) statement. The column name(s) that you specify in the `ORDER BY` clause do not need to be the `SELECT` list.

### *ColumnPosition*

An integer that identifies the number of the column in the `SelectItems` in the underlying query of the `SELECT` statement. `ColumnPosition` must be greater than 0 and not greater than the number of columns in the result table. In other words, if you want to order by a column, that column must be specified in the `SELECT` list.

### *Expression*

A sort key expression, such as numeric, string, and datetime expressions. *Expression* can also be a row value expression such as a scalar subquery or case expression.

### *ASC*

Specifies that the results should be returned in ascending order. If the order is not specified, `ASC` is the default.

### *DESC*

Specifies that the results should be returned in descending order.

#### ***NULLS FIRST***

Specifies that `NULL` values should be returned before non-`NULL` values. This is the default value for descending (`DESC`) order.

#### ***NULLS LAST***

Specifies that `NULL` values should be returned after non-`NULL` values. This is the default value for ascending (`ASC`) order.

## Using

If `SELECT DISTINCT` is specified or if the `SELECT` statement contains a `GROUP BY` clause, the `ORDER BY` columns must be in the `SELECT` list.

## Example using a correlation name

You can sort the result set by a correlation name, if the correlation name is specified in the select list. For example, to return from the `CITIES` database all of the entries in the `CITY_NAME` and `COUNTRY` columns, where the `COUNTRY` column has the correlation name `NATION`, you specify this `SELECT` statement:

```
SELECT CITY_NAME, COUNTRY AS NATION
FROM CITIES
ORDER BY NATION;
```

## Example using a numeric expression

You can sort the result set by a numeric expression, for example:

```
SELECT name, salary, bonus FROM employee
ORDER BY salary+bonus;
```

In this example, the `salary` and `bonus` columns are `DECIMAL` data types.

## Example using a function

You can sort the result set by invoking a function, for example:

```
SELECT i, len FROM measures
ORDER BY sin(i);
```

## Example of specifying a NULL ordering

You can sort the result set by invoking a function, for example:

```
SELECT * FROM Players
ORDER BY BirthDate DESC NULLS LAST;
```

## See Also

- » [GROUP BY](#) clause
- » [WHERE](#) clause
- » [SELECT](#) expression
- » [VALUES](#) expression
- » [CREATE VIEW](#) statement
- » [INSERT](#) statement
- » [SELECT](#) statement

# OVER

The `OVER` clause is used in window functions to define the window on which the function operates. Window functions are permitted only in the [ORDER BY](#) clause of queries.

For general information about and examples of Window functions in Splice Machine, see the [Using Window Functions](#) topic.

## Syntax

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

### *expression*

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

### *partitionClause*

Optional. Specifies how the window function is broken down over groups, in the same way that `GROUP BY` specifies groupings for regular aggregate functions. If you omit this clause, there is one partition that contains all rows.

The syntax for this clause is essentially the same as for the [GROUP BY](#) clause for queries; To recap:

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

### *expression [,...]*

A list of expressions that define the partitioning.

### *orderClause*

Optional. Controls the ordering. It is important for ranking functions, since it specifies by which variables ranking is performed. It is also needed for cumulative functions. The syntax for this clause is essentially the same as for the [SQL Reference](#). To recap:

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

**NOTE:** The default ordering is ascending (ASC). For ascending order, NULL values are returned last unless you specify `NULLS FIRST`; for descending order, NULL values are returned first unless you specify `NULLS LAST`.

### *frameClause*

Optional. Defines which of the rows (which *frame*) that are passed to the window function should be included in the computation. The *frameClause* provides two offsets that determine the start and end of the frame.

The syntax for the frame clause is:

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

The syntax for both *frameStart* and *frameEnd* is:

```
UNBOUNDED PRECEDING |
<n> PRECEDING        |
CURRENT ROW          |
<n> FOLLOWING         |
UNBOUNDED FOLLOWING
```

*<n>*

A non-negative integer value.

## Usage Restrictions

Because window functions are only allowed in [HAVING](#) clauses, 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 *, rank() OVER (PARTITION BY "playerID" ORDER BY "G") as rank
FROM Batting
WHERE rank = 1;
```

Instead, you need to use a subquery:

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

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

## Simple Window Function Examples

The examples in this section are fairly simple because they don't use the frame clause.

```
--- Rank each year within a player by the number of home runs hit by that player
RANK() OVER (PARTITION BY playerID ORDER BY desc(H));

--- Compute the change in number of games played from one year to the next:
G - LAG(G) OVER (PARTITION G playerID ORDER BY yearID);
```

## Examples with Frame Clauses

The frame clause can be confusing, given all of the options that it presents. There are three commonly used frame clauses:

Frame Clause Type	Example
<i>Recycled</i>	BETWEEN UNBOUNDED PRECEDING AND UNBOUNDED FOLLOWING
<i>Cumulative</i>	BETWEEN UNBOUNDED PRECEDING AND CURRENT ROW
<i>Rolling</i>	BETWEEN 2 PRECEDING AND 2 FOLLOWING

Here are some examples of window functions using frame clauses:

```
--- Compute the running sum of G for each player:
SUM(G) OVER (PARTITION BY playerID ORDER BY yearID
  BETWEEN UNBOUNDED PRECEDING AND CURRENT ROW);

--- Compute the career year:
YearID - min(YEARID) OVER (PARTITION BY playerID
  BETWEEN UNBOUNDED PRECEDING AND UNBOUNDED FOLLOWING) + 1;

--- Compute a rolling average of games by player:
MEAN(G) OVER (PARTITION BY playerID ORDER BY yearID
  BETWEEN 2 PRECEDING AND 2 FOLLOWING);
```

## See Also

- » [Window and Aggregate](#) functions
- » [SELECT](#) expression

- » [HAVING](#) clause
- » [ORDER BY](#) clause
- » [WHERE](#) clause
- » The [Using Window Functions](#) section in our *Splice Machine Developer's Guide*



# RESULT OFFSET and FETCH FIRST

The *result offset clause* provides a way to skip the N first rows in a result set before starting to return any rows.

The *fetch first clause*, which can be combined with the *result offset clause*, limits the number of rows returned in the result set. The *fetch first clause* can sometimes be useful for retrieving only a few rows from an otherwise large result set, usually in combination with an `ORDER BY` clause. Use of this clause can increase efficiency and make programming simpler.

## Syntax

```
OFFSET { integer-literal | ? }
      {ROW | ROWS}
```

### *integer-literal*

An integer value that specifies the number of rows to skip. The default value is 0.

If non-zero, this must be a positive integer value. If you specify a value greater than the number of rows in the underlying result set, no rows are returned.

```
FETCH { FIRST | NEXT }
      [integer-literal | ? ]
      {ROW | ROWS} ONLY
```

### *integer-literal*

An integer value that specifies the maximum number of rows to return in the result set. The default value is 1.

This must be a positive integer value greater than or equal to 1.

## Usage

Note that:

- » `ROW` and `ROWS` are synonymous
- » `FIRST` and `NEXT` are synonymous

Be sure to specify the `ORDER BY` clause if you expect to retrieve a sorted result set.

## Examples

```
-- Fetch the first row of T
SELECT * FROM T FETCH FIRST ROW ONLY;

-- Sort T using column I, then fetch rows 11 through 20
--   of the sorted rows (inclusive)
SELECT * FROM T ORDER BY I
      OFFSET 10 ROWS
      FETCH NEXT 10 ROWS ONLY;

-- Skip the first 100 rows of T
-- If the table has fewer than 101 records,
--   an empty result set is returned
SELECT * FROM T OFFSET 100 ROWS;

-- Use of ORDER BY and FETCH FIRST in a subquery
SELECT DISTINCT A.ORIG_AIRPORT, B.FLIGHT_ID FROM
  (SELECT FLIGHT_ID, ORIG_AIRPORT
   FROM FLIGHTS
   ORDER BY ORIG_AIRPORT DESC
   FETCH FIRST 40 ROWS ONLY)
  AS A, FLIGHTAVAILABILITY AS B
WHERE A.FLIGHT_ID = B.FLIGHT_ID;

-- JDBC (using a dynamic parameter):
PreparedStatement p =
  con.prepareStatement("SELECT * FROM T
                      ORDER BY I
                      OFFSET ? ROWS");

p.setInt(1, 100);
ResultSet rs = p.executeQuery();
```

## See Also

- » [LIMIT n](#) clause
- » [SELECT](#) statement
- » [TOP n](#) clause

# TOP n

A `TOP` clause, also called the `TOP n` clause, limits the results of a query to the first `n` result records.

## Syntax

```
TOP [number] column-Name *
```

### *number*

Optional. An integer value that specifies the maximum number of rows to return from the query. If you omit this parameter, the default value of 1 is used.

### *column-Name*

A column name, as described in the [Column Name](#) topic.

You can specify `*` as the column name to represent all columns.

## Examples

```
splice> select * from toptest order by a;
```

```
A  |B  |C  |D
```

```
-----
a1 |b1 |c1 |d1
a2 |b2 |c2 |d2
a3 |b3 |c3 |d3
a4 |b4 |c4 |d4
a5 |b5 |c5 |d5
a6 |b6 |c6 |d6
a7 |b7 |c7 |d7
a8 |b8 |c8 |d8
8 rows selected
```

```
splice> select top * from toptest order by a;
```

```
A  |B  |C  |D
```

```
-----
a1 |b1 |c1 |d1
1 row selected
```

```
splice> select top 3 a, b, c from toptest order by a;
```

```
A  |B  |C
```

```
-----
a1 |b1 |c1
a2 |b2 |c2
a3 |b3 |c3
3 rows selected
```

```
splice> select top 10 a, b from toptest order by a;
```

```
A  |B
```

```
-----
a1 |b1
a2 |b2
a3 |b3
a4 |b4
a5 |b5
a6 |b6
a7 |b7
a8 |b8
8 rows selected
```

```
splice> select top 4 * from toptest order by a offset 1 row;
```

```
A  |B  |C  |D
```

```
-----
a2 |b2 |c2 |d2
a3 |b3 |c3 |d3
a4 |b4 |c4 |d4
a5 |b5 |c5 |d5
4 rows selected
```

```
splice> select top 4 * from toptest order by a offset 2 row;
```

```
A |B |C |D
```

```
-----
```

```
a3 |b3 |c3 |d3
```

```
a4 |b4 |c4 |d4
```

```
a5 |b5 |c5 |d5
```

```
a6 |b6 |c6 |d6
```

```
4 rows selected
```

```
splice> select top 4 * from toptest order by a offset -1 row ;
```

```
ERROR 2201X: Invalid row count for OFFSET, must be >= 0.
```

```
splice> select top 4 * from toptest order by a offset 10 row;
```

```
A |B |C |D
```

```
-----
```

```
0 rows selected
```

```
splice> select top -1 * from toptest;
```

```
ERROR 2201W: Row count for FIRST/NEXT/TOP must be >= 1 and row count for LIMIT must be >= 0.
```

## See Also

- » [LIMIT n](#) clause
- » [RESULT OFFSET](#) clause
- » [SELECT](#) expression

# UNION

The `UNION` operator combines the result set of two or more similar `SELECT` queries, and returns distinct rows.

## Syntax

```
SELECT expression
```

### *SELECT expression*

A `SELECT` expression that does not include an `ORDER BY` clause.

If you include an `ORDER BY` clause, that clause applies to the intersection operation.

### *DISTINCT*

(Optional). Indicates that only distinct (non-duplicate) rows from the queries are included. This is the default.

### *ALL*

(Optional). Indicates that all rows from the queries are included, including duplicates.

## Usage

Each `SELECT` statement in the union must contain the same number of columns, with similar data types, in the same order. Although the number, data types, and order of the fields in the select queries that you combine in a `UNION` clause must correspond, you can use expressions, such as calculations or subqueries, to make them correspond.

Each `UNION` keyword combines the `SELECT` statements that immediately precede and follow it. If you use the `ALL` keyword with some of the `UNION` keywords in your query, but not with others, the results will include duplicate rows from the pairs of `SELECT` statements that are combined by using `UNION ALL`, but will not include duplicate rows from the `SELECT` statements that are combined by using `UNION` without the `ALL` keyword.

## Results

A result set.

# Examples

```
CREATE TABLE t1( id INTEGER NOT NULL PRIMARY KEY,
                  i1 INTEGER, i2 INTEGER,
                  c10 char(10), c30 char(30), tm time);

CREATE TABLE t2( id INTEGER NOT NULL PRIMARY KEY,
                  i1 INTEGER, i2 INTEGER,
                  vc20 varchar(20), d double, dt date);

INSERT INTO t1(id,i1,i2,c10,c30) VALUES
  (1,1,1,'a','123456789012345678901234567890'),
  (2,1,2,'a','bb'),
  (3,1,3,'b','bb'),
  (4,1,3,'zz','5'),
  (5,NULL,NULL,NULL,'1.0'),
  (6,NULL,NULL,NULL,'a');

INSERT INTO t2(id,i1,i2,vc20,d) VALUES
  (1,1,1,'a',1.0),
  (2,1,2,'a',1.1),
  (5,NULL,NULL,'12345678901234567890',3),
  (100,1,3,'zz',3),
  (101,1,2,'bb',NULL),
  (102,5,5,'',NULL),
  (103,1,3,' a',NULL),
  (104,1,3,'NULL',7.4);
```

```
splice> SELECT id,i1,i2 FROM t1 UNIONSELECT id,i1,i2 FROM t2 ORDER BY id,i1,i2;
ID          |I1          |I2
-----1          |1          |1
2           |1           |2
3           |1           |3
4           |1           |3
5           |NULL        |NULL
6           |NULL        |NULL
100         |1           |3
101         |1           |2
102         |5           |5
103         |1           |3
104         |1           |3

11 rows selected
```



```
splice> SELECT id,i1,i2 FROM t1 UNION ALLSELECT id,i1,i2 FROM t2 ORDER BY id,i1,i2;
```

ID	I1	I2
1	1	1
1	1	1
2	1	2
2	1	2
3	1	3
4	1	3
5	NULL	NULL
5	NULL	NULL
6	NULL	NULL
100	1	3
101	1	2
102	5	5
103	1	3
104	1	3

14 rows selected

## See Also

» [Except clause](#)

# USING

The `USING` clause specifies which columns to test for equality when two tables are joined. It can be used instead of an `ON` clause in `JOIN` operations that have an explicit join clause.

## Syntax

```
USING ( [ Simple-column-Name ] * )
```

### *SimpleColumnName*

The name of a table column, as described in the [Simple Column Name](#) topic.

## Using

The columns listed in the `USING` clause must be present in both of the tables being joined. The `USING` clause will be transformed to an `ON` clause that checks for equality between the named columns in the two tables.

When a `USING` clause is specified, an asterisk (\*) in the select list of the query will be expanded to the following list of columns (in this order):

- » All the columns in the `USING` clause
- » All the columns of the first (left) table that are not specified in the `USING` clause
- » All the columns of the second (right) table that are not specified in the `USING` clause

An asterisk qualified by a table name (for example, `COUNTRIES.*`) will be expanded to every column of that table that is not listed in the `USING` clause.

If a column in the `USING` clause is referenced without being qualified by a table name, the column reference points to the column in the first (left) table if the join is a [LEFT OUTER JOIN](#). If it is a [RIGHT OUTER JOIN](#), unqualified references to a column in the `USING` clause point to the column in the second (right) table.

## Examples

The following query performs an inner join between the `COUNTRIES` table and the `CITIES` table on the condition that `COUNTRIES.COUNTRY` is equal to `CITIES.COUNTRY`:

```
SELECT * FROM COUNTRIES JOIN CITIES
  USING (COUNTRY);
```

The next query is similar to the one above, but it has the additional join condition that `COUNTRIES.COUNTRY_ISO_CODE` is equal to `CITIES.COUNTRY_ISO_CODE`:

```
SELECT * FROM COUNTRIES JOIN CITIES  
  USING (COUNTRY, COUNTRY_ISO_CODE);
```

## See Also

- » [Join Operations](#)
- » [SELECT](#) statement

# WHERE

The `WHERE` clause is an optional part of an [UPDATE](#) statement.

The `WHERE` clause lets you select rows based on a Boolean expression. Only rows for which the expression evaluates to `TRUE` are selected to return or operate upon (delete or update).

## Syntax

```
WHERE BooleanExpression
```

### *BooleanExpression*

A Boolean expression. For more information, see the [Boolean Expressions](#) topic.

## Example

```
-- find the flights where no business-class seats have been booked
SELECT *
FROM FlightAvailability
WHERE business_seats_taken IS NULL
      OR business_seats_taken = 0;

-- Join the EMP_ACT and EMPLOYEE tables
-- select all the columns from the EMP_ACT table and
-- add the employee's surname (LASTNAME) from the EMPLOYEE table
-- to each row of the result.
SELECT SAMP.EMP_ACT.*, LASTNAME
FROM SAMP.EMP_ACT, SAMP.EMPLOYEE
WHERE EMP_ACT.EMPNO = EMPLOYEE.EMPNO;

-- Determine the employee number and salary of sales representatives
-- along with the average salary and head count of their departments.
-- This query must first create a new-column-name specified in the AS clause
-- which is outside the fullselect (DINFO)
-- in order to get the AVGSALARY and EMPCOUNT columns,
-- as well as the DEPTNO column that is used in the WHERE clause
SELECT THIS_EMP.EMPNO, THIS_EMP.SALARY, DINFO.AVGSALARY, DINFO.EMPCOUNT
FROM EMPLOYEE THIS_EMP,
     (SELECT OTHERS.WORKDEPT AS DEPTNO,
        AVG(OTHERS.SALARY) AS AVGSALARY,
        COUNT(*) AS EMPCOUNT
     FROM EMPLOYEE OTHERS
     GROUP BY OTHERS.WORKDEPT
     )AS DINFO
WHERE THIS_EMP.JOB = 'SALESREP'
      AND THIS_EMP.WORKDEPT = DINFO.DEPTNO;
```

## See Also

- » [Select](#) expressions
- » [DELETE](#) statement
- » [SELECT](#) statement
- » [UPDATE](#) statement

## WITH CLAUSE (Common Table Expression)

You can use Common Table Expressions, also known as the `WITH` clause, to break down complicated queries into simpler parts by naming and referring to subqueries within queries.

A Common Table Expression (CTE) provides a way of defining a temporary result set whose definition is available only to the query in which the CTE is defined. The result of the CTE is not stored; it exists only for the duration of the query. CTEs are helpful in reducing query complexity and increasing readability. They can be used as substitutions for views in cases where either you don't have permission to create a view or the query would be the only one using the view. CTEs allow you to more easily enable grouping by a column that is derived from a scalar sub select or a function that is non deterministic.

**NOTE:** The `WITH` clause is also known as the *subquery factoring clause*.

The handling and syntax of `WITH` queries are similar to the handling and syntax of views. The `WITH` clause can be processed as an inline view and shares syntax with `CREATE VIEW`. The `WITH` clause can also resolve as a temporary table, which may enhance the efficiency of a query.

### Syntax

```
WITH queryName
  AS SELECT Query
```

*queryName*

An identifier that names the subquery clause.

### Restrictions

You cannot currently use a temporary table in a `WITH` clause. This is being addressed in a future release of Splice Machine.

### Examples

If we create the following table:

```
CREATE TABLE BANKS (
  INSTITUTION_ID INTEGER NOT NULL,
  INSTITUTION_NAME VARCHAR(100),
  CITY VARCHAR(100),
  STATE VARCHAR(2),
  TOTAL_ASSETS DECIMAL(19,2),
  NET_INCOME DECIMAL(19,2),
  OFFICES INTEGER,
  PRIMARY KEY(INSTITUTION_ID)
);
```

We can then use a common table expression to improve the readability of a statement that finds the per-city total assets and income for the states with the top net income:

```
WITH state_sales AS (  
    SELECT STATE, SUM(NET_INCOME) AS total_sales  
    FROM BANKS  
    GROUP BY STATE  
) , top_states AS (  
    SELECT STATE  
    FROM state_sales  
    WHERE total_sales > (SELECT SUM(total_sales)/10 FROM state_sales)  
)  
SELECT STATE,  
    CITY,  
    SUM(TOTAL_ASSETS) AS assets,  
    SUM(NET_INCOME) AS income  
FROM BANKS  
WHERE STATE IN (SELECT STATE FROM top_states)  
GROUP BY STATE, CITY;
```

## See Also

- » [SELECT](#) expression
- » [Query](#)

# Expressions

This section contains the reference documentation for the Splice Machine SQL Expressions, in the following topics:

Topic	Description
<a href="#">About Expressions</a>	Overview of expression syntax and rules.
<a href="#">Boolean Expressions</a>	Syntax for and examples of <code>Boolean</code> expressions.
<a href="#">CASE Expression</a>	Syntax for and examples of <code>CASE</code> expressions.
<a href="#">Dynamic Parameters</a>	Description of using dynamic parameters in expressions in prepared statements.
<a href="#">Expression Precedence</a>	Specifies operator precedence in expressions.
<a href="#">NEXT VALUE FOR Expression</a>	Retrieves the next value from a sequence generator.
<a href="#">SELECT Expression</a>	Builds a table value based on filtering and projecting values from other tables.
<a href="#">TABLE Expression</a>	Specifies a table, view, or function in a <a href="#">FROM clause</a> .
<a href="#">VALUES Expression</a>	Constructs a row or a table from other values.



For access to the source code for the Community Edition of Splice Machine, visit [our open source GitHub repository](#).



## About Expressions

Syntax for many statements and expressions includes the term *Expression*, or a term for a specific kind of expression such as TableSubquery. Expressions are allowed in these specified places within statements.

Some locations allow only a specific type of expression or one with a specific property. If not otherwise specified, an expression is permitted anywhere the word *Expression* appears in the syntax. This includes:

- » [ORDER BY clause](#)
- » [SelectExpression](#)
- » [UPDATE statement](#) (SET portion)
- » [VALUES Expression](#)
- » [WHERE clause](#)

Of course, many other statements include these elements as building blocks, and so allow expressions as part of these elements.

The following tables list all the possible SQL expressions and indicate where the expressions are allowed.

## General Expressions

General expressions are expressions that might result in a value of any type. The following table lists the types of general expressions.

Expression Type	Explanation
Column reference	<p>A <a href="#">column-Name</a> that references the value of the column made visible to the expression containing the Column reference.</p> <p>You must qualify the column-Name by the table name or correlation name if it is ambiguous.</p> <p>The qualifier of a column-Name must be the correlation name, if a correlation name is given to a table that is in a <a href="#">SelectExpressions</a>, UPDATE statements, and the WHERE clauses of data manipulation statements.</p>
Constant	Most built-in data types typically have constants associated with them (as shown in the Data types section).
NULL	<p>NULL is an untyped constant representing the unknown value.</p> <p>Allowed in <a href="#">CAST</a> expressions or in INSERT VALUES lists and UPDATE SET clauses. Using it in a CAST expression gives it a specific data type.</p>

Expression Type	Explanation
Dynamic parameter	<p>A dynamic parameter is a parameter to an SQL statement for which the value is not specified when the statement is created. Instead, the statement has a question mark (?) as a placeholder for each dynamic parameter. See <a href="#">Dynamic parameters</a>.</p> <p>Dynamic parameters are permitted only in prepared statements. You must specify values for them before the prepared statement is executed. The values specified must match the types expected.</p> <p>Allowed anywhere in an expression where the data type can be easily deduced. See <a href="#">Dynamic parameters</a>.</p>
CAST expression	Allows you to specify the type of NULL or of a dynamic parameter or convert a value to another type. See <a href="#">CAST function</a> .
Scalar subquery	Subquery that returns a single row with a single column. See <a href="#">ScalarSubquery</a> .
Table subquery	<p>Subquery that returns more than one column and more than one row. See <a href="#">TableSubquery</a>.</p> <p>Allowed as a tableExpression in a FROM clause and with EXISTS, IN, and quantified comparisons.</p>
Conditional expression	A conditional expression chooses an expression to evaluate based on a boolean test. Conditional expressions include the <a href="#">COALESCE function</a> .

## Boolean Expressions

[Boolean expressions](#) are expressions that result in boolean values. Most general expressions can result in boolean values. Boolean expressions commonly used in a WHERE clause are made of operands operated on by SQL operators.

## Numeric Expressions

Numeric expressions are expressions that result in numeric values. Most of the general expressions can result in numeric values. Numeric values have one of the following types:

- » BIGINT
- » DECIMAL
- » DOUBLE PRECISION
- » INTEGER
- » REAL
- » SMALLINT

The following table lists the types of numeric expressions.

Expression Type	Explanation
+, -, *, /, unary + and - expressions	<p>Evaluate the expected math operation on the operands. If both operands are the same type, the result type is not promoted, so the division operator on integers results in an integer that is the truncation of the actual numeric result. When types are mixed, they are promoted as described in the Data types section.</p> <p>Unary + is a noop (i.e., +4 is the same as 4).</p> <p>Unary - is the same as multiplying the value by -1, effectively changing its sign.</p>
AVG	<a href="#">AVG</a> function
SUM	<a href="#">SUM</a> function
LENGTH	<a href="#">LENGTH</a> function.
LOWER	<a href="#">LOWER</a> function.
COUNT	<a href="#">COUNT</a> function, including COUNT ( * ) .

## Character expressions

Character expressions are expressions that result in a `CHAR` or `VARCHAR` value. Most general expressions can result in a `CHAR` or `VARCHAR` value. The following table lists the types of character expressions.

Expression Type	Explanation
A <code>CHAR</code> or <code>VARCHAR</code> value that uses wildcards.	The wildcards <code>%</code> and <code>_</code> make a character string a pattern against which the <code>LIKE</code> operator can look for a match.
Concatenation expression	In a concatenation expression, the concatenation operator, <code>  </code> , concatenates its right operand to the end of its left operand. Operates on character and bit strings. See <a href="#">Concatenation operator</a> .
Built-in string functions	The built-in string functions act on a String and return a string. See <a href="#">UCASE or UPPER function</a> .

## Date and Time Expressions

A date or time expression results in a `DATE`, `TIME`, or `TIMESTAMP` value. Most of the general expressions can result in a date or time value. The following table lists the types of date and time expressions.

Expression Type	Explanation
<code>CURRENT_DATE</code>	Returns the current date. See the <a href="#">CURRENT_DATE</a> function.
<code>CURRENT_TIME</code>	Returns the current time. See the <a href="#">CURRENT_TIME</a> function.
<code>CURRENT_TIMESTAMP</code>	Returns the current timestamp. See the <a href="#">CURRENT_TIMESTAMP</a> function.

## See Also

- » [AVG](#) function
- » [CAST](#) function
- » [COUNT](#) function
- » [CURRENT\\_DATE](#) function
- » [CURRENT\\_TIME](#) function
- » [CURRENT\\_TIMESTAMP](#) function
- » [Concatenation](#) operator
- » [LCASE](#) function
- » [LENGTH](#) function
- » [LTRIM](#) function
- » [ORDER BY](#) clause
- » [RTRIM](#) function
- » [SUBSTR](#) function
- » [SUM](#) function
- » [Select](#) expression
- » [TRIM](#) function
- » [UPDATE](#) statement
- » [VALUES](#) expression
- » [WHERE](#) clause

# Boolean Expressions

Boolean expressions are allowed in [CONSTRAINT clause](#) for more information. Boolean expressions in a WHERE clause have a highly liberal syntax; see WHERE clause, for example.

A Boolean expression can include zero or more Boolean operators.

## Syntax

The following table shows the syntax for the Boolean operators

Operator	Syntax
AND, OR, NOT	<pre>{   Expression AND Expression   Expression OR  Expression   NOT Expression }</pre>
Comparisons	<pre>Expression {   &lt;   =   &gt;   &lt;=   &gt;=   &lt;&gt; }</pre>
IS NULL, IS NOT NULL	<pre>Expression IS [ NOT ] NULL</pre>
LIKE	<pre>CharacterExpression [ NOT ] LIKE CharacterExpression WithWildCard [ ESCAPE 'escapeCharacter']</pre>
BETWEEN	<pre>Expression [ NOT ] BETWEEN Expression AND Express ion</pre>

Operator	Syntax
IN	<pre>{   Expression [ NOT ] IN <a href="#">TableSubquery</a>     Expression [ NOT ] IN   ( Expression [, Expression ]* ) }</pre>
EXISTS	<pre>[NOT] EXISTS <a href="#">TableSubquery</a></pre>
Quantified comparison	<pre>Expression ComparisonOperator {   ALL     ANY     SOME } <a href="#">TableSubquery</a></pre>

## Examples

The following example presents examples of the Boolean operators.

Operator	Explanation and Example
AND, OR, NOT	<p>Evaluate any operand(s) that are boolean expressions:</p> <pre>(orig_airport = 'SFO') OR (dest_airport = 'GR U') -- returns true</pre>
Comparisons	<p>&lt;, =, &gt;, &lt;=, &gt;=, &lt;&gt; are applicable to all of the built-in types.</p> <pre>DATE('1998-02-26') &lt; DATE('1998-03-01') -- returns true</pre> <div><b>NOTE:</b> Splice Machine also accepts the != operator, which is not included in the SQL standard.</div>

Operator	Explanation and Example
IS NULL, IS NOT NULL	<p>Test whether the result of an expression is null or not.</p> <pre>WHERE MiddleName IS NULL</pre>
LIKE	<p>Attempts to match a character expression to a character pattern, which is a character string that includes one or more wildcards.</p> <p>% matches any number (zero or more) of characters in the corresponding position in first character expression.</p> <p>_ matches one character in the corresponding position in the character expression.</p> <p>Any other character matches only that character in the corresponding position in the character expression.</p> <pre>city LIKE 'Sant_'</pre> <p>To treat % or _ as constant characters, escape the character with an optional escape character, which you specify with the ESCAPE clause.</p> <pre>SELECT a FROM tabA WHERE a LIKE '%=_ ' ESCAPE '='</pre> <div><p><b>NOTE:</b> When LIKE comparisons are used, Splice Machine compares one character at a time for non-metacharacters. This is different than the way Splice Machine processes = comparisons. The comparisons with the = operator compare the entire character string on left side of the = operator with the entire character string on the right side of the = operator.</p></div>

Operator	Explanation and Example
BETWEEN	<p>Tests whether the first operand is between the second and third operands. The second operand must be less than the third operand. Applicable only to types to which <code>&lt;=</code> and <code>&gt;=</code> can be applied.</p> <pre>WHERE booking_date       BETWEEN DATE('1998-02-26')       AND DATE('1998-03-01')</pre> <p><b>NOTE:</b> Using the <code>BETWEEN</code> operator is logically equivalent to specifying that you want to select values that are greater than or equal to the first operand and less than or equal to the second operand: <code>col between X and Y</code> is equivalent to <code>col &gt;= X and col &lt;= Y</code>. Which means that the result set will be empty if your second operand is less than your first.</p>
IN	<p>Operates on table subquery or list of values. Returns <code>TRUE</code> if the left expression's value is in the result of the table subquery or in the list of values. Table subquery can return multiple rows but must return a single column.</p> <pre>WHERE booking_date NOT IN       (SELECT booking_date        FROM HotelBookings        WHERE rooms_available = 0)</pre>
EXISTS	<p>Operates on a table subquery. Returns <code>TRUE</code> if the table subquery returns any rows, and <code>FALSE</code> if it returns no rows. A table subquery can return multiple columns and rows.</p> <pre>WHERE EXISTS       (SELECT *        FROM Flights        WHERE dest_airport = 'SFO'        AND orig_airport = 'GRU')</pre>



Operator	Explanation and Example
Quantified comparison	<p>A quantified comparison is a comparison operator (&lt;, =, &gt;, &lt;=, &gt;=, &lt;&gt;) with ALL or ANY or SOME applied.</p> <p>Operates on table subqueries, which can return multiple rows but must return a single column.</p> <p>If ALL is used, the comparison must be true for all values returned by the table subquery. If ANY or SOME is used, the comparison must be true for at least one value of the table subquery. ANY and SOME are equivalent.</p> <pre>WHERE normal_rate &lt; ALL       (SELECT budget/550 FROM Groups)</pre>

## See Also

» [CONSTRAINT](#) clause

» [WHERE](#) clause

# CASE Expression

The `CASE` expression can be used for conditional expressions in Splice Machine.

## Syntax

You can place a `CASE` expression anywhere an expression is allowed. It chooses an expression to evaluate based on a boolean test.

```
CASE
  WHEN booleanExpression THEN thenExpression
  [ WHEN booleanExpression
    THEN thenExpression ]...
  ELSE elseExpression
END
```

### *thenExpression and elseExpression*

Both are both that must be type-compatible. For built-in types, this means that the types must be the same or a built-in broadening conversion must exist between the types.

## Example

```
-- returns 3
CASE WHEN 1=1 THEN 3 ELSE 4 END;

-- returns 7
CASE
  WHEN 1 = 2 THEN 3
  WHEN 4 = 5 THEN 6
  ELSE 7
END;
```

## Dynamic Parameters

You can prepare statements that are allowed to have parameters for which the value is not specified when the statement is repared using *PreparedStatement* methods in the JDBC API. These parameters are called dynamic parameters and are represented by a `?`.

The JDBC API documents refer to dynamic parameters as `IN`, `INOUT`, or `OUT` parameters. In SQL, they are always `IN` parameters.

You must specify values for dynamic parameters before executing the statement, and the types of the specified values must match the expected types.

### Example

```
PreparedStatement ps2 = conn.prepareStatement(
    "UPDATE HotelAvailability SET rooms_available = " +
    "(rooms_available - ?) WHERE hotel_id = ? " +
    "AND booking_date BETWEEN ? AND ?");

    -- this sample code sets the values of dynamic parameters
    -- to be the values of program variables
ps2.setInt(1, numberRooms);
ps2.setInt(2, theHotel.hotelId);
ps2.setDate(3, arrival);
ps2.setDate(4, departure);
updateCount = ps2.executeUpdate();
```

### Where Dynamic Parameters are Allowed

You can use dynamic parameters anywhere in an expression where their data type can be easily deduced.

- » Use as the first operand of `BETWEEN` is allowed if one of the second and third operands is not also a dynamic parameter. The type of the first operand is assumed to be the type of the non-dynamic parameter, or the union result of their types if both are not dynamic parameters.

```
WHERE ? BETWEEN DATE('1996-01-01') AND ?
    -- types assumed to be DATE
```

- » Use as the second or third operand of `BETWEEN` is allowed. Type is assumed to be the type of the left operand.

```
WHERE DATE('1996-01-01') BETWEEN ? AND ?
    -- types assumed to be DATE
```

- » Use as the left operand of an `IN` list is allowed if at least one item in the list is not itself a dynamic parameter. Type for the left operand is assumed to be the union result of the types of the non-dynamic parameters in the list.

```
WHERE ? NOT IN (?, ?, 'Santiago')
-- types assumed to be CHAR
```

- » Use in the values list in an `IN` predicate is allowed if the first operand is not a dynamic parameter or its type was determined in the previous rule. Type of the dynamic parameters appearing in the values list is assumed to be the type of the left operand.

```
WHERE FloatColumn IN (?, ?, ?)
-- types assumed to be FLOAT
```

- » For the binary operators `+`, `-`, `*`, `/`, `AND`, `OR`, `<`, `>`, `=`, `<>`, `<=`, and `=`, use of a dynamic parameter as one operand but not both is permitted. Its type is taken from the other side.

```
WHERE ? < CURRENT_TIMESTAMP
-- type assumed to be a TIMESTAMP
```

- » Use in a `CAST` is always permitted. This gives the dynamic parameter a type.

```
CALL valueOf(CAST (? AS VARCHAR(10)))
```

- » Use on either or both sides of `LIKE` operator is permitted. When used on the left, the type of the dynamic parameter is set to the type of the right operand, but with the maximum allowed length for the type. When used on the right, the type is assumed to be of the same length and type as the left operand. (`LIKE` is permitted on `CHAR` and `VARCHAR` types; see [Concatenation operator](#) for more information.)

```
WHERE ? LIKE 'Santi%'
-- type assumed to be CHAR with a length of
-- java.lang.Integer.MAX_VALUE
```

- » In a conditional expression, which uses a `?`, use of a dynamic parameter (which is also represented as a `?`) is allowed. The type of a dynamic parameter as the first operand is assumed to be boolean. Only one of the second and third operands can be a dynamic parameter, and its type will be assumed to be the same as that of the other (that is, the third and second operand, respectively).

```
SELECT c1 IS NULL ? ? : c1
-- allows you to specify a "default" value at execution time
-- dynamic parameter assumed to be the type of c1
-- you cannot have dynamic parameters on both sides
-- of the :
```

- » A dynamic parameter is allowed as an item in the values list or select list of an `INSERT` statement. The type of the dynamic parameter is assumed to be the type of the target column.

```
INSERT INTO t VALUES (?)
-- dynamic parameter assumed to be the type
-- of the only column in table t
INSERT INTO t SELECT ?
FROM t2
-- not allowed
```

- » A `?` parameter in a comparison with a subquery takes its type from the expression being selected by the subquery. For

example:

```
SELECT *
FROM tab1
WHERE ? = (SELECT x FROM tab2)
SELECT *
FROM tab1
WHERE ? = ANY (SELECT x FROM tab2)
-- In both cases, the type of the dynamic parameter is
-- assumed to be the same as the type of tab2.x.
```

- » A dynamic parameter is allowed as the value in an `UPDATE` statement. The type of the dynamic parameter is assumed to be the type of the column in the target table.

```
UPDATE t2 SET c2 =?
-- type is assumed to be type of c2
```

- » Dynamic parameters are allowed as the operand of the unary operators `-` or `+`. For example:

```
CREATE TABLE t1 (c11 INT, c12 SMALLINT, c13 DOUBLE, c14 CHAR(3))
SELECT * FROM t1 WHERE c11 BETWEEN -? AND +?
-- The type of both of the unary operators is INT
-- based on the context in which they are used (that is,
-- because c11 is INT, the unary parameters also get the
-- type INT.
```

- » `LENGTH` allow a dynamic parameter. The type is assumed to be a maximum length `VARCHAR` type.

```
SELECT LENGTH(?)
```

- » Qualified comparisons.

```
? = SOME (SELECT 1 FROM t)
-- is valid. Dynamic parameter assumed to be INTEGER type

1 = SOME (SELECT ? FROM t)
-- is valid. Dynamic parameter assumed to be INTEGER type.
```

- » A dynamic parameter is allowed as the left operand of an `IS` expression and is assumed to be a `Boolean`.

## Expression Precedence

The precedence of operations from highest to lowest is:

- » `()`, `?`, Constant (including sign), `NULL`, *ColumnReference*, *ScalarSubquery*, `CAST`
- » `LENGTH`, `CURRENT_DATE`, `CURRENT_TIME`, `CURRENT_TIMESTAMP`, and other built-ins
- » unary `+` and `-`
- » `*`, `/`, `||` (concatenation)
- » binary `+` and `-`
- » comparisons, quantified comparisons, `EXISTS`, `IN`, `IS NULL`, `LIKE`, `BETWEEN`, `IS`
- » `NOT`
- » `AND`
- » `OR`

You can explicitly specify precedence by placing expressions within parentheses. An expression within parentheses is evaluated before any operations outside the parentheses are applied to it.

### Example

```
(3+4)*9  
(age < 16 OR age > 65) AND employed = TRUE
```

## NEXT VALUE FOR Expression

The `NEXT VALUE FOR` expression retrieves the next value from a sequence generator that was created with a [CREATE SEQUENCE statement](#).

### Syntax

```
NEXT VALUE FOR sequenceName
```

#### *sequenceName*

A sequence name is an identifier that can optionally be qualified by a schema name:

```
[ SQLIdentifier
```

If *schemaName* is not provided, the current schema is the default schema. If a qualified sequence name is specified, the schema name cannot begin with the `SYS.` prefix.

### Usage

If this is the first use of the sequence generator, the generator returns its `START` value. Otherwise, the `INCREMENT` value is added to the previous value returned by the sequence generator. The data type of the value is the *dataType* specified for the sequence generator.

If the sequence generator wraps around, then one of the following happens:

- » If the sequence generator was created using the `CYCLE` keyword, the sequence generator is reset to its `START` value.
- » If the sequence generator was created with the default `NO CYCLE` behavior, Splice Machine throws an exception.

In order to retrieve the next value of a sequence generator, you or your session's current role must have `USAGE` privilege on the generator.

A `NEXT VALUE FOR` expression may occur in the following places:

- » [SELECT statement](#): As part of the expression defining a returned column in a `SELECT` list
- » [VALUES expression](#): As part of the expression defining a column in a row constructor (`VALUES` expression)
- » [UPDATE statement](#): As part of the expression defining the new value to which a column is being set

The next value of a sequence generator is not affected by whether the user commits or rolls back a transaction which invoked the sequence generator.

### Restrictions

Only one `NEXT VALUE FOR` expression is allowed per sequence per statement.

The `NEXT VALUE FOR` expression is not allowed in any statement which has a `DISTINCT` or `ORDER BY` expression.

A `NEXT VALUE` expression **may not appear** in any of these situations:

- » [CASE](#) expression
- » [WHERE](#) clause
- » [ORDER BY](#) clause
- » [Aggregate expression](#)
- » [Window functions](#)
- » [ROW\\_NUMBER](#) function
- » `DISTINCT` select list

## Examples

```
VALUES (NEXT VALUE FOR order_id);

INSERT INTO re_order_table
  SELECT NEXT VALUE FOR order_id, order_date, quantity
  FROM orders
  WHERE back_order = 1;

UPDATE orders
  SET oid = NEXT VALUE FOR order_id
  WHERE expired = 1;
```

## See Also

- » [CREATE SEQUENCE](#) function
- » [SELECT](#) statement
- » [VALUES](#) expression
- » [UPDATE](#) statement



# SELECT Expression

A *SelectExpression* is the basic `SELECT-FROM-WHERE` construct used to build a table value based on filtering and projecting values from other tables.

## Syntax

```
SELECT [ DISTINCT | ALL ] SelectItem [ , SelectItem ]*
  FROM clause
  [ WHERE clause ]
  [ GROUP BY clause ]
  [ HAVING clause ]
  [ ORDER BY clause ]
  [ result offset clause ]
  [ fetch first clause ]
```

### SELECT clause

The `SELECT` clause contains a list of expressions and an optional quantifier that is applied to the results of the [WHERE clause](#).

If `DISTINCT` is specified, only one copy of any row value is included in the result. Nulls are considered duplicates of one another for the purposes of `DISTINCT`.

If no quantifier, or `ALL`, is specified, no rows are removed from the result in applying the `SELECT` clause. This is the default behavior.

### SelectItem:

```
{
  * |
  { <a href="correlation-Name" > .* |
    Expression [AS Simple-column-Name] }
}
```

A *SelectItem* projects one or more result column values for a table result being constructed in a *SelectExpression*.

For queries that do not select a specific column from the tables involved in the *SelectExpression* (for example, queries that use `COUNT (*)`), the user must have at least one column-level `SELECT` privilege or table-level `SELECT` privilege. See [GRANT statement](#) for more information.

### FROM clause

The result of the [FROM clause](#) is the cross product of the `FROM` items.

### WHERE clause

The [WHERE clause](#) can further qualify the result of the `FROM` clause.

### GROUP BY clause

The [GROUP BY clause](#) groups rows in the result into subsets that have matching values for one or more columns.

GROUP BY clauses are typically used with aggregates. If there is a GROUP BY clause, the SELECT clause must contain *only* aggregates or grouping columns. If you want to include a non-grouped column in the SELECT clause, include the column in an aggregate expression. For example, this query computes the average salary of each team in a baseball league:

```
splice> SELECT COUNT(*) AS PlayerCount, Team, AVG(Salary) AS AverageSalary
        FROM Players JOIN Salaries ON Players.ID=Salaries.ID
        GROUP BY Team
        ORDER BY AverageSalary;
```

If there is no GROUP BY clause, but a *SelectItem* contains an aggregate not in a subquery, the query is implicitly grouped. The entire table is the single group.

#### HAVING clause

The [HAVING clause](#) can further qualify the result of the FROM clause. This clause restricts a grouped table, specifying a search condition (much like a WHERE clause) that can refer only to grouping columns or aggregates from the current scope.

The HAVING clause is applied to each group of the grouped table. If the HAVING clause evaluates to TRUE, the row is retained for further processing; if it evaluates to FALSE or NULL, the row is discarded. If there is a HAVING clause but no GROUP BY, the table is implicitly grouped into one group for the entire table.

#### ORDER BY clause

The [ORDER BY clause](#) allows you to specify the order in which rows appear in the result set. In subqueries, the ORDER BY clause is meaningless unless it is accompanied by one or both of the result offset and fetch first clauses.

#### result offset and fetch first clauses

The [fetch first clause](#), which can be combined with the result offset clause, limits the number of rows returned in the result set.

## Usage

The result of a *SelectExpression* is always a table.

Splice Machine processes the clauses in a Select expression in the following order:

- » FROM clause
- » WHERE clause
- » GROUP BY (or implicit GROUP BY)
- » HAVING clause
- » ORDER BY clause
- » Result offset clause
- » Fetch first clause

## » SELECT clause

When a query does not have a `FROM` clause (when you are constructing a value, not getting data out of a table), use a [VALUES](#) expression, not a *SelectExpression*. For example:

```
VALUES CURRENT_TIMESTAMP;
```

## The \* wildcard

The wildcard character (`***`) expands to all columns in the tables in the associated `FROM` clause.

- » [correlation-Name](#) identifiers expand to all columns in the identified table. That table must be listed in the associated `FROM` clause.

## Naming columns

You can name a *SelectItem* column using the `AS` clause.

If a column of a *SelectItem* is not a simple *ColumnReference* expression or named with an `AS` clause, it is given a generated unique name.

These column names are useful in several cases:

- » They are made available on the JDBC *ResultSetMetaData*.
- » They are used as the names of the columns in the resulting table when the *SelectExpression* is used as a table subquery in a `FROM` clause.
- » They are used in the `ORDER BY` clause as the column names available for sorting.

## Examples

This example shows using a `SELECT` with `WHERE` and `ORDER BY` clauses; it selects the name, team, and birth date of all players born in 1985 and 1989:

```
splice> SELECT DisplayName, Team, BirthDate
FROM Players
WHERE YEAR(BirthDate) IN (1985, 1989)
ORDER BY BirthDate;
```

DISPLAYNAME	TEAM	BIRTHDATE
Jeremy Johnson	Cards	1985-03-15
Gary Kosovo	Giants	1985-06-12
Michael Hillson	Cards	1985-11-07
Mitch Canepa	Cards	1985-11-26
Edward Erdman	Cards	1985-12-21
Jeremy Packman	Giants	1989-01-01
Nathan Nickels	Giants	1989-05-04
Ken Straiter	Cards	1989-07-20
Marcus Bamburger	Giants	1989-08-01
George Goomba	Cards	1989-08-08
Jack Hellman	Cards	1989-08-09
Elliot Andrews	Giants	1989-08-21
Henry Socomy	Giants	1989-11-17

13 rows selected

This example shows using correlation names for the tables:

```
splice> SELECT CONSTRAINTNAME, COLUMNNAME
FROM SYS.SYSTABLES t, SYS.SYSCOLUMNS col,
SYS.SYSCONSTRAINTS cons, SYS.SYSCHECKS checks
WHERE t.TABLENAME = 'FLIGHTS'
AND t.TABLEID = col.REFERENCEID
AND t.TABLEID = cons.TABLEID
AND cons.CONSTRAINTID = checks.CONSTRAINTID
ORDER BY CONSTRAINTNAME;
```

This example shows using the DISTINCT clause:

```
SELECT DISTINCT SALARY FROM Salaries;
```

This example shows how to rename an expression. We use the name BOSS as the maximum department salary for all departments whose maximum salary is less than the average salary in all other departments:

```
SELECT WORKDEPT AS DPT, MAX(SALARY) AS BOSS
FROM EMPLOYEE EMP_COR
GROUP BY WORKDEPT
HAVING MAX(SALARY) < (SELECT AVG(SALARY)
FROM EMPLOYEE
WHERE NOT WORKDEPT = EMP_COR.WORKDEPT)
ORDER BY BOSS;
```

## See Also

- » [FROM](#) clause
- » [GROUP BY](#) clause
- » [HAVING](#) clause
- » [ORDER BY](#) clause
- » [WHERE](#) clause

## TABLE Expression

A *TableExpression* specifies a table, view, or function in a [FROM clause](#). It is the source from which a [TableExpression](#) selects a result.

### Syntax

```
{
    JOIN operations
}
```

### Usage

A correlation name can be applied to a table in a *TableExpression* so that its columns can be qualified with that name.

- » If you do not supply a correlation name, the table name qualifies the column name.
- » When you give a table a correlation name, you cannot use the table name to qualify columns.
- » You must use the correlation name when qualifying column names.
- » No two items in the `FROM` clause can have the same correlation name, and no correlation name can be the same as an unqualified table name specified in that `FROM` clause.

In addition, you can give the columns of the table new names in the `AS` clause. Some situations in which this is useful:

- » When a *TableSubquery*, since there is no other way to name the columns of a `VALUES` expression.
- » When column names would otherwise be the same as those of columns in other tables; renaming them means you don't have to qualify them.

The Query in a [TableSubquery](#).

### Example

```
-- SELECT from a JOIN expression
SELECT E.EMPNO, E.LASTNAME, M.EMPNO, M.LASTNAME
FROM EMPLOYEE E LEFT OUTER JOIN
    DEPARTMENT INNER JOIN EMPLOYEE M
    ON MGRNO = M.EMPNO
    ON E.WORKDEPT = DEPTNO;
```

## TableViewOrFunctionExpression

```
{
  { view-Name }
  [ CorrelationClause ] |
  { TableSubquery | TableFunctionInvocation }
  CorrelationClause
}
```

where *CorrelationClause* is

```
[ AS ]
correlation-Name
[ ( Simple-column-Name * ) ]
```

## TableFunctionExpression

```
{
  TABLE function-name( [ [ function-arg ] [, function-arg ]* ] )
}
```

Note that when you invoke a table function, you must bind it to a correlation name. For example:

```
splice> SELECT s.* FROM TABLE( externalEmployees( 42 ) ) s;
```

## See Also

- » [FROM](#) clause
- » [JOIN](#) operations
- » [SELECT](#) statement
- » [VALUES](#) expression

# VALUES Expression

The `VALUES` expression allows construction of a row or a table from other values.

## Syntax

```
{
  VALUES ( Value {, Value }* )
    [ , ( Value {, Value }* ) ]* |
  VALUES Value [ , Value ]*
}
[ ORDER BY clause ]
[ result offset clause ]
[ fetch first clause ]
```

### Value

```
Expression | DEFAULT
```

The first form constructs multi-column rows. The second form constructs single-column rows, each expression being the value of the column of the row.

The `DEFAULT` keyword is allowed only if the `VALUES` expression is in an `INSERT` statement. Specifying `DEFAULT` for a column inserts the column's default value into the column. Another way to insert the default value into the column is to omit the column from the column list and only insert values into other columns in the table.

### ORDER BY clause

The [ORDER BY clause](#) allows you to specify the order in which rows appear in the result set.

### result offset and fetch first clauses

The [fetch first clause](#), which can be combined with the `result offset clause`, limits the number of rows returned in the result set.

## Usage

A `VALUES` expression can be used in all the places where a query can, and thus can be used in any of the following ways:

- » As a statement that returns a *ResultSet*
- » Within expressions and statements wherever subqueries are permitted
- » As the source of values for an [INSERT](#) statement (in an `INSERT` statement, you normally use a `VALUES` expression when you do not use a [SelectExpression](#))

You can use a `VALUES` expression to generate new data values with a query that selects from a `VALUES` clause; for example:



```
SELECT R1,R2
FROM (VALUES ('GROUP 1','GROUP 2')) AS MYTBL(R1,R2);
```

A `VALUES` expression that is used in an `INSERT` statement cannot use an `ORDER BY` clause. However, if the `VALUES` expression does not contain the `DEFAULT` keyword, the `VALUES` clause can be put in a subquery and ordered, as in the following statement:

```
INSERT INTO t SELECT * FROM (VALUES 'a','c','b') t ORDER BY 1;
```

## Examples

```
-- 3 rows of 1 column
splice> VALUES (1),(2),(3);

-- 3 rows of 1 column
splice> VALUES 1, 2, 3;

-- 1 row of 3 columns
splice> VALUES (1, 2, 3);

-- 3 rows of 2 columns
splice> VALUES (1,21),(2,22),(3,23);

-- using ORDER BY and FETCH FIRST
splice> VALUES (3,21),(1,22),(2,23) ORDER BY 1 FETCH FIRST 2 ROWS ONLY;

-- using ORDER BY and OFFSET
splice> VALUES (3,21),(1,22),(2,23) ORDER BY 1 OFFSET 1 ROW;

-- constructing a derived table
splice> VALUES ('orange', 'orange'), ('apple', 'red'), ('banana', 'yellow');

-- Insert two new departments using one statement into the DEPARTMENT table,
-- but do not assign a manager to the new department.
splice> INSERT INTO DEPARTMENT (DEPTNO, DEPTNAME, ADMRDEPT)
VALUES ('B11', 'PURCHASING', 'B01'),
      ('E41', 'DATABASE ADMINISTRATION', 'E01');

-- insert a row with a DEFAULT value for the MAJPROJ column
splice> INSERT INTO PROJECT (PROJNO, PROJNAME, DEPTNO, RESPEMP, PRSTDAT, MAJPROJ)
VALUES ('PL2101', 'ENSURE COMPAT PLAN', 'B01', '000020', CURRENT_DATE, DEFAULT);

-- using a built-in function
splice> VALUES CURRENT_DATE;

-- getting the value of an arbitrary expression
splice> VALUES (3*29, 26.0E0/3);

-- getting a value returned by a built-in function
splice> values char(1);
```

## See Also

- » [FROM](#) clause
- » [ORDER BY](#) clause
- » [INSERT](#) statement

# Join Operations

This section contains the reference documentation for the Splice Machine SQL Join Operations, in the following topics:

Topic	Description
<a href="#">About Join Operations</a>	Overview of joins.
<a href="#">CROSS JOIN</a>	Produces the Cartesian product of two tables: it produces rows that combine each row from the first table with each row from the second table.
<a href="#">INNER JOIN</a>	Selects all rows from both tables as long as there is a match between the columns in both tables.
<a href="#">LEFT OUTER JOIN</a>	Returns all rows from the left table (table1), with the matching rows in the right table (table2). The result is <code>NULL</code> in the right side when there is no match.
<a href="#">NATURAL JOIN</a>	Creates an implicit join clause for you based on the common columns (those with the same name in both tables) in the two tables being joined.
<a href="#">RIGHT OUTER JOIN</a>	Returns all rows from the right table (table2), with the matching rows in the left table (table1). The result is <code>NULL</code> in the left side when there is no match.



For access to the source code for the Community Edition of Splice Machine, visit [our open source GitHub repository](#).

# About Join Operations

The JOIN operations, which are among the possible [FROM clause](#), perform joins between two tables.

## Syntax

JOIN Operation

The following table describes the JOIN operations:

Join Operation	Description
INNER JOIN	Specifies a join between two tables with an explicit join clause.
LEFT OUTER JOIN	Specifies a join between two tables with an explicit join clause, preserving unmatched rows from the first table.
RIGHT OUTER JOIN	Specifies a join between two tables with an explicit join clause, preserving unmatched rows from the second table.
CROSS JOIN	Specifies a join that produces the Cartesian product of two tables. It has no explicit join clause.
NATURAL JOIN	Specifies an inner or outer join between two tables. It has no explicit join clause. Instead, one is created implicitly using the common columns from the two tables. <div><b>NOTE:</b> Splice Machine does not currently support NATURAL SELF JOIN operations.</div>

In all cases, you can specify additional restrictions on one or both of the tables being joined in outer join clauses or in the [WHERE clause](#).

## Usage

Note that you can also perform a join between two tables using an explicit equality test in a [WHERE clause](#), such as:

```
WHERE t1.col1 = t2.col2.
```

## See Also

- » [FROM](#) clause
- » [JOIN operations](#)
- » [TABLE](#) expressions
- » [WHERE](#) clause

# CROSS JOIN

A `CROSS JOIN` is a [JOIN operation](#) that produces the Cartesian product of two tables. Unlike other `JOIN` operators, it does not let you specify a join clause. You may, however, specify a `WHERE` clause in the `SELECT` statement.

## Syntax

```
TableExpression CROSS JOIN ( TableExpression )
```

## Examples

The following `SELECT` statements are equivalent:

```
splice> SELECT * FROM CITIES CROSS JOIN FLIGHTS;

splice> SELECT * FROM CITIES, FLIGHTS;
```

The following `SELECT` statements are equivalent:

```
splice> SELECT * FROM CITIES CROSS JOIN FLIGHTS
      WHERE CITIES.AIRPORT = FLIGHTS.ORIG_AIRPORT;

splice> SELECT * FROM CITIES INNER JOIN FLIGHTS
      ON CITIES.AIRPORT = FLIGHTS.ORIG_AIRPORT;
```

The following example is more complex. The `ON` clause in this example is associated with the `LEFT OUTER JOIN` operation. Note that you can use parentheses around a `JOIN` operation.

```
splice> SELECT * FROM CITIES LEFT OUTER JOIN
      (FLIGHTS CROSS JOIN COUNTRIES)
      ON CITIES.AIRPORT = FLIGHTS.ORIG_AIRPORT
      WHERE COUNTRIES.COUNTRY_ISO_CODE = 'US';
```

A `CROSS JOIN` operation can be replaced with an `INNER JOIN` where the join clause always evaluates to true (for example, `1=1`). It can also be replaced with a sub-query. So equivalent queries would be:

```
splice> SELECT * FROM CITIES LEFT OUTER JOIN  
  FLIGHTS INNER JOIN COUNTRIES ON 1=1  
  ON CITIES.AIRPORT = FLIGHTS.ORIG_AIRPORT  
  WHERE COUNTRIES.COUNTRY_ISO_CODE = 'US';  
  
splice> SELECT * FROM CITIES LEFT OUTER JOIN  
  (SELECT * FROM FLIGHTS, COUNTRIES) S  
  ON CITIES.AIRPORT = S.ORIG_AIRPORT  
  WHERE S.COUNTRY_ISO_CODE = 'US';
```

## See Also

» [JOIN operations](#)

» [USING](#) clause

# INNER JOIN

An `INNER JOIN` is a [JOIN operation](#) that allows you to specify an explicit join clause.

## Syntax

```
TableExpression  
{ ON booleanExpression | USING clause }
```

You can specify the join clause by specifying `ON` with a boolean expression.

The scope of expressions in the `ON` clause includes the current tables and any tables in outer query blocks to the current `SELECT`. In the following example, the `ON` clause refers to the current tables:

```
SELECT *  
  FROM SAMP.EMPLOYEE INNER JOIN SAMP.STAFF  
    ON EMPLOYEE.SALARY < STAFF.SALARY;
```

The `ON` clause can reference tables not being joined and does not have to reference either of the tables being joined (though typically it does).



## Examples

```

-- Join the EMP_ACT and EMPLOYEE tables
-- select all the columns from the EMP_ACT table and
-- add the employee's surname (LASTNAME) from the EMPLOYEE table
-- to each row of the result
splice> SELECT SAMP.EMP_ACT.*, LASTNAME
FROM SAMP.EMP_ACT JOIN SAMP.EMPLOYEE
ON EMP_ACT.EMPNO = EMPLOYEE.EMPNO;

-- Join the EMPLOYEE and DEPARTMENT tables,
-- select the employee number (EMPNO),
-- employee surname (LASTNAME),
-- department number (WORKDEPT in the EMPLOYEE table and DEPTNO in the
-- DEPARTMENT table)
-- and department name (DEPTNAME)
-- of all employees who were born (BIRTHDATE) earlier than 1930.
splice> SELECT EMPNO, LASTNAME, WORKDEPT, DEPTNAME
FROM SAMP.EMPLOYEE JOIN SAMP.DEPARTMENT
ON WORKDEPT = DEPTNO
AND YEAR(BIRTHDATE) < 1930;

-- Another example of "generating" new data values,
-- using a query which selects from a VALUES clause (which is an
-- alternate form of a fullselect).
-- This query shows how a table can be derived called "X"
-- having 2 columns "R1" and "R2" and 1 row of data
splice> SELECT *
FROM (VALUES (3, 4), (1, 5), (2, 6))
AS VALUETABLE1(C1, C2)
JOIN (VALUES (3, 2), (1, 2),
(0, 3)) AS VALUETABLE2(c1, c2)
ON VALUETABLE1.c1 = VALUETABLE2.c1;
-- This results in:
-- C1          |C2          |C1          |2
-- -----
-- 3           |4           |3           |2
-- 1           |5           |1           |2

-- List every department with the employee number and
-- last name of the manager
splice> SELECT DEPTNO, DEPTNAME, EMPNO, LASTNAME
FROM DEPARTMENT INNER JOIN EMPLOYEE
ON MGRNO = EMPNO;

-- List every employee number and last name
-- with the employee number and last name of their manager
splice> SELECT E.EMPNO, E.LASTNAME, M.EMPNO, M.LASTNAME
FROM EMPLOYEE E INNER JOIN
DEPARTMENT INNER JOIN EMPLOYEE M
ON MGRNO = M.EMPNO
ON E.WORKDEPT = DEPTNO;

```

## See Also

- » [JOIN operations](#)
- » [USING](#) clause

# LEFT OUTER JOIN

A `LEFT OUTER JOIN` is one of the [JOIN operations](#) that allow you to specify a join clause. It preserves the unmatched rows from the first (left) table, joining them with a `NULL` row in the shape of the second (right) table.

## Syntax

```
TableExpression
{
    ON booleanExpression |
    USING clause
}
```

The scope of expressions in either the `ON` clause includes the current tables and any tables in query blocks outer to the current `SELECT`. The `ON` clause can reference tables not being joined and does not have to reference either of the tables being joined (though typically it does).

## Example 1

```
-- match cities to countries in Asia
splice> SELECT CITIES.COUNTRY, CITIES.CITY_NAME, REGION
FROM Countries
LEFT OUTER JOIN Cities
ON CITIES.COUNTRY_ISO_CODE = COUNTRIES.COUNTRY_ISO_CODE
WHERE REGION = 'Asia';

-- use the synonymous syntax, LEFT JOIN, to achieve exactly
-- the same results as in the example above
splice> SELECT COUNTRIES.COUNTRY, CITIES.CITY_NAME, REGION
FROM COUNTRIES
LEFT JOIN CITIES
ON CITIES.COUNTRY_ISO_CODE = COUNTRIES.COUNTRY_ISO_CODE
WHERE REGION = 'Asia';
```

## Example 2

```
-- Join the EMPLOYEE and DEPARTMENT tables,
-- select the employee number (EMPNO),
-- employee surname (LASTNAME),
-- department number (WORKDEPT in the EMPLOYEE table
-- and DEPTNO in the DEPARTMENT table)
-- and department name (DEPTNAME)
-- of all employees who born (BIRTHDATE) earlier than 1930
splice> SELECT EMPNO, LASTNAME, WORKDEPT, DEPTNAME
FROM SAMP.EMPLOYEE LEFT OUTER JOIN SAMP.DEPARTMENT
ON WORKDEPT = DEPTNO
AND YEAR(BIRTHDATE) < 1930;

-- List every department with the employee number and
-- last name of the manager,
-- including departments without a manager
splice> SELECT DEPTNO, DEPTNAME, EMPNO, LASTNAME
FROM DEPARTMENT LEFT OUTER JOIN EMPLOYEE
ON MGRNO = EMPNO;
```

## See Also

- » [JOIN](#) operations
- » [TABLE](#) expression
- » [USING](#) clause

# NATURAL JOIN

A `NATURAL JOIN` is a [JOIN operation](#) that creates an implicit join clause for you based on the common columns in the two tables being joined. Common columns are columns that have the same name in both tables.

## Syntax

```
TableExpression NATURAL
[ { LEFT | RIGHT }
  [ OUTER ] | INNER ] JOIN
{ TableViewOrFunctionExpression | ( TableExpression ) }
```

## Usage

A `NATURAL JOIN` can be an `INNER join`, a `LEFT OUTER join`, or a `RIGHT OUTER join`. The default is `INNER join`.

If the `SELECT` statement in which the `NATURAL JOIN` operation appears has an asterisk (\*) in the select list, the asterisk will be expanded to the following list of columns (in the shown order):

- » All the common columns
- » Every column in the first (left) table that is not a common column
- » Every column in the second (right) table that is not a common column

An asterisk qualified by a table name (for example, `COUNTRIES . *`) will be expanded to every column of that table that is not a common column.

If a common column is referenced without being qualified by a table name, the column reference points to the column in the first (left) table if the join is an `INNER JOIN` or a `LEFT OUTER JOIN`. If it is a `RIGHT OUTER JOIN`, unqualified references to a common column point to the column in the second (right) table.

**NOTE:** Splice Machine does not currently support `NATURAL SELF JOIN` operations.

## Examples

If the tables `COUNTRIES` and `CITIES` have two common columns named `COUNTRY` and `COUNTRY_ISO_CODE`, the following two `SELECT` statements are equivalent:

```
splice> SELECT *  
  FROM COUNTRIES  
  NATURAL JOIN CITIES;  
  
splice> SELECT *  
  FROM COUNTRIES  
  JOIN CITIES  
  USING (COUNTRY, COUNTRY_ISO_CODE);
```

The following example is similar to the one above, but it also preserves unmatched rows from the first (left) table:

```
splice> SELECT *  
  FROM COUNTRIES  
  NATURAL LEFT JOIN CITIES;
```

## See Also

- » [JOIN](#) operations
- » [TABLE](#) expression
- » [USING](#) clause

## RIGHT OUTER JOIN

A `RIGHT OUTER JOIN` is one of the [JOIN operations](#) that allow you to specify a `JOIN` clause. It preserves the unmatched rows from the second (right) table, joining them with a `NULL` in the shape of the first (left) table. A `Right Outer JOIN B` is equivalent to `B RIGHT OUTER JOIN A`, with the columns in a different order.

### Syntax

```
TableExpression
{
  ON booleanExpression | USING clause
}
```

The scope of expressions in the `ON` clause includes the current tables and any tables in query blocks outer to the current `SELECT`. The `ON` clause can reference tables not being joined and does not have to reference either of the tables being joined (though typically it does).

### Example 1

```
-- get all countries and corresponding cities, including
-- countries without any cities
splice> SELECT COUNTRIES.COUNTRY, CITIES.CITY_NAME
FROM CITIES RIGHT OUTER JOIN COUNTRIES
ON CITIES.COUNTRY_ISO_CODE = COUNTRIES.COUNTRY_ISO_CODE;

-- get all countries in Africa and corresponding cities,
-- including countries without any cities
splice> SELECT COUNTRIES.COUNTRY, CITIES.CITY_NAME
FROM CITIES
RIGHT OUTER JOIN COUNTRIES
ON CITIES.COUNTRY_ISO_CODE = COUNTRIES.COUNTRY_ISO_CODE
WHERE Countries.region = 'Africa';

-- use the synonymous syntax, RIGHT JOIN, to achieve exactly
-- the same results as in the example above
splice> SELECT COUNTRIES.COUNTRY, CITIES.CITY_NAME
FROM CITIES
RIGHT JOIN COUNTRIES
ON CITIES.COUNTRY_ISO_CODE = COUNTRIES.COUNTRY_ISO_CODE
WHERE Countries.region = 'Africa';
```



## Example 2

```
-- a TableExpression can be a joinOperation. Therefore
-- you can have multiple join operations in a FROM clause
-- List every employee number and last name
-- with the employee number and last name of their manager
splice> SELECT E.EMPNO, E.LASTNAME, M.EMPNO, M.LASTNAME
FROM EMPLOYEE E RIGHT OUTER JOIN
DEPARTMENT RIGHT OUTER JOIN EMPLOYEE M
ON MGRNO = M.EMPNO
ON E.WORKDEPT = DEPTNO;
```

## See Also

- » [JOIN](#) operations
- » [TABLE](#) expression
- » [USING](#) clause

# Queries

This section contains the reference documentation for the Splice Machine SQL Queries, in the following topics:

Topic	Description
<a href="#">Query</a>	Creates a virtual table based on existing tables or constants built into tables.
<a href="#">Scalar Subquery</a>	A subquery that returns a single row with a single column.
<a href="#">Table Subquery</a>	A subquery that returns multiple rows.



For access to the source code for the Community Edition of Splice Machine, visit [our open source GitHub repository](#).

# Query

A *Query* creates a virtual table based on existing tables or constants built into tables.

## Syntax

```
{
  ( Query
    [ ORDER BY clause ]
    [ result offset clause ]
    [ fetch first clause ]
  ) |
  Query EXCEPT [ ALL | DISTINCT ] Query |
  Query UNION [ ALL | DISTINCT ] Query |
  VALUES Expression
}
```

You can arbitrarily put parentheses around queries, or use the parentheses to control the order of evaluation of the UNION operations. These operations are evaluated from left to right when no parentheses are present.

## Duplicates in UNION and EXCEPT ALL results

The ALL and DISTINCT keywords determine whether duplicates are eliminated from the result of the operation. If you specify the DISTINCT keyword, then the result will have no duplicate rows. If you specify the ALL keyword, then there may be duplicates in the result, depending on whether there were duplicates in the input. DISTINCT is the default, so if you don't specify ALL or DISTINCT, the duplicates will be eliminated. For example, UNION builds an intermediate *ResultSet* with all of the rows from both queries and eliminates the duplicate rows before returning the remaining rows. UNION ALL returns all rows from both queries as the result.

Depending on which operation is specified, if the number of copies of a row in the left table is L and the number of copies of that row in the right table is R, then the number of duplicates of that particular row that the output table contains (assuming the ALL keyword is specified) is:

- » UNION: ( L + R ).
- » EXCEPT: the maximum of ( L - R ) and 0 (zero).

## Examples

Here's a simple SELECT expression:

```
SELECT *
FROM ORG;
```

Here's a `SELECT` with a subquery:

```
SELECT *  
  FROM (SELECT CLASS_CODE FROM CL_SCHED) AS CS;
```

Here's a `SELECT` with a subquery:

```
SELECT *  
  FROM (SELECT CLASS_CODE FROM CL_SCHED) AS CS;
```

Here's a `UNION` that lists all employee numbers from certain departments who are assigned to specified project numbers:

```
SELECT EMPNO, 'emp'  
  FROM EMPLOYEE  
 WHERE WORKDEPT LIKE 'E%'  
 UNION  
  SELECT EMPNO, 'emp_act'  
    FROM EMP_ACT  
  WHERE PROJNO IN('MA2100', 'MA2110', 'MA2112');
```

## See Also

- » [ORDER BY](#) clause
- » [SELECT](#) expression
- » [SELECT](#) statement
- » [VALUES](#) expression

## Scalar Subquery

A *ScalarSubquery* turns a [SelectExpression](#) result into a scalar value because it returns only a single row and column value.

### Syntax

```
( Query
  [ ORDER BY clause ]
  [ result offset clause ]
  [ fetch first clause ]
)
```

### Usage

You can place a *ScalarSubquery* anywhere an *Expression* is permitted. The query must evaluate to a single row with a single column.

Scalar subqueries are also called *expression subqueries*.

### Examples

The AVG function always returns a single value; thus, this is a scalar subquery:

```
SELECT NAME, COMM
FROM STAFF
WHERE EXISTS
  (SELECT AVG (BONUS + 800)
   FROM EMPLOYEE
   WHERE COMM < 5000
   AND EMPLOYEE.LASTNAME = UPPER (STAFF.NAME)
  );
```

### See Also

- » [ORDER BY](#) clause
- » [SELECT](#) expression

## Table Subquery

A *TableSubquery* is a subquery that returns multiple rows.

### Syntax

```
( Query
  [ ORDER BY clause ]
  [ result offset clause ]
  [ fetch first clause ]
)
```

### Usage

Unlike a [ScalarSubquery](#), a *TableSubquery* is allowed only:

- » as a [TableExpression](#) in a [FROM clause](#)
- » with EXISTS, IN, or quantified comparisons.

When used as a [TableExpression](#) in a [FROM clause](#), or with EXISTS, it can return multiple columns.

When used with IN or quantified comparisons, it must return a single column.

### Example

This example shows a subquery used as a table expression in a FROM clause:

```
SELECT VirtualFlightTable.flight_ID
FROM
  (SELECT flight_ID, orig_airport, dest_airport
   FROM Flights
   WHERE (orig_airport = 'SFO' OR dest_airport = 'SCL'))
AS VirtualFlightTable;
```

This shows one subquery used with EXISTS and another used with IN:

```
SELECT *
  FROM Flights
 WHERE EXISTS
    (SELECT *
      FROM Flights
     WHERE dest_airport = 'SFO'
        AND orig_airport = 'GRU');

SELECT flight_id, segment_number
  FROM Flights
 WHERE flight_id IN
    (SELECT flight_ID
      FROM Flights
     WHERE orig_airport = 'SFO'
        OR dest_airport = 'SCL');
```

## See Also

- » [FROM](#) clause
- » [ORDER BY](#) clause
- » [SELECT](#) expression
- » [TABLE](#) expression

## Built-in SQL Functions

This section contains the reference documentation for the SQL Functions that are built into Splice Machine, which are grouped into the following subsections:

- » [Conversion Functions](#)
- » [Current Session Functions](#)
- » [Date and Time Functions](#)
- » [Miscellaneous Functions](#)
- » [Numeric Functions](#)
- » [String Functions](#)
- » [Trigonometric Functions](#)
- » [Window and Aggregate Functions](#)

## Conversion Functions

These are the built-in [conversion functions](#):

Function Name	Description
<a href="#">BIGINT</a>	Returns a 64-bit integer representation of a number or character string in the form of an integer constant.
<a href="#">CAST</a>	Converts a value from one data type to another and provides a data type to a dynamic parameter (?) or a NULL value.
<a href="#">CHAR</a>	Returns a fixed-length character string representation.
<a href="#">DOUBLE</a>	Returns a floating-point number
<a href="#">INTEGER</a>	Returns an integer representation of a number or character string in the form of an integer constant.
<a href="#">SMALLINT</a>	Returns a small integer representation of a number or character string in the form of a small integer constant.
<a href="#">TO_CHAR</a>	Formats a date value into a string.
<a href="#">TO_DATE</a>	Formats a date string according to a formatting specification, and returns a date value.
<a href="#">VARCHAR</a>	Returns a varying-length character string representation of a character string.



## Current Session Functions

These are the built-in [current session functions](#):

Function Name	Description
<a href="#">CURRENT_ROLE</a>	Returns the authorization identifier of the current role.
<a href="#">CURRENT_SCHEMA</a>	Returns the schema name used to qualify unqualified database object references.
<a href="#">CURRENT_USER</a>	Depending on context, returns the authorization identifier of either the user who created the SQL session or the owner of the schema.
<a href="#">SESSION_USER</a>	Depending on context, returns the authorization identifier of either the user who created the SQL session or the owner of the schema.
<a href="#">USER</a>	Depending on context, returns the authorization identifier of either the user who created the SQL session or the owner of the schema.

## Date and Time Functions

These are the built-in [date and time functions](#):

Function Name	Description
<a href="#">ADD_MONTHS</a>	Returns the date resulting from adding a number of months added to a specified date.
<a href="#">CURRENT_DATE</a>	Returns the current date.
<a href="#">CURRENT_TIME</a>	Returns the current time;
<a href="#">CURRENT_TIMESTAMP</a>	Returns the current timestamp;
<a href="#">DATE</a>	Returns a date from a value.
<a href="#">DAY</a>	Returns the day part of a value.
<a href="#">EXTRACT</a>	Extracts various date and time components from a date expression.
<a href="#">HOUR</a>	Returns the hour part of a value.
<a href="#">LAST_DAY</a>	Returns the date of the last day of the specified month.
<a href="#">MINUTE</a>	Returns the minute part of a value.

<a href="#">MONTH</a>	Returns the numeric month part of a value.
<a href="#">MONTH_BETWEEN</a>	Returns the number of months between two dates.
<a href="#">MONTHNAME</a>	Returns the string month part of a value.
<a href="#">NEXT_DAY</a>	Returns the date of the next specified day of the week after a specified date.
<a href="#">NOW</a>	Returns the current date and time as a timestamp value.
<a href="#">QUARTER</a>	Returns the quarter number (1-4) from a date expression.
<a href="#">SECOND</a>	Returns the seconds part of a value.
<a href="#">TIME</a>	Returns a time from a value.
<a href="#">TIMESTAMP</a>	Returns a timestamp from a value or a pair of values.
<a href="#">TIMESTAMPADD</a>	Adds the value of an interval to a timestamp value and returns the sum as a new timestamp
<a href="#">TIMESTAMPDIFF</a>	Finds the difference between two timestamps, in terms of the specified interval.
<a href="#">TO_CHAR</a>	Formats a date value into a string.
<a href="#">TO_DATE</a>	Formats a date string according to a formatting specification, and returns a date value.
<a href="#">TRUNC</a> or <a href="#">TRUNCATE</a>	Truncates numeric, date, and timestamp values.
<a href="#">WEEK</a>	Returns the year part of a value.
<a href="#">YEAR</a>	Returns the year part of a value.

## Miscellaneous Functions

These are the built-in [miscellaneous functions](#):

Function Name	Description
<a href="#">COALESCE</a>	Takes two or more compatible arguments and Returns the first argument that is not null.
<a href="#">NULLIF</a>	Returns NULL if the two arguments are equal, and it Returns the first argument if they are not equal.
<a href="#">NVL</a>	Takes two or more compatible arguments and Returns the first argument that is not null.

<a href="#">ROWID</a>	A <i>pseudocolumn</i> that uniquely defines a single row in a database table.
-----------------------	---

## Numeric Functions

These are the built-in [numeric functions](#):

Function Name	Description
<a href="#">ABS</a> or <a href="#">ABSVAL</a>	Returns the absolute value of a numeric expression.
<a href="#">CEIL</a> or <a href="#">CEILING</a>	Round the specified number up, and return the smallest number that is greater than or equal to the specified number.
<a href="#">EXP</a>	Returns e raised to the power of the specified number.
<a href="#">FLOOR</a>	Rounds the specified number down, and Returns the largest number that is less than or equal to the specified number.
<a href="#">LN</a> or <a href="#">LOG</a>	Return the natural logarithm (base e) of the specified number.
<a href="#">LOG10</a>	Returns the base-10 logarithm of the specified number.
<a href="#">MOD</a>	Returns the remainder (modulus) of one number divided by another.
<a href="#">RAND</a>	Returns a random number given a seed number
<a href="#">RANDOM</a>	Returns a random number.
<a href="#">SIGN</a>	Returns the sign of the specified number.
<a href="#">SQRT</a>	Returns the square root of a floating point number;
<a href="#">TRUNC</a> or <a href="#">TRUNCATE</a>	Truncates numeric, date, and timestamp values.

## String Functions

These are the built-in [string functions](#):

Function Name	Description
---------------	-------------

<a href="#">Concatenate</a>	Concatenates a character string value onto the end of another character string. Can also be used on bit string values.
<a href="#">INITCAP</a>	Converts the first letter of each word in a string to uppercase, and converts any remaining characters in each word to lowercase.
<a href="#">INSTR</a>	Returns the index of the first occurrence of a substring in a string.
<a href="#">LCASE</a> or <a href="#">LOWER</a>	Takes a character expression as a parameter and Returns a string in which all alpha characters have been converted to lowercase.
<a href="#">LENGTH</a>	Applied to either a character string expression or a bit string expression and Returns the number of characters in the result.
<a href="#">LOCATE</a>	Used to search for a string within another string.
<a href="#">LTRIM</a>	Removes blanks from the beginning of a character string expression.
<a href="#">REGEXP_LIKE</a>	Returns true if a string matches a regular expression.
<a href="#">REPLACE</a>	Replaces all occurrences of a substring with another substring
<a href="#">RTRIM</a>	Removes blanks from the end of a character string expression.
<a href="#">SUBSTR</a>	Return a portion of string beginning at the specified position for the number of characters specified or rest of the string.
<a href="#">TRIM</a>	Takes a character expression and Returns that expression with leading and/or trailing pad characters removed.
<a href="#">UCASE</a> or <a href="#">UPPER</a>	Takes a character expression as a parameter and Returns a string in which all alpha characters have been converted to uppercase.

## Trigonometric Functions

These are the built-in [trigonometric functions](#):

Function Name	Description
<a href="#">ACOS</a>	Returns the arc cosine of a specified number.
<a href="#">ASIN</a>	Returns the arc sine of a specified number.
<a href="#">ATAN</a>	Returns the arc tangent of a specified number.
<a href="#">ATAN2</a>	Returns the arctangent, in radians, of the quotient of the two arguments.

<a href="#">COS</a>	Returns the cosine of a specified number.
<a href="#">COSH</a>	Returns the hyperbolic cosine of a specified number.
<a href="#">COT</a>	Returns the cotangens of a specified number.
<a href="#">DEGREES</a>	Converts a specified number from radians to degrees.
<a href="#">PI</a>	Returns a value that is closer than any other value to pi.
<a href="#">RADIANS</a>	Converts a specified number from degrees to radians.
<a href="#">SIN</a>	Returns the sine of a specified number.
<a href="#">SINH</a>	Returns the hyperbolic sine of a specified number.
<a href="#">TAN</a>	Returns the tangent of a specified number.
<a href="#">TANH</a>	Returns the hyperbolic tangent of a specified number

## Window and Aggregate Functions

These are the built-in [window and aggregate functions](#):

Function Name	Description
<a href="#">AVG</a>	Returns the average computed over a subset (partition) of a table.
<a href="#">COUNT</a>	Returns the number of rows in a partition.
<a href="#">DENSE_RANK</a>	Returns the ranking of a row within a partition.
<a href="#">FIRST_VALUE</a>	Returns the first value within a partition..
<a href="#">LAG</a>	Returns the value of an expression evaluated at a specified offset number of rows <i>before</i> the current row in a partition.
<a href="#">LAST_VALUE</a>	Returns the last value within a partition..
<a href="#">LEAD</a>	Returns the value of an expression evaluated at a specified offset number of rows <i>after</i> the current row in a partition.
<a href="#">MAX</a>	Returns the maximum value computed over a partition.
<a href="#">MIN</a>	Returns the minimum value computed over a partition.

Function Name	Description
<a href="#">RANK</a>	Returns the ranking of a row within a subset of a table.
<a href="#">ROW_NUMBER</a>	Returns the row number of a row within a partition.
<a href="#">STDDEV_POP</a>	Returns the population standard deviation of a set of numeric values
<a href="#">STDDEV_SAMP</a>	Returns the sample standard deviation of a set of numeric values
<a href="#">SUM</a>	Returns the sum of a value calculated over a partition.



For access to the source code for the Community Edition of Splice Machine, visit [our open source GitHub repository](#).

## ABS or ABSVAL

ABS or ABSVAL returns the absolute value of a numeric expression.

### Syntax

```
ABS (NumericExpression)
```

*NumericExpression*

A numeric expression; all built-in numeric types are supported: [SMALLINT](#)

### Results

The return type is the type of the input parameter.

### Example

```
splice> VALUES ABS (-3);  
1  
-----  
3  
  
1 row selected
```

### See Also

» [About Data Types](#)

# ACOS

The ACOS function returns the arc cosine of a specified number.

## Syntax

```
ACOS ( number )
```

*number*

A [DOUBLE PRECISION](#) number that specifies the cosine, in radians, of the angle that you want.

## Results

The data type of the returned value is a [DOUBLE PRECISION](#) number. The returned value, in radians, is in the range of zero (0) to pi.

- » If the specified *number* is NULL, the result of this function is NULL.
- » If the absolute value of the specified number is greater than 1, an exception is returned that indicates that the value is out of range (SQL state 22003).

## Example

```
splice> VALUES ACOS(0.5);
1
-----
1.0471975511965979

1 row selected
```

## See Also

- » [DOUBLE PRECISION](#) data type
- » [ASIN](#) function
- » [ATAN](#) function
- » [ATAN2](#) function
- » [COS](#) function
- » [COSH](#) function



- » [COT](#) function
- » [DEGREES](#) function
- » [RADIANS](#) function
- » [SIN](#) function
- » [SINH](#) function
- » [TAN](#) function
- » [TANH](#) function

# ADD\_MONTHS

The `ADD_MONTHS` function returns the date resulting from adding a number of months added to a specified date.

## Syntax

```
ADD_MONTHS(Date source, int numOfMonth);
```

### *source*

The source date. This can be a `DATE` value, or any value that can be implicitly converted to `DATE`.

### *numOfMonth*

An integer value that specifies the number of months to add to the source date.

## Results

The returned string always has data type `DATE`.

If date is the last day of the month or if the resulting month has fewer days than the day component of date, then the result is the last day of the resulting month. Otherwise, the result has the same day component as date.

## Examples

```
splice> VALUES (ADD_MONTHS (CURRENT_DATE, 5) );
1
-----
2015-02-22
1 row selected

splice> VALUES (ADD_MONTHS (CURRENT_DATE, -5) );
1
-----
2014-04-22
1 row selected

splice> VALUES (ADD_MONTHS (DATE (CURRENT_TIMESTAMP), -5) );
1
-----
2014-04-22
1 row selected

splice> VALUES (ADD_MONTHS (DATE ('2014-01-31'), 1) );
1
-----
2014-02-28
1 row selected
```

# ASIN

The `ASIN` function returns the arc sine of a specified number.

## Syntax

```
ASIN ( number )
```

*number*

A [DOUBLE PRECISION](#) number that specifies the sine, in radians, of the angle that you want.

## Results

The data type of the returned value is a [DOUBLE PRECISION](#) number. The returned value, in radians, is in the range  $\pi/2$  to  $\pi/2$ .

- » If the specified number is `NULL`, the result of this function is `NULL`.
- » If the specified number is zero (0), the result of this function is zero with the same sign as the specified number.
- » If the absolute value of the specified number is greater than 1, an exception is returned that indicates that the value is out of range (SQL state 22003).

## Example

```
splice> VALUES ASIN(0.5);
1
-----
0.5235987755982989

1 row selected
```

## See Also

- » [DOUBLE PRECISION](#) data type
- » [ACOS](#) function
- » [ATAN](#) function
- » [ATAN2](#) function
- » [COS](#) function

- » [COSH](#) function
- » [COT](#) function
- » [DEGREES](#) function
- » [RADIANS](#) function
- » [SIN](#) function
- » [SINH](#) function
- » [TAN](#) function
- » [TANH](#) function

# ATAN

The ATAN function returns the arc tangent of a specified number.

## Syntax

```
ATAN ( number )
```

*number*

A [DOUBLE PRECISION](#) number that specifies the tangent, in radians, of the angle that you want.

## Results

The data type of the returned value is a [DOUBLE PRECISION](#) number. The returned value, in radians, is in the range  $\pi/2$  to  $\pi/2$ .

- » If the specified number is `NULL`, the result of this function is `NULL`.
- » If the specified number is zero (0), the result of this function is zero with the same sign as the specified number.
- » If the absolute value of the specified number is greater than 1, an exception is returned that indicates that the value is out of range (SQL state 22003).

## Example

```
splice> VALUES ATAN(0.5);
1
-----
0.46364760900008061

1 row selected
```

## See Also

- » [DOUBLE PRECISION](#) data type
- » [ACOS](#) function
- » [ASIN](#) function
- » [ATAN2](#) function
- » [COS](#) function

- » [COSH](#) function
- » [COT](#) function
- » [DEGREES](#) function
- » [RADIANS](#) function
- » [SIN](#) function
- » [SINH](#) function
- » [TAN](#) function
- » [TANH](#) function

# ATAN2

The ATAN2 function returns the arctangent, in radians, of the quotient of the two arguments.

## Syntax

```
ATAN2 ( y, x )
```

*y*

A [DOUBLE PRECISION](#) number.

*x*

A [DOUBLE PRECISION](#) number.

## Results

ATAN2 returns the arc tangent of  $y/x$  in the range  $-pi$  to  $pi$  radians, as a [DOUBLE PRECISION](#) number.

- » If either argument is NULL, the result of the function is NULL.
- » If the first argument is zero and the second argument is positive, the result of the function is zero.
- » If the first argument is zero and the second argument is negative, the result of the function is the double value closest to  $pi$ .
- » If the first argument is positive and the second argument is zero, the result is the double value closest to  $pi/2$ .
- » If the first argument is negative and the second argument is zero, the result is the double value closest to  $-pi/2$ .

## Example

```
splice> VALUES ATAN2(1, 0);
1
-----
1.5707963267948966

1 row selected
```

## See Also

- » [DOUBLE PRECISION](#) data type
- » [ACOS](#) function



- » [ASIN](#) function
- » [ATAN](#) function
- » [COS](#) function
- » [COSH](#) function
- » [COT](#) function
- » [DEGREES](#) function
- » [RADIANS](#) function
- » [SIN](#) function
- » [SINH](#) function
- » [TAN](#) function
- » [TANH](#) function

# AVG

AVG evaluates the average of an expression over a set of rows. You can use it as an [window \(analytic\) function](#).

## Syntax

```
AVG ( [ DISTINCT | ALL ] Expression )
```

### *DISTINCT*

If this qualifier is specified, duplicates are eliminated

### *ALL*

If this qualifier is specified, all duplicates are retained. This is the default value.

### *Expression*

An expression that evaluates to a numeric data type: [SMALLINT](#).

The expression can contain multiple column references or expressions, but it cannot contain another aggregate or subquery, and it must evaluate to an ANSI SQL numeric data type. This means that you can call methods that evaluate to ANSI SQL data types.

If an expression evaluates to NULL, the aggregate skips that value.

## Usage

Only one `DISTINCT` aggregate expression per [Expression](#) is allowed. For example, the following query is not valid:

```
--- query not valid
SELECT AVG (DISTINCT AtBats), SUM (DISTINCT Hits)
FROM Batting;
```

**NOTE:** Note that specifying `DISTINCT` can result in a different value, since a smaller number of values may be averaged. For example, if a column contains the values 1.0, 1.0, 1.0, 1.0, and 2.0, `AVG (col)` returns a smaller value than `AVG (DISTINCT col)`.

## Results

The resulting data type is the same as the expression on which it operates; it will never overflow.

The following query, for example, returns the `INTEGER 1`, which might not be what you would expect:

```
SELECT AVG(c1)
  FROM (VALUES (1), (1), (1), (1), (2))
  AS myTable (c1);
```

[CAST](#) the expression to another data type if you want more precision:

```
SELECT AVG(CAST (c1 AS DOUBLE PRECISION))
  FROM (VALUES (1), (1), (1), (1), (2)) AS myTable (c1);
```

## Aggregate Example

```
splice> SELECT AVG(salary) "Average" FROM Salaries;
Average
-----
2949737

1 row selected
```

## Analytic Example

The following example shows the average salary paid, per position, for the San Francisco Giants in 2015:

```
splice> SELECT Position, Players.ID, Salary, AVG(Cast(Salary as DECIMAL(11,3)))
OVER (PARTITION by Position) "Average for Position"
FROM players join Salaries on players.ID=salaries.ID
WHERE Team='Giants' and Season=2015;
```

POS&	ID	SALARY	Average for Po&
-----			
C	1	17277777	3733139.8000
C	13	468674	3733139.8000
C	18	800000	3733139.8000
C	20	41598	3733139.8000
C	24	77650	3733139.8000
IF	23	91516	91516.0000
1B	2	3600000	1815252.5000
1B	26	30505	1815252.5000
LF	6	4000000	1792987.0000
LF	16	278961	1792987.0000
LF	27	1100000	1792987.0000
P	28	6950000	4759511.3333
P	29	485314	4759511.3333
P	30	4000000	4759511.3333
P	31	12000000	4759511.3333
P	32	9000000	4759511.3333
P	33	18000000	4759511.3333
P	34	20833333	4759511.3333
P	35	3578825	4759511.3333
P	36	2100000	4759511.3333
P	37	210765	4759511.3333
P	38	507500	4759511.3333
P	39	507500	4759511.3333
P	40	6000000	4759511.3333
P	41	6000000	4759511.3333
P	42	374385	4759511.3333
P	43	4000000	4759511.3333
P	44	72103	4759511.3333
P	45	91516	4759511.3333
P	46	5000000	4759511.3333
P	47	74877	4759511.3333
P	48	163620	4759511.3333
3B	5	509000	2654500.0000
3B	14	4800000	2654500.0000
MI	15	288767	288767.0000
SS	4	3175000	3175000.0000
RF	8	18500000	9166666.6666
RF	10	1000000	9166666.6666
RF	12	8000000	9166666.6666
2B	3	507500	339719.5000
2B	11	171939	339719.5000
CF	7	10250000	10250000.0000
UT	17	1450000	749959.0000
UT	22	49918	749959.0000

OF	9	3600000	962397.2500
OF	19	91516	962397.2500
OF	21	149754	962397.2500
OF	25	8319	962397.2500

48 rows selected

## See Also

- » [About Data Types](#)
- » [Window and aggregate functions](#)
- » [COUNT](#) function
- » [MAX](#) function
- » [MIN](#) function
- » [SUM](#) function
- » [OVER](#) clause
- » [Using Window Functions](#) in the *Developer Guide*.

# BIGINT

The `BIGINT` function returns a 64-bit integer representation of a number or character string in the form of an integer constant.

## Syntax

```
BIGINT (CharacterExpression | NumericExpression )
```

### *CharacterExpression*

An expression that returns a character string value of length not greater than the maximum length of a character constant. Leading and trailing blanks are eliminated and the resulting string must conform to the rules for forming an SQL integer constant. The character string cannot be a long string. If the argument is a `CharacterExpression`, the result is the same number that would occur if the corresponding integer constant were assigned to a big integer column or variable.

### *NumericExpression*

An expression that returns a value of any built-in numeric data type. If the argument is a `NumericExpression`, the result is the same number that would occur if the argument were assigned to a big integer column or variable. If the whole part of the argument is not within the range of integers, an error occurs. The decimal part of the argument is truncated if present.

## Results

The result of the function is a big integer.

If the argument can be `NULL`, the result can be `NULL`; if the argument is `NULL`, the result is the `NULL` value.

## Example

Using the `Batting` table from our Doc Examples database, select the `TotalBases` column in big integer form for further processing in the application:

```
splice> SELECT ID, BIGINT(TotalBases) "TotalBases"  
        FROM Batting  
        WHERE ID < 11;
```

ID	TotalBases
1	262
2	235
3	174
4	234
5	245
6	135
7	170
8	99
9	135
10	85

10 rows selected

## See Also

- » [About Data Types](#)
- » [BIGINT](#) data type

# CAST

The `CAST` function converts a value from one data type to another and provides a data type to a dynamic parameter or a `NULL` value.

`CAST` expressions are permitted anywhere expressions are permitted.

## Syntax

```
CAST ( [ Expression | NULL | ? ]
      AS Datatype)
```

The data type to which you are casting an expression is the *target type*. The data type of the expression from which you are casting is the *source type*.

## CAST conversions among ANSI SQL data types

The following table shows valid explicit conversions between source types and target types for SQL data types. This table shows which explicit conversions between data types are valid. The first column on the table lists the source data types. The first row lists the target data types. A “Y” indicates that a conversion from the source to the target is valid. For example, the first cell in the second row lists the source data type `SMALLINT`. The remaining cells on the second row indicate the whether or not you can convert `SMALLINT` to the target data types that are listed in the first row of the table.

TYPES	B O O L E A N	S M A L L I N T	I N T E G E R	B I G I N T	D E C I M A L	R E A L	D O U B L E	F L O A T	C H A R	V A R C H A R	L O N G  V A R C H A R	C L O B	B L O B	D A T E	T I M E	T I M E S T A M P
BOOLEAN	Y	-	-	-	-	-	-	-	Y	Y	Y	Y	-	-	-	-
SMALLINT	-	Y	Y	Y	Y	Y	Y	Y	Y	-	-	-	-	-	-	-
INTEGER	-	Y	Y	Y	Y	Y	Y	Y	Y	-	-	-	-	-	-	-
BIGINT	-	Y	Y	Y	Y	Y	Y	Y	Y	-	-	-	-	-	-	-



TYPES	B O O L E A N	S M A L L I N T	I N T E G E R	B I G I N T	D E C I M A L	R E A L	D O U B L E	F L O A T	C H A R	V A R C H A R	L O N G V A R C H A R	C L O B	B L O B	D A T E	T I M E	T I M E S T A M P
DECIMAL	-	Y	Y	Y	Y	Y	Y	Y	Y	-	-	-	-	-	-	-
REAL	-	Y	Y	Y	Y	Y	Y	Y	-	-	-	-	-	-	-	-
DOUBLE	-	Y	Y	Y	Y	Y	Y	Y	-	-	-	-	-	-	-	-
FLOAT	-	Y	Y	Y	Y	Y	Y	Y	-	-	-	-	-	-	-	-
CHAR	Y	Y	Y	Y	Y	-	-	-	Y	Y	Y	Y	-	Y	Y	Y
VARCHAR	Y	Y	Y	Y	Y	-	-	-	Y	Y	Y	Y	-	Y	Y	Y
LONG VARCHAR	Y	-	-	-	-	-	-	-	Y	Y	Y	Y	-	-	-	-
CLOB	Y	-	-	-	-	-	-	-	Y	Y	Y	Y	-	-	-	-
BLOB	-	-	-	-	-	-	-	-	-	-	-	-	Y	-	-	-
DATE	-	-	-	-	-	-	-	-	Y	Y	-	-	-	Y	-	-
TIME	-	-	-	-	-	-	-	-	Y	Y	-	-	-	-	Y	-
TIMESTAMP	-	-	-	-	-	-	-	-	Y	Y	-	-	-	Y	Y	Y

If a conversion is valid, CASTs are allowed. Size incompatibilities between the source and target types might cause runtime errors.

## Type Categories

This section lists information about converting specific data types. The Splice Machine ANSI SQL data types are categorized as follows:

Category	Data Types
<i>logical</i>	BOOLEAN
<i>numeric</i>	Exact numeric: SMALLINT, INTEGER, BIGINT, DECIMAL, NUMERIC Approximate numeric: FLOAT, REAL, DOUBLE PRECISION
<i>string</i>	Character string: CLOB, CHAR, VARCHAR, LONG VARCHAR Bit string: BLOB
<i>date and time</i>	DATE, TIME, TIMESTAMP

## Conversion Notes

This section lists additional information about casting of certain data types.

### Applying Multiple Conversions

As shown in the above table, you cannot convert freely among all types. For example, you cannot CAST an INTEGER value to a VARCHAR value. However, you may be able to achieve your conversion by using multiple CAST operations.

For example, since you can convert an INTEGER value to a CHAR value, and you can convert a CHAR value to a VARCHAR value, you can use multiple CAST operations, as shown here:

```
CAST(CAST(123 AS CHAR(10)) AS VARCHAR(10));SELECT CAST(CAST(myId as CHAR(20)) as VAR
CHAR(20));
```

### Conversions to and from logical types

These notes apply to converting logical values to strings and vice-versa:

- » A BOOLEAN value can be cast explicitly to any of the string types. The result is 'true', 'false', or null.
- » Conversely, string types can be cast to BOOLEAN; however, an error is raised if the string value is not 'true', 'false', 'unknown', or null.
- » Casting 'false' to BOOLEAN results in a null value.

### Conversions from numeric types

A numeric type can be converted to any other numeric type. These notes apply:

- » If the target type cannot represent the non-fractional component without truncation, an exception is raised.

- » If the target numeric cannot represent the fractional component (scale) of the source numeric, then the source is silently truncated to fit into the target. For example, casting `763.1234` as `INTEGER` yields `763`.

## Conversions from and to bit strings

Bit strings can be converted to other bit strings, but not to character strings. Strings that are converted to bit strings are padded with trailing zeros to fit the size of the target bit string. The `BLOB` type is more limited and requires explicit casting. In most cases the `BLOB` type cannot be cast to and from other types: you can cast a `BLOB` only to another `BLOB`, but you can cast other bit string types to a `BLOB`.

## Conversions of date/time values

A date/time value can always be converted to and from a `TIMESTAMP`.

If a `DATE` is converted to a `TIMESTAMP`, the `TIME` component of the resulting `TIMESTAMP` is always `00:00:00`.

If a `TIME` data value is converted to a `TIMESTAMP`, the `DATE` component is set to the value of `CURRENT_DATE` at the time the `CAST` is executed.

If a `TIMESTAMP` is converted to a `DATE`, the `TIME` component is silently truncated.

If a `TIMESTAMP` is converted to a `TIME`, the `DATE` component is silently truncated.

## Examples

```
splice> SELECT CAST (TotalBases AS BIGINT)
FROM Batting;

-- convert timestamps to text
splice> INSERT INTO mytable (text_column)
VALUES (CAST (CURRENT_TIMESTAMP AS VARCHAR(100)));

-- you must cast NULL as a data type to use it
splice> SELECT airline
FROM Airlines
UNION ALL
VALUES (CAST (NULL AS CHAR(2)));

-- cast a double as a decimal
splice> SELECT CAST (FLYING_TIME AS DECIMAL(5,2))
FROM FLIGHTS;

-- cast a SMALLINT to a BIGINT
splice> VALUES CAST (CAST (12 as SMALLINT) as BIGINT);
```

## See Also

» [About Data Types](#)

## CEIL or CEILING

The `CEIL` and `CEILING` functions round the specified number up, and return the smallest number that is greater than or equal to the specified number.

### Syntax

```
CEIL ( number )
```

```
CEILING ( number )
```

*number*

A [DOUBLE PRECISION](#) value.

The expression can contain multiple column references or expressions, but it cannot contain another aggregate or subquery, and it must evaluate to an ANSI SQL numeric data type. This means that you can call methods that evaluate to ANSI SQL data types.

If an expression evaluates to `NULL`, the aggregate skips that value.

### Results

The data type of the returned value is a [DOUBLE PRECISION](#) number.

The returned value is the smallest (closest to negative infinity) double floating point value that is greater than or equal to the specified number. The returned value is equal to a mathematical integer.

- » If the specified number is `NULL`, the result of these functions is `NULL`.
- » If the specified number is equal to a mathematical integer, the result of these functions is the same as the specified number.
- » If the specified number is zero (0), the result of these functions is zero.
- » If the specified number is less than zero but greater than -1.0, then the result of these functions is zero.

## Example

```
splice> VALUES CEIL(3.33);
1
-----
4

1 row selected

splice> VALUES CEILING(3.67);
1
-----
4

1 row selected
```

## See Also

» [DOUBLE PRECISION](#) data type

# CHAR

The `CHAR` function returns a fixed-length character string representation. The representations are:

- » A character string, if the first argument is any type of character string.
- » A datetime value, if the first argument is a date, time, or timestamp.
- » A decimal number, if the first argument is a decimal number.
- » A double-precision floating-point number, if the first argument is a `DOUBLE` or `REAL`.
- » An integer number, if the first argument is a `SMALLINT`, `INTEGER`, or `BIGINT`.

The first argument must be of a built-in data type.

The result of the `CHAR` function is a fixed-length character string. If the first argument can be `NULL`, the result can be `NULL`. If the first argument is `NULL`, the result is the `NULL` value.

## Character to character syntax

```
CHAR (CharacterExpression [, integer] )
```

### *CharacterExpression*

An expression that returns a value that is `CHAR`, `VARCHAR`, `LONG VARCHAR`, or `CLOB` data type.

### *integer*

The length attribute for the resulting fixed length character string. The value must be between 0 and 254.

## Results

If the length of the character-expression is less than the length attribute of the result, the result is padded with blanks up to the length of the result.

If the length of the character-expression is greater than the length attribute of the result, truncation is performed. A warning is returned unless the truncated characters were all blanks and the character-expression was not a long string (`LONG VARCHAR` or `CLOB`).

## Integer to character syntax

```
CHAR (IntegerExpression )
```

### *IntegerExpression*

An expression that returns a value that is an integer data type (either `SMALLINT`, `INTEGER` or `BIGINT`).

## Results

The result is the character string representation of the argument in the form of an SQL integer constant. The result consists of *n* characters that are the significant digits that represent the value of the argument with a preceding minus sign if the argument is negative. It is left justified.

- » If the first argument is a small integer: the length of the result is 6. If the number of characters in the result is less than 6, then the result is padded on the right with blanks to length 6.
- » If the first argument is a large integer: the length of the result is 11. If the number of characters in the result is less than 11, then the result is padded on the right with blanks to length 11.
- » If the first argument is a big integer: the length of the result is 20. If the number of characters in the result is less than 20, then the result is padded on the right with blanks to length 20.

## Datetime to character syntax

```
CHAR (DatetimeExpression )
```

### *DatetimeExpression*

An expression that is one of the following three data types:

Type	Description
DATE	The result is the character representation of the date. The length of the result is 10.
TIME	The result is the character representation of the time. The length of the result is 8.
TIMESTAMP	The result is the character string representation of the timestamp. The length of the result is 26.

## Decimal to character

```
CHAR (DecimalExpression )
```

### *DecimalExpression*

An expression that returns a value that is a decimal data type.

If a different precision and scale is desired, you can use the `DECIMAL` scalar function first to make the change.

## Floating point to character syntax

```
CHAR (FloatingPointExpression )
```



### *FloatingPointExpression*

An expression that returns a value that is a floating-point data type (DOUBLE or REAL).

## Example

Use the CHAR function to return the values for PlateAppearances (defined as smallint) as a fixed length character string:

```
splice> SELECT CHAR (AtBats * 2) "DoubledAtBats"
      FROM Batting WHERE ID <= 10;
DoubledAtBats
-----
1246
1112
864
1122
1224
784
1102
446
744
548

10 rows selected
```

Since AtBats is declared as SMALLINT in our Examples database, each of the resulting values is padded with blank characters to make it 6 characters long.

## See Also

» [About Data Types](#)

# COALESCE

The `COALESCE` function returns the first non-NULL expression from a list of expressions.

You can also use `COALESCE` as a variety of a `CASE` expression. For example:

```
COALESCE ( expresssion_1, expression_2,...expression_n);
```

is equivalent to:

```
CASE WHEN expression_1 IS NOT NULL THEN expression_1
      ELSE WHEN expression_1 IS NOT NULL THEN expression_2
      ...
      ELSE expression_n;
```

## Syntax

```
COALESCE ( expression1, expression2 [, expressionN]* )
```

*expression1*

An expression.

*expression1*

An expression.

*expressionN*

You can specify more than two arguments; **you MUST specify at least two arguments**.

## Usage

`VALUE` is a synonym for `COALESCE` that is accepted by Splice Machine, but is not recognized by the SQL standard.

## Results

The result is `NULL` only if all of the arguments are `NULL`.

An error occurs if all of the parameters of the function call are dynamic.

## Example

```
-- create table with three different integer types
splice> SELECT ID, FldGames, PassedBalls, WildPitches, Pickoffs,
  COALESCE(PassedBalls, WildPitches, Pickoffs) as "FirstNonNull"
  FROM Fielding
  WHERE FldGames>50
  ORDER BY ID;
```

ID	FLDGA&	PASSE&	WILDP&	PICKO&	First&
1	142	4	20	0	4
2	131	NULL	NULL	NULL	NULL
3	99	NULL	NULL	NULL	NULL
4	140	NULL	NULL	NULL	NULL
5	142	NULL	NULL	NULL	NULL
6	88	NULL	NULL	NULL	NULL
7	124	NULL	NULL	NULL	NULL
8	51	NULL	NULL	NULL	NULL
9	93	NULL	NULL	NULL	NULL
10	79	NULL	NULL	NULL	NULL
39	73	NULL	NULL	0	0
40	52	NULL	NULL	0	0
41	70	NULL	NULL	2	2
42	55	NULL	NULL	0	0
43	77	NULL	NULL	0	0
46	67	NULL	NULL	0	0
49	134	4	34	2	4
50	119	NULL	NULL	NULL	NULL
51	147	NULL	NULL	NULL	NULL
52	148	NULL	NULL	NULL	NULL
53	152	NULL	NULL	NULL	NULL
54	64	NULL	NULL	NULL	NULL
55	93	NULL	NULL	NULL	NULL
56	147	NULL	NULL	NULL	NULL
57	85	NULL	NULL	NULL	NULL
58	62	NULL	NULL	NULL	NULL
59	64	NULL	NULL	NULL	NULL
62	53	1	11	0	1
64	59	NULL	NULL	NULL	NULL
81	76	NULL	NULL	0	0
82	71	NULL	NULL	1	1
84	68	NULL	NULL	0	0
92	81	NULL	NULL	3	3

33 rows selected

## Concatenation Operator

The concatenation operator, `||`, concatenates its right operand onto the end of its left operand; it operates on character string or bit string expressions.

**NOTE:** Since all built-in data types are implicitly converted to strings, this function can act on all built-in data types.

## Syntax

```
{
  { CharacterExpression || CharacterExpression } |
  { BitExpression || BitExpression }
}
```

*CharacterExpression*

An expression.

*expression1*

An expression.

*expressionN*

You can specify more than two argument; you **MUST** specify at least two arguments.

## Results

For character strings:

- » If both the left and right operands are of type [VARCHAR](#).
- » The normal blank padding/trimming rules for `CHAR` and `VARCHAR` apply to the result of this operator.
- » The length of the resulting string is the sum of the lengths of both operands.

## Examples

```
-- returns 'San Francisco Giants'
splice> VALUES 'San' || ' ' || 'Francisco' || ' ' || 'Giants';

-- returns NULL
splice> VALUES CAST (null AS VARCHAR(7)) || 'Something';

-- returns 'Today it is: 93'
splice> VALUES 'Today it is: ' || '93';
```

## See Also

- » [About Data Types](#)
- » [INITCAP](#) function
- » [INSTR](#) function
- » [LCASE](#) function
- » [LENGTH](#) function
- » [LTRIM](#) function
- » [REGEX\\_LIKE](#) operator
- » [REPLACE](#) function
- » [RTRIM](#) function
- » [SUBSTR](#) function
- » [TRIM](#) function
- » [UCASE](#) function

# COS

The `COS` function returns the cosine of a specified number.

## Syntax

```
COS ( number )
```

*number*

A [DOUBLE PRECISION](#) number that specifies the angle, in radians, for which you want the cosine computed.

## Results

The data type of the returned value is a [DOUBLE PRECISION](#) number.

If input argument is `NULL`, the result of the function is `NULL`.

## Example

```
splice> VALUES COS(84.4);  
1  
-----  
-0.9118608758306834  
  
1 row selected
```

## See Also

- » [DOUBLE PRECISION](#) data type
- » [ACOS](#) function
- » [ASIN](#) function
- » [ATAN](#) function
- » [ATAN2](#) function
- » [COSH](#) function
- » [COT](#) function
- » [DEGREES](#) function

» [RADIANS](#) function

» [SIN](#) function

» [SINH](#) function

» [TAN](#) function

» [TANH](#) function

# COSH

The `COSH` function returns the hyperbolic cosine of a specified number.

## Syntax

```
COSH ( number )
```

*number*

A [DOUBLE PRECISION](#) number that specifies the angle, in radians, for which you want the hyperbolic cosine computed.

## Results

The data type of the returned value is a [DOUBLE PRECISION](#) number.

- » If the specified number is `NULL`, the result of this function is `NULL`.
- » If the specified number is zero (0), the result of this function is one (1.0).

## Example

```
splice> VALUES COSH(1.234);
1
-----
2.2564425307671042E36

1 row selected
```

## See Also

- » [DOUBLE PRECISION](#) data type
- » [ACOS](#) function
- » [ASIN](#) function
- » [ATAN](#) function
- » [ATAN2](#) function
- » [COS](#) function
- » [COT](#) function



- » [DEGREES](#) function
- » [RADIANS](#) function
- » [SIN](#) function
- » [SINH](#) function
- » [TAN](#) function
- » [TANH](#) function

# COT

The `COT` function returns the cotangent of a specified number.

## Syntax

```
COT ( number )
```

*number*

A [DOUBLE PRECISION](#) number that specifies the angle, in radians, for which you want the cotangent computed.

## Results

The data type of the returned value is a [DOUBLE PRECISION](#) number.

- » If the specified number is NULL, the result of this function is NULL.
- » If the specified number is zero (0), the result of this function is one (1.0).

## Example

```
splice> VALUES COT(1.234);
1
-----
0.35013639786791445

1 row selected
```

## See Also

- » [DOUBLE PRECISION](#) data type
- » [ACOS](#) function
- » [ASIN](#) function
- » [ATAN](#) function
- » [ATAN2](#) function
- » [COS](#) function
- » [COSH](#) function

- » [DEGREES](#) function
- » [RADIANS](#) function
- » [SIN](#) function
- » [SINH](#) function
- » [TAN](#) function
- » [TANH](#) function

# COUNT

COUNT returns the number of rows returned by the query. You can use it as an [window \(analytic\) function](#).

The COUNT(*Expression*) version returns the number of row where *Expression* is not null. You can count either all rows, or only distinct values of *Expression*.

The COUNT(\*) version returns all rows, including duplicates and nulls.

## Syntax

```
COUNT ( [ DISTINCT | ALL ] Expression )
```

### *DISTINCT*

If this qualifier is specified, duplicates are eliminated from the count.

### *ALL*

If this qualifier is specified, all duplicates are retained. This is the default value.

### *Expression*

An expression that evaluates to a numeric data type: [SMALLINT](#).

An *Expression* can contain multiple column references or expressions, but it cannot contain another aggregate or subquery.

If an *Expression* evaluates to NULL, the aggregate skips that value.

## Usage

Only one DISTINCT aggregate expression per [Expression](#) is allowed. For example, the following query is not valid:

```
-- query not allowed
SELECT COUNT (DISTINCT flying_time), SUM (DISTINCT miles)
FROM Flights
```

**NOTE:** Note that specifying DISTINCT can result in a different value, since a smaller number of values may be counted. For example, if a column contains the values 1, 1, 1, 1, and 2, COUNT(col) returns a greater value than COUNT(DISTINCT col).

## Results

The resulting data type is [BIGINT](#).

## Aggregate Example

```
splice> Select COUNT (Name) "Players", Team
      FROM Players
      GROUP BY Team
      HAVING COUNT(Team) > 1;
Players          |TEAM
-----
-
46                |Cards
48                |Giants

2 rows selected
```

## Analytic Example

The following example shows the product ID, quantity, and count of all rows from the beginning of the data window:

```
splice> SELECT displayName, homeruns,
      COUNT(*) OVER (ORDER BY HomeRuns ROWS BETWEEN UNBOUNDED PRECEDING AND CURRENT ROW)
      as "Running Count"
      FROM Players JOIN Batting ON Players.ID=Batting.ID
      WHERE homeRuns > 5
      ORDER BY "Running Count";
DISPLAYNAME      |HOMER&|Running Count
-----
Jeremy Packman   |6      |1
Jason Minman     |7      |2
Stan Post        |7      |3
John Purser      |8      |4
Harry Pennello  |9      |5
Kelly Wacherman  |11     |6
Mitch Duffer     |12     |7
Michael Rastono  |13     |8
Jack Hellman     |13     |9
Jonathan Pearlman|17     |10
Roger Green      |17     |11
Billy Bopper     |18     |12
Buddy Painter    |19     |13
Bob Cranker      |21     |14
Mitch Canepa     |28     |15

15 rows selected
```

## See Also

- » [About Data Types](#)
- » [Window and Aggregate Functions](#)
- » [AVG](#) function
- » [MAX](#) function
- » [MIN](#) function
- » [SUM](#) function
- » [OVER](#) clause

# CURRENT SCHEMA

`CURRENT SCHEMA` returns the schema name used to qualify unqualified database object references.

**NOTE:** `CURRENT SCHEMA` and `CURRENT SQLID` are synonyms.

## Syntax

```
CURRENT SCHEMA
```

– or, alternatively:

```
CURRENT SQLID
```

## Results

The returned value is a string with a length of up to 128 characters.

## Examples

```
splice> VALUES (CURRENT SCHEMA) ;  
1
```

```
-----  
SPICE
```

```
1 row selected
```

# CURRENT\_DATE

`CURRENT_DATE` returns the current date.

**NOTE:** This function returns the same value if it is executed more than once in a single statement, which means that the value is fixed, even if there is a long delay between fetching rows in a cursor.

## Syntax

```
CURRENT_DATE
```

or, alternately

```
CURRENT DATE
```

## Results

A [DATE](#) value.

## Examples

The following query finds all players older that 33 years (as of Nov. 9, 2015) on the Cards baseball team:

```
splice> SELECT displayName, birthDate
        FROM Players
        WHERE (BirthDate+(33 * 365.25)) <= CURRENT_DATE AND Team='Cards';
DISPLAYNAME          |BIRTHDATE
-----
Yuri Milleton        |1982-07-13
Jonathan Pearlman    |1982-05-28
David Janssen         |1979-08-10
Jason Larrimore       |1978-10-23
Tam Croonster        |1980-12-19
Alex Wister          |1981-08-30
Robert Cohen         |1975-09-05
Mitch Brandon        |1980-06-06

8 rows selected
```



## See Also

- » [DATE](#) data type
- » [DATE](#) function
- » [DAY](#) function
- » [EXTRACT](#) function
- » [LASTDAY](#) function
- » [MONTH](#) function
- » [MONTH\\_BETWEEN](#) function
- » [MONTHNAME](#) function
- » [NEXTDAY](#) function
- » [NOW](#) function
- » [QUARTER](#) function
- » [TIME](#) data type
- » [TIMESTAMP](#) function
- » [TO\\_CHAR](#) function
- » [TO\\_DATE](#) function
- » [WEEK](#) function
- » [\*Working with Dates\*](#) in the *Developer's Guide*

# CURRENT\_ROLE

`CURRENT_ROLE` returns the authorization identifier of the current role. If there is no current role, it returns `NULL`.

This function returns a string of up to 258 characters. This is twice the length of an identifier ( $128 \times 2$ ) + 2, to allow for quoting.

## Syntax

```
CURRENT_ROLE
```

## Example

```
splice> VALUES CURRENT_ROLE;
```

## See Also

- » [CREATE\\_ROLE](#) statement
- » [DROP\\_ROLE](#) statement
- » [GRANT](#) statement
- » [REVOKE](#) statement
- » [SET\\_ROLE](#) statement
- » [SYSROLES](#) system table

# CURRENT\_TIME

`CURRENT_TIME` returns the current time.

**NOTE:** This function returns the same value if it is executed more than once in a single statement, which means that the value is fixed, even if there is a long delay between fetching rows in a cursor.

## Syntax

```
CURRENT_TIME
```

or, alternately

```
CURRENT TIME
```

## Results

A time value.

## Examples

```
splice> VALUES CURRENT_TIME;  
1  
-----  
11:02:57  
  
1 row selected
```

# CURRENT\_TIMESTAMP

`CURRENT_TIMESTAMP` returns the current timestamp.

**NOTE:** This function returns the same value if it is executed more than once in a single statement, which means that the value is fixed, even if there is a long delay between fetching rows in a cursor.

## Syntax

```
CURRENT_TIMESTAMP
```

or, alternately

```
CURRENT TIMESTAMP
```

## Results

A timestamp value.

## Examples

```
splice> VALUES CURRENT_TIMESTAMP;  
1  
-----  
2015-11-19 11:03:44.095  
  
1 row selected
```

# CURRENT\_USER

When used outside stored routines, `CURRENT_USER`, [USER](#), and [SESSION\\_USER](#) all return the authorization identifier of the user who created the SQL session.

`SESSION_USER` also always returns this value when used within stored routines.

If used within a stored routine created with `EXTERNAL SECURITY DEFINER`, however, `CURRENT_USER` and [USER](#) return the authorization identifier of the user that owns the schema of the routine. This is usually the creating user, although the database owner could be the creator as well.

For information about definer's and invoker's rights, see [CREATE FUNCTION statement](#).

Each of these functions returns a string of up to 128 characters.

## Syntax

```
CURRENT_USER
```

## Example

```
splice> VALUES CURRENT_USER;  
1  
-----  
SPLICE  
  
1 row selected
```

## See Also

- » [USER](#) function
- » [SESSION\\_USER](#) function
- » [CREATE FUNCTION](#) statement
- » [CREATE PROCEDURE](#) statement

# DATE

The `DATE` function returns a date from a value.

## Syntax

```
DATE ( expression )
```

*expression*

An expression that can be any of the following:

- » A [LONG VARCHAR](#) value, which must represent a valid date in the form `yyyynnn`, where `yyyy` is a four-digit year value, and `nnn` is a three-digit day value in the range 001 to 366.

## Results

The returned result is governed by the following rules:

- » If the argument can be `NULL`, the result can be `NULL`; if the argument is `NULL`, the result is the `NULL` value.
- » If the argument is a date, timestamp, or valid string representation of a date or timestamp, the result is the date part of the value.
- » If the argument is a number, the result is the date that is  $n-1$  days after January 1, 1970, where  $n$  is the integral part of the number.
- » If the argument is a string with a length of 7, the result is a string representation of the date.

## Examples

This example results in an internal representation of '1988-12-25'.

```
splice> VALUES DATE('1988-12-25');
```

This example results in an internal representation of '1972-02-28'.

```
splice> VALUES DATE(789);
```

This example illustrates using date arithmetic with the `DATE` function:

```
splice> select Birthdate - DATE('11/22/1963') AS "DaysSinceJFK" FROM Players WHERE I
D < 20;
DaysSinceJ&
-----
8526
8916
9839
8461
9916
6619
6432
7082
7337
7289
9703
5030
9617
6899
9404
7446
7609
9492
9172

19 rows selected
```

## See Also

- » [CURRENT\\_DATE](#) function
- » [DATE](#) data type
- » [DAY](#) function
- » [EXTRACT](#) function
- » [LASTDAY](#) function
- » [MONTH](#) function
- » [MONTH\\_BETWEEN](#) function
- » [MONTHNAME](#) function
- » [NEXTDAY](#) function
- » [NOW](#) function
- » [QUARTER](#) function
- » [TIME](#) data type

- » [TIMESTAMP](#) function
- » [TO\\_CHAR](#) function
- » [TO\\_DATE](#) function
- » [WEEK](#) function
- » [\*Working with Dates\*](#) in the *Developer's Guide*



# DAY

The `DAY` function returns the day part of a value.

## Syntax

```
DAY ( expression )
```

*expression*

An expression that can be any of the following:

» A [LONG VARCHAR](#) value.

## Results

The returned result is an integer value in the range 1 to 31.

If the argument can be `NULL`, the result can be `NULL`; if the argument is `NULL`, the result is the `NULL` value.

## Examples

Get the current date:

```
splice> VALUES (CURRENT_DATE) ;
1
-----
2015-10-25

1 row selected
```

Now get the current day only:

```
splice> VALUES (DAY (CURRENT_DATE)) ;
1
-----
25

1 row selected
```

Get the day number for each player's birthdate:

```
splice> select Day(Birthdate) AS "Day-of-Birth"
        FROM Players
        WHERE ID < 20
        ORDER BY "Day-of-Birth";
Day-of-Bir&
-----
1
2
5
6
11
12
13
15
16
17
20
21
21
21
22
24
27
30
30

19 rows selected
```

## See Also

- » [CURRENT\\_DATE](#) function
- » [DATE](#) data type
- » [DATE](#) function
- » [EXTRACT](#) function
- » [LASTDAY](#) function
- » [MONTH](#) function
- » [MONTH\\_BETWEEN](#) function
- » [MONTHNAME](#) function
- » [NEXTDAY](#) function
- » [NOW](#) function
- » [QUARTER](#) function

- » [TIME](#) data type
- » [TIMESTAMP](#) function
- » [TO\\_CHAR](#) function
- » [TO\\_DATE](#) function
- » [WEEK](#) function
- » [\*Working with Dates\*](#) in the *Developer's Guide*

# DEGREES

The `DEGREES` function converts (approximately) a specified number from radians to degrees.

**NOTE:** The conversion from radians to degrees is not exact. You should not expect `DEGREES (ACOS (0.5))` to return exactly `60.0`.

## Syntax

```
DEGREES ( number )
```

*number*

A [DOUBLE PRECISION](#) number that specifies the angle you want converted, in radians.

## Example

```
splice> VALUES DEGREES (ACOS (0.5)) ;
1
-----
60.000000000000001
1 row selected
```

## Results

The data type of the returned value is a [DOUBLE PRECISION](#) number.

## See Also

- » [DOUBLE PRECISION](#) data type
- » [ACOS](#) function
- » [ASIN](#) function
- » [ATAN](#) function
- » [ATAN2](#) function
- » [COS](#) function
- » [COSH](#) function

- » [COT](#) function
- » [RADIANS](#) function
- » [SIN](#) function
- » [SINH](#) function
- » [TAN](#) function
- » [TANH](#) function

## DENSE\_RANK()

`DENSE_RANK()` is a *ranking function* that returns the rank of a value within the ordered partition of values defined by its `OVER` clause. Ranking functions are a subset of [window functions](#).

### Syntax

```
DENSE_RANK() OVER ( overClause )
```

*overClause*

See the [OVER](#) clause documentation.

**NOTE:** Ranking functions such as `DENSE_RANK` must include an [ORDER BY](#) clause in the `OVER` clause. This is because the ranking is calculated based on the ordering.

### Results

The resulting data type is [BIGINT](#).

### Usage

The `DENSE_RANK()` and [RANK\(\)](#) analytic functions are very similar. The difference shows up when there are multiple input rows that have the same ranking value. When that happens:

- » The `DENSE_RANK()` function always returns consecutive rankings: if values in the ranking column are the same, they receive the same rank, and the next number in the ranking sequence is then used to rank the row or rows that follow.
- » The `RANK()` function can generate non-consecutive ranking result values: if values in the ranking column are the same (tie values), 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.

Here's a simple example that shows the ranking produced by the two functions for input with duplicate values to illustrate that difference:

Value	RANK	DENSE_RANK
a	1	1
a	1	1
a	1	1
b	4	2
c	5	3
c	5	3
d	7	4
e	8	5

Example

The following query ranks the salaries of players, per team, whose salary is at least \$1 million.

```

SELECT DisplayName, Team, Season, Salary,
       DENSE_RANK() OVER (PARTITION BY Team ORDER BY Salary Desc) "RANK"
FROM Players JOIN Salaries ON Salaries.ID=Players.ID
WHERE Salary>999999 AND Season=2015;

```

DISPLAYNAME	TEAM	SEASON	SALARY	RANK
Mitch Hassleman	Cards	2015	17000000	1
Yuri Milleton	Cards	2015	15200000	2
James Grasser	Cards	2015	9375000	3
Jack Hellman	Cards	2015	8300000	4
Larry Lintos	Cards	2015	7000000	5
Jeremy Johnson	Cards	2015	4125000	6
Mitch Canepa	Cards	2015	3750000	7
Mitch Brandon	Cards	2015	3500000	8
Robert Cohen	Cards	2015	3000000	9
James Woegren	Cards	2015	2675000	10
Sam Culligan	Cards	2015	2652732	11
Barry Morse	Cards	2015	2379781	12
Michael Rastono	Cards	2015	2000000	13
Carl Vanamos	Cards	2015	2000000	13
Alex Wister	Cards	2015	1950000	14
Pablo Bonjourno	Cards	2015	1650000	15
Jonathan Pearlman	Cards	2015	1500000	16
Jan Bromley	Cards	2015	1200000	17
Martin Cassman	Giants	2015	20833333	1
Harry Pennello	Giants	2015	18500000	2
Tam Lassiter	Giants	2015	18000000	3
Buddy Painter	Giants	2015	17277777	4
Thomas Hillman	Giants	2015	12000000	5
Alex Paramour	Giants	2015	10250000	6
Jack Peepers	Giants	2015	9000000	7
Mark Briste	Giants	2015	8000000	8
Marcus Bamburger	Giants	2015	6950000	9
Jalen Ardson	Giants	2015	6000000	10
Steve Raster	Giants	2015	6000000	10
Sam Castleman	Giants	2015	5000000	11
Craig McGawn	Giants	2015	4800000	12
Norman Aikman	Giants	2015	4000000	13
Randy Varner	Giants	2015	4000000	13
Jason Lilliput	Giants	2015	4000000	13
Billy Bopper	Giants	2015	3600000	14
Greg Brown	Giants	2015	3600000	14
Mitch Lovell	Giants	2015	3578825	15
Bob Cranker	Giants	2015	3175000	16
Yuri Piamam	Giants	2015	2100000	17
Joseph Arkman	Giants	2015	1450000	18
Trevor Imhof	Giants	2015	1100000	19
Jason Minman	Giants	2015	1000000	20

42 rows selected



Here's the same query using `RANK` instead of `DENSE_RANK`. Note how tied rankings are handled differently:

```

SELECT DisplayName, Team, Season, Salary,
       RANK() OVER (PARTITION BY Team ORDER BY Salary Desc) "RANK"
FROM Players JOIN Salaries ON Salaries.ID=Players.ID
WHERE Salary>999999 AND Season=2015;

```

DISPLAYNAME	TEAM	SEASON	SALARY	RANK
Mitch Hassleman	Cards	2015	17000000	1
Yuri Milleton	Cards	2015	15200000	2
James Grasser	Cards	2015	9375000	3
Jack Hellman	Cards	2015	8300000	4
Larry Lintos	Cards	2015	7000000	5
Jeremy Johnson	Cards	2015	4125000	6
Mitch Canepa	Cards	2015	3750000	7
Mitch Brandon	Cards	2015	3500000	8
Robert Cohen	Cards	2015	3000000	9
James Woegren	Cards	2015	2675000	10
Sam Culligan	Cards	2015	2652732	11
Barry Morse	Cards	2015	2379781	12
Michael Rastono	Cards	2015	2000000	13
Carl Vanamos	Cards	2015	2000000	13
Alex Wister	Cards	2015	1950000	15
Pablo Bonjourno	Cards	2015	1650000	16
Jonathan Pearlman	Cards	2015	1500000	17
Jan Bromley	Cards	2015	1200000	18
Martin Cassman	Giants	2015	20833333	1
Harry Pennello	Giants	2015	18500000	2
Tam Lassiter	Giants	2015	18000000	3
Buddy Painter	Giants	2015	17277777	4
Thomas Hillman	Giants	2015	12000000	5
Alex Paramour	Giants	2015	10250000	6
Jack Peepers	Giants	2015	9000000	7
Mark Briste	Giants	2015	8000000	8
Marcus Bamburger	Giants	2015	6950000	9
Jalen Ardson	Giants	2015	6000000	10
Steve Raster	Giants	2015	6000000	10
Sam Castleman	Giants	2015	5000000	12
Craig McGawn	Giants	2015	4800000	13
Norman Aikman	Giants	2015	4000000	14
Randy Varner	Giants	2015	4000000	14
Jason Lilliput	Giants	2015	4000000	14
Billy Bopper	Giants	2015	3600000	17
Greg Brown	Giants	2015	3600000	17
Mitch Lovell	Giants	2015	3578825	19
Bob Cranker	Giants	2015	3175000	20
Yuri Piamam	Giants	2015	2100000	21
Joseph Arkman	Giants	2015	1450000	22
Trevor Imhof	Giants	2015	1100000	23
Jason Minman	Giants	2015	1000000	24

42 rows selected

## See Also

- » [Window and Aggregate](#) functions
- » [BIGINT](#) data type
- » [RANK](#) function
- » [OVER](#) clause
- » [\*Using Window Functions\*](#) in the *Developer Guide*.

# DOUBLE

The `DOUBLE` function returns a floating-point number corresponding to a:

- » number if the argument is a numeric expression
- » character string representation of a number if the argument is a string expression

## Numeric to Double

```
DOUBLE [PRECISION] (NumericExpression )
```

### *NumericExpression*

The argument is an expression that returns a value of any built-in numeric data type.

## Results

The data type of the returned value is a [DOUBLE PRECISION](#) number.

If the argument can be `NULL`, the result can be `NULL`; if the argument is `NULL`, the result is the `NULL` value.

The result is the same value that would result if the argument were assigned to a double-precision floating-point column or variable.

## Character String to Double

```
DOUBLE (StringExpression )
```

### *StringExpression*

The argument can be of type [VARCHAR](#) in the form of a numeric constant. Leading and trailing blanks in argument are ignored.

## Results

The data type of the returned value is a [DOUBLE PRECISION](#) number.

If the argument can be `NULL`, the result can be `NULL`; if the argument is `NULL`, the result is the `NULL` value.

The result is the same value that would result if the string was considered a constant and assigned to a double-precision floating-point column or variable.

## Example

```
splice> VALUES DOUBLE(84.4);  
1  
-----  
84.4  
  
1 row selected
```

## See Also

» [About Data Types](#)

# EXP

The `EXP` function returns  $e$  raised to the power of the specified number. The constant  $e$  is the base of the natural logarithms.

## Syntax

```
EXP ( number )
```

*number*

A [DOUBLE PRECISION](#) number that specifies the exponent to which you want to raise  $e$ .

## Example

```
splice> VALUES EXP(1.234);  
1  
-----  
3.43494186080076  
  
1 row selected
```

## Results

The data type of the result is a [DOUBLE PRECISION](#) number.

## See Also

- » [About Data Types](#)
- » [DOUBLE PRECISION](#) data type

# EXTRACT

You can use the `EXTRACT` built-in function can use to extract specific information from date and time values.

## Syntax

```
EXTRACT ( infoType FROM dateExpr );
```

### *infoType*

The value (information) that you want to extract and return from the date-time expression. This can be one of the following values:

#### *YEAR*

The four-digit year value is extracted from the date-time expression.

#### *QUARTER*

The single digit (1–4) quarter number is extracted from the date-time expression.

#### *MONTH*

The month number (1–12) is extracted from the date-time expression.

#### *MONTHNAME*

The full month name (e.g. `September`) is extracted from the date-time expression.

#### *WEEK*

The week-of-year number (1 is the first week) is extracted from the date-time expression.

#### *WEEKDAY*

The day-of-week number (1–7, with Monday as 1 and Sunday as 7) is extracted from the date-time expression.

#### *WEEKDAYNAME*

The day-of-week name (e.g. `Tuesday`) is extracted from the date-time expression.

#### *DAYOFYEAR*

The numeric day-of-year (0–366) is extracted from the date-time expression.

#### *DAY*

The numeric day-of-month (0–31) is extracted from the date-time expression.

#### *HOURL*

The numeric hour (0–23) is extracted from the date-time expression.

Note that Splice Machine [DATE](#) values do not include time information and will not work correctly with this *infoType*.

#### *MINUTE*

The numeric minute (0–59) is extracted from the date-time expression.

Note that Splice Machine [DATE](#) values do not include time information and will not work correctly with this *infoType*.

#### *SECOND*

The numeric second (0–59) is extracted from the date-time expression.

Note that Splice Machine [DATE](#) values do not include time information and will not work correctly with this *infoType*.

#### *dateExpr*

The date-time expression from which you wish to extract information.

Note that Splice Machine [DATE](#) values do not include time information and thus will not produce correct values if you specify `HOUR`, `MINUTE`, or `SECOND` *infoTypes*.



## Examples

```
splice> SELECT Birthdate,
      EXTRACT (Quarter FROM Birthdate) "Quarter",
      EXTRACT (Week FROM Birthdate) "Week",
      EXTRACT(WeekDay FROM Birthdate) "Weekday"
      FROM Players
      WHERE ID < 20
      ORDER BY "Quarter";
```

BIRTHDATE	Quarter	Week	Weekday
1987-03-27	1	13	5
1987-01-21	1	4	3
1991-01-15	1	3	2
1982-01-05	1	1	2
1990-03-22	1	12	4
1989-01-01	1	52	7
1988-04-20	2	16	3
1983-04-13	2	15	3
1990-06-16	2	24	6
1984-04-11	2	15	3
1981-07-02	3	27	4
1977-08-30	3	35	2
1989-08-21	3	34	1
1984-09-21	3	38	5
1990-10-30	4	44	2
1983-12-24	4	51	6
1983-11-06	4	44	7
1982-10-12	4	41	2
1989-11-17	4	46	5

19 rows selected

```
splice> values EXTRACT(monthname FROM '2009-09-02 11:22:33.04');
```

1

September

```
splice> values EXTRACT(weekdayname FROM '2009-11-07 11:22:33.04');
```

1

Saturday

1 row selected

```
splice> values EXTRACT(dayofyear FROM '2009-02-01 11:22:33.04');
```

1

32

1 row selected

```
splice> values EXTRACT(hour FROM '2009-07-02 11:22:33.04');
```

1

```

-----
11
1 row selected

splice> values EXTRACT(minute FROM '2009-07-02 11:22:33.04');
1
-----
22
1 row selected

splice> values EXTRACT(second FROM '2009-07-02 11:22:33.04');
1
-----
33
1 row selected

```

## See Also

- » [CURRENT\\_DATE](#) function
- » [DATE](#) data type
- » [DATE](#) function
- » [DAY](#) function
- » [LASTDAY](#) function
- » [MONTH](#) function
- » [MONTH\\_BETWEEN](#) function
- » [MONTHNAME](#) function
- » [NEXTDAY](#) function
- » [NOW](#) function
- » [QUARTER](#) function
- » [TIME](#) data type
- » [TIMESTAMP](#) function
- » [TO\\_CHAR](#) function
- » [TO\\_DATE](#) function
- » [WEEK](#) function
- » *[Working with Dates](#) in the *Developer's Guide**

# FIRST\_VALUE

`FIRST_VALUE` is a window function that returns the values of a specified expression that is evaluated at the first row of a window for the current row. This means that you can select a first value from a set of rows without having to use a self join.

## Syntax

```
FIRST_VALUE ( expression [ {IGNORE | RESPECT} NULLS ] ) OVER ( overClause )
```

### *expression*

The expression to evaluate; typically a column name or computation involving a column name.

### *IGNORE NULLS*

If this optional qualifier is specified, `NULL` values are ignored, and the first non-`NULL` value is evaluated.

If you specify this and all values are `NULL`, `FIRST_VALUE` returns `NULL`.

### *RESPECT NULLS*

This qualifier is the default behavior: it specifies that the first value is always returned, even if it is `NULL`.

### *overClause*

See the [OVER](#) clause documentation.

## Usage Notes

Splice Machine recommends that you use the `FIRST_VALUE` function with the [ORDER BY](#) clause to produce deterministic results.

## Results

Returns value(s) resulting from the evaluation of the specified expression; the return type is of the same value type as the data stored in the column used in the expression..

- » `FIRST_VALUE` returns the first value in the set, unless that value is `NULL` and you have specified the `IGNORE NULLS` qualifier; if you've specified `IGNORE NULLS`, this function returns the first non-`NULL` value in the set.
- » If all values in the set are `NULL`, `FIRST_VALUE` always returns `NULL`.

**NOTE:** Splice Machine always sorts `NULL` values first in the results.

## Examples

The following query finds all players with 10 or more HomeRuns, and compares each player's home run count with the lowest total within that group on his team:

```
splice> SELECT Team, DisplayName, HomeRuns,
  FIRST_VALUE(HomeRuns) OVER (PARTITION BY Team ORDER BY HomeRuns
    ROWS BETWEEN UNBOUNDED PRECEDING AND UNBOUNDED FOLLOWING) "Least"
  FROM Players JOIN Batting ON Players.ID=Batting.ID
  WHERE HomeRuns > 10
  ORDER BY Team, HomeRuns DESC;
```

TEAM	DISPLAYNAME	HOMER&	Least
Cards	Mitch Canepa	28	11
Cards	Jonathan Pearlman	17	11
Cards	Roger Green	17	11
Cards	Michael Rastono	13	11
Cards	Jack Hellman	13	11
Cards	Kelly Wacherman	11	11
Giants	Bob Cranker	21	12
Giants	Buddy Painter	19	12
Giants	Billy Bopper	18	12
Giants	Mitch Duffer	12	12

10 rows selected

## See Also

- » [Window and Aggregate](#) functions
- » [AVG](#) function
- » [COUNT](#) function
- » [LAG](#) function
- » [LAST\\_VALUE](#) function
- » [LEAD](#) function
- » [MIN](#) function
- » [SUM](#) function
- » [OVER](#) clause
- » [Using Window Functions](#)

# FLOOR

The `FLOOR` function rounds the specified number down, and returns the largest number that is less than or equal to the specified number.

## Syntax

```
FLOOR ( number )
```

*number*

A [DOUBLE PRECISION](#) number.

## Example

```
splice> VALUES FLOOR(84.4);
1
-----
84
1 row selected
```

## Results

The data type of the result is a [DOUBLE PRECISION](#) number. The returned value is equal to a mathematical integer.

- » If the specified number is `NULL`, the result of this function is `NULL`.
- » If the specified number is equal to a mathematical integer, the result of this function is the same as the specified number.
- » If the specified number is zero (0), the result of this function is zero.

## See Also

- » [About Data Types](#)
- » [DOUBLE PRECISION](#) data type

# HOUR

The `HOUR` function returns the hour part of a value.

## Syntax

```
HOUR ( expression )
```

*expression*  
An expression that can be a time, timestamp, or a valid character string representation of a time or timestamp.

## Results

The returned result is an integer value in the range 0 to 24.

If the argument can be `NULL`, the result can be `NULL`; if the argument is `NULL`, the result is the `NULL` value.

## Example

```
splice> values ( NOW, HOUR(NOW), MINUTE(NOW), SECOND(NOW) );
1 | 2 | 3 | 4
-----
2015-11-12 17:48:55.217 | 17 | 48 | 55.217

1 row selected
```

## See Also

- >> [About Data Types](#)
- >> [TIME](#) data value
- >> [TIMESTAMP](#) data value
- >> [MINUTE](#) function
- >> [TIMESTAMP](#) function
- >> [TIMESTAMPADD](#) function
- >> [TIMESTAMPDIFF](#) function

# INITCAP

The `INITCAP` function converts the first letter of each word in a string to uppercase, and converts any remaining characters in each word to lowercase. Words are delimited by white space characters, or by characters that are not alphanumeric.

## Syntax

```
INITCAP( charExpression );
```

### *charExpression*

The string to be converted. This can be a `CHAR` or `VARCHAR` data type, or another type that gets implicitly converted.

## Results

The returned string has the same data type as the input `charExpression`.

## Examples

```
splice> VALUES( INITCAP('this is a test') );
1
-----
This Is A Test
1 row selected

splice> VALUES( INITCAP('tHIS iS a test') );
1
-----
This Is A Test
1 row selected
```

## See Also

- » [About Data Types](#)
- » [Concatenation](#) operator
- » [INSTR](#) function
- » [LCASE](#) function
- » [LENGTH](#) function



- » [LOCATE](#) function
- » [LTRIM](#) function
- » [REGEX\\_LIKE](#) operator
- » [REPLACE](#) function
- » [RTRIM](#) function
- » [SUBSTR](#) function
- » [TRIM](#) function
- » [UCASE](#) function

# INSTR

The `INSTR` function returns the index of the first occurrence of a substring in a string.

## Syntax

```
INSTR(str, substring)
```

*str*

The string in which to search for the substring.

*substring*

The substring to search for.

## Results

Returns the index in `str` of the first occurrence of `substring`.

The first index is 1.

If `substring` is not found, `INSTR` returns 0.

## Examples

```
splice> SELECT DisplayName, INSTR(DisplayName, 'Pa') "Position"
FROM Players
WHERE (INSTR(DisplayName, 'Pa') > 0)
ORDER BY DisplayName;
```

DISPLAYNAME	Position
Alex Paramour	6
Buddy Painter	7
Jeremy Packman	8
Pablo Bonjourno	1
Paul Kaster	1

5 rows selected

## See Also

» [About Data Types](#)

- » [Concatenation](#) operator
- » [INITCAP](#) function
- » [LCASE](#) function
- » [LENGTH](#) function
- » [LOCATE](#) function
- » [LTRIM](#) function
- » [REGEX\\_LIKE](#) operator
- » [REPLACE](#) function
- » [RTRIM](#) function
- » [SUBSTR](#) function
- » [TRIM](#) function
- » [UCASE](#) function

# INTEGER

The `INTEGER` function returns an integer representation of a number or character string in the form of an integer constant.

## Syntax

```
INT[EGER] (NumericExpression | CharacterExpression )
```

### *NumericExpression*

An expression that returns a value of any built-in numeric data type.

### *CharacterExpression*

An expression that returns a character string value of length not greater than the maximum length of a character constant. Leading and trailing blanks are eliminated and the resulting string must conform to the rules for forming an SQL integer constant. The character string cannot be a long string.

## Results

The result of the function is a large integer.

- » If the argument can be `NULL`, the result can be `NULL`; if the argument is `NULL`, the result is the `NULL` value.
- » If the argument is a numeric-expression, the result is the same number that would occur if the argument were assigned to a large integer column or variable. If the whole part of the argument is not within the range of integers, an error occurs. The decimal part of the argument is truncated if present.
- » If the argument is a character-expression, the result is the same number that would occur if the corresponding integer constant were assigned to a large integer column or variable.

## Example

The following query truncates the number of innings pitches by using the `INTEGER` function:

```
splice> SELECT DisplayName, INTEGER(Innings) "Innings"
        FROM Pitching JOIN Players ON Pitching.ID=Players.ID
        WHERE Innings > 50
        ORDER BY Innings DESC;
```

DISPLAYNAME	Innings
Marcus Bamburger	218
Jason Larrimore	218
Milt Warrimore	181
Carl Marin	179
Charles Heillman	177
Larry Lintos	175
Randy Varner	135
James Grasser	129
Thomas Hillman	123
Jack Peepers	110
Tam Lassiter	76
Yuri Piamam	76
Ken Straiter	74
Gary Kosovo	73
Tom Rather	68
Steve Mossely	63
Carl Vanamos	61
Martin Cassman	60
Tim Lentleson	60
Sam Castleman	58
Steve Raster	57
Mitch Lovell	55
Harold Sermer	51

23 rows selected

## See Also

» [About Data Types](#)

# LAG

LAG returns the values of a specified expression that is evaluated at the specified offset number of rows before the current row in a window.

## Syntax

```
LAG ( expression [ , offset ] ) OVER ( overClause )
```

### *expression*

The expression to evaluate; typically a column name or computation involving a column name.

### *offset*

An integer value that specifies the offset (number of rows) from the current row at which you want the expression evaluated.

The default value is 1.

### *overClause*

See the [OVER](#) clause documentation.

Our current implementation of this function does not allow for specifying a default value, as is possible in some other database software.

## Usage Notes

Splice Machine recommends that you use the LAG function with the [ORDER BY](#) clause to produce deterministic results.

## Results

Returns value(s) resulting from the evaluation of the specified expression; the return type is of the same value type as the data stored in the column used in the expression.

## Examples

The following example shows the salaries per position for players in our baseball database, grouped by position, and ordered from highest salary to lowest for each position:

```
splice> SELECT Position, Players.ID, Salary,
      LAG(Salary) OVER (PARTITION BY Position ORDER BY Salary DESC) "PrevHigherSalary"
      FROM Players JOIN Salaries ON Players.ID=Salaries.ID
      WHERE Salary > 999999
      ORDER BY Position, Salary DESC;
```

POS&	ID	SALARY	PrevHigherSalary
------	----	--------	------------------

1B	2	3600000	NULL
1B	63	2379781	3600000
1B	50	2000000	2379781
3B	14	4800000	NULL
3B	53	3750000	4800000
C	1	17277777	NULL
C	49	15200000	17277777
CF	7	10250000	NULL
CF	59	4125000	10250000
CF	55	1650000	4125000
LF	54	17000000	2000000
LF	6	4000000	17000000
LF	27	1100000	4000000
P	34	20833333	NULL
P	33	18000000	20833333
P	31	12000000	18000000
P	76	9375000	12000000
P	32	9000000	9375000
P	75	7000000	9000000
P	28	6950000	7000000
P	40	6000000	6950000
P	41	6000000	6000000
P	46	5000000	6000000
P	30	4000000	5000000
P	43	4000000	4000000
P	35	3578825	4000000
P	86	3500000	3578825
P	82	3000000	3500000
P	88	2675000	3000000
P	90	2652732	2675000
P	36	2100000	2652732
P	79	2000000	2100000
P	80	1950000	2000000
P	94	1200000	1950000
RF	8	18500000	NULL
RF	56	8300000	18500000
RF	12	8000000	8300000
RF	10	1000000	8000000
SS	4	3175000	NULL
SS	52	1500000	3175000
UT	17	1450000	NULL

41 rows selected

## See Also

- » [Window and Aggregate](#) functions
- » [AVG](#) function
- » [COUNT](#) function
- » [FIRST\\_VALUE](#) function
- » [LAST\\_VALUE](#) function
- » [LEAD](#) function
- » [MIN](#) function
- » [SUM](#) function
- » [OVER](#) clause
- » [\*Using Window Functions\*](#) in the *Developer Guide*.



# LAST\_DAY

The `LAST_DAY` function returns the date of the last day of the month that contains the input date.

## Syntax

```
LAST_DAY ( dateExpression )
```

*dateExpression*

A date value.

## Results

The return type is always [DATE](#), regardless of the data type of the *dateExpression*.

## Examples

```
Examples:
splice> values (LAST_DAY(CURRENT_DATE));
1
-----
2015-11-30

splice> values (LAST_DAY(DATE(CURRENT_TIMESTAMP)));
1
-----
2015-11-30

splice> SELECT DISPLAYNAME, BirthDate, LAST_DAY(BirthDate) "MonthEnd"
      FROM Players
      WHERE MONTH(BirthDate) IN (2, 5, 12);
DISPLAYNAME          |BIRTHDATE |MonthEnd
-----
Tam Croonster        |1980-12-19|1980-12-31
Jack Peepers         |1981-05-31|1981-05-31
Jason Martell        |1982-02-01|1982-02-28
Kameron Fannais      |1982-05-24|1982-05-31
Jonathan Pearlman    |1982-05-28|1982-05-31
Greg Brown           |1983-12-24|1983-12-31
Edward Erdman        |1985-12-21|1985-12-31
Jonathan Wilson      |1986-05-14|1986-05-31
Reed Lister          |1986-12-16|1986-12-31
Larry Lintos         |1987-05-12|1987-05-31
Taylor Trantula      |1987-12-17|1987-12-31
Tim Lentleson        |1988-02-21|1988-02-29
Cameron Silliman     |1988-12-21|1988-12-31
Nathan Nickels       |1989-05-04|1989-05-31
Tom Rather           |1990-05-29|1990-05-31
Mo Grandosi          |1992-02-16|1992-02-29

16 rows selected
```

## See Also

- » [CURRENT\\_DATE](#) function
- » [DATE](#) data type
- » [DATE](#) function
- » [DAY](#) function
- » [EXTRACT](#) function
- » [MONTH](#) function

- » [MONTH\\_BETWEEN](#) function
- » [MONTHNAME](#) function
- » [NEXTDAY](#) function
- » [NOW](#) function
- » [QUARTER](#) function
- » [TIME](#) data type
- » [TIMESTAMP](#) function
- » [TO\\_CHAR](#) function
- » [TO\\_DATE](#) function
- » [WEEK](#) function
- » *[Working with Dates](#) in the *Developer's Guide**

# LAST\_VALUE

`LAST_VALUE` is a window function that returns the values of a specified expression that is evaluated at the last row of a window for the current row. This means that you can select a last value from a set of rows without having to use a self join.

## Syntax

```
LAST_VALUE ( expression [ {IGNORE | RESPECT} NULLS ] ) OVER ( overClause )
```

### *expression*

The expression to evaluate; typically a column name or computation involving a column name.

### *IGNORE NULLS*

If this optional qualifier is specified, `NULL` values are ignored, and the first non-`NULL` value is evaluated.

If you specify this and all values are `NULL`, `LAST_VALUE` returns `NULL`.

### *RESPECT NULLS*

This qualifier is the default behavior: it specifies that the last value is always returned, even if it is `NULL`.

### *overClause*

See the [OVER](#) clause documentation.

## Usage Notes

Splice Machine recommends that you use the `LAST_VALUE` function with the [ORDER BY](#) clause to produce deterministic results.

## Results

Returns value(s) resulting from the evaluation of the specified expression; the return type is of the same value type as the data stored in the column used in the expression..

- » `LAST_VALUE` returns the last value in the set, unless that value is `NULL` and you have specified the `IGNORE NULLS` qualifier; if you've specified `IGNORE NULLS`, this function returns the last non-`NULL` value in the set.
- » If all values in the set are `NULL`, `LAST_VALUE` always returns `NULL`.

**NOTE:** Splice Machine always sorts `NULL` values first in the results.

## Examples

The following query finds all players with 10 or more HomeRuns, and compares each player's home run count with the highest total on his team:

```
splice> SELECT Team, DisplayName, HomeRuns,
  LAST_VALUE(HomeRuns) OVER (PARTITION BY Team ORDER BY HomeRuns
    ROWS BETWEEN UNBOUNDED PRECEDING AND UNBOUNDED FOLLOWING) "Most"
  FROM Players JOIN Batting ON Players.ID=Batting.ID
  WHERE HomeRuns > 10
  ORDER BY Team, HomeRuns DESC;
```

TEAM	DISPLAYNAME	HOMER&	Most
Cards	Mitch Canepa	28	28
Cards	Jonathan Pearlman	17	28
Cards	Roger Green	17	28
Cards	Michael Rastono	13	28
Cards	Jack Hellman	13	28
Cards	Kelly Wacherman	11	28
Giants	Bob Cranker	21	21
Giants	Buddy Painter	19	21
Giants	Billy Bopper	18	21
Giants	Mitch Duffer	12	21

10 rows selected

## See Also

- » [Window and Aggregate](#) functions
- » [AVG](#) function
- » [COUNT](#) function
- » [FIRST\\_VALUE](#) function
- » [LAG](#) function
- » [LEAD](#) function
- » [MIN](#) function
- » [SUM](#) function
- » [OVER](#) clause
- » [Using Window Functions](#) in the *Developer Guide*.

## LCASE or LOWER

LCASE or LOWER returns a string in which all alphabetic characters in the input character expression have been converted to lowercase.

**NOTE:** LOWER and LCASE follow the database locale.

### Syntax

```
LCASE or LOWER ( CharacterExpression )
```

*CharacterExpression*

A [LONG VARCHAR](#) data type, or any built-in type that is implicitly converted to a string (but not a bit expression).

### Results

The data type of the result is as follows:

- » If the *CharacterExpression* evaluates to NULL, this function returns NULL.
- » If the *CharacterExpression* is of type [CHAR](#).
- » If the *CharacterExpression* is of type [LONG VARCHAR](#).
- » Otherwise, the return type is [VARCHAR](#).

The length and maximum length of the returned value are the same as the length and maximum length of the parameter.

## Examples

```
splice> SELECT LCASE(DisplayName)
        FROM Players
        WHERE ID < 11;
1
-----
buddy painter
billy bopper
john purser
bob cranker
mitch duffer
norman aikman
alex paramour
harry pennello
greg brown
jason minman

10 rows selected
```

## See Also

- » [About Data Types](#)
- » [Concatenation](#) operator
- » [INITCAP](#) function
- » [INSTR](#) function
- » [LENGTH](#) function
- » [LOCATE](#) function
- » [LTRIM](#) function
- » [REGEX\\_LIKE](#) operator
- » [REPLACE](#) function
- » [RTRIM](#) function
- » [SUBSTR](#) function
- » [TRIM](#) function
- » [UCASE](#) function

# LEAD

**LEAD** is a window function that returns the values of a specified expression that is evaluated at the specified offset number of rows after the current row in a window.

## Syntax

```
LEAD ( expression [ , offset ] ) OVER ( overClause )
```

### *expression*

The expression to evaluate; typically a column name or computation involving a column name.

### *offset*

An integer value that specifies the offset (number of rows) from the current row at which you want the expression evaluated.

The default value is 1.

### *overClause*

See the [OVER](#) clause documentation.

Our current implementation of this function does not allow for specifying a default value, as is possible in some other database software.

## Usage Notes

Splice Machine recommends that you use the **LEAD** function with the [ORDER BY](#) clause to produce deterministic results.

## Results

Returns value(s) resulting from the evaluation of the specified expression; the return type is of the same value type as the data stored in the column used in the expression.



## Examples

```
splice> SELECT DisplayName, Position, Salary,
      LEAD(SALARY) OVER (PARTITION BY Position ORDER BY Salary DESC) "NextLowerSalary"
      FROM Players JOIN Salaries ON Players.ID=Salaries.ID
      WHERE Salary>999999
      ORDER BY Position, Salary DESC;
```

DISPLAYNAME	POS&	SALARY	NextLowerSalary
Billy Bopper	1B	3600000	2379781
Barry Morse	1B	2379781	2000000
Michael Rastono	1B	2000000	NULL
Craig McGawn	3B	4800000	3750000
Mitch Canepa	3B	3750000	NULL
Buddy Painter	C	17277777	15200000
Yuri Milleton	C	15200000	NULL
Alex Paramour	CF	10250000	4125000
Jeremy Johnson	CF	4125000	1650000
Pablo Bonjourno	CF	1650000	NULL
Mitch Hassleman	LF	17000000	4000000
Norman Aikman	LF	4000000	1100000
Trevor Imhof	LF	1100000	NULL
Greg Brown	OF	3600000	NULL
Martin Cassman	P	20833333	18000000
Tam Lassiter	P	18000000	12000000
Thomas Hillman	P	12000000	9375000
James Grasser	P	9375000	9000000
Jack Peepers	P	9000000	7000000
Larry Lintos	P	7000000	6950000
Marcus Bamburger	P	6950000	6000000
Jalen Ardson	P	6000000	6000000
Steve Raster	P	6000000	5000000
Sam Castleman	P	5000000	4000000
Randy Varner	P	4000000	4000000
Jason Lilliput	P	4000000	3578825
Mitch Lovell	P	3578825	3500000
Mitch Brandon	P	3500000	3000000
Robert Cohen	P	3000000	2675000
James Woegren	P	2675000	2652732
Sam Culligan	P	2652732	2100000
Yuri Piamam	P	2100000	2000000
Carl Vanamos	P	2000000	1950000
Alex Wister	P	1950000	1200000
Jan Bromley	P	1200000	NULL
Harry Pennello	RF	18500000	8300000
Jack Hellman	RF	8300000	8000000
Mark Briste	RF	8000000	1000000
Jason Minman	RF	1000000	NULL
Bob Cranker	SS	3175000	1500000
Jonathan Pearlman	SS	1500000	NULL
Joseph Arkman	UT	1450000	NULL

42 rows selected

## See Also

- » [Window and Aggregate](#) functions
- » [AVG](#) function
- » [COUNT](#) function
- » [FIRST\\_VALUE](#) function
- » [LAG](#) function
- » [LAST\\_VALUE](#) function
- » [MIN](#) function
- » [SUM](#) function
- » [OVER](#) clause
- » [\*Using Window Functions\*](#) in the *Developer Guide*.

# LENGTH

The `LENGTH` function returns the number of characters in a character string expression or bit string expression.

**NOTE:** Since all built-in data types are implicitly converted to strings, this function can act on all built-in data types.

## Syntax

```
LENGTH ( { CharacterExpression | BitExpression } )
```

*CharacterExpression*

A character string expression.

*BitExpression*

A bit string expression.

## Results

The result data type is an integer value.

## Examples

The following three examples show the values returned by the `LENGTH` function for string, integer, and bit string values.

```
splice> SELECT DisplayName, LENGTH(DisplayName) "NameLen"
        FROM Players
        WHERE ID < 11
        ORDER BY "NameLen";
```

DISPLAYNAME	NameLen
Greg Brown	10
John Purser	11
Bob Cranker	11
Billy Bopper	12
Mitch Duffer	12
Jason Minman	12
Buddy Painter	13
Norman Aikman	13
Alex Paramour	13
Harry Pennello	14

10 rows selected

```
splice> SELECT ID,
        LENGTH(CAST(ID AS SMALLINT)) "SMALLINT",
        LENGTH(CAST(ID AS INT)) "INT",
        LENGTH(CAST(ID AS BIGINT)) "BIGINT",
        LENGTH(CAST(ID AS DECIMAL)) "DECIMAL5",
        LENGTH(CAST(ID AS DECIMAL(15,10))) "DECIMAL15",
        LENGTH(CAST(ID AS DECIMAL(30,25))) "DECIMAL30"
        FROM Players
        WHERE ID<11;
```

ID	SMALLINT	INT	BIGINT	DECIMAL5	DECIMAL15	DECIMAL30
1	2	4	8	3	8	16
2	2	4	8	3	8	16
3	2	4	8	3	8	16
4	2	4	8	3	8	16
5	2	4	8	3	8	16
6	2	4	8	3	8	16
7	2	4	8	3	8	16
8	2	4	8	3	8	16
9	2	4	8	3	8	16
10	2	4	8	3	8	16

10 rows selected

```
splice> VALUES LENGTH(X'FF'),
        LENGTH(X'FFFF'),
        LENGTH(X'FFFFFFFF'),
        LENGTH(X'FFFFFFFFFFFFFFFF');
```

```
-----
1
```

```
2  
4  
8  
  
4 rows selected
```

## See Also

- » [About Data Types](#)
- » [Concatenation](#) operator
- » [INITCAP](#) function
- » [INSTR](#) function
- » [LCASE](#) function
- » [LOCATE](#) function
- » [LTRIM](#) function
- » [REGEX\\_LIKE](#) operator
- » [REPLACE](#) function
- » [RTRIM](#) function
- » [SUBSTR](#) function
- » [TRIM](#) function
- » [UCASE](#) function

## LN or LOG

The `LN` and `LOG` functions return the natural logarithm (base  $e$ ) of the specified number.

### Syntax

```
LN ( number )
LOG ( number )
```

*number*

A [DOUBLE PRECISION](#) number that is greater than zero (0).

### Example

```
splice> VALUES ( LOG(84.4), LN(84.4) );
1                | 2
-----
4.435674016019115 | 4.435674016019115

1 row selected
```

### Results

The data type of the returned value is a [DOUBLE PRECISION](#) number.

- » If the specified number is `NULL`, the result of these functions is `NULL`.
- » If the specified number is zero or a negative number, an exception is returned that indicates that the value is out of range (SQL state 22003).

### See Also

- » [About Data Types](#)
- » [DOUBLE PRECISION](#) data type

# LOCATE

The `LOCATE` function is used to search for a string (the *needle*) within another string (the *haystack*). If the desired string is found, `LOCATE` returns the index at which it is found. If the desired string is not found, `LOCATE` returns 0.

## Syntax

```
LOCATE ( CharacterExpression1, CharacterExpression2 [, StartPosition] )
```

### *CharacterExpression1*

A character expression that specifies the string to search **for** in *CharacterExpression2*, sometimes called the needle.

### *CharacterExpression2*

A character expression that specifies the string in which to search, sometimes called the haystack.

### *StartPosition*

(Optional). Specifies the position in *CharacterExpression2* at which the search is to start. This defaults to the start of *CharacterExpression2*, which is the value 1.

## Results

The return type for `LOCATE` is an integer that indicates the index position within the second argument at which the first argument was first located. Index positions start with 1.

- » If the first argument is not found in the second argument, `LOCATE` returns 0.
- » If the first argument is an empty string ( ' ' ), `LOCATE` returns the value of the third argument (or 1 if it was not provided), even if the second argument is also an empty string.
- » If a `NULL` value is passed for either of the `CharacterExpression` arguments, `NULL` is returned



## Examples

```
splice> SELECT DisplayName, LOCATE('Pa', DisplayName, 3) "Position"
        FROM Players
        WHERE (INSTR(DisplayName, 'Pa') > 0)
        ORDER BY DisplayName;
DISPLAYNAME          |Position
-----
Alex Paramour        |6
Buddy Painter        |7
Jeremy Packman       |8
Pablo Bonjourno     |0
Paul Kaster          |0

5 rows selected
```

## See Also

- » [About Data Types](#)
- » [Concatenation](#) operator
- » [INITCAP](#) function
- » [INSTR](#) function
- » [LCASE](#) function
- » [LOCATE](#) function
- » [LTRIM](#) function
- » [REGEX\\_\\_LIKE](#) operator
- » [REPLACE](#) function
- » [RTRIM](#) function
- » [SUBSTR](#) function
- » [TRIM](#) function
- » [UCASE](#) function

# LOG10

The LOG10 function returns the base-10 logarithm of the specified number.

## Syntax

```
LOG10 ( number )
```

*number*

A [DOUBLE PRECISION](#) number that is greater than zero (0).

## Results

The data type of the returned value is a [DOUBLE PRECISION](#) number.

- » If the specified number is NULL, the result of this function is NULL.
- » If the specified number is zero or a negative number, an exception is returned that indicates that the value is out of range (SQL state 22003).

## Example

```
splice> VALUES LOG10(84.4);
1
-----
1.926342446625655

1 row selected
```

## See Also

- » [About Data Types](#)
- » [DOUBLE PRECISION](#) data type

# LTRIM

LTRIM removes blanks from the beginning of a character string expression.

## Syntax

```
LTRIM(CharacterExpression)
```

*CharacterExpression*

A [LONG VARCHAR](#) data type, or any built-in type that is implicitly converted to a string.

## Results

A character string expression. If the *CharacterExpression* evaluates to NULL, this function returns NULL.

## Example

```
splice> VALUES LTRIM('      Space Case  ');
1
-----
Space Case          --- This is the string 'Space Case  '
```

## See Also

- » [About Data Types](#)
- » [Concatenation](#) operator
- » [INITCAP](#) function
- » [INSTR](#) function
- » [LCASE](#) function
- » [LENGTH](#) function
- » [LOCATE](#) function
- » [REGEX\\_LIKE](#) operator
- » [REPLACE](#) function
- » [RTRIM](#) function
- » [SUBSTR](#) function

» [TRIM](#) function

» [UCASE](#) function

# MAX

MAX evaluates the maximum of an expression over a set of rows. You can use it as an [window \(analytic\) function](#).

## Syntax

```
MAX ( [ DISTINCT | ALL ] Expression )
```

### *DISTINCT*

If this qualifier is specified, duplicates are eliminated.

### *ALL*

If this qualifier is specified, all duplicates are retained. This is the default value.

### *Expression*

An expression that evaluates to a numeric data type: [SMALLINT](#).

The expression can contain multiple column references or expressions, but it cannot contain another aggregate or subquery, and it must evaluate to an ANSI SQL numeric data type. This means that you can call methods that evaluate to ANSI SQL data types.

If an expression evaluates to NULL, the aggregate skips that value.

## Usage

Only one *DISTINCT* aggregate expression per *Expression* is allowed. For example, the following query is not valid:

```
--- Not a valid query:
SELECT COUNT(DISTINCT flying_time),
       MAX (DISTINCT miles)
FROM Flights;
```

**NOTE:** Since duplicate values do not change the computation of the maximum value, the *DISTINCT* and *ALL* qualifiers have no impact on this function.

The *Expression* can contain multiple column references or expressions, but it cannot contain another aggregate or subquery. It must evaluate to a built-in data type. You can therefore call methods that evaluate to built-in data types. (For example, a method that returns a *java.lang.Integer* or *int* evaluates to an *INTEGER*.) If an expression evaluates to NULL, the aggregate skips that value.

## Results

The resulting data type is the same as the expression on which it operates; it will never overflow.

The comparison rules for the *Expression*'s type determine the resulting maximum value. For example, if you supply a [VARCHAR](#) argument, the number of blank spaces at the end of the value can affect how the maximum value is evaluated: if the values ' z ' and ' z ' are both stored in a column, you cannot control which one will be returned as the maximum, because blank spaces are ignored for character comparisons.

## Examples

This example finds the birthdate of the youngest player in our database:

```
splice> SELECT MAX (BirthDate) FROM Players;
1
-----
1992-10-19
```

This example finds the maximum number of singles, doubles, triples and homeruns by any player in the database:

```
splice> SELECT MAX(Singles) "Singles", MAX(DOUBLES) "Doubles",
              MAX(Triples) "Triples", Max(HomeRuns) "HomeRuns"
FROM Batting;
Singl&|Doubl&|Tripl&|HomeR&
-----
130    |44      |7       |28
1 row selected
```

## Analytic Example

The following shows the homeruns hit by all batters who hit more than 10, compared to the most Homeruns by a player who hit 10 or more on his team:

```
splice> SELECT Team, DisplayName, HomeRuns,
      MAX(HomeRuns) OVER (PARTITION BY Team ORDER BY HomeRuns
      ROWS BETWEEN UNBOUNDED PRECEDING AND UNBOUNDED FOLLOWING) "Most"
      FROM Players JOIN Batting ON Players.ID=Batting.ID
      WHERE HomeRuns > 10
      ORDER BY Team, HomeRuns DESC;
```

TEAM	DISPLAYNAME	HOMER&	Most
-----	-----	-----	-----
Cards	Mitch Canepa	28	28
Cards	Jonathan Pearlman	17	28
Cards	Roger Green	17	28
Cards	Michael Rastono	13	28
Cards	Jack Hellman	13	28
Cards	Kelly Wacherman	11	28
Giants	Bob Cranker	21	21
Giants	Buddy Painter	19	21
Giants	Billy Bopper	18	21
Giants	Mitch Duffer	12	21

10 rows selected

## See Also

- » [About Data Types](#)
- » [Window and Aggregate Functions](#)
- » [AVG](#) function
- » [COUNT](#) function
- » [MIN](#) function
- » [SUM](#) function
- » [OVER](#) clause

# MIN

MIN evaluates the minimum of an expression over a set of rows. You can use it as an [window \(analytic\) function](#).

## Syntax

```
MIN ( [ DISTINCT | ALL ] Expression )
```

### *DISTINCT*

If this qualifier is specified, duplicates are eliminated.

### *ALL*

If this qualifier is specified, all duplicates are retained. This is the default value.

### *Expression*

An expression that evaluates to a numeric data type: [SMALLINT](#).

The expression can contain multiple column references or expressions, but it cannot contain another aggregate or subquery, and it must evaluate to an ANSI SQL numeric data type. This means that you can call methods that evaluate to ANSI SQL data types.

If an expression evaluates to NULL, the aggregate skips that value.

## Usage

Only one *DISTINCT* aggregate expression per *Expression* is allowed. For example, the following query is not valid:

```
--- Not a valid query:
SELECT COUNT (DISTINCT flying_time),
           MIN (DISTINCT miles)
FROM Flights;
```

**NOTE:** Since duplicate values do not change the computation of the minimum value, the *DISTINCT* and *ALL* qualifiers have no impact on this function.

The *Expression* can contain multiple column references or expressions, but it cannot contain another aggregate or subquery. It must evaluate to a built-in data type. You can therefore call methods that evaluate to built-in data types. (For example, a method that returns a *java.lang.Integer* or *int* evaluates to an *INTEGER*.) If an expression evaluates to NULL, the aggregate skips that value.

## Results

The resulting data type is the same as the expression on which it operates; it will never overflow.



The comparison rules for the *Expression*'s type determine the resulting minimum value. For example, if you supply a [VARCHAR](#) argument, the number of blank spaces at the end of the value can affect how the minimum value is evaluated: if the values ' z ' and ' z ' are both stored in a column, you cannot control which one will be returned as the minimum, because blank spaces are ignored for character comparisons.

## Examples

```
splice> SELECT MIN (BirthDate) FROM Players;
1
-----
1975-07-13
```

This example finds the minimum number of walks and strikeouts by any pitcher in the database:

```
splice> SELECT MIN(Walks) "Walks", Min(Strikeouts) "Strikeouts"
        FROM Pitching JOIN Players on Pitching.ID=Players.ID
        WHERE Position='P';
Walks |Strik&
-----
1      |1

1 row selected
```

## Analytic Example

The following shows the homeruns hit by all batters who hit more than 10, compared to the least number of Homeruns by a player who hit 10 or more on his team:

```
splice> SELECT Team, DisplayName, HomeRuns,
  MIN(HomeRuns) OVER (PARTITION BY Team ORDER BY HomeRuns
    ROWS BETWEEN UNBOUNDED PRECEDING AND UNBOUNDED FOLLOWING) "Least"
  FROM Players JOIN Batting ON Players.ID=Batting.ID
  WHERE HomeRuns > 10
  ORDER BY Team, HomeRuns DESC;
```

TEAM	DISPLAYNAME	HOMER& Least
-----		
Cards	Mitch Canepa	28  11
Cards	Jonathan Pearlman	17  11
Cards	Roger Green	17  11
Cards	Michael Rastono	13  11
Cards	Jack Hellman	13  11
Cards	Kelly Wacherman	11  11
Giants	Bob Cranker	21  12
Giants	Buddy Painter	19  12
Giants	Billy Bopper	18  12
Giants	Mitch Duffer	12  12

10 rows selected

## See Also

- » [Window and Aggregate functions](#)
- » [About Data Types](#)
- » [AVG](#) function
- » [COUNT](#) function
- » [MAX](#) function
- » [SUM](#) function
- » [OVER](#) clause

# MINUTE

The `MINUTE` function returns the minute part of a value.

## Syntax

```
MINUTE ( expression )
```

*expression*  
An expression that can be a time, timestamp, or a valid character string representation of a time or timestamp.

## Results

The returned result is an integer value in the range 0 to 59.

If the argument can be `NULL`, the result can be `NULL`; if the argument is `NULL`, the result is the `NULL` value.

## Example

```
splice> VALUES ( NOW, HOUR(NOW), MINUTE(NOW), SECOND(NOW) );
1 | 2 | 3 | 4
-----
2015-11-12 17:48:55.217 | 17 | 48 | 55.217

1 row selected
```

## See Also

- >> [About Data Types](#)
- >> [TIMESTAMP](#) data value
- >> [HOUR](#) function
- >> [SECOND](#) function
- >> [TIMESTAMP](#) function
- >> [TIMESTAMPADD](#) function
- >> [TIMESTAMPDIFF](#) function

# MOD

MOD returns the remainder (modulus) of argument 1 divided by argument 2.

## Syntax

```
mod(number, divisor)
```

*number*

The number for which you want to find the remainder after the division is performed.

*divisor*

The number by which you want to divide.

## Results

The result is negative only if *number* is negative.

The result of the function is:

- » NULL if any argument is NULL.
- » [SMALLINT](#).
- » [SMALLINT](#).
- » [SMALLINT](#).

The result can be NULL; if any argument is NULL, the result is the NULLvalue.

## Examples

```
splice> VALUES MOD(37, 3);
```

```
1
```

```
-----
```

```
1
```

```
1 row selected
```

```
---select players with odd-numbered IDs:
```

```
splice> SELECT ID, Team, DisplayName
        FROM Players
        WHERE MOD(ID, 2) = 1
        ORDER BY ID;
```

ID	TEAM	DISPLAYNAME
-----		
1	Giants	Buddy Painter
3	Giants	John Purser
5	Giants	Mitch Duffer
7	Giants	Alex Paramour
9	Giants	Greg Brown
11	Giants	Kelly Tamlin
13	Giants	Andy Sussman
15	Giants	Elliot Andrews
17	Giants	Joseph Arkman
19	Giants	Jeremy Packman
21	Giants	Jason Pratter
23	Giants	Nathan Nickels
25	Giants	Reed Lister
27	Giants	Trevor Imhof
29	Giants	Charles Heillman
31	Giants	Thomas Hillman
33	Giants	Tam Lassiter
35	Giants	Mitch Lovell
37	Giants	Justin Oscar
39	Giants	Gary Kosovo
41	Giants	Steve Raster
43	Giants	Jason Lilliput
45	Giants	Cory Hammersmith
47	Giants	Barry Bochner
49	Cards	Yuri Milleton
51	Cards	Kelly Wacherman
53	Cards	Mitch Canepa
55	Cards	Pablo Bonjourno
57	Cards	Roger Green
59	Cards	Jeremy Johnson
61	Cards	Tad Philomen
63	Cards	Barry Morse
65	Cards	George Goomba
67	Cards	David Janssen
69	Cards	Edward Erdman
71	Cards	Don Allison
73	Cards	Carl Marin

```
75      |Cards      |Larry Lintos
77      |Cards      |Tim Lentleson
79      |Cards      |Carl Vanamos
81      |Cards      |Steve Mossely
83      |Cards      |Manny Stolanaro
85      |Cards      |Michael Hillson
87      |Cards      |Neil Gaston
89      |Cards      |Mo Grandosi
91      |Cards      |Mark Hasty
93      |Cards      |Stephen Tuvesco
```

```
47 rows selected
```

## See Also

» [About Data Types](#)

» [DOUBLE PRECISION](#) data type

# MONTH

The `MONTH` function returns the month part of a value.

## Syntax

```
MONTH ( expression )
```

*expression*

An expression that can be a time, timestamp, or a valid character string representation of a time or timestamp.

## Results

The returned result is an integer value in the range 1 to 12.

If the argument can be `NULL`, the result can be `NULL`; if the argument is `NULL`, the result is the `NULL` value.

## Examples

Get the current date:

```
splice> VALUES (CURRENT_DATE);
1
-----
2014-05-15
```

Now get the current month only:

```
splice> VALUES (MONTH (CURRENT_DATE) );
1
-----
5
```

Get the month of one week from now:

```
splice> VALUES (MONTH (CURRENT_DATE+7) );
1
-----
5
```

Select all players who were born in December:



```
splice> SELECT DisplayName, Team, BirthDate
        FROM Players
        WHERE MONTH(BirthDate)=12;
DISPLAYNAME          | TEAM          | BIRTHDATE
-----
Greg Brown           | Giants        | 1983-12-24
Reed Lister           | Giants        | 1986-12-16
Cameron Silliman     | Cards         | 1988-12-21
Edward Erdman         | Cards         | 1985-12-21
Taylor Trantula       | Cards         | 1987-12-17
Tam Croonster         | Cards         | 1980-12-19

6 rows selected
```

## See Also

- » [CURRENT\\_DATE](#) function
- » [DATE](#) data type
- » [DATE](#) function
- » [DAY](#) function
- » [EXTRACT](#) function
- » [LASTDAY](#) function
- » [MONTH\\_BETWEEN](#) function
- » [MONTHNAME](#) function
- » [NEXTDAY](#) function
- » [NOW](#) function
- » [QUARTER](#) function
- » [TIME](#) data type
- » [TIMESTAMP](#) function
- » [TO\\_CHAR](#) function
- » [TO\\_DATE](#) function
- » [WEEK](#) function
- » [Working with Dates](#) in the *Developer's Guide*

# MONTHNAME

The `MONTHNAME` function returns a character string containing month name from a date expression.

## Syntax

```
MONTHNAME ( dateExpr );
```

*dateExpr*

The date-time expression from which you wish to extract information.

## Results

The returned month name is specific to the data source location; for English, the returned name will be in the range `January` through `December`, or `Jan .` through `Dec .` For a data source that uses German, the returned name will be in the range `Januar` through `Dezember`.

## Examples

The following query displays the birth month of players:

```
splice> SELECT DisplayName, MONTHNAME(BirthDate) "Month"
        FROM Players
        WHERE ID<20
        ORDER BY MONTH(BirthDate);
```

DISPLAYNAME	Month
Bob Cranker	January
Mitch Duffer	January
Norman Aikman	January
Jeremy Packman	January
Buddy Painter	March
Andy Sussman	March
Billy Bopper	April
Harry Pennello	April
Alex Darba	April
Kelly Tamlin	June
Alex Paramour	July
Mark Briste	August
Elliot Andrews	August
Joseph Arkman	September
John Purser	October
Craig McGawn	October
Jason Minman	November
Henry Socomy	November
Greg Brown	December

19 rows selected

## See Also

- » [CURRENT\\_DATE](#) function
- » [DATE](#) data type
- » [DATE](#) function
- » [DAY](#) function
- » [EXTRACT](#) function
- » [LASTDAY](#) function
- » [MONTH](#) function
- » [MONTH\\_BETWEEN](#) function
- » [NEXTDAY](#) function
- » [NOW](#) function
- » [QUARTER](#) function

- » [TIME](#) data type
- » [TIMESTAMP](#) function
- » [TO\\_CHAR](#) function
- » [TO\\_DATE](#) function
- » [WEEK](#) function
- » [\*Working with Dates\*](#) in the *Developer's Guide*

# MONTH\_BETWEEN

The `MONTH_BETWEEN` function returns the number of months between two dates.

## Syntax

```
MONTH_BETWEEN( date1, date2 );
```

*date1*

The first date.

*date2*

The second date

## Results

If `date2` is later than `date1`, then the result is positive.

If `date2` is earlier than `date1`, then the result is negative.

If `date1` and `date2` are either the same days of the month or both last days of months, then the result is always an integer.

## Examples

```
splice> VALUES (MONTH_BETWEEN(CURRENT_DATE, DATE('2015-8-15')));
1
-----
3.0

splice> SELECT MIN(BirthDate) "Oldest",
               MAX(Birthdate) "Youngest",
               MONTH_BETWEEN(MIN(Birthdate), MAX(BirthDate)) "Months Between"
               FROM Players;
Oldest      |Youngest    |Months Between
-----
1975-07-14  |1992-10-19  |207.0

1 row selected
```

## See Also

» [CURRENT\\_DATE](#) function

- » [DATE](#) data type
- » [DATE](#) function
- » [DAY](#) function
- » [EXTRACT](#) function
- » [LASTDAY](#) function
- » [MONTH](#) function
- » [MONTHNAME](#) function
- » [NEXTDAY](#) function
- » [NOW](#) function
- » [QUARTER](#) function
- » [TIME](#) data type
- » [TIMESTAMP](#) function
- » [TO\\_CHAR](#) function
- » [TO\\_DATE](#) function
- » [WEEK](#) function
- » [\*Working with Dates\*](#) in the *Developer's Guide*

# NEXT\_DAY

The `NEXT_DAY` function returns the date of the next specified day of the week after a specified date.

## Syntax

```
NEXT_DAY( source_date, day_of_week );
```

*source\_date*  
The source date.

*day\_of\_week*  
The day of the week. This is the case-insensitive name of a day in the date language of your session. You can also specify day-name abbreviations, in which case any characters after the recognized abbreviation are ignored. For example, if you're using English, you can use the following values (again, the case of the characters is ignored):

Day Name	Abbreviation
Sunday	Sun
Monday	Mon
Tuesday	Tue
Wednesday	Wed
Thursday	Thu
Friday	Fri
Saturday	Sat

## Results

This function returns the date of the first weekday, as specified by `day_of_week`, that is later than the specified date.

The return type is always `DATE`, regardless of the data type of the `source_date` parameter.

The return value has the same hours, minutes, and seconds components as does the `source_date` parameter value.

## Examples

```
splice> values (NEXT_DAY(CURRENT_DATE, 'tuesday'));
1
-----
2014-09-23
1 row selected

splice> values (NEXT_DAY(CURRENT_DATE, 'monday'));
1
-----
2014-09-29
1 row selected

SELECT DisplayName, BirthDate, NEXT_DAY(BirthDate, 'sunday') as "FirstSunday"
   FROM Players
  WHERE ID < 20;
DISPLAYNAME          |BIRTHDATE |FirstSund&
-----
Buddy Painter        |1987-03-27|1987-03-29
Billy Bopper         |1988-04-20|1988-04-24
John Purser          |1990-10-30|1990-11-04
Bob Cranker          |1987-01-21|1987-01-25
Mitch Duffer         |1991-01-15|1991-01-20
Norman Aikman        |1982-01-05|1982-01-10
Alex Paramour        |1981-07-02|1981-07-05
Harry Pennello      |1983-04-13|1983-04-17
Greg Brown           |1983-12-24|1983-12-25
Jason Minman         |1983-11-06|1983-11-06
Kelly Tamlin         |1990-06-16|1990-06-17
Mark Briste          |1977-08-30|1977-09-04
Andy Sussman         |1990-03-22|1990-03-25
Craig McGawn         |1982-10-12|1982-10-17
Elliot Andrews       |1989-08-21|1989-08-27
Alex Darba           |1984-04-11|1984-04-15
Joseph Arkman        |1984-09-21|1984-09-23
Henry Socomy         |1989-11-17|1989-11-19
Jeremy Packman       |1989-01-01|1989-01-01

19 rows selected
```

## See Also

- » [CURRENT\\_DATE](#) function
- » [DATE](#) data type
- » [DATE](#) function
- » [DAY](#) function



- » [EXTRACT](#) function
- » [LASTDAY](#) function
- » [MONTH](#) function
- » [MONTH\\_BETWEEN](#) function
- » [MONTHNAME](#) function
- » [NOW](#) function
- » [QUARTER](#) function
- » [TIME](#) data type
- » [TIMESTAMP](#) function
- » [TO\\_CHAR](#) function
- » [TO\\_DATE](#) function
- » [WEEK](#) function
- » [\*Working with Dates\*](#) in the *Developer's Guide*

# NOW

The `NOW` function returns the current date and time as a [TIMESTAMP](#) value.

## Syntax

```
NOW ( ) ;
```

## Results

Returns the current date and time as a [TIMESTAMP](#) value.

## Examples

```
splice> VALUES ( NOW ( ) , HOUR (NOW) , MINUTE (NOW) , SECOND (NOW) ) ;
1          | 2          | 3          | 4
-----
2015-11-12 17:48:55.217      | 17         | 48         | 55.217

1 row selected
```

## See Also

- >> [CURRENT\\_DATE](#) function
- >> [DATE](#) data type
- >> [DATE](#) function
- >> [DAY](#) function
- >> [EXTRACT](#) function
- >> [LASTDAY](#) function
- >> [MONTH](#) function
- >> [MONTH\\_BETWEEN](#) function
- >> [MONTHNAME](#) function
- >> [NEXTDAY](#) function
- >> [QUARTER](#) function

- » [TIME](#) data type
- » [TIMESTAMP](#) function
- » [TO\\_CHAR](#) function
- » [TO\\_DATE](#) function
- » [WEEK](#) function
- » [\*Working with Dates\*](#) in the *Developer's Guide*

# NULLIF

The `NULLIF` function compares the values of two expressions; if they are equal, it returns `NULL`; otherwise, it returns the value of the first expression.

## Syntax

```
NULLIF (expression1, expression2 )
```

*expression1*

The first .expression whose value you want to compare.

**NOTE:** You cannot specify the literal `NULL` for *expression1*.

*expression2*

The first .expression whose value you want to compare.

## Results

The `NULLIF` function is logically similar to the following [CASE](#) expression:

```
CASE WHEN expression1 = expression2 THEN NULL ELSE expression1 END;
```

## Example

```
splice> Select DisplayName "Position Player", NULLIF(Position,'P') "Position"
FROM Players
WHERE MOD(ID, 2)=1
ORDER BY Position;
```

Position Player	Pos&
Barry Morse	1B
David Janssen	1B
John Purser	2B
Kelly Tamlin	2B
Kelly Wacherman	2B
Mitch Duffer	3B
Mitch Canepa	3B
Buddy Painter	C
Andy Sussman	C
Yuri Milleton	C
Edward Erdman	C
Alex Paramour	CF
Pablo Bonjourno	CF
Jeremy Johnson	CF
Tad Philomen	CF
Nathan Nickels	IF
George Goomba	IF
Don Allison	IF
Trevor Imhof	LF
Elliot Andrews	MI
Greg Brown	OF
Jeremy Packman	OF
Jason Pratter	OF
Reed Lister	OF
Roger Green	OF
Charles Heillman	NULL
Thomas Hillman	NULL
Tam Lassiter	NULL
Mitch Lovell	NULL
Justin Oscar	NULL
Gary Kosovo	NULL
Steve Raster	NULL
Jason Lilliput	NULL
Cory Hammersmith	NULL
Barry Bochner	NULL
Carl Marin	NULL
Larry Lintos	NULL
Tim Lentleson	NULL
Carl Vanamos	NULL
Steve Mossely	NULL
Manny Stolanaro	NULL
Michael Hillson	NULL
Neil Gaston	NULL
Mo Grandosi	NULL
Mark Hasty	NULL

```
Stephen Tuvesco      | NULL  
Joseph Arkman       | UT
```

```
47 rows selected
```

## See Also

» [CASE](#) expression

## NVL

The `NVL` function returns the first non-`NULL` expression from a list of expressions.

You can also use `NVL` as a variety of a `CASE` expression. For example:

```
NVL( expresssion_1, expression_2,...expression_n);
```

is equivalent to:

```
CASE WHEN expression_1 IS NOT NULL THEN expression_1
      ELSE WHEN expression_1 IS NOT NULL THEN expression_2
      ...
      ELSE expression_n;
```

## Syntax

```
NVL ( expression1, expression2 [, expressionN]* )
```

*expression1*

An expression.

*expression1*

An expression.

*expressionN*

You can specify more than two arguments; **you MUST specify at least two arguments**.

## Usage

`VALUE` is a synonym for `NVL` that is accepted by Splice Machine, but is not recognized by the SQL standard.

## Results

The result is `NULL` only if all of the arguments are `NULL`.

An error occurs if all of the parameters of the function call are dynamic.



## Example

```
-- create table with three different integer types
splice> SELECT ID, FldGames, PassedBalls, WildPitches, Pickoffs,
  NVL(PassedBalls, WildPitches, Pickoffs) as "FirstNonNull"
  FROM Fielding
 WHERE FldGames>50
 ORDER BY ID;
```

ID	FLDGA&	PASSE&	WILDP&	PICKO&	First&
1	142	4	20	0	4
2	131	NULL	NULL	NULL	NULL
3	99	NULL	NULL	NULL	NULL
4	140	NULL	NULL	NULL	NULL
5	142	NULL	NULL	NULL	NULL
6	88	NULL	NULL	NULL	NULL
7	124	NULL	NULL	NULL	NULL
8	51	NULL	NULL	NULL	NULL
9	93	NULL	NULL	NULL	NULL
10	79	NULL	NULL	NULL	NULL
39	73	NULL	NULL	0	0
40	52	NULL	NULL	0	0
41	70	NULL	NULL	2	2
42	55	NULL	NULL	0	0
43	77	NULL	NULL	0	0
46	67	NULL	NULL	0	0
49	134	4	34	2	4
50	119	NULL	NULL	NULL	NULL
51	147	NULL	NULL	NULL	NULL
52	148	NULL	NULL	NULL	NULL
53	152	NULL	NULL	NULL	NULL
54	64	NULL	NULL	NULL	NULL
55	93	NULL	NULL	NULL	NULL
56	147	NULL	NULL	NULL	NULL
57	85	NULL	NULL	NULL	NULL
58	62	NULL	NULL	NULL	NULL
59	64	NULL	NULL	NULL	NULL
62	53	1	11	0	1
64	59	NULL	NULL	NULL	NULL
81	76	NULL	NULL	0	0
82	71	NULL	NULL	1	1
84	68	NULL	NULL	0	0
92	81	NULL	NULL	3	3

33 rows selected

# PI

The `PI` function returns a value that is closer than any other value to `pi`. The constant `pi` is the ratio of the circumference of a circle to the diameter of a circle.

## Syntax

```
PI ( )
```

## Syntax

The data type of the returned value is a [DOUBLE PRECISION](#) number.

## Example

```
splice> VALUES PI();  
1  
-----  
3.14159265358793  
  
1 row selected
```

## See Also

» [DOUBLE PRECISION](#) data type

# QUARTER

The `QUARTER` function returns an integer value representing the quarter of the year from a date expression.

## Syntax

```
QUARTER ( dateExpr );
```

*dateExpr*

The date-time expression from which you wish to extract information.

## Results

The returned week number is in the range 1 to 4. January 1 through March 31 is Quarter 1.

## Examples

```
splice> VALUES QUARTER('2009-01-02 11:22:33.04');
1
-----
1
1 row selected

splice> SELECT DisplayName, BirthDate, Quarter(BirthDate) "Quarter"
      FROM Players
      WHERE ID<20
      ORDER BY "Quarter", BirthDate;
DISPLAYNAME          |BIRTHDATE |Quarter
-----
Norman Aikman         |1982-01-05|1
Bob Cranker           |1987-01-21|1
Buddy Painter         |1987-03-27|1
Jeremy Packman        |1989-01-01|1
Andy Sussman          |1990-03-22|1
Mitch Duffer          |1991-01-15|1
Harry Pennello       |1983-04-13|2
Alex Darba            |1984-04-11|2
Billy Bopper          |1988-04-20|2
Kelly Tamlin          |1990-06-16|2
Mark Briste           |1977-08-30|3
Alex Paramour         |1981-07-02|3
Joseph Arkman         |1984-09-21|3
Elliot Andrews        |1989-08-21|3
Craig McGawn          |1982-10-12|4
Jason Minman          |1983-11-06|4
Greg Brown            |1983-12-24|4
Henry Socomy          |1989-11-17|4
John Purser           |1990-10-30|4

19 rows selected
```

## See Also

- » [CURRENT\\_DATE](#) function
- » [DATE](#) data type
- » [DATE](#) function
- » [DAY](#) function
- » [EXTRACT](#) function
- » [LASTDAY](#) function
- » [MONTH](#) function

- » [MONTH\\_BETWEEN](#) function
- » [MONTHNAME](#) function
- » [NEXTDAY](#) function
- » [NOW](#) function
- » [TIME](#) data type
- » [TIMESTAMP](#) function
- » [TO\\_CHAR](#) function
- » [TO\\_DATE](#) function
- » [WEEK](#) function
- » [\*Working with Dates\*](#) in the *Developer's Guide*

# RADIANS

The `RADIANS` function converts a specified number from degrees to radians.

The specified number is an angle measured in degrees, which is converted to an approximately equivalent angle measured in radians. The specified number must be a [DOUBLE PRECISION](#) number.

**NOTE:** The conversion from degrees to radians is not exact.

The data type of the returned value is a `DOUBLE PRECISION` number.

## Syntax

```
RADIANS ( number )
```

## Example

```
splice> VALUES RADIANS(90);
1
-----
1.5707963267948966

1 row selected
```

## See Also

- » [DOUBLE PRECISION](#) data type
- » [ACOS](#) function
- » [ASIN](#) function
- » [ATAN](#) function
- » [ATAN2](#) function
- » [COS](#) function
- » [COSH](#) function
- » [COT](#) function
- » [DEGREES](#) function
- » [SIN](#) function

» [SINH](#) function

» [TAN](#) function

» [TANH](#) function

# RAND

The `RAND` function returns a random number given a seed number

The `RAND` function returns an [INTEGER](#) seed number.

## Syntax

```
RAND ( seed )
```

## Example

```
splice> VALUES RAND(13);  
1  
-----  
0.7298032243379924  
  
1 row selected
```

## See Also

- » [About Data Types](#)
- » [DOUBLE PRECISION](#) data type



# RANDOM

The `RANDOM` function returns a random number.

The `RANDOM` function returns an [INTEGER](#) seed number.

## Syntax

```
RANDOM()
```

## Example

```
splice> VALUES RANDOM();  
1  
-----  
0.2826393098638572  
  
1 row selected
```

## See Also

- » [About Data Types](#)
- » [DOUBLE PRECISION](#) data type

# RANK()

`RANK()` is a *ranking function* that returns the rank of a value within the ordered partition of values defined by its `OVER` clause. Ranking functions are a subset of [window functions](#).

## Syntax

```
RANK() OVER ( overClause )
```

*overClause*

See the [OVER](#) clause documentation.

**NOTE:** Ranking functions such as `RANK` must include an [ORDER BY](#) clause in the `OVER` clause. This is because the ranking is calculated based on the ordering.

## Results

The resulting data type is [BIGINT](#).

## Usage

The `RANK()` and [DENSE\\_RANK\(\)](#) analytic functions are very similar. The difference shows up when there are multiple input rows that have the same ranking value. When that happens:

- » The `RANK()` function can generate non-consecutive ranking result values: 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.
- » The `DENSE_RANK()` function always returns consecutive rankings: if values in the ranking column are the same, they receive the same rank, and the next number in the ranking sequence is then used to rank the row or rows that follow.

Here's a simple example that shows the ranking produced by the two functions for input with duplicate values to illustrate that difference:

Value	RANK	DENSE_RANK
a	1	1
a	1	1
a	1	1
b	4	2
c	5	3
c	5	3
d	7	4
e	8	5

Example

The following query ranks the salaries of players, per team, whose salary is at least \$1 million.

```

SELECT DisplayName, Team, Season, Salary,
       RANK() OVER (PARTITION BY Team ORDER BY Salary Desc) "RANK"
FROM Players JOIN Salaries ON Salaries.ID=Players.ID
WHERE Salary>999999 AND Season=2015;

```

DISPLAYNAME	TEAM	SEASON	SALARY	RANK
Mitch Hassleman	Cards	2015	17000000	1
Yuri Milletton	Cards	2015	15200000	2
James Grasser	Cards	2015	9375000	3
Jack Hellman	Cards	2015	8300000	4
Larry Lintos	Cards	2015	7000000	5
Jeremy Johnson	Cards	2015	4125000	6
Mitch Canepa	Cards	2015	3750000	7
Mitch Brandon	Cards	2015	3500000	8
Robert Cohen	Cards	2015	3000000	9
James Woegren	Cards	2015	2675000	10
Sam Culligan	Cards	2015	2652732	11
Barry Morse	Cards	2015	2379781	12
Michael Rastono	Cards	2015	2000000	13
Carl Vanamos	Cards	2015	2000000	13
Alex Wister	Cards	2015	1950000	15
Pablo Bonjourno	Cards	2015	1650000	16
Jonathan Pearlman	Cards	2015	1500000	17
Jan Bromley	Cards	2015	1200000	18
Martin Cassman	Giants	2015	20833333	1
Harry Pennello	Giants	2015	18500000	2
Tam Lassiter	Giants	2015	18000000	3
Buddy Painter	Giants	2015	17277777	4
Thomas Hillman	Giants	2015	12000000	5
Alex Paramour	Giants	2015	10250000	6
Jack Peepers	Giants	2015	9000000	7
Mark Briste	Giants	2015	8000000	8
Marcus Bamburger	Giants	2015	6950000	9
Jalen Ardson	Giants	2015	6000000	10
Steve Raster	Giants	2015	6000000	10
Sam Castleman	Giants	2015	5000000	12
Craig McGawn	Giants	2015	4800000	13
Norman Aikman	Giants	2015	4000000	14
Randy Varner	Giants	2015	4000000	14
Jason Lilliput	Giants	2015	4000000	14
Billy Bopper	Giants	2015	3600000	17
Greg Brown	Giants	2015	3600000	17
Mitch Lovell	Giants	2015	3578825	19
Bob Cranker	Giants	2015	3175000	20
Yuri Piamam	Giants	2015	2100000	21
Joseph Arkman	Giants	2015	1450000	22
Trevor Imhof	Giants	2015	1100000	23
Jason Minman	Giants	2015	1000000	24

42 rows selected

Here's the same query using `DENSE_RANK` instead of `RANK`. Note how tied rankings are handled differently:

```

SELECT DisplayName, Team, Season, Salary,
       DENSE_RANK() OVER (PARTITION BY Team ORDER BY Salary Desc) "RANK"
FROM Players JOIN Salaries ON Salaries.ID=Players.ID
WHERE Salary>999999 AND Season=2015;

```

DISPLAYNAME	TEAM	SEASON	SALARY	RANK
Mitch Hassleman	Cards	2015	17000000	1
Yuri Milletteon	Cards	2015	15200000	2
James Grasser	Cards	2015	9375000	3
Jack Hellman	Cards	2015	8300000	4
Larry Lintos	Cards	2015	7000000	5
Jeremy Johnson	Cards	2015	4125000	6
Mitch Canepa	Cards	2015	3750000	7
Mitch Brandon	Cards	2015	3500000	8
Robert Cohen	Cards	2015	3000000	9
James Woegren	Cards	2015	2675000	10
Sam Culligan	Cards	2015	2652732	11
Barry Morse	Cards	2015	2379781	12
Michael Rastono	Cards	2015	2000000	13
Carl Vanamos	Cards	2015	2000000	13
Alex Wister	Cards	2015	1950000	14
Pablo Bonjourno	Cards	2015	1650000	15
Jonathan Pearlman	Cards	2015	1500000	16
Jan Bromley	Cards	2015	1200000	17
Martin Cassman	Giants	2015	20833333	1
Harry Pennello	Giants	2015	18500000	2
Tam Lassiter	Giants	2015	18000000	3
Buddy Painter	Giants	2015	17277777	4
Thomas Hillman	Giants	2015	12000000	5
Alex Paramour	Giants	2015	10250000	6
Jack Peepers	Giants	2015	9000000	7
Mark Briste	Giants	2015	8000000	8
Marcus Bamburger	Giants	2015	6950000	9
Jalen Ardson	Giants	2015	6000000	10
Steve Raster	Giants	2015	6000000	10
Sam Castleman	Giants	2015	5000000	11
Craig McGawn	Giants	2015	4800000	12
Norman Aikman	Giants	2015	4000000	13
Randy Varner	Giants	2015	4000000	13
Jason Lilliput	Giants	2015	4000000	13
Billy Bopper	Giants	2015	3600000	14
Greg Brown	Giants	2015	3600000	14
Mitch Lovell	Giants	2015	3578825	15
Bob Cranker	Giants	2015	3175000	16
Yuri Piamam	Giants	2015	2100000	17
Joseph Arkman	Giants	2015	1450000	18
Trevor Imhof	Giants	2015	1100000	19
Jason Minman	Giants	2015	1000000	20

42 rows selected

## See Also

- » [Window and Aggregate](#) functions
- » [BIGINT](#) data type
- » [DENSE\\_RANK](#) function
- » [OVER](#) clause
- » [Working with Dates](#) in the *Developer's Guide*

## REGEXP\_LIKE Operator

The `REGEXP_LIKE` operator returns `true` if the string matches the regular expression. This function is similar to the `LIKE` predicate, except that it uses regular expressions rather than simple wildcard character matching.

### Syntax

```
REGEXP_LIKE( sourceString, patternString )
```

#### *sourceString*

The character expression to match against the regular expression.

#### *patternString*

The regular expression string used to search for a match in *sourceString*.

The pattern is a `java.util.regex` pattern. You can find documentation for the JDK 8 version here: <http://docs.oracle.com/javase/8/docs/api/java/util/regex/package-summary.html>.

### Results

Returns `true` if the *sourcestring* you are testing matches the specified regular expression in *patternString*.

### Examples

The following query finds all players whose name begins with *Ste*:

```
splice> SELECT DisplayName
        FROM Players
        WHERE REGEXP_LIKE(DisplayName, '^Ste.*');

DISPLAYNAME
-----
Steve Raster
Steve Mossely
Stephen Tuvesco

3 rows selected
```

### See Also

- » [About Data Types](#)
- » [CONCATENATION](#) operator



- » [INITCAP](#) function
- » [INSTR](#) function
- » [LCASE](#) function
- » [LENGTH](#) function
- » [LTRIM](#) function
- » [REPLACE](#) function
- » [RTRIM](#) function
- » [SUBSTR](#) function
- » [TRIM](#) function
- » [UCASE](#) function

# REPLACE

The `REPLACE` function replaces all occurrences of a substring within a string and returns the new string.

## Syntax

```
REPLACE(subjectStr, searchStr, replaceStr)
```

### *subjectStr*

The string you want modified. This can be a literal string or a reference to a `char` or `varchar` value.

### *searchStr*

The substring to replace within *subjectStr*. This can be a literal string or a reference to a `char` or `varchar` value.

### *replaceStr*

The replacement substring. This can be a literal string or a reference to a `char` or `varchar` value.

## Results

A string value.

## Examples

The first examples shows the players on each team with averages greater than .300. The second example shows the result of replacing the team of those players with averages greater than 0.300 who play on one team (the Cards):

```
splice> SELECT DisplayName, Average, Team
        FROM Players JOIN Batting on Players.ID=Batting.ID
        WHERE Average > 0.300 AND Games>50;
```

DISPLAYNAME	AVERAGE	TEAM
Buddy Painter	0.31777	Giants
John Purser	0.31151	Giants
Kelly Tamlin	0.30337	Giants
Stan Post	0.30472	Cards

4 rows selected

```
splice> SELECT DisplayName, Average,
        REPLACE(Team, 'Cards', 'Giants') "TRADED"
        FROM PLAYERS JOIN Batting ON Players.ID=Batting.ID
        WHERE Team='Cards' AND Average > 0.300 AND Games > 50;
```

DISPLAYNAME	AVERAGE	TRADED
Stan Post	0.30472	Giants

1 row selected

## See Also

- » [About Data Types](#)
- » [Concatenation](#) operator
- » [INITCAP](#) function
- » [INSTR](#) function
- » [LCASE](#) function
- » [LENGTH](#) function
- » [LOCATE](#) function
- » [LTRIM](#) function
- » [REGEX\\_LIKE](#) operator
- » [RTRIM](#) function
- » [SUBSTR](#) function
- » [TRIM](#) function
- » [UCASE](#) function

# ROWID

ROWID is a *pseudocolumn* that uniquely defines a single row in a database table.

The term pseudocolumn is used because you can refer to ROWID in the [WHERE](#) clauses of a query as you would refer to a column stored in your database; the difference is you cannot insert, update, or delete ROWID values.

The ROWID value for a given row in a table remains the same for the life of the row, with one exception: the ROWID may change if the table is an index organized table and you change its primary key.

## Syntax

```
ROWID
```

## Usage

You can use a ROWID value to refer to a row in a table in the [WHERE](#) clauses of a query. These values have several valuable uses:

- » They are the fastest way to access a single row.
- » They are a built-in, unique identifier for every row in a table.
- » They provide information about how the rows in a table are stored.

Some important notes about ROWID values:

- » Do not use ROWID as the primary key of a table.
- » The ROWID of a deleted row can later be reassigned to a new row.
- » A ROWID value is associated with a table row when the row is created.
- » ROWID values are unique within a table, but not necessarily unique within a database.
- » If you delete and re-import a row in a table, the ROWID may change.
- » The ROWID value for a row may change if the row is in an index organized table and you change the table's primary key.

## Using ROWID with JDBC

You can access ROWID with JDBC result sets; for example:

```
() ResultSet.getRowId(int);
```

You can also use ROWID in JDBC queries; for example:

```
( ) CallableStatement.setRowId(int, RowId);
( ) PreparedStatement.setRowId(int, RowId);
```

## Examples

This statement selects the unique row address and salary of all records in the employees database in the engineering department:

```
splice> SELECT ROWID, DisplayName, Position
        FROM Players
        WHERE Team='Giants' and Position='OF';
```

ROWID	DISPLAYNAME	POS&
89	Greg Brown	OF
93	Jeremy Packman	OF
95	Jason Pratter	OF
99	Reed Lister	OF

4 rows selected

This statement updates column `c` in all rows in which column `b` equals 10:

```
UPDATE mytable SET c=100 WHERE rowid=(SELECT rowid FROM mytable WHERE b=10);
```

## See Also

- » [SELECT](#) expression
- » [SELECT](#) statement
- » [UPDATE](#) statement
- » [WHERE](#) clause

# ROW\_NUMBER

`ROW_NUMBER()` is a *ranking function* that numbers the rows within the ordered partition of values defined by its `OVER` clause. Ranking functions are a subset of [window functions](#).

## Syntax

```
ROW_NUMBER() OVER ( overClause )
```

*overClause*

See the [OVER](#) clause documentation.

**NOTE:** Ranking functions such as `ROW_NUMBER` must include an [ORDER BY](#) clause in the `OVER` clause. This is because the ranking is calculated based on the ordering.

## Results

The resulting data type is [BIGINT](#).

## Example

The following query ranks the salaries of players on the Cards whose salaries are at least \$1 million:

```
splice> SELECT DisplayName, Salary,
        ROW_NUMBER() OVER (PARTITION BY Team ORDER BY Salary DESC) "RowNum"
        FROM Players JOIN Salaries ON Players.ID=Salaries.ID
        WHERE Team='Cards' and Salary>9999999;
```

DISPLAYNAME	SALARY	RowNum
Mitch Hassleman	17000000	1
Yuri Milletton	15200000	2
James Grasser	9375000	3
Jack Hellman	8300000	4
Larry Lintos	7000000	5
Jeremy Johnson	4125000	6
Mitch Canepa	3750000	7
Mitch Brandon	3500000	8
Robert Cohen	3000000	9
James Woegren	2675000	10
Sam Culligan	2652732	11
Barry Morse	2379781	12
Michael Rastono	2000000	13
Carl Vanamos	2000000	14
Alex Wister	1950000	15
Pablo Bonjourno	1650000	16
Jonathan Pearlman	1500000	17
Jan Bromley	1200000	18

18 rows selected

## See Also

- » [Window and Aggregate](#) functions
- » [BIGINT](#) data type
- » [OVER](#) clause
- » [OVER](#) clause
- » [Using Window Functions](#) in the *Developer Guide*.

# RTRIM

RTRIM removes blanks from the end of a character string expression.

## Syntax

```
RTRIM(CharacterExpression)
```

*CharacterExpression*

A [LONG VARCHAR](#) data type, any built-in type that is implicitly converted to a string.

## Results

A character string expression. If the *CharacterExpression* evaluates to NULL, this function returns NULL.

## Examples

```
splice> VALUES RTRIM('      Space Case      ');
1
-----
      Space Case      --- This is the string '      Space Case'
```

## See Also

- » [About Data Types](#)
- » [Concatenation](#) operator
- » [INITCAP](#) function
- » [INSTR](#) function
- » [LCASE](#) function
- » [LENGTH](#) function
- » [LOCATE](#) function
- » [LTRIM](#) function
- » [REGEX LIKE](#) operator
- » [REPLACE](#) function
- » [SUBSTR](#) function



» [TRIM](#) function

» [UCASE](#) function

# SECOND

The `SECOND` function returns the seconds part of a value.

## Syntax

```
SECOND( expression )
```

*expression*  
An expression that can be a time, timestamp, or a valid character string representation of a time or timestamp.

## Results

The returned result is an integer value in the range 0 to 59.

If the argument can be `NULL`, the result can be `NULL`; if the argument is `NULL`, the result is the `NULL` value.

## Example

```
splice> VALUES ( NOW() , HOUR(NOW) , MINUTE(NOW) , SECOND(NOW) );
1 | 2 | 3 | 4
-----
2015-11-12 17:48:55.217 | 17 | 48 | 55.217

1 row selected
```

## See Also

- >> [About Data Types](#)
- >> [TIMESTAMP](#) data value
- >> [HOUR](#) function
- >> [MINUTE](#) function
- >> [TIMESTAMP](#) function
- >> [TIMESTAMPADD](#) function
- >> [TIMESTAMPDIFF](#) function

# SESSION\_USER

When used outside stored routines, [CURRENT\\_USER](#), [USER](#), and `SESSION_USER` all return the authorization identifier of the user who created the SQL session.

`SESSION_USER` also always returns this value when used within stored routines.

If used within a stored routine created with `EXTERNAL SECURITY DEFINER`, however, `CURRENT_USER` and `USER` return the authorization identifier of the user that owns the schema of the routine. This is usually the creating user, although the database owner could be the creator as well.

For information about definer's and invoker's rights, see [CREATE FUNCTION statement](#).

## Syntax

```
SESSION_USER
```

## Example

```
VALUES SESSION_USER;  
1  
-----  
SPICE  
  
1 row selected
```

## See Also

- » [CURRENT\\_USER](#) function
- » [USER](#) function
- » [CREATE FUNCTION](#) statement
- » [CREATE PROCEDURE](#) statement

# SIGN

The `SIGN` function returns the sign of the specified number.

## Syntax

```
SIGN ( number )
```

*number*

A [DOUBLE PRECISION](#) number that specifies the value whose sign you want.

## Results

The data type of the returned value is [INTEGER](#):

- » If the specified number is `NULL`, the result of this function is `NULL`.
- » If the specified number is zero (0), the result of this function is zero (0).
- » If the specified number is greater than zero (0), the result of this function is plus one (+1).
- » If the specified number is less than zero (0), the result of this function is minus one (-1).

## Example

```
splice> VALUES ( SIGN(84.4), SIGN(-85.5), SIGN(0), SIGN(NULL) );
1          | 2          | 3          | 4
-----
1          | -1         | 0          | NULL
1 row selected
```

## See Also

- » [DOUBLE PRECISION](#) data type

# SIN

The `SIN` function returns the sine of a specified number.

## Syntax

```
SIN ( number )
```

*number*

A [DOUBLE PRECISION](#) number that specifies the angle, in radians, for which you want the sine computed.

## Results

The data type of the returned value is a [DOUBLE PRECISION](#) number.

If *number* is `NULL`, the result of the function is `NULL`.

If *number* is 0, the result of the function is 0.

## Example

```
splice> VALUES SIN(84.4);
1
-----
0.4104993826174394

1 row selected
```

## See Also

- » [DOUBLE PRECISION](#) data type
- » [ACOS](#) function
- » [ASIN](#) function
- » [ATAN](#) function
- » [ATAN2](#) function
- » [COS](#) function
- » [COSH](#) function

- » [COT](#) function
- » [DEGREES](#) function
- » [RADIANS](#) function
- » [SINH](#) function
- » [TAN](#) function
- » [TANH](#) function

# SINH

The `SINH` function returns the hyperbolic sine of a specified number.

## Syntax

```
SINH ( number )
```

*number*

A [DOUBLE PRECISION](#) number that specifies the angle, in radians, for which you want the hyperbolic sine computed.

## Results

The data type of the returned value is a [DOUBLE PRECISION](#) number.

If *number* is `NULL`, the result of the function is `NULL`.

If *number* is 0, the result of the function is 0.

## Example

```
splice> VALUES SINH(84.4);
1
-----
2.2564425307671042E36

1 row selected
```

## See Also

- » [DOUBLE PRECISION](#) data type
- » [ACOS](#) function
- » [ASIN](#) function
- » [ATAN](#) function
- » [ATAN2](#) function
- » [COS](#) function

- » [COSH](#) function
- » [COT](#) function
- » [DEGREES](#) function
- » [RADIANS](#) function
- » [SIN](#) function
- » [TAN](#) function
- » [TANH](#) function



# SMALLINT

The `SMALLINT` function returns a small integer representation of a number or character string, in the form of a small integer constant.

## Syntax

```
SMALLINT ( NumericExpression | CharacterExpression )
```

### *NumericExpression*

An expression that returns a value of any built-in numeric data type.

### *CharacterExpression*

An expression that returns a character string value of length not greater than the maximum length of a character constant. Leading and trailing blanks are eliminated and the resulting string must conform to the rules for forming an SQL integer constant. The value of the constant must be in the range of small integers. The character string cannot be a long string.

## Results

The result of the function is a [SMALLINT](#). If the argument can be `NULL`, the result can be `NULL`. If the argument is `NULL`, the result is the `NULL` value.

If the argument is a *NumericExpression*, the result is the same number that would occur if the argument were assigned to a small integer column or variable. If the whole part of the argument is not within the range of small integers, an error occurs. The decimal part of the argument is truncated if present.

If the argument is a *CharacterExpression*, the result is the same number that would occur if the corresponding integer constant were assigned to a small integer column or variable.

## Examples

Using the `Pitching` table from our Doc Examples database, select the `Era` column in big integer form for further processing in the application:

```
splice> SELECT ID, SMALLINT(Era) "ERA"  
        FROM Pitching  
        WHERE MOD(ID,2) = 0;
```

ID	ERA
28	2
30	4
32	3
34	5
36	3
38	5
40	5
42	2
44	5
46	2
48	5
72	2
74	3
76	2
78	3
80	1
82	3
84	2
86	2
88	0
90	2
92	2
94	2

23 rows selected

## See Also

» [About Data Types](#)

# SQRT

The `SQRT` function returns the square root of a floating point number.

**NOTE:** To execute `SQRT` on data types other than floating point numbers, you must first cast them to floating point types.

## Syntax

```
SQRT (FloatingPointExpression)
```

*FloatingPointExpression*

A `DOUBLE PRECISION` number.

## Results

The return type for `SQRT` is the type of the input parameter value.

## Examples

```
splice> VALUES sqrt(3421E+09);
```

```
1
```

```
-----  
1849594.5501649815
```

```
1 row selected
```

```
-- Shows using SQRT on a SMALLINT column
```

```
splice> select Strikeouts, SQRT(Strikeouts) "SQRT"
```

```
FROM Batting
```

```
WHERE Strikeouts > 50
```

```
ORDER BY Strikeouts;
```

```
STRIK&|SQRT
```

```
-----  
52      |7.211102550927978
```

```
56      |7.483314773547883
```

```
59      |7.681145747868608
```

```
59      |7.681145747868608
```

```
59      |7.681145747868608
```

```
76      |8.717797887081348
```

```
90      |9.486832980505138
```

```
93      |9.643650760992955
```

```
95      |9.746794344808963
```

```
96      |9.797958971132712
```

```
110     |10.488088481701515
```

```
111     |10.535653752852738
```

```
119     |10.908712114635714
```

```
121     |11.0
```

```
147     |12.12435565298214
```

```
151     |12.288205727444508
```

```
16 rows selected
```

```
splice> SELECT ID, FieldingIndependent, SQRT(FieldingIndependent) "SQRT"
```

```
FROM Pitching
```

```
WHERE Mod(ID, 2)=1;
```

```
ID      |FIELDI&|SQRT
```

```
-----  
29      |4.02      |2.004993765576342
```

```
31      |4.53      |2.1283796653792764
```

```
33      |4.29      |2.071231517720798
```

```
35      |4.83      |2.1977260975835913
```

```
37      |3.90      |1.9748417658131499
```

```
39      |4.02      |2.004993765576342
```

```
41      |1.91      |1.3820274961085253
```

```
43      |3.36      |1.833030277982336
```

```
45      |4.81      |2.1931712199461306
```

```
47      |3.13      |1.7691806012954132
```

```
73      |3.21      |1.7916472867168918
```

```
75      |3.44      |1.8547236990991407
```

```
77      |4.53      |2.1283796653792764
```

79	3.74	1.9339079605813716
81	3.78	1.944222209522358
83	2.76	1.6613247725836149
85	5.39	2.32163735324878
89	8.38	2.894822965226026
91	4.63	2.151743479135001
93	3.82	1.9544820285692064

20 rows selected

See Also

» [About Data Types](#)

## STDDEV\_POP

The `STDDEV_POP()` function returns the population standard deviation of a set of numeric values.

It returns `NULL` if no matching row is found.

### Syntax

```
STDDEV_POP ( [ DISTINCT | ALL ] expression )
```

#### *DISTINCT*

If this qualifier is specified, duplicates are eliminated

#### *ALL*

If this qualifier is specified, all duplicates are retained. This is the default value.

#### *expression*

An expression that evaluates to a numeric data type: [SMALLINT](#).

The expression can contain multiple column references or expressions, but it cannot contain another aggregate or subquery, and it must evaluate to an ANSI SQL numeric data type. This means that you can call methods that evaluate to ANSI SQL data types.

If an expression evaluates to `NULL`, the aggregate skips that value.

### Results

This function returns a double-precision number.

If the input expression consists of a single value, the result of the function is `NULL`, not 0.

### Execute Privileges

If authentication and SQL authorization are both enabled, only the database owner has execute privileges on this function by default. The database owner can grant access to other users.

### Example

The following example shows computing the average, population standard deviation, and sample standard deviation from our `Salaries` table:

```
splice> SELECT AVG(Salary) as AvgSalary, STDDEV_POP(Salary) AS PopStdDev, STDDEV_SAMP(Salary) As SampStdDev FROM Salaries;
```

AVGSALARY	POPSTDDEV	SAMPSTDDEV
2949737	4694155.715951055	4719325.63212163

```
1 row selected
```



## STDDEV\_SAMP

The `STDDEV_POP()` function returns the sample standard deviation of a set of numeric values.

It returns `NULL` if no matching row is found.

### Syntax

```
STDDEV_SAMP ( [ DISTINCT | ALL ] expression )
```

#### *DISTINCT*

If this qualifier is specified, duplicates are eliminated

#### *ALL*

If this qualifier is specified, all duplicates are retained. This is the default value.

#### *expression*

An expression that evaluates to a numeric data type: [SMALLINT](#).

The expression can contain multiple column references or expressions, but it cannot contain another aggregate or subquery, and it must evaluate to an ANSI SQL numeric data type. This means that you can call methods that evaluate to ANSI SQL data types.

If an expression evaluates to `NULL`, the aggregate skips that value.

### Results

This function returns a double-precision number.

If the input expression consists of a single value, the result of the function is `NULL`, not 0.

### Execute Privileges

If authentication and SQL authorization are both enabled, only the database owner has execute privileges on this function by default. The database owner can grant access to other users.

### Example

The following example shows computing the average, population standard deviation, and sample standard deviation from our `Salaries` table:

```
splice> SELECT AVG(Salary) as AvgSalary, STDDEV_POP(Salary) AS PopStdDev, STDDEV_SAMP(Salary) As SampStdDev FROM Salaries;
```

AVGSALARY	POPSTDDEV	SAMPSTDDEV
2949737	4694155.715951055	4719325.63212163

```
1 row selected
```

# SUBSTR

The `SUBSTR` function extracts and returns a portion of a character string or bit string expression, starting at the specified character or bit position. You can specify the number of characters or bits you want returned, or use the default length, which is to extract from the specified starting position to the end of the string.

## Syntax

```
SUBSTR({ CharacterExpression },
      StartPosition [, LengthOfSubstring ] )
```

### *CharacterExpression*

A `CHAR`, `VARCHAR`, or `LONG VARCHAR` data type or any built-in type that is implicitly converted to a string (except a bit expression).

### *StartPosition*

An integer expression; for character expressions, this is the starting character position of the returned substring. For bit expressions, this is the bit position of the returned substring.

The first character or bit has a *StartPosition* of 1. If you specify 0, Splice Machine assumes that you mean 1.

If the *StartPosition* is positive, it refers to the position from the start of the source expression (counting the first character as 1) to the beginning of the substring you want extracted. The *StartPosition* value cannot be a negative number.

### *LengthOfSubstring*

An optional integer expression that specifies the length of the extracted substring; for character expressions, this is number of characters to return. For bit expressions, this is the number of bits to return.

If this value is not specified, then `SUBSTR` extracts a substring of the expression from the *StartPosition* to the end of the source expression.

If *LengthOfString* is specified, `SUBSTR` returns a `VARCHAR` or `VARBIT` of length *LengthOfString* starting at the *StartPosition*.

The `SUBSTR` function returns an error if you specify a negative number for the parameter *LengthOfString*.

## Results

For character string expressions, the result type is a [VARCHAR](#) value.

The length of the result is the maximum length of the source type.

## Examples

The following query extracts the first four characters of each player's name, and then extracts the remaining characters:

```
splice> SELECT DisplayName,
      SUBSTR(DisplayName, 1, 4) "1to4",
      SUBSTR(DisplayName, 4) "5ToEnd"
      FROM Players
      WHERE ID < 11;
DISPLAYNAME          |1To4|5ToEnd
-----
Buddy Painter        |Budd|dy Painter
Billy Bopper         |Bill|ly Bopper
John Purser          |John|n Purser
Bob Cranker          |Bob | Cranker
Mitch Duffer         |Mitc|ch Duffer
Norman Aikman        |Norm|man Aikman
Alex Paramour        |Alex|x Paramour
Harry Pennello      |Harr|ry Pennello
Greg Brown           |Greg|g Brown
Jason Minman         |Jaso|on Minman

10 rows selected
```

## See Also

- » [About Data Types](#)
- » [Concatenation](#) operator
- » [INITCAP](#) function
- » [INSTR](#) function
- » [LCASE](#) function
- » [LENGTH](#) function
- » [LOCATE](#) function
- » [LTRIM](#) function
- » [REGEX\\_LIKE](#) operator
- » [REPLACE](#) function
- » [RTRIM](#) function
- » [TRIM](#) function
- » [UCASE](#) function

# SUM

`SUM` returns the sum of values of an expression over a set of rows. You can use it as an [window \(analytic\) function](#).

The `SUM` function takes as an argument any numeric data type or any non-numeric data type that can be implicitly converted to a numeric data type. The function returns the same data type as the numeric data type of the argument.

## Syntax

```
SUM ( [ DISTINCT | ALL ] Expression )
```

### *DISTINCT*

If this qualifier is specified, duplicates are eliminated

If you specify `DISTINCT` in the analytic version of `SUM`, the [OVER](#) clause for your window function cannot include an `ORDER BY` clause or a *frame clause*.

### *ALL*

If this qualifier is specified, all duplicates are retained. This is the default value.

### *Expression*

An expression that evaluates to a numeric data type: [SMALLINT](#).

An *Expression* can contain multiple column references or expressions, but it cannot contain another aggregate or subquery.

If an *Expression* evaluates to `NULL`, the aggregate skips that value.

## Usage

The *Expression* can contain multiple column references or expressions, but it cannot contain another aggregate or subquery. It must evaluate to a built-in numeric data type. If an expression evaluates to `NULL`, the aggregate skips that value.

Only one `DISTINCT` aggregate expression per *Expression* is allowed. For example, the following query is not valid:

```
-- query not allowed
SELECT AVG (DISTINCT flying_time),
       SUM (DISTINCT miles)
FROM Flights;
```

Note that specifying `DISTINCT` can result in a different value, since a smaller number of values may be summed. For example, if a column contains the values 1, 1, 1, 1, and 2, `SUM(col)` returns a greater value than `SUM(DISTINCT col)`.

## Results

The resulting data type is the same as the expression on which it operates (it might overflow).

## Aggregate Examples

These queries compute the total of all salaries for all teams, and then the total for each individually.

```
splice> SELECT SUM(Salary) FROM Salaries;
1
-----
277275362

1 row selected
splice> SELECT SUM(Salary) FROM Salaries JOIN Players ON Salaries.ID=Players.ID WHERE
E Team='Cards';
1
-----
97007230

1 row selected
splice> SELECT SUM(Salary) FROM Salaries JOIN Players ON Salaries.ID=Players.ID WHERE
E Team='Giants';
1
-----
180268132

1 row selected
```

## Analytic Example

This example computes the running total of salaries, per team, counting only the players who make at least \$5 million in salary.

```
splice> SELECT Team, DisplayName, Salary,
      SUM(Salary) OVER(PARTITION BY Team ORDER BY Salary ASC
        ROWS BETWEEN UNBOUNDED PRECEDING AND CURRENT ROW) "Running Total"
      FROM Players JOIN Salaries ON Players.ID=Salaries.ID
      WHERE Salary>5000000
      ORDER BY Team;
```

TEAM	DISPLAYNAME	SALARY	RUNNING TOTAL
Cards	Larry Lintos	7000000	7000000
Cards	Jack Hellman	8300000	15300000
Cards	James Grasser	9375000	24675000
Cards	Yuri Milleton	15200000	39875000
Cards	Mitch Hassleman	17000000	56875000
Giants	Jalen Ardson	6000000	6000000
Giants	Steve Raster	6000000	12000000
Giants	Marcus Bamburger	6950000	18950000
Giants	Mark Briste	8000000	26950000
Giants	Jack Peepers	9000000	35950000
Giants	Alex Paramour	10250000	46200000
Giants	Thomas Hillman	12000000	58200000
Giants	Buddy Painter	17277777	75477777
Giants	Tam Lassiter	18000000	93477777
Giants	Harry Pennello	18500000	111977777
Giants	Martin Cassman	20833333	132811110

16 rows selected

## See Also

- » [About Data Types](#)
- » [Window and aggregate](#) functions
- » [AVG](#) function
- » [COUNT](#) function
- » [MAX](#) function
- » [MIN](#) function
- » [OVER](#) clause
- » [Using Window Functions](#) in the *Developer Guide*.

# TAN

The `TAN` function returns the tangent of a specified number.

## Syntax

```
TAN ( number )
```

*number*

A [DOUBLE PRECISION](#) number that specifies the angle, in radians, for which you want the tangent computed.

## Results

The data type of the returned value is a [DOUBLE PRECISION](#) number.

If *number* is `NULL`, the result of the function is `NULL`.

If *number* is 0, the result of the function is 0.

## Example

```
splice> VALUES TAN(84.4);  
1  
-----  
-0.45017764606194366  
  
1 row selected
```

## See Also

- » [DOUBLE PRECISION](#) data type
- » [ACOS](#) function
- » [ASIN](#) function
- » [ATAN](#) function
- » [ATAN2](#) function
- » [COS](#) function
- » [COSH](#) function



- » [COT](#) function
- » [DEGREES](#) function
- » [RADIANS](#) function
- » [SIN](#) function
- » [SINH](#) function
- » [TANH](#) function

# TANH

The `TANH` function returns the hyperbolic tangent of a specified number.

## Syntax

```
TANH ( number )
```

*number*

A [DOUBLE PRECISION](#) number that specifies the angle, in radians, for which you want the hyperbolic tangent computed.

## Results

The data type of the returned value is a [DOUBLE PRECISION](#) number.

If *number* is `NULL`, the result of the function is `NULL`.

If *number* is 0, the result of the function is 0.

## Example

```
splice> VALUES TANH(1.234);  
1  
-----  
0.8437356625893302  
  
1 row selected
```

## See Also

- » [DOUBLE PRECISION](#) data type
- » [ACOS](#) function
- » [ASIN](#) function
- » [ATAN](#) function
- » [ATAN2](#) function
- » [COS](#) function

- » [COSH](#) function
- » [COT](#) function
- » [DEGREES](#) function
- » [RADIANS](#) function
- » [SIN](#) function
- » [SINH](#) function
- » [TAN](#) function

# TIME

The `TIME` function returns a time from a value.

## Syntax

```
TIME ( expression )
```

*expression*

An expression that can be any of the following:

- » A [TIMESTAMP](#) value
- » A valid string representation of a time or timestamp

## Results

The returned result is governed by the following rules:

- » If the argument can be `NULL`, the result can be `NULL`; if the argument is `NULL`, the result is the `NULL` value.
- » If the argument is a time, the result is that time value.
- » If the argument is a timestamp, the result is the time part of the timestamp.
- » If the argument is a string, the result is the time represented by the string.

## Syntax

```
TIME ( expression )
```

## Example

```
splice> VALUES TIME( CURRENT_TIMESTAMP );
1
-----
18:53:13

1 row selected
```

## See Also

- » [TIME](#) data type
- » [TIMESTAMP](#) data type

# TIMESTAMP

The `TIMESTAMP` function returns a timestamp from a value or a pair of values.

## Syntax

```
TIMESTAMP ( expression1 [, expression2 ] )
```

### *expression1*

If *expression2* is also specified, *expression1* must be a date or a valid string representation of a date.

If only *expression1* is specified, it must be one of the following:

- » A [DATE](#) value
- » A valid SQL string representation of a timestamp

### *expression2*

(Optional). A time or a valid string representation of a time.

## Results

The data type of the result depends on how the input expression(s) were specified:

- » If both *expression1* and *expression2* are specified, the result is a timestamp with the date specified by *expression1* and the time specified by *expression2*. The microsecond part of the timestamp is zero.
- » If only *expression1* is specified and it is a timestamp, the result is that timestamp.
- » If only *expression1* is specified and it is a string, the result is the timestamp represented by that string. If *expression1* is a string of length 14, the timestamp has a microsecond part of zero.

## Examples

This example converts date and time strings into a timestamp value:

```
splice> VALUES TIMESTAMP('2015-11-12', '19:02:43');
1
-----
2015-11-12 19:02:43.0

1 row selected
```

This query shows the timestamp version of the birth date of each player born in the final quarter of the year:

```
splice> SELECT TIMESTAMP(BirthDate)
        FROM Players
        WHERE MONTH(BirthDate) > 10
        ORDER BY BirthDate;
```

```
1
```

```
-----
1980-12-19 00:00:00.0
1983-11-06 00:00:00.0
1983-11-28 00:00:00.0
1983-12-24 00:00:00.0
1984-11-22 00:00:00.0
1985-11-07 00:00:00.0
1985-11-26 00:00:00.0
1985-12-21 00:00:00.0
1986-11-13 00:00:00.0
1986-11-24 00:00:00.0
1986-12-16 00:00:00.0
1987-11-12 00:00:00.0
1987-11-16 00:00:00.0
1987-12-17 00:00:00.0
1988-12-21 00:00:00.0
1989-11-17 00:00:00.0
1991-11-15 00:00:00.0
```

```
17 rows selected
```

## See Also

- » [CURRENT\\_DATE](#) function
- » [DATE](#) data type
- » [DATE](#) function
- » [DAY](#) function
- » [EXTRACT](#) function
- » [LASTDAY](#) function
- » [MONTH](#) function
- » [MONTH\\_BETWEEN](#) function
- » [MONTHNAME](#) function
- » [NEXTDAY](#) function
- » [NOW](#) function
- » [QUARTER](#) function

- » [TIME](#) data type
- » [TO\\_CHAR](#) function
- » [TO\\_DATE](#) function
- » [WEEK](#) function
- » [\*Working with Dates\*](#) in the *Developer's Guide*



# TIMESTAMPADD

The `TIMESTAMPADD` function adds the value of an interval to a timestamp value and returns the sum as a new timestamp. You can supply a negative interval value to subtract from a timestamp.

## Syntax

```
TIMESTAMPADD ( interval, count, timeStampl )
```

### *interval*

One of the following timestamp interval constants:

- » `SQL_TSI_FRAC_SECOND`
- » `SQL_TSI_SECOND`
- » `SQL_TSI_MINUTE`
- » `SQL_TSI_HOUR`
- » `SQL_TSI_DAY`
- » `SQL_TSI_WEEK`
- » `SQL_TSI_MONTH`
- » `SQL_TSI_QUARTER`
- » `SQL_TSI_YEAR`

### *count*

An integer specifying the number of times the interval is to be added to the timestamp. Use a negative integer value to subtract.

### *timeStamp1*

The [timestamp](#) value to which the count of intervals is added.

**NOTE:** If you use a `datetime` column inside the `TIMESTAMPADD` function in a `WHERE` clause, the optimizer cannot use indexes on that column. We strongly recommend not doing this!

## Results

The `TIMESTAMPADD` function returns a [timestamp](#) value that is the result of adding *count intervals* to *timeStamp1*.

## Examples

The following example displays the current timestamp, and the timestamp value two months from now:

```
splice> VALUES ( CURRENT_TIMESTAMP, TIMESTAMPADD(SQL_TSI_MONTH, 2, CURRENT_TIMESTAMP) );
1 | 2
-----
2015-11-23 13:54:16.728 | 2016-01-23 13:54:16.728

1 row selected
```

## See Also

- » [About Data Types](#)
- » [TIMESTAMP](#) data value
- » [HOUR](#) function
- » [MINUTE](#) function
- » [SECOND](#) function
- » [TIMESTAMP](#) function
- » [TIMESTAMPDIFF](#) function
- » [Working with Dates](#) in the *Developer's Guide*

# TIMESTAMPDIFF

The `TIMESTAMPDIFF` function finds the difference between two timestamps, in terms of the specified interval.

## Syntax

```
TIMESTAMPDIFF ( interval, timeStamp1, timeStamp2 )
```

*interval*

One of the following timestamp interval constants:

- » `SQL_TSI_FRAC_SECOND`
- » `SQL_TSI_SECOND`
- » `SQL_TSI_MINUTE`
- » `SQL_TSI_HOUR`
- » `SQL_TSI_DAY`
- » `SQL_TSI_WEEK`
- » `SQL_TSI_MONTH`
- » `SQL_TSI_QUARTER`
- » `SQL_TSI_YEAR`

*timeStamp1*

The first [timestamp](#) value.

*timeStamp2*

The second [timestamp](#) value.

**NOTE:** If you use a `datetime` column inside the `TIMESTAMPDIFF` function in a `WHERE` clause, the optimizer cannot use indexes on that column. We strongly recommend not doing this!

## Results

The `TIMESTAMPDIFF` function returns an integer value representing the count of intervals between the two timestamp values.

## Examples

These examples shows the number of years a player was born after Nov 22, 1963:.

```
splice> SELECT ID, BirthDate, TIMESTAMPDIFF(SQL_TSI_YEAR, Date('11/22/1963'), BirthDate) "YearsSinceJFK"
        FROM Players WHERE ID < 11
        ORDER BY Birthdate;
ID      |BIRTHDATE |YearsSinceJFK
-----|-----|-----
7       |1981-07-02|17
6       |1982-01-05|18
8       |1983-04-13|19
10      |1983-11-06|19
9       |1983-12-24|20
4       |1987-01-21|23
1       |1987-03-27|23
2       |1988-04-20|24
3       |1990-10-30|26
5       |1991-01-15|27

10 rows selected
```

## See Also

- » [About Data Types](#)
- » [TIMESTAMP](#) data value
- » [HOUR](#) function
- » [MINUTE](#) function
- » [SECOND](#) function
- » [TIMESTAMP](#) function
- » [TIMESTAMPADD](#) function
- » [Working with Dates](#) in the *Developer's Guide*

# TO\_CHAR

The `TO_CHAR` function formats a date value into a string.

## Syntax

```
TO_CHAR( dateExpr, format );
```

### *dateExpr*

The date value that you want to format.

### *format*

A string that specifies the format you want applied to the `date`. You can specify formats such as the following:

*yyyy-mm-dd*

*mm/dd/yyyy*

*dd.mm.yy*

*dd-mm-yy*

## Results

This function returns a string (CHAR) value.

## Examples

```
splice> VALUES TO_CHAR(CURRENT_DATE, 'mm/dd/yyyy');
1
```

```
-----
09/22/2014
```

```
1 row selected
```

```
splice> VALUES TO_CHAR(CURRENT_DATE, 'dd-mm-yyyy');
1
```

```
-----
22-09-2014
```

```
1 row selected
```

```
splice> VALUES TO_CHAR(CURRENT_DATE, 'dd-mm-yy');
1
```

```
-----
22-09-14
```

## See Also

- » [CURRENT\\_DATE](#) function
- » [DATE](#) data type
- » [DATE](#) function
- » [DAY](#) function
- » [EXTRACT](#) function
- » [LASTDAY](#) function
- » [MONTH](#) function
- » [MONTH\\_BETWEEN](#) function
- » [MONTHNAME](#) function
- » [NEXTDAY](#) function
- » [NOW](#) function
- » [QUARTER](#) function
- » [TIME](#) data type
- » [TIMESTAMP](#) function
- » [TO\\_DATE](#) function
- » [WEEK](#) function
- » [\*Working with Dates\*](#) in the *Developer's Guide*

# TO\_DATE

The `TO_DATE` function formats a date string according to a formatting specification, and returns a [DATE](#) value. `DATE` values do not store time components.

## Syntax

```
TO_DATE( dateStrExpr, formatStr );
```

### *dateStrExpr*

A string expression that contains a date that is formatted according to the format string.

### *formatStr*

A string that specifies the format you want applied to the `dateStr`. See the [Date and Time Formats](#) section below for more information about format specification.

## Results

The result is always a [DATE](#) value.

## Date and Time Formats

Splice Machine supports date and time format specifications based on the Java [SimpleDateFormat](#) class.

Date and time value formats are used for both parsing input values and for formatting output values. For example, the format specification `yyyy-MM-dd HH:mm:ssZ` parses or formats values like `2014-03-02 11:47:44-0800`.

The remainder of this topic describes format specifications in these sections:

- » [Pattern Specifications](#) contains a table showing details for all of the pattern letters you can use.
- » [Presentation Types](#) describes how certain pattern letters are interpreted for parsing and/or formatting.
- » [Examples](#) contains a number of examples that will help you understand how to use formats.

## Pattern Specifications

You can specify formatting or parsing patterns for date-time values using the pattern letters shown in the following table. Note that pattern letters are typically repeated in a format specification. For example, `YYYY` or `YY`. Refer to the next section for specific information about multiple pattern letters in the different [presentation types](#).

Pattern Letter	Meaning	Presentation Type	Example
G	Era designator	<i>Text</i>	BC
y	Year	<i>Year</i>	2015 -or- 15
Y	Week year	<i>Year</i>	2011 -or- 11
M	Month in year	<i>Month</i>	July -or- Jul -or- 07
w	Week in year	<i>Number</i>	27
W	Week in month	<i>Number</i>	3
D	Day in year	<i>Number</i>	212  A common usage error is to mistakenly specify DD for the day field:  >> use dd to specify day of month >> use DD to specify the day of the year
d	Day in month	<i>Number</i>	13
F	Day of week in month	<i>Number</i>	2
E	Day name in week	<i>Text</i>	Tuesday -or- Tue
u	Day number of week (1=Monday, 7=Sunday)	<i>Number</i>	4
a	AM / PM marker	<i>Text</i>	PM
H	Hour in day (0 - 23)	<i>Number</i>	23
k	Hour in day (1 - 24)	<i>Number</i>	24
K	Hour in AM/PM (0 - 11)	<i>Number</i>	11



Pattern Letter	Meaning	Presentation Type	Example
h	Hour in AM/PM (1 - 12)	<i>Number</i>	12
m	Minute in hour	<i>Number</i>	33
s	Second in minute	<i>Number</i>	55
S	Millisecond	<i>Number</i>	959
z	Time zone	<i>General time zone</i>	Pacific Standard Time -or- PST -or- GMT-08:00
Z	Time zone	<i>RFC 822 time zone</i>	-0800
X	Time zone	<i>ISO 8601 time zone</i>	-08 -or- -0800 -or- -08:00
'	Escape char for text	<i>Delimiter</i>	
' '	Single quote	<i>Literal</i>	'

## Presentation Types

How a presentation type is interpreted for certain pattern letters depends on the number of repeated letters in the pattern. In some cases, as noted in the following table, other factors can influence how the pattern is interpreted.

Presentation Type	Description
<i>Text</i>	<p>For formatting, if the number of pattern letters is 4 or more, the full form is used. Otherwise, a short or abbreviated form is used, if available.</p> <p>For parsing, both forms are accepted, independent of the number of pattern letters.</p>
<i>Number</i>	<p>For formatting, the number of pattern letters is the minimum number of digits, and shorter numbers are zero-padded to this amount.</p> <p>For parsing, the number of pattern letters is ignored unless it's needed to separate two adjacent fields.</p>

Presentation Type	Description
<i>Year (for Gregorian calendar)</i>	<p>For formatting, if the number of pattern letters is 2, the year is truncated to 2 digits; otherwise it is interpreted as a number.</p> <p>For parsing, if the number of pattern letters is more than 2, the year is interpreted literally, regardless of the number of digits; e.g.:</p> <ul style="list-style-type: none"> <li>» if you use the pattern <code>MM/dd/yyyy</code>, the value <code>01/11/12</code> parses to January 11, 12 A.D.</li> <li>» if you use the pattern <code>MM/dd/yy</code>, the value <code>01/11/12</code> parses to January 11, 2012.</li> </ul> <p>If the number of pattern letters is one or two, (<code>y</code> or <code>yy</code>), the abbreviated year is interpreted as relative to a century; this is done by adjusting dates to be within 80 years before and 20 years after the current date.</p>
<i>Year (other calendar systems)</i>	<p>Calendar-system specific forms are applied.</p> <p>For both formatting and parsing, if the number of pattern letters is 4 or more, a calendar specific long form is used. Otherwise, a calendar specific short or abbreviated form is used.</p>
<i>Month</i>	<p>If the number of pattern letters is 3 or more, the month is interpreted as text; otherwise, it is interpreted as a number.</p>
<i>General time zone</i>	<p>Time zones are interpreted as text if they have names.</p> <p>For time zones representing a GMT offset value, the following syntax is used:</p> <pre>GMT Sign Hours : Minutes</pre> <p>where:</p> <ul style="list-style-type: none"> <li>Sign is + or -</li> <li>Hours is either Digit or Digit Digit, between 0 and 23.</li> <li>Minutes is Digit Digit and must be between 00 and 59.</li> </ul> <p>For parsing, RFC 822 time zones are also accepted.</p>

Presentation Type	Description
<i>RFC 822 time zone</i>	<p>For formatting, use the RFC 822 4-digit time zone format is used:</p> <p>Sign TwoDigitHours Minutes</p> <p>TwoDigitHours must be between 00 and 23.</p> <p>For parsing General time zones are also accepted.</p>
<i>ISO 8601 time zone</i>	<p>The number of pattern letters designates the format for both formatting and parsing as follows:</p> <p>Sign TwoDigitHours Z   Sign TwoDigitHours Minutes Z   Sign TwoDigitHours : Minutes Z</p> <p>For formatting:</p> <ul style="list-style-type: none"><li>» if the offset value from GMT is 0, Z value is produced</li><li>» if the number of pattern letters is 1, any fraction of an hour is ignored</li></ul> <p>For parsing, Z is parsed as the UTC time zone designator. Note that General time zones <i>are not accepted</i>.</p>
<i>Delimiter</i>	Use the single quote to escape characters in text strings.
<i>Literal</i>	<p>You can include literals in your format specification by enclosing the character(s) in single quotes.</p> <p>Note that you must escape single quotes to include them as literals, e.g. use ' 'T' ' to include the literal string 'T'.</p>

Formatting Examples

The following table contains a number of examples of date time formats:

Date and Time Pattern	Result
"yyyy.MM.dd G 'at' HH:mm:ss z"	2001.07.04 AD at 12:08:56 PDT
"EEE, MMM d, ''yy"	Wed, Jul 4, '01
"h:mm a"	12:08 PM

Date and Time Pattern	Result
"hh 'o''clock' a, zzzz"	12 o'clock PM, Pacific Daylight Time
"K:mm a, z"	0:08 PM, PDT
"yyyyy.MMMMM.dd GGG hh:mm aaa"	02001.July.04 AD 12:08 PM
"EEE, d MMM yyyy HH:mm:ss Z"	Wed, 4 Jul 2001 12:08:56 -0700
"yyMMddHHmmssZ"	010704120856-0700
"yyyy-MM-dd'T'HH:mm:ss.SSSZ"	2001-07-04T12:08:56.235-0700
"yyyy-MM-dd'T'HH:mm:ss.SSSXXX"	2001-07-04T12:08:56.235-07:00
"YYYY-'W'ww-u"	2001-W27-3

## Examples of Using TO\_DATE

Here are several simple examples:

```

splice> VALUES TO_DATE('2015-01-01', 'YYYY-MM-dd');
1
-----
2015-01-01
1 row selected

splice> VALUES TO_DATE('01-01-2015', 'MM-dd-YYYY');
1
-----
2015-01-01
1 row selected

splice> VALUES (TO_DATE('01-01-2015', 'MM-dd-YYYY') + 30);
1
-----
2015-01-31
1

splice> VALUES (TO_DATE('2015-126', 'MM-DDD'));
1
-----
2015-05-06
1 row selected

splice> VALUES (TO_DATE('2015-026', 'MM-DDD'));
1
-----
2015-01-26

splice> VALUES (TO_DATE('2015-26', 'MM-DD'));
1
-----
2015-01-26
1 row selected

```

And here is an example that shows two interesting aspects of using `TO_DATE`. In this example, the input includes the literal `T`), which means that the format pattern must delimit that letter with single quotes. Since we're delimiting the entire pattern in single quotes, we then have to escape those marks and specify `'T'` in our parsing pattern.

And because this example specifies a time zone (`Z`) in the parsing pattern but not in the input string, the timezone information is not preserved. In this case, that means that the parsed date is actually a day earlier than intended:

```

splice> VALUES TO_DATE('2013-06-18T01:03:30.000-0800','yyyy-MM-dd''T''HH:mm:ss.SSS
Z');
1
-----
2013-06-17

```

The solution is to explicitly include the timezone for your locale in the input string:

```
splice> VALUES TO_DATE('2013-06-18T01:03:30.000-08:00','yyyy-MM-dd''T''HH:mm:ss.SSS
Z');
1
-----
2013-06-18
```

## See Also

- » [CURRENT\\_DATE](#) function
- » [DATE](#) data type
- » [DATE](#) function
- » [DAY](#) function
- » [EXTRACT](#) function
- » [LASTDAY](#) function
- » [MONTH](#) function
- » [MONTH\\_BETWEEN](#) function
- » [MONTHNAME](#) function
- » [NEXTDAY](#) function
- » [NOW](#) function
- » [QUARTER](#) function
- » [TIME](#) data type
- » [TIMESTAMP](#) function
- » [TO\\_CHAR](#) function
- » [WEEK](#) function
- » [\*Working with Dates\*](#) in the *Developer's Guide*

# TRIM

The `TRIM` function that takes a character expression and returns that expression with leading and/or trailing pad characters removed. Optional parameters indicate whether leading, or trailing, or both leading and trailing pad characters should be removed, and specify the pad character that is to be removed.

## Syntax

```
TRIM( [ trimOperands ] trimSource)
```

### *trimOperands*

```
{ { trimType [trimCharacter] FROM
  | trimCharacter FROM
}
```

### *trimCharacter*

A character expression that specifies which character to trim from the source. If this is specified, it must evaluate to either `NULL` or to a character string whose length is exactly one. If left unspecified, it defaults to the space character (' ').

### *trimType*

```
{LEADING | TRAILING | BOTH}
```

If this value is not specified, the default value of `BOTH` is used.

### *trimSource*

The character expression to be trimmed

## Results

If either *trimCharacter* or *trimSource* evaluates to `NULL`, the result of the `TRIM` function is `NULL`. Otherwise, the result is defined as follows:

- » If *trimType* is `LEADING`, the result will be the *trimSource* value with all leading occurrences of *trimCharacter* removed.
- » If *trimType* is `TRAILING`, the result will be the *trimSource* value with all trailing occurrences of *trimCharacter* removed.
- » If *trimType* is `BOTH`, the result will be the *trimSource* value with all leading AND trailing occurrences of *trimCharacter* removed.

If *trimSource*'s data type is `CHAR` or `VARCHAR`, the return type of the `TRIM` function will be `VARCHAR`. Otherwise the return type of the `TRIM` function will be `CLOB`.

## Examples

```
splice> VALUES TRIM('      Space Case      ');
1
-----
Space Case      --- This is the string 'Space Case'

splice> VALUES TRIM(BOTH ' ' FROM '      Space Case      ');
1
-----
Space Case      --- This is the string 'Space Case'

splice> VALUES TRIM(TRAILING ' ' FROM '      Space Case      ');
1
-----
      Space Case      --- This is the string '      Space Case'

splice> VALUES TRIM(CAST NULL AS CHAR(1) FROM '      Space Case      ');
1
-----
NULL

splice> VALUES TRIM('o' FROM 'VooDoo');
1
-----
VooD

-- results in an error because trimCharacter can only be 1 character
splice> VALUES TRIM('Do' FROM 'VooDoo');
```

## See Also

- » [About Data Types](#)
- » [Concatenation](#) operator
- » [INITCAP](#) function
- » [INSTR](#) function
- » [LCASE](#) function
- » [LENGTH](#) function
- » [LOCATE](#) function
- » [LTRIM](#) function
- » [REGEX\\_LIKE](#) operator
- » [REPLACE](#) function



- » [RTRIM](#) function
- » [SUBSTR](#) function
- » [UCASE](#) function

# TRUNC or TRUNCATE

This topic describes the `TRUNCATE` built-in function, which you can use to truncate numeric, date, and timestamp values. You can use the abbreviation `TRUNC` interchangeably with the full name, `TRUNCATE`.

## Syntax

```
TRUNCATE ( number      [, numPlaces] |
           date        [, truncPoint] |
           timestamp   [, truncPoint] );
```

### *number*

An integer or decimal number to be truncated.

### *date*

A [DATE](#) value to be truncated.

### *timestamp*

A [TIMESTAMP](#) value to be truncated.

### *numPlaces*

An optional integer value that specifies the number of digits to truncate (made zero) when applying this function to a *number*.

- » If this value is positive, that many of the least significant digits (to the right of the decimal point) are truncated: `truncate(123.456, 2)` returns `123.450`.
- » If this value is negative, that many of the least significant digits to the left of the decimal point are truncated: `truncate(123.456, -1)` returns `120.000`.
- » If this value is zero, the decimal portion of the number is truncated: `truncate(123.456, 0)` returns `123.000`.
- » If this value is not specified, the decimal portion of the number is zero'ed, which means that `truncate(123.456)` returns `123.000`.

See the [Truncating Numbers](#) examples below.

### *truncPoint*

An optional string that specifies the point at which to truncate (zero) a date or timestamp value. This can be one of the following values:

*YEAR or YR*

The year value is retained; other values are set to their minimum values.

*MONTH or MON or MO*

The year and month values are retained; other values are set to their minimum values.

*DAY*

The year, month, and day values are retained; other values are set to their minimum values.

*HOUR or HR*

The year, month, day, and hour values are retained; other values are set to their minimum values.

*MINUTE or MIN*

The year, month, day, hour, and minute values are retained; other values are set to their minimum values.

*SECOND or SEC*

The year, month, day, hour, minute, and second values are retained; the milliseconds value is set to 0.

*MILLISECOND or MILLI*

All of the values, including year, month, day, hour, minute, second, and milliseconds are retained.

The default value, if nothing is specified, is *DAY*.

## Examples

### Truncating Numbers

```

splice> VALUES TRUNC(1234.456, 2);
1
-----
1234.450

splice> VALUES TRUNCATE(123.456,-1);
1
-----
120.000

splice> VALUES TRUNCATE(123.456,0);
1
-----
123.000

splice> VALUES TRUNCATE(123.456);
1
-----
123.000

splice> VALUES TRUNC(1234.456, 2);
1
-----
1234.450

splice> VALUES TRUNCATE(123.456,-1);
1
-----
120.000

splice> VALUES TRUNCATE(123.456,0);
1
-----
123.000
1 row selected

splice> VALUES TRUNCATE(123.456);
1
-----
123.000

VALUES TRUNCATE(1234.6789, 1);
-----
12345.6000

VALUES TRUNCATE(12345.6789, 2);
-----
12345.6700

VALUES TRUNCATE(12345.6789, -1);
-----

```

```

12340.0000

VALUES TRUNCATE(12345.6789, 0);
-----
12345.0000

VALUES TRUNCATE(12345.6789);
-----
12345.0000

```

## Truncating Dates

```

VALUES TRUNCATE (DATE ('1988-12-26'), 'year');
-----
1988-01-01

VALUES TRUNCATE (DATE ('1988-12-26'), 'month');
-----
1988-12-01

VALUES TRUNCATE (DATE ('1988-12-26'), 'day');
-----
1988-12-26

VALUES TRUNCATE (DATE ('1988-12-26'));
-----
1988-12-26

VALUES TRUNCATE (DATE ('2011-12-26'), 'MONTH');
-----
2011-12-01

```

## Truncating Timestamps

```
VALUES TRUNCATE(TIMESTAMP('2000-06-07 17:12:30.0'), 'year');
-----
2000-01-01 00:00:00.0

VALUES TRUNCATE(TIMESTAMP('2000-06-07 17:12:30.0'), 'month');
-----
2000-06-01 00:00:00.0

VALUES TRUNCATE(TIMESTAMP('2000-06-07 17:12:30.0'), 'day');
-----
2000-06-07 00:00:00.0

VALUES TRUNCATE(TIMESTAMP('2000-06-07 17:12:30.0'), 'hour');
-----
2000-06-07 17:00:00.0

VALUES TRUNCATE(TIMESTAMP('2000-06-07 17:12:30.0'), 'minute');
-----
2000-06-07 17:12:00.0

VALUES TRUNCATE(TIMESTAMP('2000-06-07 17:12:30.0'), 'second');
-----
2000-06-07 17:12:30.0

VALUES TRUNCATE(TIMESTAMP('2000-06-07 17:12:30.0'), 'MONTH');
-----
2011-12-01 00:00:00.0

VALUES TRUNCATE(TIMESTAMP('2000-06-07 17:12:30.0'));
-----
2011-12-26 00:00:00.0
```

## LN or UPPER

UCASE or UPPER returns a string in which all alphabetic characters in the input character expression have been converted to uppercase.

**NOTE:** UPPER and UCASE follow the database locale.

### Syntax

```
UCASE or UPPER ( CharacterExpression )
```

*CharacterExpression*

A [LONG VARCHAR](#) data type, or any built-in type that is implicitly converted to a string (but not a bit expression).

### Results

The data type of the result is as follows:

- » If the *CharacterExpression* evaluates to NULL, this function returns NULL.
- » If the *CharacterExpression* is of type [CHAR](#).
- » If the *CharacterExpression* is of type [LONG VARCHAR](#).
- » Otherwise, the return type is [VARCHAR](#).

The length and maximum length of the returned value are the same as the length and maximum length of the parameter.

### Example

To return the names of players, use the following clause:



```
splice> SELECT UCASE(DisplayName)
        FROM Players
        WHERE ID < 11;
1
-----
BUDDY PAINTER
BILL BOPPER
JOHN PURSER
BOB CRANKER
MITCH DUFFER
NORMAN AIKMAN
ALEX PARAMOUR
HARRY PENNELLO
GREG BROWN
JASON MINMAN

10 rows selected
```

## See Also

- » [About Data Types](#)
- » [Concatenation](#) operator
- » [INITCAP](#) function
- » [INSTR](#) function
- » [LCASE](#) function
- » [LENGTH](#) function
- » [LOCATE](#) function
- » [LTRIM](#) function
- » [REGEX\\_\\_LIKE](#) operator
- » [REPLACE](#) function
- » [RTRIM](#) function
- » [SUBSTR](#) function
- » [TRIM](#) function

# USER

When used outside stored routines, [CURRENT\\_USER](#), `USER`, and [SESSION\\_USER](#) all return the authorization identifier of the user who created the SQL session.

`SESSION_USER` also always returns this value when used within stored routines.

If used within a stored routine created with `EXTERNAL SECURITY DEFINER`, however, `CURRENT_USER` and `USER` return the authorization identifier of the user that owns the schema of the routine. This is usually the creating user, although the database owner could be the creator as well.

For information about definer's and invoker's rights, see [CREATE\\_FUNCTION statement](#).

## Syntax

```
USER
```

## Example

```
splice> VALUES USER;  
1  
-----  
SPLICE
```

## See Also

- » [CURRENT\\_USER](#) function
- » [SESSION\\_USER](#) function
- » [CREATE\\_FUNCTION](#) statement
- » [CREATE\\_PROCEDURE](#) statement

# VARCHAR

The `VARCHAR` function returns a varying-length character string representation of a character string.

## Character to varchar syntax

```
VARCHAR (CharacterStringExpression )
```

### *CharacterStringExpression*

An expression whose value must be of a character-string data type with a maximum length of 32,762 bytes.

## Datetime to varchar syntax

```
VARCHAR (DatetimeExpression )
```

### *DatetimeExpression*

An expression whose value must be of a date, time, or timestamp data type.

## Example

The `Position` column in our `Players` table is defined as `CHAR(2)`. The following query shows how to access position values as `VARCHARS`:

```
splice> SELECT VARCHAR(Position)
        FROM Players
        WHERE ID < 11;
1
----
C
1B
2B
SS
3B
LF
CF
RF
OF
RF

10 rows selected
```

## See Also

- » [About Data Types](#)
- » [VARCHAR](#) data type

# WEEK

The `WEEK` function returns an integer value representing the week of the year from a date expression.

## Syntax

```
WEEK ( dateExpr );
```

*dateExpr*

The date-time expression from which you wish to extract information.

## Results

The returned week number is in the range 1 to 53.

## Examples

```
splice> SELECT BirthDate, Week(BirthDate) "BirthWeek"
        FROM Players
        WHERE ID < 15;
BIRTHDATE | BIRTHWEEK
-----
1987-03-27 | 13
1988-04-20 | 16
1990-10-30 | 44
1987-01-21 | 4
1991-01-15 | 3
1982-01-05 | 1
1981-07-02 | 27
1983-04-13 | 15
1983-12-24 | 51
1983-11-06 | 44
1990-06-16 | 24
1977-08-30 | 35
1990-03-22 | 12
1982-10-12 | 41

14 rows selected
```

## See Also

» [CURRENT\\_DATE](#) function

- » [DATE](#) data type
- » [DATE](#) function
- » [DAY](#) function
- » [EXTRACT](#) function
- » [LASTDAY](#) function
- » [MONTH](#) function
- » [MONTH\\_BETWEEN](#) function
- » [MONTHNAME](#) function
- » [NEXTDAY](#) function
- » [NOW](#) function
- » [QUARTER](#) function
- » [TIME](#) data type
- » [TIMESTAMP](#) function
- » [TO\\_CHAR](#) function
- » [TO\\_DATE](#) function
- » [\*Working with Dates\*](#) in the *Developer's Guide*

# YEAR

The `YEAR` function returns the year part of a value. The argument must be a date, timestamp, or a valid character string representation of a date or timestamp. The result of the function is an integer between 1 and 9999.

## Syntax

```
YEAR ( expression )
```

## Usage

If the argument is `NULL`, the result is the `NULL` value.

## Examples

Get the current date:

```
splice> value(current_date);  
1  
-----  
2014-02-25
```

Now get the current year only:

```
splice> value(year(current_date));  
1  
-----  
2015
```

Now get the year value from 60 days ago:

```
splice> value(year(current_date-60));  
1  
-----  
2014
```

Select all players born in 1985 or 1989:

```
splice> SELECT DisplayName, Team, BirthDate
        FROM Players
        WHERE YEAR(BirthDate) IN (1985, 1989)
        ORDER BY BirthDate;
```

DISPLAYNAME	TEAM	BIRTHDATE
-----	-----	-----
Jeremy Johnson	Cards	1985-03-15
Gary Kosovo	Giants	1985-06-12
Michael Hillson	Cards	1985-11-07
Mitch Canepa	Cards	1985-11-26
Edward Erdman	Cards	1985-12-21
Jeremy Packman	Giants	1989-01-01
Nathan Nickels	Giants	1989-05-04
Ken Straiter	Cards	1989-07-20
Marcus Bamburger	Giants	1989-08-01
George Goomba	Cards	1989-08-08
Jack Hellman	Cards	1989-08-09
Elliot Andrews	Giants	1989-08-21
Henry Socomy	Giants	1989-11-17

13 rows selected

## See Also

- » [CURRENT\\_DATE](#) function
- » [DATE](#) function
- » [DAY](#) function
- » [LASTDAY](#) function
- » [MONTH](#) function
- » [MONTH\\_BETWEEN](#) function
- » [NEXTDAY](#) function
- » [TIMESTAMP](#) function
- » [Working with Dates](#) in the *Developer's Guide*



## Built-in System Procedures and Functions

This section contains the reference documentation for the Splice Machine Built-in SQL System Procedures and Functions, in the following subsections:

- » [Database Admin Procedures and Functions](#)
- » [Database Property Procedures and Functions](#)
- » [Importing Data Procedures and Functions](#)
- » [Jar File Procedures and Functions](#)
- » [Logging Procedures and Functions](#)
- » [Statements and Stored Procedures System Procedures](#)
- » [Statistics Procedures and Functions](#)
- » [System Status Procedures and Functions](#)
- » [Transaction Procedures and Functions](#)

### Database Admin Procedures and Functions

These are the system procedures and functions for administering your database:

Procedure / Function Name	Description
<a href="#">SYSCS_UTIL.SYSCS BACKUP DATABASE</a>	<p>Backs up the database to a specified backup directory.</p> <p>This procedure is only available in our <i>On-Premise Database</i> product.</p>
<a href="#">SYSCS_UTIL.SYSCS CANCEL BACKUP</a>	<p>Cancels a backup.</p> <p>This procedure is only available in our <i>On-Premise Database</i> product.</p>
<a href="#">SYSCS_UTIL.SYSCS CANCEL DAILY BACKUP</a>	<p>Cancels a scheduled daily backup.</p> <p>This procedure is only available in our <i>On-Premise Database</i> product.</p>
<a href="#">SYSCS_UTIL.COMPACT_REGION</a>	<p>Performs a minor compaction on a table or index region.</p>

Procedure / Function Name	Description
<a href="#"><code>SYSCS_UTIL.SYSCS_CREATE_USER</code></a>	Adds a new user account to a database.
<a href="#"><code>SYSCS_UTIL.SYSCS_DELETE_BACKUP</code></a>	<p>Delete a specific backup.</p> <p>This procedure is only available in our <i>On-Premise Database</i> product.</p>
<a href="#"><code>SYSCS_UTIL.SYSCS_DELETE_OLD_BACKUPS</code></a>	<p>Deletes all backups that were created more than a certain number of days ago.</p> <p>This procedure is only available in our <i>On-Premise Database</i> product.</p>
<a href="#"><code>SYSCS_UTIL.SYSCS_DROP_USER</code></a>	Removes a user account from a database.
<a href="#"><code>SYSCS_UTIL.GET_ENCODED_REGION_NAME</code></a>	Returns the encoded name of the HBase region that contains the specified, unencoded Splice Machine table primary key or index values.
<a href="#"><code>SYSCS_UTIL.GET_REGIONS</code></a>	Retrieves the list of regions containing a range of key values.
<a href="#"><code>SYSCS_UTIL.GET_RUNNING_OPERATIONS</code></a>	Displays information about each Splice Machine operations running on a server.
<a href="#"><code>SYSCS_UTIL.GET_START_KEY</code></a>	Retrieves the unencoded start key for a specified HBase table or index region.
<a href="#"><code>SYSCS_UTIL.KILL_OPERATION</code></a>	Terminates a Splice Machine operation running on the server to which you are connected.
<a href="#"><code>SYSCS_UTIL.MAJOR_COMPACT_REGION</code></a>	Performs a major compaction on a table or index region.
<a href="#"><code>SYSCS_UTIL.MERGE_REGIONS</code></a>	Merges two adjacent table or index regions.
<a href="#"><code>SYSCS_UTIL.SYSCS_MODIFY_PASSWORD</code></a>	Called by a user to change that user's own password.
<a href="#"><code>SYSCS_UTIL.SYSCS_PERFORM_MAJOR_COMPACTION_ON_SCHEMA</code></a>	Performs a major compaction on a schema
<a href="#"><code>SYSCS_UTIL.SYSCS_PERFORM_MAJOR_COMPACTION_ON_TABLE</code></a>	Performs a major compaction on a table.

Procedure / Function Name	Description
<a href="#"><code>SYSCS_UTIL.SYSCS_REFRESH_EXTERNAL_TABLE</code></a>	Refreshes the schema of an external table in Splice Machine; use this when the schema of the table's source file has been modified outside of Splice Machine.
<a href="#"><code>SYSCS_UTIL.SYSCS_RESET_PASSWORD</code></a>	Resets a password that has expired or has been forgotten.
<a href="#"><code>SYSCS_UTIL.SYSCS_RESTORE_DATABASE</code></a>	Restores a database from a previous backup.  This procedure is only available in our <i>On-Premise Database</i> product.
<a href="#"><code>SYSCS_UTIL.SYSCS_SCHEDULE_DAILY_BACKUP</code></a>	Schedules a full or incremental database backup to run at a specified time daily.  This procedure is only available in our <i>On-Premise Database</i> product.
<a href="#"><code>SYSCS_UTIL.SYSCS_UPDATE_SCHEMA_OWNER</code></a>	Changes the owner of a schema.
<a href="#"><code>SYSCS_UTIL.VACUUM</code></a>	Performs clean-up operations on the system.

## Database Properties Procedures and Functions

These are the system procedures and functions for working with your database properties:

Procedure / Function Name	Description
<a href="#"><code>SYSCS_UTIL.SYSCS_GET_ALL_PROPERTIES</code></a>	Displays all of the Splice Machine Derby properties.
<a href="#"><code>SYSCS_UTIL.SYSCS_GET_GLOBAL_DATABASE_PROPERTY</code> <code>function</code></a>	Fetches the value of the specified property of the database.
<a href="#"><code>SYSCS_UTIL.SYSCS_GET_SCHEMA_INFO</code></a>	Displays table information for all user schemas, including the HBase regions occupied and their store file size.

Procedure / Function Name	Description
<a href="#">SYSCS_UTIL.SYSCS_PEEK_AT_SEQUENCE function</a>	Allows users to observe the instantaneous current value of a sequence generator without having to query the <a href="#">SYSSEQUENCES system table</a> .
<a href="#">SYSCS_UTIL.SYSCS_SET_GLOBAL_DATABASE_PROPERTY</a>	Sets or deletes the value of a property of the database.

## Importing Data Procedures and Functions

These are the system procedures and functions for importing data into your database:

Procedure / Function Name	Description
<a href="#">SYSCS_UTIL.BULK_IMPORT_HFILE</a>	Imports data from an HFile.
<a href="#">SYSCS_UTIL.COMPUTE_SPLIT_KEY</a>	Computes split keys for a table or index.
<a href="#">SYSCS_UTIL.DELETE_SNAPSHOT</a>	Deletes a stored snapshot.
<a href="#">SYSCS_UTIL.IMPORT_DATA</a>	Imports data to a subset of columns in a table.
<a href="#">SYSCS_UTIL.SYSCS_MERGE_DATA_FROM_FILE</a>	Imports data from external files, inserting new records and updating existing records.
<a href="#">SYSCS_UTIL.SET_PURGE_DELETED_ROWS</a>	Enables (or disables) physical deletion of logically deleted rows from a specific table.
<a href="#">SYSCS_UTIL.RESTORE_SNAPSHOT</a>	Restores a table or schema from a stored snapshot.
<a href="#">SYSCS_UTIL.SNAPSHOT_SCHEMA</a>	Creates a Splice Machine snapshot of a schema.
<a href="#">SYSCS_UTIL.SNAPSHOT_TABLE</a>	Creates a Splice Machine snapshot of a specific table.
<a href="#">SYSCS_UTIL.SPLIT_TABLE_OR_INDEX_AT_POINTS</a>	Sets up a table or index in your database with split keys computed by the <code>COMPUTE_SPLIT_KEY</code> procedure.
<a href="#">SYSCS_UTIL.SPLIT_TABLE_OR_INDEX</a>	<p>Computes split keys for a table or index and then sets up the table or index.</p> <p>This combines the functionality of <code>SYSCS_UTIL.COMPUTE_SPLIT_KEY</code> and <code>SYSCS_UTIL.SPLIT_TABLE_OR_INDEX_AT_POINTS</code>.</p>

Procedure / Function Name	Description
<a href="#">SYSCS_UTIL.SYSCS UPSERT DATA FROM FILE</a>	Imports data from external files, inserting new records and updating existing records.

## Jar File Procedures and Functions

These are the system procedures and functions for working with JAR files:

Procedure / Function Name	Description
<a href="#">SQLJ.INSTALL_JAR</a>	Stores a jar file in a database.
<a href="#">SQLJ.REMOVE_JAR</a>	Removes a jar file from a database.
<a href="#">SQLJ.REPLACE_JAR</a>	Replaces a jar file in a database.

## Logging Procedures and Functions

These are the system procedures and functions for working with system logs:

Procedure / Function Name	Description
<a href="#">SYSCS_UTIL.SYSCS GET_LOGGER_LEVEL</a>	Displays the log level of the specified logger.
<a href="#">SYSCS_UTIL.SYSCS GET_LOGGERS</a>	Displays the names of all Splice Machine loggers in the system.
<a href="#">SYSCS_UTIL.SYSCS SET_LOGGER_LEVEL</a>	Changes the log level of the specified logger.

## Statement and Stored Procedures System Procedures

These are the system procedures and functions for working with executing statements and stored procedures:

Procedure / Function Name	Description
<a href="#">SYSCS_UTIL.SYSCS EMPTY_GLOBAL_STATEMENT_CACHE</a>	Removes as many compiled statements (plans) as possible from the database-wide statement cache (across all region servers).

Procedure / Function Name	Description
<a href="#"><code>SYSCS_UTIL.SYSCS_EMPTY_STATEMENT_CACHE</code></a>	Removes as many compiled statements (plans) as possible from the database statement cache on your current region server.
<a href="#"><code>SYSCS_UTIL.SYSCS_INVALIDATE_STORED_STATEMENTS</code></a>	Invalidates all system prepared statements and forces the query optimizer to create new execution plans.
<a href="#"><code>SYSCS_UTIL.SYSCS_UPDATE_METADATA_STORED_STATEMENTS</code></a>	Updates the execution plan for stored procedures in your database.
<a href="#"><code>SYSCS_UTIL.SYSCS_UPDATE_ALL_SYSTEM_PROCEDURES</code></a>	Updates the signatures of all of the system procedures in a database.
<a href="#"><code>SYSCS_UTIL.SYSCS_UPDATE_SYSTEM_PROCEDURE</code></a>	Updates the stored declaration of a specific system procedure in the data dictionary.

## Statistics Procedures and Functions

These are the system procedures and functions for managing database statistics:

Procedure / Function Name	Description
<a href="#"><code>SYSCS_UTIL.COLLECT_SCHEMA_STATISTICS</code></a>	Collects statistics on a specific schema in your database.
<a href="#"><code>SYSCS_UTIL.DISABLE_COLUMN_STATISTICS</code></a>	Disables collection of statistics on a specific column in a table.
<a href="#"><code>SYSCS_UTIL.DROP_SCHEMA_STATISTICS</code></a>	Drops statistics for a specific schema in your database.
<a href="#"><code>SYSCS_UTIL.ENABLE_COLUMN_STATISTICS</code></a>	Enables collection of statistics on a specific column in a table.

## System Status Procedures and Functions

These are the system procedures and functions for monitoring and adjusting system status:

Procedure / Function Name	Description
<a href="#"><code>SYSCS_UTIL.SYSCS_GET_ACTIVE_SERVERS</code></a>	Displays the number of active servers in the Splice cluster.

Procedure / Function Name	Description
<a href="#">SYSCS_UTIL.SYSCS_GET_REGION_SERVER_STATS_INFO</a>	Displays input and output statistics about the cluster.
<a href="#">SYSCS_UTIL.SYSCS_GET_REQUESTS</a>	Displays information about the number of RPC requests that are coming into Splice Machine.
<a href="#">SYSCS_UTIL.SYSCS_GET_RUNNING_OPERATIONS</a>	Displays information about all Splice Machine operations running on the server to which you are connected.
<a href="#">SYSCS_UTIL.SYSCS_GET_SESSION_INFO</a>	Displays session information, including the hostname and session IDs.
<a href="#">SYSCS_UTIL.SYSCS_GET_VERSION_INFO</a>	Displays the version of Splice Machine installed on each server in your cluster.
<a href="#">SYSCS_UTIL.SYSCS_GET_WRITE_INTAKE_INFO</a>	Displays information about the number of writes coming into Splice Machine.
<a href="#">SYSCS_UTIL.KILL_OPERATION</a>	Terminates a Splice Machine operation running on the server to which you are connected.

## Transaction Procedures and Functions

These are the system procedures and functions for working with transactions in your database

Procedure / Function Name	Description
<a href="#">SYSCS_UTIL.SYSCS_GET_CURRENT_TRANSACTION</a>	Displays summary information about the current transaction.



For access to the source code for the Community Edition of Splice Machine, visit [our open source GitHub repository](#).

## SYSCS\_UTIL.SYSCS\_BACKUP\_DATABASE

The `SYSCS_UTIL.SYSCS_BACKUP_DATABASE` system procedure performs an immediate full or incremental backup of your database to a specified backup directory.

Splice Machine supports both full and incremental backups:

- » A *full backup* backs up all of the files/blocks that constitute your database.
- » An *incremental backup* only stores database files/blocks that have changed since a previous backup.

**NOTE:** The first time that you run an incremental backup, a full backup is performed. Subsequent runs of the backup will only copy information that has changed since the previous backup.

For more information, see the [Backing Up and Restoring](#) topic.

## Syntax

```
SYSCS_UTIL.SYSCS_BACKUP_DATABASE ( VARCHAR backupDir,
                                     VARCHAR(30) backupType );
```

### *backupDir*

Specifies the path to the directory in which you want the backup stored. This can be a local directory if you're using the standalone version of Splice Machine, or a directory in your cluster's file system (HDFS or MapR-FS).

**NOTE:** You must have permissions set properly to use cloud storage as a backup destination. See [Backing Up to Cloud Storage](#) for information about setting backup permissions properties.

Relative paths are resolved based on the current user directory. To avoid confusion, we strongly recommend that you use an absolute path when specifying the backup destination.

### *backupType*

Specifies the type of backup that you want performed. This must be one of the following values: `full` or `incremental`; any other value produces an error and the backup is not run.

Note that if you specify `'incremental'`, Splice Machine checks the [SYS.SYSBACKUP](#) table to determine if there already is a backup for the system; if not, Splice Machine will perform a full backup, and subsequent backups will be incremental.

## Results

This procedure does not return a result.



## Backup Resource Allocation

Splice Machine backup jobs use a Map Reduce job to copy HFiles; this process may hang up if the resources required for the Map Reduce job are not available from Yarn. See the [Backup Resource Allocation](#) section of our *Troubleshooting Guide* for specific information about allocation of resources.

## Usage Notes

There's a subtle issue with performing a backup when you're using a temporary table in your session: although the temporary table is (correctly) not backed up, the temporary table's entry in the system tables will be backed up. When the backup is restored, the table entries will be restored, but the temporary table will be missing.

There's a simple workaround:

1. Exit your current session, which will automatically delete the temporary table and its system table entries.
2. Start a new session (reconnect to your database).
3. Start your backup job.

## Execute Privileges

If authentication and SQL authorization are both enabled, only the database owner has execute privileges on this function by default. The database owner can grant access to other users.

## JDBC example

The following example performs an immediate full backup to a subdirectory of the `hdfs:///home/backup` directory:

```
CallableStatement cs = conn.prepareCall
("CALL SYSCS_UTIL.SYSCS_BACKUP_DATABASE(?,?)");
cs.setString(1, 'hdfs:///home/backup');
cs.setString(2, 'full');
cs.execute();
cs.close();
```

## SQL Example

Backing up a database may take several minutes, depending on the size of your database and how much of it you're backing up.

The following example runs an immediate incremental backup to the `hdfs:///home/backup/` directory:

```
splice> CALL SYSCS_UTIL.SYSCS_BACKUP_DATABASE( 'hdfs:///home/backup', 'incremental'
);
Statement executed.
```

The following example runs the same backup and stores it on AWS:

```
splice> CALL SYSCS_UTIL.SYSCS_BACKUP_DATABASE( 's3://backup1234', 'incremental' );
Statement executed.
```

And this example does a full backup to a relative directory (relative to your `splicemachine` directory) on a standalone version of Splice Machine:

```
splice> CALL SYSCS_UTIL.SYSCS_BACKUP_DATABASE( './dbBackups', 'full' );
Statement executed.
```

## See Also

- » [\*Backing Up and Restoring Databases\*](#)
- » [SYSCS\\_UTIL.SYSCS\\_CANCEL\\_DAILY\\_BACKUP](#)
- » [SYSCS\\_UTIL.SYSCS\\_DELETE\\_BACKUP](#)
- » [SYSCS\\_UTIL.SYSCS\\_DELETE\\_OLD\\_BACKUPS](#)
- » [SYSCS\\_UTIL.SYSCS\\_RESTORE\\_DATABASE](#)
- » [SYSCS\\_UTIL.SYSCS\\_SCHEDULE\\_DAILY\\_BACKUP](#)
- » [SYSBACKUP](#)
- » [SYSBACKUPITEMS](#)
- » [SYSBACKUPJOBS](#)

## SYSCS\_UTIL.BULK\_IMPORT\_HFILE

The `SYSCS_UTIL.BULK_IMPORT_HFILE` system procedure imports data into your Splice Machine database by first generating HFiles and then importing those HFiles.

Our HFile data import procedure leverages HBase bulk loading, which allows it to import your data at a faster rate; however, using this procedure instead of our standard [SYSCS\\_UTIL.IMPORT\\_DATA](#) procedure means that **constraint checks are not performed during data importation**.

### Selecting an Import Procedure

Splice Machine provides four system procedures for importing data:

- » The [SYSCS\\_UTIL.IMPORT\\_DATA](#) procedure imports each input record into a new record in your database.
- » The [SYSCS\\_UTIL.UPSERT\\_DATA\\_FROM\\_FILE](#) procedure updates existing records and adds new records to your database. It only differs from `SYSCS_UTIL.MERGE_DATA_FROM_FILE` in that upserting **overwrites** the generated or default value of a column that *is not specified* in your `insertColumnList` parameter when updating a record.
- » The [SYSCS\\_UTIL.MERGE\\_DATA\\_FROM\\_FILE](#) procedure updates existing records and adds new records to your database. It only differs from `SYSCS_UTIL.UPSERT_DATA_FROM_FILE` in that merging **does not overwrite** the generated or default value of a column that *is not specified* in your `insertColumnList` parameter when updating a record.
- » This procedure, [SYSCS\\_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 has improved performance for large tables; however, the bulk HFile import requires extra work on your part and lacks constraint checking.

Our [Importing Data Tutorial](#) includes a decision tree and brief discussion to help you determine which procedure best meets your needs.

## Syntax

```
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
);
```

**NOTE:** If you have specified `skipSampling=true` to indicate that you're computing the splits yourself, as described [below](#), the parameter values that you pass to the [SYSCS\\_UTIL.COMPUTE\\_SPLIT\\_KEY](#) or [SYSCS\\_UTIL.SYSCS\\_SPLIT\\_TABLE\\_OR\\_INDEX](#) procedures should match the values that you pass to this procedure.

## Parameters

The following table summarizes the parameters used by `SYSCS_UTIL.BULK_IMPORT_HFILE` and other Splice Machine data importation procedures. Each parameter name links to a more detailed description in our [Importing Data Tutorial](#).

Category	Parameter	Description	Example Value
Table Info	<a href="#">schemaName</a>	The name of the schema of the table in which to import.	SPLICE
	<a href="#">tableName</a>	The name of the table in which to import	playerTeams
Data Location	<a href="#">insertColumnList</a>	The names, in single quotes, of the columns to import. If this is <code>null</code> , all columns are imported.	'ID, TEAM'

Category	Parameter	Description	Example Value
	<u>fileOrDirectoryName</u>	<p>Either a single file or a directory. If this is a single file, that file is imported; if this is a directory, all of the files in that directory are imported. You can import compressed or uncompressed files.</p> <div> <p>The SYSCS_UTIL.MERGE_DATA_FROM_FILE procedure only works with single files; <b>you cannot specify a directory name</b> when calling SYSCS_UTIL.MERGE_DATA_FROM_FILE.</p> </div> <p>On a cluster, the files to be imported <b>MUST</b> be on S3, HDFS (or MapR-FS). If you're using our Database Service product, files can only be imported from S3.</p>	<p>/data/mydata/mytable.csv</p> <p>'s3a://splice-benchmark-data/flat/TPCH/100/region'</p>
Data Formats	<u>oneLineRecords</u>	A Boolean value that specifies whether ( <code>true</code> ) each record in the import file is contained in one input line, or ( <code>false</code> ) if a record can span multiple lines.	<code>true</code>
	<u>charset</u>	The character encoding of the import file. The default value is UTF-8.	<code>null</code>
	<u>columnDelimiter</u>	The character used to separate columns, Specify <code>null</code> if using the comma (,) character as your delimiter.	<code>' '</code>
	<u>characterDelimiter</u>	The character is used to delimit strings in the imported data.	<code>'\"'</code>
	<u>timestampFormat</u>	<p>The format of timestamps stored in the file. You can set this to <code>null</code> if there are no time columns in the file, or if the format of any timestamps in the file match the <code>Java.sql.Timestamp</code> default format, which is: "<code>yyyy-MM-dd HH:mm:ss</code>".</p> <div>  <p>All of the timestamps in the file you are importing must use the same format.</p> </div>	<code>'yyyy-MM-dd HH:mm:ss.SSZ'</code>

Category	Parameter	Description	Example Value
	<u>dateFormat</u>	The format of timestamps stored in the file. You can set this to <code>null</code> if there are no date columns in the file, or if the format of any dates in the file match pattern: " <code>yyyy-MM-dd</code> ".	<code>yyyy-MM-dd</code>
	<u>timeFormat</u>	The format of time values stored in the file. You can set this to <code>null</code> if there are no time columns in the file, or if the format of any times in the file match pattern: " <code>HH:mm:ss</code> ".	<code>HH:mm:ss</code>
<b>Problem Logging</b>	<u>badRecordsAllowed</u>	The number of rejected (bad) records that are tolerated before the import fails. If this count of rejected records is reached, the import fails, and any successful record imports are rolled back. Specify 0 to indicate that no bad records are tolerated, and specify -1 to indicate that all bad records should be logged and allowed.	25
	<u>badRecordDirectory</u>	<p>The directory in which bad record information is logged. Splice Machine logs information to the <code>&lt;import_file_name&gt;.bad</code> file in this directory; for example, bad records in an input file named <code>foo.csv</code> would be logged to a file named <code>badRecordDirectory/foo.csv.bad</code>.</p> <p>On a cluster, this directory <b>MUST be on S3, HDFS (or MapR-FS)</b>. If you're using our Database Service product, files can only be imported from S3.</p>	<code>'importErrsDir'</code>
<b>Bulk HFile Import</b>	<u>bulkImportDirectory</u> ( <u>outputDirectory</u> )	<p>For <code>SYSCS_UTIL.BULK_IMPORT_HFILE</code>, this is the name of the directory into which the generated HFiles are written prior to being imported into your database.</p> <p>For the <code>SYSCS_UTIL.COMPUTE_SPLIT_KEY</code> procedure, where it is named <code>outputDirectory</code>, this parameter specifies the directory into which the split keys are written.</p>	<code>hdfs:///tmp/test_hfile_import/</code>
	<u>skipSampling</u>	<p>The <code>skipSampling</code> parameter is a Boolean value that specifies how you want the split keys used for the bulk HFile import to be computed. Set to <code>false</code> to have <code>SYSCS_UTIL.BULK_IMPORT_HFILE</code> automatically determine splits for you.</p> <p>This parameter is only used with the <code>SYSCS_UTIL.BULK_IMPORT_HFILE</code> system procedure.</p>	<code>false</code>

## Usage

The [SYSCS\\_UTIL.BULK\\_IMPORT\\_HFILE](#) procedure needs the data that you're importing split into multiple HFiles before it actually imports the data into your database. You can achieve these splits in three ways:

- » You can call `SYSCS_UTIL.BULK_IMPORT_HFILE` with the `skipSampling` parameter to `false`. `SYSCS_UTIL.BULK_IMPORT_HFILE` samples the data to determine the splits, then splits the data into multiple HFiles, and then imports the data.
- » You can split the data into HFiles with the [SYSCS\\_UTIL.SYSCS\\_SPLIT\\_TABLE\\_OR\\_INDEX](#) system procedure, which both computes the keys and performs the splits. You then call `SYSCS_UTIL.BULK_IMPORT_HFILE` with the `skipSampling` parameter to `true` to import your data.
- » You can split the data into HFiles by first calling the [SYSCS\\_UTIL.COMPUTE\\_SPLIT\\_KEY](#) procedure and then calling the [SYSCS\\_UTIL.SYSCS\\_SPLIT\\_TABLE\\_OR\\_INDEX\\_AT\\_POINTS](#) procedure to split the table or index. You then call `SYSCS_UTIL.BULK_IMPORT_HFILE` with the `skipSampling` parameter to `true` to import your data.

In all cases, `SYSCS_UTIL.BULK_IMPORT_HFILE` automatically deletes the HFiles after the import process has completed.

The [Bulk HFile Import Examples](#) section of our *Importing Data Tutorial* describes how these methods differ and provides examples of using them to import data.

## Results

`SYSCS_UTIL.BULK_IMPORT_HFILE` displays a summary of the import process results that looks like this:

rowsImported	failedRows	files	dataSize	failedLog
94	0	1	4720	NONE

## Examples

The [Importing Data: Bulk HFile Examples](#) topic walks you through several examples of importing data with bulk HFiles.

## See Also

- » [SYSCS\\_UTIL.COMPUTE\\_SPLIT\\_KEY](#)
- » [SYSCS\\_UTIL.SYSCS\\_SPLIT\\_TABLE\\_OR\\_INDEX](#)
- » [SYSCS\\_UTIL.SYSCS\\_SPLIT\\_TABLE\\_OR\\_INDEX\\_AT\\_POINTS](#)

# SYSCS\_UTIL.SYSCS\_CANCEL\_BACKUP

The SYSCS\_UTIL.SYSCS\_CANCEL\_BACKUP system procedure cancels an in-progress backup.

## Syntax

```
SYSCS_UTIL.SYSCS_CANCEL_BACKUP ( ) ;
```

## Results

This procedure does not return a result.

## Execute Privileges

If authentication and SQL authorization are both enabled, only the database owner has execute privileges on this function by default. The database owner can grant access to other users.

## Example

This cancels the currently running backup:

```
CALL SYSCS_UTIL.SYSCS_CANCEL_BACKUP ( ) ;
```

## See Also

- » [\*Backing Up and Restoring Databases\*](#)
- » [SYSCS\\_UTIL.SYSCS\\_BACKUP\\_DATABASE](#)
- » [SYSCS\\_UTIL.SYSCS\\_DELETE\\_BACKUP](#)
- » [SYSCS\\_UTIL.SYSCS\\_DELETE\\_OLD\\_BACKUPS](#)
- » [SYSCS\\_UTIL.SYSCS\\_RESTORE\\_DATABASE](#)
- » [SYSCS\\_UTIL.SYSCS\\_SCHEDULE\\_DAILY\\_BACKUP](#)



## SYSCS\_UTIL.SYSCS\_CANCEL\_DAILY\_BACKUP

The `SYSCS_UTIL.SYSCS_CANCEL_DAILY_BACKUP` system procedure cancels a scheduled backup job.

**NOTE:** Once you cancel a daily backup, it will no longer be scheduled to run.

### Syntax

```
SYSCS_UTIL.SYSCS_CANCEL_DAILY_BACKUP( BIGINT jobId );
```

*jobId*

A `BIGINT` value that specifies which scheduled backup job you want to cancel.

To find the *jobId* you want to cancel, see the [Backing Up and Restoring](#) topic.

### Results

This procedure does not return a result.

### Execute Privileges

If authentication and SQL authorization are both enabled, only the database owner has execute privileges on this function by default. The database owner can grant access to other users.

### Example

If necessary, you can first query the [SYSBACKUPJOBS](#) system table to find the `jobId` of the job you want to cancel.

And then cancel that job; for example:

```
splice> SELECT * FROM SYS.SYSBACKUPJOBS;
JOB_ID      | FILESYSTEM                                | TYPE      | HOUR_OF_DAY | BEGI
N_TIMESTAMP
-----
-----
41069      | /~/Documents/splicemachine/dbBackups/    | full      | 18          | 09:4
1:05.455

1 row selected

splice> CALL SYSCS_UTIL.SYSCS_CANCEL_DAILY_BACKUP(41069);
Statement executed.
```

## See Also

- » [\*Backing Up and Restoring Databases\*](#)
- » [SYSCS\\_UTIL.SYSCS\\_BACKUP\\_DATABASE](#)
- » [SYSCS\\_UTIL.SYSCS\\_DELETE\\_BACKUP](#)
- » [SYSCS\\_UTIL.SYSCS\\_DELETE\\_OLD\\_BACKUPS](#)
- » [SYSCS\\_UTIL.SYSCS\\_RESTORE\\_DATABASE](#)
- » [SYSCS\\_UTIL.SYSCS\\_SCHEDULE\\_DAILY\\_BACKUP](#)
- » [SYSBACKUP](#)
- » [SYSBACKUPITEMS](#)
- » [SYSBACKUPJOBS](#)

# SYSCS\_UTIL.COLLECT\_SCHEMA\_STATISTICS

The `SYSCS_UTIL.COLLECT_SCHEMA_STATISTICS` system procedure collects statistics on a specific schema in your database.

**NOTE:** Once statistics have been collected for a schema, they are automatically used by the query optimizer.

This procedure collects statistics for every table in the schema. It also collects statistics for the index associated with every table in the schema. For example, if you have :

- » a schema named `mySchema`
- » `mySchema` contains two tables: `myTable1` and `myTable2`
- » `myTable1` has two indices: `myTable1Index1` and `myTable1Index2`

Then `SYSCS_UTIL.COLLECT_SCHEMA_STATISTICS` will collect statistics for `myTable1`, `myTable2`, `myTable1Index1`, and `myTable1Index2`.

## Syntax

```
SYSCS_UTIL.COLLECT_SCHEMA_STATISTICS ( VARCHAR(128) schema,
                                       BOOLEAN staleOnly)
```

### *schemaName*

Specifies the schema for which you want to collect statistics. Passing a `null` or non-existent schema name generates an error.

### *staleOnly*

A `BOOLEAN` value that specifies:

- » If this is `true`, data is only re-collected for partitions that are known to have out of date statistics.
- » If this is `false`, data is collected on all partitions. **Note** that this can significantly increase the time required to collect statistics, and is typically used only when you are not sure about the current quality of statistics in the entire schema.



The `staleOnly` parameter value is currently ignored, but must be specified in your call to this procedure. Its value is always set to `false` in the system code.

## Results

This procedure returns a results table that contains:

- » one row for each table
- » one row per index for every table and its associated index in the schema

Each row contains the following columns:

Column Name	Type	Contents
schemaName	VARCHAR	The name of the schema.
tableName	VARCHAR	The name of the table.
partition	VARCHAR	The name of the region on which statistics were collected.
rowsCollected	INTEGER	The number of rows of statistics that were collected..
partitionSize	BIGINT	The size of the partition in bytes.

## Usage Notes

Collecting statistics on a schema can take some time.

## SQL Examples

```
splice> CALL SYSCS_UTIL.COLLECT_SCHEMA_STATISTICS( 'SPICE', false );
schemaName |tableName |partition |rowsCollec&|partiti
onSize
-----
-----
SPICE      |PLAYERS   |splice:1440,,1467393447889.cbc33f4635ade|76 |351
5
SPICE      |SALARIES  |splice:1456,,1467393749257.7724e0cb12af3|76 |1420
SPICE      |BATTING   |splice:1472,,1467393754889.b34f5da64c36e|44 |2257
1
SPICE      |PITCHING  |splice:1488,,1467393760434.35ee9880e5090|32 |21212
SPICE      |FIELDING  |splice:1504,,1467393775949.674b34acdb182|44 |9876

5 rows selected
```

## See Also

- » [Data Assignments and Comparisons](#)
- » [SYSCS\\_UTIL.ENABLE\\_COLUMN\\_STATISTICS](#)

- » [SYSCS\\_UTIL.DISABLE\\_COLUMN\\_STATISTICS](#)
- » [SYSCS\\_UTIL.DROP\\_SCHEMA\\_STATISTICS](#)
- » [Using Statistics](#) in the *Developer's Guide*

# SYSCS\_UTIL.COMPACT\_REGION

The `SYSCS_UTIL.COMPACT_REGION` system procedure performs a minor compaction on a table region or an index region.

Region names must be specified in HBase-encoded format. You can retrieve the encoded name for a region by calling the [SYSCS\\_UTIL.GET\\_ENCODED\\_REGION\\_NAME](#) system procedure.

A common reason for calling this procedure is to improve compaction performance by only compacting recent updates in a table. For example, you might confine any updates to regions of the current month, so older regions need not be re-compacted.

## Syntax

```
SYSCS_UTIL.COMPACT_REGION ( VARCHAR schemaName,
                           VARCHAR tableName,
                           VARCHAR indexName,
                           VARCHAR startKey)
```

### *schemaName*

The name of the schema of the table.

### *tableName*

The name of the table to compact.

### *indexName*

`NULL` or the name of the index.

Specify the name of the index you want to compact; if you are compacting the table, specify `NULL` for this parameter.

### *regionName*

The **encoded** HBase name of the region you want compacted. You can call the [SYSCS\\_UTIL.GET\\_ENCODED\\_REGION\\_NAME](#) procedure to look up the region name for an unencoded Splice Machine table or index key.

## Usage

You can compact a table region by specifying `NULL` for the index name. To compact an index region, specify both the table name and the index name.

Region compaction is asynchronous, which means that when you invoke this procedure from the command line, Splice Machine issues a compaction request to HBase, and returns control to you immediately; HBase will determine when to subsequently run the compaction.

## Results

This procedure does not return a result.

## Examples

The following example will perform a minor compaction on the region with encoded key value 8ffc80e3f8ac3b180441371319ea90e2 for table testTable. The encoded key value is first retrieved by passing the unencoded key value, 1|2, into the SYSCS\_UTIL.GET\_ENCODED\_REGION\_NAME procedure:

```
splice> CALL SYSCS_UTIL.GET_ENCODED_REGION_NAME('SPLICE', 'TESTTABLE', null, '1|2',
'|', null, null, null, null);
ENCODED_REGION_NAME          |START_KE
Y                             |END_KEY
-----
8ffc80e3f8ac3b180441371319ea90e2  |\x81\x00\x8
2                                |\x81\x00\x84

1 row selected

splice> CALL SYSCS_UTIL.COMPACT_REGION('SPLICE', 'testTable', NULL, '8ffc80e3f8ac3b1
80441371319ea90e2');
Statement executed.
```

And this example performs a minor compaction on the region with encoded index key value ff8f9e54519a31e15f264ba6d2b828a4 for index testIndex on table testTable. The encoded key value is first retrieved by passing the unencoded index key value, 1996-04-12|155190|21168.23|0.04, into the SYSCS\_UTIL.GET\_ENCODED\_REGION\_NAME procedure:

```
splice> CALL SYSCS_UTIL.GET_ENCODED_REGION_NAME('SPLICE', 'TESTTABLE', 'SHIP_INDE
X','1996-04-12|155190|21168.23|0.04', '|', null, null, null, null);
ENCODED_REGION_NAME          |START_KE
Y                             |END_KEY
-----
ff8f9e54519a31e15f264ba6d2b828a4  |\xEC\xC1\x15\xAD\xCD\x80\x00\xE1\x06\xEE\x0
0\xE4V&|

1 row selected

splice> CALL SYSCS_UTIL.COMPACT_REGION('SPLICE', 'testTable', 'testIndex', 'ff8f9e54
519a31e15f264ba6d2b828a4');
Statement executed.
```

## See Also

- » [SYSCS\\_UTIL.GET\\_ENCODED\\_REGION\\_NAME](#)
- » [SYSCS\\_UTIL.GET\\_REGIONS](#)
- » [SYSCS\\_UTIL.GET\\_START\\_KEY](#)
- » [SYSCS\\_UTIL.MAJOR\\_COMPACT\\_REGION](#)
- » [SYSCS\\_UTIL.MERGE\\_REGIONS](#)



# SYSCS\_UTIL.COMPUTE\_SPLIT\_KEY

Use the SYSCS\_UTIL.COMPUTE\_SPLIT\_KEY system procedure to compute the split keys for a table or index prior to calling the [SYSCS\\_UTIL.SYSCS\\_SPLIT\\_TABLE\\_OR\\_INDEX\\_AT\\_POINTS](#) procedure to split the data into HFiles. Once you've done that, call [SYSCS\\_UTIL.BULK\\_IMPORT\\_HFILE](#) system procedure to import your data in HFile format.

## Syntax

```
call SYSCS_UTIL.COMPUTE_SPLIT_KEY (
    schemaName,
    tableName,
    indexName,
    columnList | null,
    fileName,
    columnDelimiter | null,
    characterDelimiter | null,
    timestampFormat | null,
    dateFormat | null,
    timeFormat | null,
    maxBadRecords,
    badRecordDirectory | null,
    oneLineRecords | null,
    charset | null,
    outputDirectory
);
```

## Parameters

The following table summarizes the parameters used by SYSCS\_UTIL.COMPUTE\_SPLIT\_KEY and other Splice Machine data importation procedures. Each parameter name links to a more detailed description in our [Importing Data Tutorial](#).



The parameter values that you pass into this procedure should match the values that you use when you subsequently call the [SYSCS\\_UTIL.BULK\\_IMPORT\\_HFILE](#) procedure to perform the import.

Category	Parameter	Description	Example Value
Table Info	<a href="#">schemaName</a>	The name of the schema of the table in which to import.	SPLICE
	<a href="#">tableName</a>	The name of the table in which to import	playerTeams
Data Location	<a href="#">insertColumnList</a>	The names, in single quotes, of the columns to import. If this is null, all columns are imported.	'ID, TEAM'

Category	Parameter	Description	Example Value
	<u>fileOrDirectoryName</u>	<p>Either a single file or a directory. If this is a single file, that file is imported; if this is a directory, all of the files in that directory are imported. You can import compressed or uncompressed files.</p> <div> <p>The SYSCS_UTIL.MERGE_DATA_FROM_FILE procedure only works with single files; <b>you cannot specify a directory name</b> when calling SYSCS_UTIL.MERGE_DATA_FROM_FILE.</p> </div> <p>On a cluster, the files to be imported <b>MUST</b> be on S3, HDFS (or MapR-FS). If you're using our Database Service product, files can only be imported from S3.</p>	<p>/data/mydata/mytable.csv</p> <p>'s3a://splice-benchmark-data/flat/TPCH/100/region'</p>
Data Formats	<u>oneLineRecords</u>	A Boolean value that specifies whether ( <code>true</code> ) each record in the import file is contained in one input line, or ( <code>false</code> ) if a record can span multiple lines.	<code>true</code>
	<u>charset</u>	The character encoding of the import file. The default value is UTF-8.	<code>null</code>
	<u>columnDelimiter</u>	The character used to separate columns, Specify <code>null</code> if using the comma (,) character as your delimiter.	<code>' '</code>
	<u>characterDelimiter</u>	The character is used to delimit strings in the imported data.	<code>'\"'</code>
	<u>timestampFormat</u>	<p>The format of timestamps stored in the file. You can set this to <code>null</code> if there are no time columns in the file, or if the format of any timestamps in the file match the <code>Java.sql.Timestamp</code> default format, which is: "<code>yyyy-MM-dd HH:mm:ss</code>".</p> <div>  <p>All of the timestamps in the file you are importing must use the same format.</p> </div>	<code>'yyyy-MM-dd HH:mm:ss.SSZ'</code>

Category	Parameter	Description	Example Value
	<u>dateFormat</u>	The format of timestamps stored in the file. You can set this to <code>null</code> if there are no date columns in the file, or if the format of any dates in the file match pattern: " <code>yyyy-MM-dd</code> ".	<code>yyyy-MM-dd</code>
	<u>timeFormat</u>	The format of time values stored in the file. You can set this to <code>null</code> if there are no time columns in the file, or if the format of any times in the file match pattern: " <code>HH:mm:ss</code> ".	<code>HH:mm:ss</code>
<b>Problem Logging</b>	<u>badRecordsAllowed</u>	The number of rejected (bad) records that are tolerated before the import fails. If this count of rejected records is reached, the import fails, and any successful record imports are rolled back. Specify 0 to indicate that no bad records are tolerated, and specify -1 to indicate that all bad records should be logged and allowed.	25
	<u>badRecordDirectory</u>	<p>The directory in which bad record information is logged. Splice Machine logs information to the <code>&lt;import_file_name&gt;.bad</code> file in this directory; for example, bad records in an input file named <code>foo.csv</code> would be logged to a file named <code>badRecordDirectory/foo.csv.bad</code>.</p> <p>On a cluster, this directory <b>MUST be on S3, HDFS (or MapR-FS)</b>. If you're using our Database Service product, files can only be imported from S3.</p>	<code>'importErrsDir'</code>
<b>Bulk HFile Import</b>	<u>bulkImportDirectory (outputDirectory)</u>	<p>For <code>SYSCS_UTIL.BULK_IMPORT_HFILE</code>, this is the name of the directory into which the generated HFiles are written prior to being imported into your database.</p> <p>For the <code>SYSCS_UTIL.COMPUTE_SPLIT_KEY</code> procedure, where it is named <code>outputDirectory</code>, this parameter specifies the directory into which the split keys are written.</p>	<code>hdfs:///tmp/test_hfile_import/</code>
	<u>skipSampling</u>	<p>The <code>skipSampling</code> parameter is a Boolean value that specifies how you want the split keys used for the bulk HFile import to be computed. Set to <code>false</code> to have <code>SYSCS_UTIL.BULK_IMPORT_HFILE</code> automatically determine splits for you.</p> <p>This parameter is only used with the <code>SYSCS_UTIL.BULK_IMPORT_HFILE</code> system procedure.</p>	<code>false</code>

## Usage

The [SYSCS\\_UTIL.BULK\\_IMPORT\\_HFILE](#) procedure needs the data that you're importing split into multiple HFiles before it actually imports the data into your database. You can achieve these splits in three ways:

- » You can call `SYSCS_UTIL.BULK_IMPORT_HFILE` with the `skipSampling` parameter to `false`. `SYSCS_UTIL.BULK_IMPORT_HFILE` samples the data to determine the splits, then splits the data into multiple HFiles, and then imports the data.
- » You can split the data into HFiles with the `[SYSCS_UTIL.SYSCS_SPLIT_TABLE_OR_INDEX]` procedure, which both computes the keys and performs the splits. You then call `SYSCS_UTIL.BULK_IMPORT_HFILE` with the `skipSampling` parameter to `true` to import your data.
- » You can split the data into HFiles by first calling this procedure, `SYSCS_UTIL.COMPUTE_SPLIT_KEY`, and then calling the [SYSCS\\_UTIL.SYSCS\\_SPLIT\\_TABLE\\_OR\\_INDEX\\_AT\\_POINTS](#) procedure to split the table or index. You then call `SYSCS_UTIL.BULK_IMPORT_HFILE` with the `skipSampling` parameter to `true` to import your data.

In all cases, `SYSCS_UTIL.BULK_IMPORT_HFILE` automatically deletes the HFiles after the import process has completed.

The [Bulk HFile Import Examples](#) section of our *Importing Data Tutorial* describes how these methods differ and provides examples of using them to import data.

## Examples

The [Importing Data: Bulk HFile Examples](#) topic walks you through several examples of importing data with bulk HFiles.

## See Also

- » [SYSCS\\_UTIL.BULK\\_IMPORT\\_HFILE](#)
- » [SYSCS\\_UTIL.SYSCS\\_SPLIT\\_TABLE\\_OR\\_INDEX](#)
- » [SYSCS\\_UTIL.SYSCS\\_SPLIT\\_TABLE\\_OR\\_INDEX\\_AT\\_POINTS](#)

# SYSCS\_UTIL.SYSCS\_CREATE\_USER

The `SYSCS_UTIL.SYSCS_CREATE_USER` system procedure adds a new user account to a database.

This procedure creates users for use with NATIVE authentication.

If NATIVE authentication is not already turned on when you call this procedure:

- » The first user whose credentials are stored must be the database owner.
- » Calling this procedure will turn on NATIVE authentication the next time the database is booted.



Once you turn on NATIVE authentication with this procedure, it remains turned on permanently. There is no way to turn it off.

## Syntax

```
SYSCS_UTIL.SYSCS_CREATE_USER (
    IN userName VARCHAR(128),
    IN password VARCHAR(32672)
)
```

### *userName*

A user name that is case-sensitive if you place the name string in double quotes. This user name is an authorization identifier.

Case sensitivity is very specific with user names: if you specify the user name in single quotes, e.g. 'Fred', the system automatically converts it into all Uppercase.

**NOTE:** The user name is only case sensitive if you double-quote it inside of the single quotes. For example, '"Fred"' is a different user name than 'Fred', because 'Fred' is assumed to be case-insensitive.

### *password*

A case-sensitive password.

## Results

When you add a new user, a new schema is automatically created with exactly the same name as the user. For example, here's a sequence of an administrator adding a new user named `fred` and then verifying that the schema named `fred` is now active:

```

splice> CALL SYSCS_UTIL.SYSCS_CREATE_USER('fred', 'fredpassword');
Statement executed.
splice> VALUES(CURRENT SCHEMA);
1
-----
SPLICE

1 row selected
splice> SET SCHEMA fred;
0 rows inserted/updated/deleted
splice> VALUES(CURRENT SCHEMA);
1
-----
FRED

1 row selected

```

When the new user's credentials are used to connect to the database, his/her default schema will be that new schema. If you want the new user to have access to data in other schemas, such as the `SPLICE` schema, an administrator will need to explicitly [grant](#) those access privileges.

## Execute Privileges

If authentication and SQL authorization are both enabled, only the database owner has execute privileges on this function by default. The database owner can grant access to other users.

## JDBC example

Create a user named FRED:

```

CallableStatement cs = conn.prepareCall
    ("CALL SYSCS_UTIL.SYSCS_CREATE_USER(?, ?)");
cs.setString(1, "fred");
cs.setString(2, "fredpassword");
cs.execute();
cs.close();

```

Create a user named FreD:

```

CallableStatement cs = conn.prepareCall
    ("CALL SYSCS_UTIL.SYSCS_CREATE_USER(?, ?)");
cs.setString(1, "\"FreD\"");
cs.setString(2, "fredpassword");
cs.execute();
cs.close();

```

## SQL Example

Create a user named FRED:

```
splice> CALL SYSCS_UTIL.SYSCS_CREATE_USER('fred', 'fredpassword');  
Statement executed.
```

Create a (case sensitive) user named MrBaseball:

```
CALL SYSCS_UTIL.SYSCS_CREATE_USER('MrBaseball', 'pinchhitter')  
Statement executed.
```

## See Also

» [SYSCS\\_UTIL.SYSCS\\_DROP\\_USER](#) built-in system procedure

# SYSCS\_UTIL.SYSCS\_DELETE\_BACKUP

The SYSCS\_UTIL.SYSCS\_DELETE\_BACKUP system procedure deletes a backup that you previously created using either the [SYSCS\\_UTIL.SYSCS\\_SCHEDULE\\_DAILY\\_BACKUP](#) system procedures.

## Syntax

```
SYSCS_UTIL.SYSCS_DELETE_BACKUP( BIGINT backupId );
```

*backupId*  
Specifies the ID of the backup job you want to delete.  
To find the *jobId* you want to cancel, see the [Backing Up and Restoring](#) topic.

## Results

This procedure does not return a result.

## Execute Privileges

If authentication and SQL authorization are both enabled, only the database owner has execute privileges on this function by default. The database owner can grant access to other users.

## Example

If necessary, you can first query the [SYS.SYSBACKUP](#) system table to find the BACKUP\_ID of the job you want to delete; entries in that table include timestamp information.

And then delete that job:

```
splice> SELECT * FROM SYS.SYSBACKUP;
BACKUP_ID  |BEGIN_TIMESTAMP                |END_TIMESTAMP                |STATUS  |FILESYSTEM
M          |                                |SCOPE |INCR&|INCREMENTAL_PARENT_&|BACKUP_ITEM
-----
-----
40975      |2015-11-25 09:32:53.04         |2015-11-25 09:33:09.081     |S       |/Users/me/Documents/splicemachine/dbBackups
ID         |false|-1                       |93
1 row selected

splice> CALL SYSCS_UTIL.SYSCS_DELETE_BACKUP(40975);
Statement executed.
```



## See Also

- » [\*Backing Up and Restoring Databases\*](#)
- » [SYSCS\\_UTIL.SYSCS\\_BACKUP\\_DATABASE](#)
- » [SYSCS\\_UTIL.SYSCS\\_CANCEL\\_DAILY\\_BACKUP](#)
- » [SYSCS\\_UTIL.SYSCS\\_DELETE\\_OLD\\_BACKUPS](#)
- » [SYSCS\\_UTIL.SYSCS\\_RESTORE\\_DATABASE](#)
- » [SYSCS\\_UTIL.SYSCS\\_SCHEDULE\\_DAILY\\_BACKUP](#)
- » [SYSBACKUP](#)
- » [SYSBACKUPITEMS](#)
- » [SYSBACKUPJOBS](#)

## SYSCS\_UTIL.SYSCS\_DELETE\_OLD\_BACKUPS

The `SYSCS_UTIL.SYSCS_DELETE_OLD_BACKUPS` system procedure deletes any backups that are older than a specified number of days (the *backup window*), retaining only those backups that fit into that window.

Backups can consume a lot of disk space, and thus, we recommend regularly scheduling both the creation of new backups and deletion of outdated backups.

### Syntax

```
SYSCS_UTIL.SYSCS_DELETE_OLD_BACKUPS ( INT backupWindow );
```

#### *backupWindow*

Specifies the number of days of backups that you want retained. Any backups created more than `backupWindow` days ago are deleted.

See the [Backing Up and Restoring](#) topic in our *Administrator's Guide* for more information.

### Results

This procedure does not return a result.

### Execute Privileges

If authentication and SQL authorization are both enabled, only the database owner has execute privileges on this function by default. The database owner can grant access to other users.

### SQL Example

The following example deletes all database backups that were created more than 30 days ago.

```
splice> CALL SYSCS_UTIL.SYSCS_DELETE_OLD_BACKUPS(30);
Statement executed.
```

### See Also

- » [Backing Up and Restoring Databases](#)
- » [SYSCS\\_UTIL.SYSCS\\_BACKUP\\_DATABASE](#)
- » [SYSCS\\_UTIL.SYSCS\\_CANCEL\\_DAILY\\_BACKUP](#)

- » [SYSCS\\_UTIL.SYSCS\\_DELETE\\_BACKUP](#)
- » [SYSCS\\_UTIL.SYSCS\\_RESTORE\\_DATABASE](#)
- » [SYSCS\\_UTIL.SYSCS\\_SCHEDULE\\_DAILY\\_BACKUP](#)
- » [SYSDUMP](#)
- » [SYSDUMPITEMS](#)
- » [SYSDUMPJOBS](#)

# SYSCS\_UTIL.DELETE\_REGION

The `SYSCS_UTIL.DELETE_REGION` system procedure deletes a Splice Machine table or index region.

This procedure is intended for use only by expert database administrators. Use of this procedure requires extreme caution: you can easily create data inconsistencies.

## Syntax

```
SYSCS_UTIL.DELETE_REGION ( VARCHAR schemaName,
                           VARCHAR tableName,
                           VARCHAR indexName,
                           VARCHAR regionName,
                           VARCHAR mergeRegion )
```

### *schemaName*

The name of the schema of the table.

### *tableName*

The name of the table.

### *indexName*

`NULL` or the name of the index.

Specify the name of the index if you are deleting an index region; if you are a table region, specify `NULL` for this parameter.

### *regionName*

The **encoded** HBase name of the first of the two regions you want merged. You can call the [SYSCS\\_UTIL.GET\\_ENCODED\\_REGION\\_NAME](#) procedure to look up the region name for an unencoded Splice Machine table or index key.

### *mergeRegion*

Specify `TRUE` (case-insensitive) to merge the region after deleting all of its HFiles.

## Usage

Before invoking `SYSCS_UTIL.DELETE_REGION()`,:

- » Check region boundaries of the base table and indexes using the [SYSCS\\_UTIL.GET\\_REGIONS](#) procedure
- » Identify the set of regions from their indexes and tables, and make sure the index regions contains indexes to base table regions.

This procedure is intended for use only by expert database administrators. Use of this procedure requires extreme caution and is intended: you can easily create data inconsistencies.

## Configuration Parameters

There are several configuration options that you need to be aware of when using `SYSCS_UTIL.DELETE_REGION`:

- » The `hbase.hbck.close.timeout` value specifies the amount of time to wait for a region to close. The default value is 2 minutes.
- » The `hbase.hbck.assign.timeout` value specifies the amount of time to wait for a region to be assigned. The default value is 2 minutes.
- » We recommend setting the value of `hbase.rpc.timeout` to 20 minutes when using this procedure.

## Results

This procedure does not display a result.

## Example

Here's an example of creating a table and then deleting a table region and an index region from it.

### Create a Table and Index, and Split Them

```
splice> create table t(a int, b int, c int, primary key(a,b));
0 rows inserted/updated/deleted
splice> create index ti on t(a);
0 rows inserted/updated/deleted
splice> insert into t values (1,1,1), (2,2,2), (4,4,4), (5,5,5);
4 rows inserted/updated/deleted
splice> call syscs_util.syscs_split_table_or_index_at_points('SPLICE','T',null,'\x83');
Statement executed.
splice> call syscs_util.syscs_split_table_or_index_at_points('SPLICE','T','TI','\x83');
Statement executed.
```

**Display region information for base table and index**

```
splice> call syscs_util.get_regions('SPlice', 'T', 'TI', null, null, null, null, null, null, null);
ENCODED_REGION_NAME          |SPlice_START_KEY |SPlice_END_KEY |HBASE_START_KEY
|HBASE_END_KEY |NUM_HFILES |SIZE |LAST_MODIFICATION_TIME |REGION_NAME
-----
-----
02953478d84fcb1a7bb44f3eba0c9036 |{ NULL }          |{ 3 }          |
|\x83                        |1                  |1073 |2017-12-12 11:12:34.0 |splice:1809,,151310595332
5.02953478d84fcb1a7bb44f3eba0c9036.
1c1ee3dd90817576ef1148d91666defa |{ 3 }            |{ NULL }       ||\x83
|                             |1                  |1073 |2017-12-12 11:12:34.0 |splice:1809,\x83,151310595
3325.1c1ee3dd90817576ef1148d91666defa.

2 rows selected
splice> call syscs_util.get_regions('SPlice', 'T', null, null, null, null, null, null, null, null);
ENCODED_REGION_NAME          |SPlice_START_KEY |SPlice_END_KEY |HBASE_START_KEY
|HBASE_END_KEY |NUM_HFILES |SIZE |LAST_MODIFICATION_TIME |REGION_NAME
-----
-----
19c21ae5b0b2767403a8beff3148b646 |{ NULL, NULL }   |{ 3, NULL }   |
|\x83                        |1                  |1045 |2017-12-12 11:12:08.0 |splice:1792,,151310592782
4.19c21ae5b0b2767403a8beff3148b646.
6c8ac07d50cc2e606562dc1949705374 |{ 3, NULL }      |{ NULL, NULL } ||\x83
|                             |1                  |1045 |2017-12-12 11:12:08.0 |splice:1792,\x83,151310592
7824.6c8ac07d50cc2e606562dc1949705374.

2 rows selected
```

**Delete one region from base table and one region from index**

```
splice> call syscs_util.delete_region('SPlice', 'T', null, '19c21ae5b0b2767403a8beff3148b646', true);
Statement executed.
splice> call syscs_util.delete_region('SPlice', 'T', 'TI', '02953478d84fcb1a7bb44f3eba0c9036', true);
Statement executed.
```

## Verify the results

```
splice> call syscs_util.get_regions('SPICE', 'T', 'TI',null,null,null,null,null,nul
1,null);
ENCODED_REGION_NAME          |SPICE_START_KEY |SPICE_END_KEY |HBASE_START_KEY
|HBASE_END_KEY |NUM_HFILES |SIZE |LAST_MODIFICATION_TIME |REGION_NAME
-----
-----
5e8d1ffdf5e8aaa4e85a851caf17a2d9 |{ NULL }      |{ NULL }      |
|              |1             |1073 |2017-12-12 11:14:15.0 |splice:1809,,151310605454
7.5e8d1ffdf5e8aaa4e85a851caf17a2d9.

1 row selected
splice> select count(*) from t --splice-properties index=null
> ;
1
-----
2

1 row selected
splice> select count(*) from t --splice-properties index=ti
> ;
1
-----
2

1 row selected
```

## See Also

- » [SYSCS\\_UTIL.GET\\_START\\_KEY](#)
- » [SYSCS\\_UTIL.GET\\_REGIONS](#)
- » [SYSCS\\_UTIL.MERGE\\_REGIONS](#)

# SYSCS\_UTIL.SYSCS\_DELETE\_SNAPSHOT

The `SYSCS_UTIL.SYSCS_DELETE_SNAPSHOT` system procedure deletes a previously created Splice Machine snapshot.

**NOTE:** Snapshots include both the data and indexes for tables.

For more information, see the [Using Snapshots](#) topic.

## Syntax

```
SYSCS_UTIL.SYSCS_DELETE_SNAPSHOT ( VARCHAR(128) snapshotName );
```

*snapshotName*

The name of the snapshot that you are deleting.

## Results

This procedure does not return a result.

## Example

The following example deletes a snapshot:

```
splice> CALL SYSCS_UTIL.DELETE_SNAPSHOT ( 'snap_myschema_070417a' );  
Statement executed.
```



# SYSCS\_UTIL.DISABLE\_COLUMN\_STATISTICS

The `SYSCS_UTIL.DISABLE_COLUMN_STATISTICS` system procedure disables collection of statistics on a specific table column in your database.

## Syntax

```
SYSCS_UTIL.DISABLE_COLUMN_STATISTICS (  
    VARCHAR(128) schema,  
    VARCHAR(128) table,  
    VARCHAR(128) columnName)
```

### *schemaName*

Specifies the schema of the table. Passing a `null` or non-existent schema name generates an error.

### *tableName*

Specifies the table name of the table. The string must exactly match the case of the table name, and the argument of "Fred" will be passed to SQL as the delimited identifier 'Fred'. Passing a `null` or non-existent table name generates an error.

### *columnName*

Specifies the name of the column for which you want statistics disabled. Passing a `null` or non-existent column name generates an error.

## Results

This procedure does not return a result.

## Usage Notes

Statistics are automatically collected on all columns by default. Attempting to disable statistics collection on a keyed column generates an error.

## SQL Examples

```
splice> CALL SYSCS_UTIL.DISABLE_COLUMN_STATISTICS('SPLICE', 'Salaries', 'Salary');  
Statement executed.
```

## See Also

- » [Data Assignments and Comparisons](#)
- » [SYSCS\\_UTIL.ENABLE\\_COLUMN\\_STATISTICS](#)
- » [SYSCS\\_UTIL.COLLECT\\_SCHEMA\\_STATISTICS](#)
- » [SYSCS\\_UTIL.DROP\\_SCHEMA\\_STATISTICS](#)
- » [Using Statistics](#)

# SYSCS\_UTIL.DROP\_SCHEMA\_STATISTICS

The `SYSCS_UTIL.DROP_SCHEMA_STATISTICS` system procedure drops statistics for a specific schema in your database.

This procedure drops statistics for every table in the schema. It also drops statistics for the index associated with every table in the schema. For example, if you have :

- » a schema named `mySchema`
- » `mySchema` contains two tables: `myTable1` and `myTable2`
- » `myTable1` has two indices: `myTable1Index1` and `myTable1Index2`

Then `SYSCS_UTIL.DROP_SCHEMA_STATISTICS` will drop statistics for `myTable1`, `myTable2`, `myTable1Index1`, and `myTable1Index2`.

## Syntax

```
SYSCS_UTIL.DROP_SCHEMA_STATISTICS ( VARCHAR(128) schema );
```

### *schemaName*

Specifies the schema for which you want to drop statistics. Passing a `null` or non-existent schema name generates an error.

## Results

This procedure does not produce a result.

## SQL Examples

```
splice> CALL SYSCS_UTIL.DROP_SCHEMA_STATISTICS ('MYSCHEMA');
Statement executed.
```

## See Also

- » [Data Assignments and Comparisons](#)
- » [SYSCS\\_UTIL.ENABLE\\_COLUMN\\_STATISTICS](#)
- » [SYSCS\\_UTIL.DISABLE\\_COLUMN\\_STATISTICS](#)
- » [SYSCS\\_UTIL.COLLECT\\_SCHEMA\\_STATISTICS](#)
- » [Using Statistics](#)



# SYSCS\_UTIL.SYSCS\_DROP\_USER

The `SYSCS_UTIL.SYSCS_DROP_USER` system procedure removes a user account from a database.

This procedure is used in conjunction with NATIVE authentication..

You are not allowed to remove the user account of the database owner.

If you use this procedure to remove a user account, the schemas and data objects owned by the user remain in the database and can be accessed only by the database owner or by other users who have been granted access to them. If the user is created again, then he or she regains access to the schemas and data objects.

## Syntax

```
SYSCS_UTIL.SYSCS_DROP_USER( IN userName VARCHAR(128) )
```

*userName*

A user name that is case-sensitive if you place the name string in double quotes. This user name is an authorization identifier. If the user name is that of the database owner, an error is raised.

## Results

This procedure does not return a result.

## Execute Privileges

If authentication and SQL authorization are both enabled, only the database owner has execute privileges on this function by default. The database owner can grant access to other users.

## JDBC example

Drop a user named FRED:

```
CallableStatement cs = conn.prepareCall  
("CALL SYSCS_UTIL.SYSCS_DROP_USER('fred')");  
cs.execute();  
cs.close();
```

## SQL Example

Drop a user named Fred:

```
splice> CALL SYSCS_UTIL.SYSCS_DROP_USER('fred');  
Statement executed;
```

## See Also

» [SYSCS\\_UTIL.SYSCS\\_CREATE\\_USER](#)

# SYSCS\_UTIL.SYSCS\_EMPTY\_GLOBAL\_STATEMENT\_CACHE

The `SYSCS_UTIL.SYSCS_EMPTY_GLOBAL_STATEMENT_CACHE` stored procedure removes as many compiled statements (plans) as possible from the database-wide statement cache (across all region servers). This procedure does not remove statements related to currently executing queries or to activations that are about to be garbage collected, so the cache is not guaranteed to be completely empty after it completes.

**NOTE:** The related procedure [SYSCS\\_UTIL.SYSCS\\_EMPTY\\_STATEMENT\\_CACHE](#) performs the same operation on a single region server.

## Syntax

```
SYSCS_UTIL.SYSCS_EMPTY_GLOBAL_STATEMENT_CACHE ()
```

## Results

This procedure does not return a result.

## Execute Privileges

If authentication and SQL authorization are both enabled, only the database owner has execute privileges on this function by default. The database owner can grant access to other users.

## JDBC Example

```
CallableStatement cs = conn.prepareCall  
    ("CALL SYSCS_UTIL.SYSCS_EMPTY_GLOBAL_STATEMENT_CACHE ()");  
cs.execute();  
cs.close();
```

## SQL Example

```
splice> CALL SYSCS_UTIL.SYSCS_EMPTY_GLOBAL_STATEMENT_CACHE ();  
Statement executed.
```

## See Also

- » [SYSCS\\_UTIL.SYSCS\\_EMPTY\\_STATEMENT\\_CACHE](#)
- » [SYSCS\\_UTIL.SYSCS\\_INVALIDATE\\_STORED\\_STATEMENTS](#)



## SYSCS\_UTIL.SYSCS\_EMPTY\_STATEMENT\_CACHE

The `SYSCS_UTIL.SYSCS_EMPTY_STATEMENT_CACHE` stored procedure removes as many compiled statements (plans) as possible from the database statement cache. on your current region server. This procedure does not remove statements related to currently executing queries or to activations that are about to be garbage collected, so the cache is not guaranteed to be completely empty after it completes.

**NOTE:** The related procedure [SYSCS\\_UTIL.SYSCS\\_EMPTY\\_GLOBAL\\_STATEMENT\\_CACHE](#) performs the same operation across the entire cluster.

### Syntax

```
SYSCS_UTIL.SYSCS_EMPTY_STATEMENT_CACHE ()
```

### Results

This procedure does not return a result.

### Execute Privileges

If authentication and SQL authorization are both enabled, only the database owner has execute privileges on this function by default. The database owner can grant access to other users.

### JDBC Example

```
CallableStatement cs = conn.prepareCall
("CALL SYSCS_UTIL.SYSCS_EMPTY_STATEMENT_CACHE ()");
cs.execute();
cs.close();
```

### SQL Example

```
splice> CALL SYSCS_UTIL.SYSCS_EMPTY_STATEMENT_CACHE ();
Statement executed.
```

## See Also

- » [SYSCS\\_UTIL.SYSCS\\_EMPTY\\_GLOBAL\\_STATEMENT\\_CACHE](#)
- » [SYSCS\\_UTIL.SYSCS\\_INVALIDATE\\_STORED\\_STATEMENTS](#)
- » [SYSCS\\_UTIL.SYSCS\\_UPDATE\\_METADATA\\_STORED\\_STATEMENTS](#)

# SYSCS\_UTIL.ENABLE\_COLUMN\_STATISTICS

The `SYSCS_UTIL.ENABLE_COLUMN_STATISTICS` system procedure enables collection of statistics on a specific table column in your database.

## Syntax

```
SYSCS_UTIL.ENABLE_COLUMN_STATISTICS (
    VARCHAR(128) schema,
    VARCHAR(128) table,
    VARCHAR(128) columnName)
```

### *schemaName*

Specifies the schema of the table. Passing a `null` or non-existent schema name generates an error.

### *tableName*

Specifies the table name of the table. The string must exactly match the case of the table name, and the argument of "Fred" will be passed to SQL as the delimited identifier 'Fred'. Passing a `null` or non-existent table name generates an error.

### *columnName*

Specifies the name of the column for which you want statistics enabled. Passing a `null` or non-existent column name generates an error.

## Results

This procedure does not return a result.

## Usage Notes

Here are some **important notes** about collecting column statistics:

- » Statistics can only be collected on columns with data types that can be ordered; numeric types, some `CHAR` types, some `BIT` types, and date/time types can be ordered.

You can determine if a data type can be ordered by examining the Comparisons table in the [Data Assignments and Comparisons](#) topic: any data type with a `Y` in any column in that table can be ordered, and thus can have statistics collected on it.

- » Statistics are automatically collected on all columns by default.

## SQL Examples

```
splice> CALL SYSCS_UTIL.ENABLE_COLUMN_STATISTICS('SPLICE', 'Salaries', 'Salary');  
Statement executed.
```

## See Also

- » [Data Assignments and Comparisons](#)
- » [SYSCS\\_UTIL.DISABLE\\_COLUMN\\_STATISTICS](#)
- » [SYSCS\\_UTIL.COLLECT\\_SCHEMA\\_STATISTICS](#)
- » [SYSCS\\_UTIL.DROP\\_SCHEMA\\_STATISTICS](#)
- » [Using Statistics](#)

## SYSCS\_UTIL.SYSCS\_ENABLE\_ENTERPRISE

The `SYSCS_UTIL.SYSCS_ENABLE_ENTERPRISE` stored procedure unlocks access to features that are only available in the Enterprise Edition of Splice Machine.

Calling `SYSCS_UTIL.SYSCS_ENABLE_ENTERPRISE` with a valid license key unlocks access to *Enterprise-only* features in Splice Machine such as backing up and restoring your database. However, to unlock bootstrapped authentication and encryption features such as LDAP and Kerberos, you must also modify your `hbase-site.xml` file and restart Splice Machine.

**NOTE:** Please see the [Upgrading to the Enterprise Edition of Splice Machine](#) topic for more information.

### Syntax

```
SYSCS_UTIL.SYSCS_ENABLE_ENTERPRISE( STRING license_key );
```

*license\_key*

The license key you received from Splice Machine.

### SQL Example

```
splice> CALL SYSCS_UTIL.SYSCS_ENABLE_ENTERPRISE (<your-license-code>);
Statement executed.
```

### Results

This procedure does not return a result; however, if you provide an invalid license key, you'll see an error message displayed:

```
splice> CALL SYSCS_UTIL.SYSCS_ENABLE_ENTERPRISE (<bogus-code>);
Error
-----

ERROR XSRSE: Unable to enable the enterprise Manager. Enterprise services are disabled. Contact your Splice Machine representative to enable.
```

### See Also

» [Upgrading to the Enterprise Edition of Splice Machine](#)

# SYSCS\_UTIL.SYSCS\_GET\_ACTIVE\_SERVERS

The `SYSCS_UTIL.SYSCS_GET_ACTIVE_SERVERS` system procedure displays the active servers in the Splice cluster.

## Syntax

```
SYSCS_UTIL.SYSCS_GET_ACTIVE_SERVERS ()
```

## Results

The displayed results of calling `SYSCS_UTIL.SYSCS_GET_ACTIVE_SERVERS` include these values:

Value	Description
HOSTNAME	The host on which the server is running.
PORT	The port on which the server is listening for requests.
STARTCODE	The system identifier for the Region Server.

## Example

```
splice> CALL SYSCS_UTIL.SYSCS_GET_ACTIVE_SERVERS ();
HOSTNAME | PORT | STARTCODE
-----
localhost | 56412 | 1447433590803

1 row selected
```

# SYSCS\_UTIL.SYSCS\_GET\_ALL\_PROPERTIES

The `SYSCS_UTIL.SYSCS_GET_ALL_PROPERTIES` system procedure displays all of the Splice Machine Derby properties.

## Syntax

```
SYSCS_UTIL.SYSCS_GET_ALL_PROPERTIES ()
```

## Results

The displayed results of calling `SYSCS_UTIL.SYSCS_GET_ALL_PROPERTIES` include these values:

Value	Description
KEY	The property name
VALUE	The property value
TYPE	The property type

## Example



```

splice> CALL SYSCS_UTIL.SYSCS_GET_ALL_PROPERTIES();
KEY                                     | VALU
E                                     | TYPE
-----
derby.authentication.builtin.algorithm | SHA-51
2                                     | JVM
derby.authentication.native.create.credentials.da& | tru
e                                     | JVM
derby.authentication.provider          | NATIVE:spliceDB:LOCA
L                                     | JVM
derby.connection.requireAuthentication | tru
e                                     | JVM
derby.database.collation               | UCS_BASI
C                                     | DATABASE
derby.database.defaultConnectionMode   | fullAcces
s                                     | SERVICE
derby.database.propertiesOnly          | fals
e                                     | SERVICE
derby.database.sqlAuthorization        | tru
e                                     | JVM
derby.engineType                      | 2
| 2                                     | SERVICE
derby.language.logQueryPlan           | fals
e                                     | SERVICE
derby.language.logStatementText       | fals
e                                     | SERVICE
derby.language.updateSystemProcs      | fals
e                                     | JVM
derby.locks.escalationThreshold       | 50
0                                     | SERVICE
derby.storage.propertiesId            | 1
6                                     | SERVICE
derby.storage.rowLocking              | fals
e                                     | SERVICE
derby.stream.error.file               | ./splice-derby.lo
g                                     | JVM
splice.authentication                 | NATIV
E                                     | JVM
splice.debug.logStatementContext       | fals
e                                     | JVM
splice.software.buildtime              | 2015-10-21 13:18 -050
0                                     | JVM
splice.software.release                | 1.5.1
| 1.5.1                               | JVM
splice.software.url                   | http://www.splicemachine.co
m                                     | JVM
splice.software.versionhash           | felb10bda
0                                     | JVM

22 rows selected

```



# SYSCS\_UTIL.SYSCS\_GET\_CURRENT\_TRANSACTION

The `SYSCS_UTIL.SYSCS_GET_CURRENT_TRANSACTION` system procedure displays summary information about the current transaction.

## Syntax

```
SYSCS_UTIL.SYSCS_GET_CURRENT_TRANSACTION()
```

## Results

The displayed results of calling `SYSCS_UTIL.SYSCS_GET_CURRENT_TRANSACTION` include these values:

Value	Description
txnId	The ID of the current transaction

## Example

```
splice> CALL SYSCS_UTIL.SYSCS_GET_CURRENT_TRANSACTION();
txnId
-----
2081

1 row selected
```

# SYSCS\_UTIL.SYSCS\_GET\_GLOBAL\_DATABASE\_PROPERTY Function

The `SYSCS_UTIL.SYSCS_GET_GLOBAL_DATABASE_PROPERTY` function fetches the value of the specified property of the database.

## Syntax

```
VARCHAR(32672) SYSCS_UTIL.SYSCS_GET_GLOBAL_DATABASE_PROPERTY (
    IN Key VARCHAR(128)
)
```

### Key

The key for the property whose value you want.

**NOTE:** An error occurs if *Key* is null.

## Results

Returns the value of the property. If the value that was set for the property is invalid, the `SYSCS_UTIL.SYSCS_GET_GLOBAL_DATABASE_PROPERTY` function returns the invalid value, but Splice Machine uses the default value.

## Execute Privileges

If authentication and SQL authorization are both enabled, only the database owner has execute privileges on this function by default. The database owner can grant access to other users.

## SQL Example

Retrieve the value of the `splicemachine.locks.deadlockTimeout` property:

```
splice> VALUES SYSCS_UTIL.SYSCS_GET_GLOBAL_DATABASE_PROPERTY( 'splicemachine.locks.d
eadlockTimeout' );
1
-----
10

1 row selected
```

## See Also

» [SYSCS\\_UTIL.SYSCS\\_SET\\_GLOBAL\\_DATABASE\\_PROPERTY](#)

## SYSCS\_UTIL.GET\_ENCODED\_REGION\_NAME

The `SYSCS_UTIL.GET_ENCODED_REGION_NAME` system procedure returns the encoded name of the HBase region that contains the Splice Machine table primary key or index values for the `unencodedKey` value that you specify.

You can call this procedure to retrieve an encoded HBase region name prior to calling the `SYSCS_UTIL.MAJOR_COMPACT_REGION`, `SYSCS_UTIL.COMPACT_REGION`, or `SYSCS_UTIL.MERGE_REGIONS` procedures.

### Syntax

```
SYSCS_UTIL.GET_ENCODED_REGION_NAME ( VARCHAR schemaName,
                                     VARCHAR tableName,
                                     VARCHAR indexName,
                                     VARCHAR unencodedKey,
                                     VARCHAR columnDelimiter,
                                     VARCHAR characterDelimiter,
                                     VARCHAR timestampFormat,
                                     VARCHAR dateFormat,
                                     VARCHAR timeFormat )
```

#### *schemaName*

The name of the schema of the table.

#### *tableName*

The name of the table.

#### *indexName*

`NULL` or the name of the index.

Specify `NULL` to indicate that the `unencodedKey` is the primary key of the base table; specify an index name to indicate that the `unencodedKey` is an index value.

#### *unencodedKey*

For a table, this is a comma-separated-value (CSV) representation of the table's primary key (unencoded). For an index, this is the CSV representation of the index columns, also unencoded.

#### *columnDelimiter*

The character used to separate columns in `unencodedKey`. Specify `null` if using the comma (,) character as your delimiter.

#### *characterDelimiter*

Specifies which character is used to delimit strings in `unencodedKey`. 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.

To use the single quote (') character as your string delimiter, you need to escape that character. This means that you specify four quotes (' ' ' ') as the value of this parameter. This is standard SQL syntax.

**NOTE:** The [Examples](#) section below contains an example that uses the single quote as the string delimiter character.

#### *timestampFormat*

The format of timestamps in `unencodedKey`. You can set this to `null` if there are no time columns in the split key, or if the format of any timestamps in the file match the `Java.sql.Timestamp` default format, which is: “`yyyy-MM-dd HH:mm:ss`”.

See the [About Timestamp Formats](#) section in the [SYSCS\\_UTIL.IMPORT\\_DATA](#) topic for more information about timestamps.

#### *dateFormat*

The format of datestamps in `unencodedKey`. You can set this to `null` if there are no date columns in the `unencodedKey`, or if the format of any dates in the split key match this pattern: “`yyyy-MM-dd`”.

#### *timeFormat*

The format of time values stored in `unencodedKey`. You can set this to `null` if there are no time columns in the file, or if the format of any times in the split key match this pattern: “`HH:mm:ss`”.

## Usage

Use this procedure to retrieve the HBase-encoded name of a table or index region in your database. These system procedures required encoded region names as parameter values:

- » [SYSCS\\_UTIL.COMPACT\\_REGION](#)
- » [SYSCS\\_UTIL.MAJOR\\_COMPACT\\_REGION](#)
- » [SYSCS\\_UTIL.MERGE\\_REGIONS](#)

## Results

The displayed results of calling `SYSCS_UTIL.SYSCS_GET_ENCODED_REGION_NAME` include these values:

Value	Description
<code>ENCODED_REGION_NAME</code>	The HBase-encoded name of the region.
<code>START_KEY</code>	The HBase starting key for the region.
<code>END_KEY</code>	The HBase ending key for the region.

## Examples

The following call will retrieve the encoded region name for TESTTABLE for a table row that has key value 1 | 2:

```
splice> CALL SYSCS_UTIL.GET_ENCODED_REGION_NAME (
          'SPlice', 'TESTTABLE', null, '1|2', '|', null, null, null, null););
ENCODED_REGION_NAME          |START_KEY          |END_KEY
-----
8ffc80e3f8ac3b180441371319ea90e2      |\x81\x00\x82      |\x81\x00\x84

1 row selected
```

This call will retrieve the encoded region name for TESTTABLE for a region that contains index value 1996-04-12,155190,21168.23,0.04:

```
splice> CALL SYSCS_UTIL.GET_ENCODED_REGION_NAME (
          'SPlice', 'TESTTABLE', 'SHIP_INDEX','1996-04-12|155190|21168.23|0.0
4',
          '|', null, null, null, null);
ENCODED_REGION_NAME          |START_KEY
|END_KEY
-----
ff8f9e54519a31e15f264ba6d2b828a4  |\xEC\xC1\x15\xAD\xCD\x80\x00\xE1\x06\xEE\x00\xE4
V&|

1 row selected
```

## See Also

- » [SYSCS\\_UTIL.COMPACT\\_REGION](#)
- » [SYSCS\\_UTIL.GET\\_START\\_KEY](#)
- » [SYSCS\\_UTIL.GET\\_REGIONS](#)
- » [SYSCS\\_UTIL.MAJOR\\_COMPACT\\_REGION](#)
- » [SYSCS\\_UTIL.MERGE\\_REGIONS](#)



# SYSCS\_UTIL.SYSCS\_GET\_LOGGERS

The SYSCS\_UTIL.SYSCS\_GET\_LOGGERS system procedure displays the names of all Splice Machine loggers in the system. Use this to find loggers of interest, if you want to determine or change their log levels.

**NOTE:** You can read more about Splice Machine loggers in the [Logging](#) topic.

## Syntax

```
SYSCS_UTIL.SYSCS_GET_LOGGERS ()
```

## Results

The displayed results of calling SYSCS\_UTIL.SYSCS\_GET\_LOGGERS include these values:

Value	Description
LOGGERNAME	The name of the logger

## Example

Here's the output from a call to SYSCS\_UTIL.SYSCS\_GET\_LOGGERS, as of Splice Machine Release 1.5:

```
splice> CALL SYSCS_UTIL.SYSCS_GET_LOGGERS();
SPLICELOGGER
-----
com.splicemachine
com.splicemachine.async.HBaseClient
com.splicemachine.async.QueueingAsyncScanner
com.splicemachine.async.RegionClient
com.splicemachine.async.RegionInfo
com.splicemachine.async.Scanner
com.splicemachine.concurrent.LoggingScheduledThreadPoolExecutor
com.splicemachine.constants.SpliceConstants
com.splicemachine.constants.environment.EnvUtils
com.splicemachine.db
com.splicemachine.db.impl.ast.AssignRSNVisitor
com.splicemachine.db.impl.ast.FindHashJoinColumns
com.splicemachine.db.impl.ast.FixSubqueryColRefs
com.splicemachine.db.impl.ast.JoinConditionVisitor
com.splicemachine.db.impl.ast.JsonTreeBuilderVisitor
com.splicemachine.db.impl.ast.PlanPrinter
com.splicemachine.db.impl.ast.RowLocationColumnVisitor
com.splicemachine.db.impl.ast.SpliceDerbyVisitorAdapter
com.splicemachine.db.impl.jdbc.authentication
com.splicemachine.db.impl.sql.catalog
com.splicemachine.db.impl.sql.catalog.DefaultSystemProcedureGenerator
com.splicemachine.db.impl.sql.compile.subquery.exists.ExistsSubqueryPredicate
com.splicemachine.db.impl.sql.execute.operations
com.splicemachine.db.shared.common.sanity
com.splicemachine.derby.ddl.AsynchronousDDLController
com.splicemachine.derby.ddl.DDLWatchRefresher
com.splicemachine.derby.ddl.DDLZookeeperClient
com.splicemachine.derby.ddl.ZooKeeperDDLWatchChecker
com.splicemachine.derby.ddl.ZookeeperDDLWatcher
com.splicemachine.derby.hbase.AbstractSpliceIndexObserver
com.splicemachine.derby.hbase.AbstractSpliceIndexObserver.Compaction
com.splicemachine.derby.hbase.AbstractSpliceIndexObserver.Split
com.splicemachine.derby.hbase.ActivationSerializer
com.splicemachine.derby.hbase.RollForwardAction
com.splicemachine.derby.hbase.RollForwardTask
com.splicemachine.derby.hbase.ShutdownRegionServerObserver
com.splicemachine.derby.hbase.SpliceBaseIndexEndpoint
com.splicemachine.derby.hbase.SpliceBaseOperationRegionScanner
com.splicemachine.derby.hbase.SpliceDerbyCoproprocessor
com.splicemachine.derby.hbase.SpliceDriver
com.splicemachine.derby.hbase.SpliceIndexEndpoint
com.splicemachine.derby.hbase.SpliceIndexObserver
com.splicemachine.derby.hbase.SpliceMasterObserver
com.splicemachine.derby.hbase.SpliceObserverInstructions
com.splicemachine.derby.hbase.SpliceOperationRegionObserver
com.splicemachine.derby.hbase.SpliceOperationRegionScanner
com.splicemachine.derby.hbase.SpliceWriteControl
com.splicemachine.derby.iapi.sql.execute.OperationResultSet
```

```

com.splicemachine.derby.iapi.sql.execute.SpliceNoPutResultSet
com.splicemachine.derby.iapi.sql.execute.SpliceOperationContext
com.splicemachine.derby.impl.SpliceMethod
com.splicemachine.derby.impl.SpliceService
com.splicemachine.derby.impl.db.SpliceDatabase
com.splicemachine.derby.impl.job.coprocessor.CoprocessorTaskScheduler
com.splicemachine.derby.impl.job.operation.SinkTask
com.splicemachine.derby.impl.job.scheduler.BaseJobControl
com.splicemachine.derby.impl.job.scheduler.DistributedJobScheduler
com.splicemachine.derby.impl.job.scheduler.JobControl
com.splicemachine.derby.impl.job.scheduler.RegionTaskControl
com.splicemachine.derby.impl.job.scheduler.TaskCallable
com.splicemachine.derby.impl.job.scheduler.WorkStealingTaskScheduler
com.splicemachine.derby.impl.services.streams.ConfiguredStream
com.splicemachine.derby.impl.spark.SpliceSpark
com.splicemachine.derby.impl.sql.catalog
com.splicemachine.derby.impl.sql.catalog.SpliceDataDictionary
com.splicemachine.derby.impl.sql.catalog.upgrade
com.splicemachine.derby.impl.sql.compile.NestedLoopJoinStrategy
com.splicemachine.derby.impl.sql.depend.SpliceDependencyManager
com.splicemachine.derby.impl.sql.execute.LazyDataValueDescriptor
com.splicemachine.derby.impl.sql.execute.LazyStringDataValueDescriptor
com.splicemachine.derby.impl.sql.execute.LazyTimestampDataValueDescriptor
com.splicemachine.derby.impl.sql.execute.SpliceExecutionFactory
com.splicemachine.derby.impl.sql.execute.SpliceGenericConstantActionFactory
com.splicemachine.derby.impl.sql.execute.SpliceGenericResultSetFactory
com.splicemachine.derby.impl.sql.execute.SpliceRealResultSetStatisticsFactory
com.splicemachine.derby.impl.sql.execute.actions.DeleteConstantOperation
com.splicemachine.derby.impl.sql.execute.actions.TransactionReadTask
com.splicemachine.derby.impl.sql.execute.operations.AnyOperation
com.splicemachine.derby.impl.sql.execute.operations.BroadcastJoinRows
com.splicemachine.derby.impl.sql.execute.operations.BroadcastJoinOperation
com.splicemachine.derby.impl.sql.execute.operations.CachedOperation
com.splicemachine.derby.impl.sql.execute.operations.CallStatementOperation
com.splicemachine.derby.impl.sql.execute.operations.DMLWriteOperation
com.splicemachine.derby.impl.sql.execute.operations.DeleteOperation
com.splicemachine.derby.impl.sql.execute.operations.IndexRowReader
com.splicemachine.derby.impl.sql.execute.operations.IndexRowToBaseRowOperation
com.splicemachine.derby.impl.sql.execute.operations.JoinOperation
com.splicemachine.derby.impl.sql.execute.operations.JoinUtils
com.splicemachine.derby.impl.sql.execute.operations.Joiner
com.splicemachine.derby.impl.sql.execute.operations.MergeSortJoinOperation
com.splicemachine.derby.impl.sql.execute.operations.NoRowsOperation
com.splicemachine.derby.impl.sql.execute.operations.NormalizeOperation
com.splicemachine.derby.impl.sql.execute.operations.OperationTree
com.splicemachine.derby.impl.sql.execute.operations.ProjectRestrictOperation
com.splicemachine.derby.impl.sql.execute.operations.RowOperation
com.splicemachine.derby.impl.sql.execute.operations.ScanOperation
com.splicemachine.derby.impl.sql.execute.operations.SpliceBaseOperation
com.splicemachine.derby.impl.sql.execute.operations.SpliceBaseOperation.close
com.splicemachine.derby.impl.sql.execute.operations.TableScanOperation

```

```

com.splicemachine.derby.impl.sql.execute.operations.UpdateOperation
com.splicemachine.derby.impl.sql.execute.operations.scanner.SITableScanner
com.splicemachine.derby.impl.stats.CachedPhysicalStatsStore
com.splicemachine.derby.impl.stats.HBaseColumnStatisticsStore
com.splicemachine.derby.impl.stats.PartitionStatsStore
com.splicemachine.derby.impl.stats.StatisticsTask
com.splicemachine.derby.impl.stats.StatsConstants
com.splicemachine.derby.impl.storage.AbstractMultiScanProvider
com.splicemachine.derby.impl.storage.AbstractScanProvider
com.splicemachine.derby.impl.storage.BaseHashAwareScanBoundary
com.splicemachine.derby.impl.storage.ClientResultScanner
com.splicemachine.derby.impl.storage.ClientScanProvider
com.splicemachine.derby.impl.storage.DistributedClientScanProvider
com.splicemachine.derby.impl.storage.MeasuredResultScanner
com.splicemachine.derby.impl.storage.MultiScanRowProvider
com.splicemachine.derby.impl.storage.RegionAwareScanner
com.splicemachine.derby.impl.storage.ReopenableScanner
com.splicemachine.derby.impl.storage.RowProviders
com.splicemachine.derby.impl.storage.SingleScanRowProvider
com.splicemachine.derby.impl.store.access.BaseSpliceTransaction
com.splicemachine.derby.impl.store.access.HBaseStore
com.splicemachine.derby.impl.store.access.PropertyConglomerate
com.splicemachine.derby.impl.store.access.SpliceAccessManager
com.splicemachine.derby.impl.store.access.SpliceLockFactory
com.splicemachine.derby.impl.store.access.SpliceTransaction
com.splicemachine.derby.impl.store.access.SpliceTransactionContext
com.splicemachine.derby.impl.store.access.SpliceTransactionFactory
com.splicemachine.derby.impl.store.access.SpliceTransactionManager
com.splicemachine.derby.impl.store.access.SpliceTransactionManagerContext
com.splicemachine.derby.impl.store.access.StatsStoreCostController
com.splicemachine.derby.impl.store.access.base.SpliceConglomerate
com.splicemachine.derby.impl.store.access.base.SpliceController
com.splicemachine.derby.impl.store.access.base.SpliceScan
com.splicemachine.derby.impl.store.access.btree.IndexConglomerate
com.splicemachine.derby.impl.store.access.btree.IndexConglomerateFactory
com.splicemachine.derby.impl.store.access.btree.IndexController
com.splicemachine.derby.impl.store.access.hbase.HBaseConglomerate
com.splicemachine.derby.impl.store.access.hbase.HBaseController
com.splicemachine.derby.impl.temp.TempTable
com.splicemachine.derby.jdbc.SpliceTransactionResourceImpl
com.splicemachine.derby.logging.DerbyOutputLoggerWriter
com.splicemachine.derby.management.StatementManager
com.splicemachine.derby.management.TransactionalsysTableWriter
com.splicemachine.derby.utils.ConglomerateUtils
com.splicemachine.derby.utils.DerbyBytesUtil
com.splicemachine.derby.utils.Scans
com.splicemachine.derby.utils.SpliceAdmin
com.splicemachine.derby.utils.SpliceUtils
com.splicemachine.derby.utils.StatisticsAdmin
com.splicemachine.hbase.AbstractBufferedRegionScanner
com.splicemachine.hbase.BufferedReaderScanner

```

```

com.splicemachine.hbase.HBaseRegionLoads
com.splicemachine.hbase.NoRetryCoprocessorRpcChannel
com.splicemachine.hbase.backup.Backup
com.splicemachine.hbase.backup.BackupHFileCleaner
com.splicemachine.hbase.backup.BackupReporter
com.splicemachine.hbase.backup.BackupSystemProcedures
com.splicemachine.hbase.backup.BackupUtils
com.splicemachine.hbase.backup.SnapshotUtilsBase
com.splicemachine.hbase.backup.SnapshotUtilsImpl
com.splicemachine.hbase.regioninfocache.HBaseRegionCache
com.splicemachine.hbase.table.BetterHTablePool
com.splicemachine.hbase.table.SpliceHTable
com.splicemachine.hbase.table.SpliceHTableFactory
com.splicemachine.job.CompositeJobResults
com.splicemachine.job.ZkTaskMonitor
com.splicemachine.mrio.api
com.splicemachine.pipeline.callbuffer.PipingCallBuffer
com.splicemachine.pipeline.callbuffer.RegionServerCallBuffer
com.splicemachine.pipeline.ddl.DDLChange
com.splicemachine.pipeline.exception.SpliceDoNotRetryIOException
com.splicemachine.pipeline.impl.BulkWriteAction
com.splicemachine.pipeline.impl.BulkWriteAction.retries
com.splicemachine.pipeline.impl.BulkWriteChannelInvoker
com.splicemachine.pipeline.threadpool.MonitoredThreadPool
com.splicemachine.pipeline.utils.PipelineConstants
com.splicemachine.pipeline.utils.PipelineUtils
com.splicemachine.pipeline.writeconfiguration.BaseWriteConfiguration
com.splicemachine.pipeline.writecontext.PipelineWriteContext
com.splicemachine.pipeline.writecontextfactory.LocalWriteContextFactory
com.splicemachine.pipeline.writehandler.RegionWriteHandler
com.splicemachine.queryPlan
com.splicemachine.si.api.Txn
com.splicemachine.si.coprocessors.SIBaseObserver
com.splicemachine.si.coprocessors.SIObserver
com.splicemachine.si.coprocessors.TimestampMasterObserver
com.splicemachine.si.coprocessors.TxnLifecycleEndpoint
com.splicemachine.si.impl.BaseSIFilter
com.splicemachine.si.impl.ClientTxnLifecycleManager
com.splicemachine.si.impl.PackedTxnFilter
com.splicemachine.si.impl.ReadOnlyTxn
com.splicemachine.si.impl.SITransactor
com.splicemachine.si.impl.WritableTxn
com.splicemachine.si.impl.readresolve.AsyncReadResolver
com.splicemachine.si.impl.readresolve.SynchronousReadResolver
com.splicemachine.si.impl.region.RegionTxnStore
com.splicemachine.si.impl.region.TransactionResolver
com.splicemachine.si.impl.rollforward.SegmentedRollForward
com.splicemachine.si.impl.timestamp.TimestampClient
com.splicemachine.si.impl.timestamp.TimestampOracle
com.splicemachine.tools.version.ManifestFinder
com.splicemachine.utils.SpliceUtilities

```

```
com.splicemachine.utils.SpliceZooKeeperManager  
com.splicemachine.utils.ZkUtils
```

```
203 rows selected
```

## See Also

» [SYSCS\\_UTIL.SYSCS\\_GET\\_LOGGER\\_LEVEL](#)

» [SYSCS\\_UTIL.SYSCS\\_SET\\_LOGGER\\_LEVEL](#)

# SYSCS\_UTIL.SYSCS\_GET\_LOGGER\_LEVEL

The SYSCS\_UTIL.SYSCS\_GET\_LOGGER\_LEVEL system procedure displays the logging level of the specified logger.

**NOTE:** You can read more about Splice Machine loggers and logging levels in the [Logging](#) topic of our *Developer's Guide*.

## Syntax

```
SYSCS_UTIL.SYSCS_GET_LOGGER_LEVEL(loggerName)
```

*loggerName*

A string specifying the name of the logger whose logging level you want to find.

You can find all of the available loggers by using the [SYSCS\\_UTIL.SYSCS\\_GET\\_LOGGERS](#) system procedure.

## Results

The displayed results of calling SYSCS\_UTIL.SYSCS\_GET\_LOGGER\_LEVEL include these values:

Value	Description
LOGLEVEL	<p>The level of the logger. This is one of the following values, which are described in the <a href="#">Logging</a> topic:</p> <ul style="list-style-type: none"><li>» TRACE</li><li>» DEBUG</li><li>» INFO</li><li>» WARN</li><li>» ERROR</li><li>» FATAL</li></ul>

## Example

Here are two examples of using this procedure:

```
splice> CALL SYSCS_UTIL.SYSCS_GET_LOGGER_LEVEL('com.splicemachine.utils.SpliceUtilities');
LOG&
----
WARN

1 row selected

splice> CALL SYSCS_UTIL.SYSCS_GET_LOGGER_LEVEL('com.splicemachine.mrio.api');
LOGL&
-----
DEBUG

1 row selected
```

## See Also

- » [SYSCS\\_UTIL.SYSCS\\_GET\\_LOGGERS](#)
- » [SYSCS\\_UTIL.SYSCS\\_SET\\_LOGGER\\_LEVEL](#)
- » [\*Splice Machine Logging\*](#) in our *Developer's Guide*.



# SYSCS\_UTIL.GET\_REGIONS

The SYSCS\_UTIL.GET\_REGIONS system procedure that retrieves the list of regions containing a range of key values.

## Syntax

```
SYSCS_UTIL.GET_REGIONS ( VARCHAR schemaName,
                        VARCHAR tableName,
                        VARCHAR indexName,
                        VARCHAR startKey,
                        VARCHAR endKey,
                        VARCHAR columnDelimiter,
                        VARCHAR characterDelimiter,
                        VARCHAR timestampFormat,
                        VARCHAR dateFormat,
                        VARCHAR timeFormat )
```

### *schemaName*

The name of the schema of the table.

### *tableName*

The name of the table.

### *indexName*

NULL or the name of the index.

Specify NULL to indicate that the `startKey` is the primary key of the base table; specify an index name to indicate that the `startKey` is an index value.

### *startKey*

For a table, this is a comma-separated-value (CSV) representation of the primary key value for the start of the regions in which you are interested. For an index, this is the CSV representation of the index columns.

### *endKey*

For a table, this is a comma-separated-value (CSV) representation of the primary key value for the end of the regions in which you are interested. For an index, this is the CSV representation of the index columns.

### *columnDelimiter*

The character used to separate columns in `startKey`. Specify `null` if using the comma (,) character as your delimiter.

### *characterDelimiter*

Specifies which character is used to delimit strings in `startKey`. 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.

To use the single quote ( ' ) character as your string delimiter, you need to escape that character. This means that you specify four quotes ( ' ' ' ' ) as the value of this parameter. This is standard SQL syntax.

**NOTE:** The [Examples](#) section below contains an example that uses the single quote as the string delimiter character.

#### *timestampFormat*

The format of timestamps in `startKey`. You can set this to `null` if there are no time columns in the split key, or if the format of any timestamps in the file match the `Java.sql.Timestamp` default format, which is: “yyyy-MM-dd HH:mm:ss”.

See the [About Timestamp Formats](#) section in the [SYSCS\\_UTIL.IMPORT\\_DATA](#) topic for more information about timestamps.

#### *dateFormat*

The format of datestamps in `startKey`. You can set this to `null` if there are no date columns in the `startKey`, or if the format of any dates in the split key match this pattern: “yyyy-MM-dd”.

#### *timeFormat*

The format of time values stored in `startKey`. You can set this to `null` if there are no time columns in the file, or if the format of any times in the split key match this pattern: “HH:mm:ss”.

## Usage

Specify the starting and ending key values, this procedure returns information about all regions that span those key values.

## Results

The displayed results of calling `SYSCS_UTIL.SYSCS_GET_REGIONS` include these values:

Value	Description
ENCODED_REGION_NAME	The HBase-encoded name of the region.
SPLICE_START_KEY	The unencoded start key, in CSV format, for the list of regions in which you are interested. This is the value you supplied in the <code>startKey</code> parameter. For example: {1, 2}.
SPLICE_END_KEY	The unencoded end key for the region, in CSV format, for the list of regions in which you are interested. This is the value you supplied in the <code>endKey</code> parameter. For example: {1, 6}.

Value	Description
HBASE_START_KEY	The start key for the region, formatted as shown in the HBase Web UI. For example: \x81\x00\x82.
HBASE_END_KEY	The end key for the region, formatted as shown in the HBase Web UI. For example: \x81\x00\x86.
NUM_HFILES	The number of HBase Files contained in the region.
SIZE	The size, in bytes, of the region.
LAST_MODIFICATION_TIME	The most recent time at which the region was modified.
REGION_NAME	The unencoded name of the region.

Example

The following call returns information about the regions that are in the key range {1,2} to {1,8}:

```
splice> CALL SYSCS_UTIL.GET_REGIONS( 'SPLICE','TestTable', null,
                                     '1|2', '1|8', '|',null,null,null,null);
ENCODED_REGION_NAME      |SPLICE_START_KEY |SPLICE_END_KEY |HBASE_START_KEY
|HBASE_END_KEY |NUM_HFILES |SIZE  |LAST_MODIFICATION_TIME |REGION_NAME
-----
132c824b9e269006a8e0a3fad577bd12 |{ 1, 2}          |{ 1, 6}          |\x81\x00\x82
|\x81\x00\x86  |1              |1645  |2017-08-17 12:44:15.0  |splice:2944,\x81\x00\x8
2,1502999053574.132c824b9e269006a8e0a3fad577bd12.
2ee995a552cbb75b7172eed27b917cab |{ 1, 6 }         |{ 1, 8 }         |\x81\x00\x86
|\x81\x00\x88  |1              |1192  |2017-08-17 08:37:56.0  |splice:2944,\x81\x00\x8
6,1502984266749.2ee995a552cbb75b7172eed27b917cab.

2 rows selected
```

See Also

- >> [SYSCS\\_UTIL.COMPACT\\_REGION](#)
- >> [SYSCS\\_UTIL.GET\\_ENCODED\\_REGION\\_NAME](#)
- >> [SYSCS\\_UTIL.GET\\_START\\_KEY](#)
- >> [SYSCS\\_UTIL.MAJOR\\_COMPACT\\_REGION](#)
- >> [SYSCS\\_UTIL.MERGE\\_REGIONS](#)

# SYSCS\_UTIL.SYSCS\_GET\_REGION\_SERVER\_STATS\_INFO

The SYSCS\_UTIL.SYSCS\_GET\_REGION\_SERVER\_STATS\_INFO system procedure displays input and output statistics about the cluster.

## Syntax

```
SYSCS_UTIL.SYSCS_GET_REGION_SERVER_STATS_INFO()
```

## Results

The displayed results of calling SYSCS\_UTIL.SYSCS\_GET\_REGION\_SERVER\_STATS\_INFO include these values:

Value	Description
HOST	The host name (or IP address).
REGIONCOUNT	The number of regions.
STOREFILECOUNT	The number of files stored.
WRITEREQUESTCOUNT	The number of write requests.
READREQUESTCOUNT	The number of read requests.
TOTALREQUESTCOUNT	The total number of requests.

## Example

```
splice> CALL SYSCS_UTIL.SYSCS_GET_REGION_SERVER_STATS_INFO();
Host          |regionCount      |storeFileCount    |writeRequestCount  |readRe
questCount    |totalRequestCount
-----
111.222.3.4   |58                |0                  |5956               |9969
7             |20517
555.666.7.8   |59                |0                  |1723               |5702
2             |6253
1 row selected
```

# SYSCS\_UTIL.SYSCS\_GET\_REQUESTS

The SYSCS\_UTIL.SYSCS\_GET\_REQUESTS system procedure displays information about the number of RPC requests that are coming into Splice Machine.

## Syntax

```
SYSCS_UTIL.SYSCS_GET_REQUESTS ()
```

## Results

The displayed results of calling SYSCS\_UTIL.SYSCS\_GET\_REQUESTS include these values:

Value	Description
HOSTNAME	The host name.
PORT	The port receiving requests.
TOTALREQUESTS	The total number of RPC requests on that port.

## Example

```
splice> CALL SYSCS_UTIL.SYSCS_GET_REQUESTS ();
HOSTNAME  |PORT  |TOTALREQUE&
-----
localhost |55709 |7296

1 row selected
```

## SYSCS\_UTIL.SYSCS\_GET\_RUNNING\_OPERATIONS

The `SYSCS_UTIL.SYSCS_GET_RUNNING_OPERATIONS` system procedure displays a list of the operations running on the server to which you are currently connected.

You can use this procedure to find the UUID for an operation, which you can then use for purposes such as terminating an operation with the [SYSCS\\_UTIL.SYSCS\\_KILL\\_OPERATION](#) system procedure.

### Syntax

```
SYSCS_UTIL.SYSCS_GET_RUNNING_OPERATIONS () ;
```

### Results

The displayed results of calling `SYSCS_UTIL.SYSCS_GET_RUNNING_OPERATIONS` include these values:

Value	Description
UUID	The operation identifier. This is the same identifier that is shown in the Spark console.
USER	The name of the database user.
HOSTNAME	The host on which the server is running.
SESSION	The session ID.
SQL	The SQL statement that is running.
SUBMITTED	The date and time that the operation was submitted.
ELAPSED	Elapsed time since the operation began running.
ENGINE	Which engine (SPARK or CONTROL) is running the operation.
JOBTYPE	The operation type.

### Example

```
splice> call SYSCS_UTIL.SYSCS_GET_RUNNING_OPERATIONS () ;
```

UUID	USER	HOSTNAME	SESSION	SQL
34b0f479-be9a-4933-9b4d-900af218a19c	SPLICE	MacBook-Pro.local:1527	264	select * from sys.systables --splice-properties useSpark=true
4099f016-3c9d-4c62-8059-ff18d3b38a19	SPLICE	MacBook-Pro.local:1527	4	call syscs_util.syscs_get_running_operations(

2 rows selected

```
splice> call SYSCS_UTIL.SYSCS_KILL_OPERATION('4099f016-3c9d-4c62-8059-ff18d3b38a19');
Statement executed.
```

# SYSCS\_UTIL.SYSCS\_GET\_SCHEMA\_INFO

The SYSCS\_UTIL.SYSCS\_GET\_SCHEMA\_INFO system procedure displays table information,including the HBase regions occupied and their store file size, for all user schemas.

## Syntax

```
SYSCS_UTIL.SYSCS_GET_SCHEMA_INFO()
```

## Results

The displayed results of calling SYSCS\_UTIL.SYSCS\_GET\_SCHEMA\_INFO include these values:

Value	Description
SCHEMANAME	The schema to which the table belongs.
TABLERNAME	The name of the table. Note that may be more than one row containing a table name; for example, this happens if the table has an index.
ISINDEX	A Boolean value that specifies whether the HBase table is an index table.
HBASEREGIONS	<p>The HBase regions on which the table resides. There can be multiple regions.</p> <p>Each region display shows (tableName, regionId.storeFileSize, memStoreSize, and storeIndexSize MB).</p>



Example

```
splice> CALL SYSCS_UTIL.SYSCS_GET_SCHEMA_INFO();
SCHEMANAME | TABLENAME | REGIONNAME | I
S_I&|HBASEREGIONS_STORES&|MEMSTORESIZE | STOREINDEXSIZE
-----
SPLICE      | PLAYERS    | 2176,,1446847689610.7211e284f7f767d7b142dbd639b4d9bf. | fals
e|0          | 0          | 0          |
SPLICE      | PITCHING   | 1968,,1446260714743.01963d7260fc9d4dc01507eccdf67e40. | fals
e|0          | 0          | 0          |
SPLICE      | BATTING    | 1984,,1446260731076.ca29785eb5b16a8752d9c4ceeaad2ce4. | fals
e|0          | 0          | 0          |
SPLICE      | FIELDING   | 2192,,1447092732332.b4aae99023002bbac1a08432e6ffc2df. | fals
e|0          | 0          | 0          |
SPLICE      | SALARIES   | 2256,,1447803176538.11ce38c9e470b4d209de4d32c96cb815. | fals
e|0          | 0          | 0          |

5 rows selected
```

## SYSCS\_UTIL.SYSCS\_GET\_SESSION\_INFO

The `SYSCS_UTIL.SYSCS_GET_SESSION_INFO` system procedure displays the hostname and session ID for your current session. You can use this information to correlate your Splice Machine query with a Spark job: the same information is displayed in the Job Id (Job Group) in the Spark console.

### Syntax

```
SYSCS_UTIL.SYSCS_GET_SESSION_INFO()
```

### Results

The displayed results of calling `SYSCS_UTIL.SYSCS_GET_SESSION` include these values:

Value	Description
HOSTNAME	The identity of your Splice Machine connection.
SESSION	The ID of your database connection session.

### Example

```
splice> CALL SYSCS_UTIL.SYSCS_GET_SESSION_INFO();
HOSTNAME                               |SESSION |
-----
localhost:1527                          | 4
1 row selected
```

For this session, you could find your Spark job ID by correlating the displayed host and session IDs with the Job Group information displayed in the Spark console. For example:

Job Id (Job Group)	Description	Submitted	Duration	Stages: Succeeded/ Total	Tasks (for all stages): Succeeded/ Total
0 (SPICE <localhost:1527,4,e5ff1a51-4ac1-4202-a74c- a2d54ab3525a,36096>)	select * from sys.systables --splice- properties useSpark=true (kill) Produce Result Set	2017/08/ 25 14:03:07	8s	0/1	0/1

## SYSCS\_UTIL.GET\_START\_KEY

The `SYSCS_UTIL.GET_START_KEY` system procedure that retrieves the starting key value, in unencoded format, for a specified HBase region.

### Syntax

```
SYSCS_UTIL.GET_START_KEY( VARCHAR schemaName,
                          VARCHAR tableName,
                          VARCHAR indexName,
                          VARCHAR encodedRegionName)
```

#### *schemaName*

The name of the schema of the table.

#### *tableName*

The name of the table.

#### *indexName*

`NULL` or the name of the index.

Specify `NULL` to indicate that the `startKey` is the primary key of the base table; specify an index name to indicate that the `startKey` is an index value.

#### *encodedRegionName*

The HBase-encoded name of the region, which you can retrieve using the [SYSCS\\_UTIL.GET\\_ENCODED\\_REGION\\_NAME](#) system procedure.

### Usage

Use this procedure to discover the starting key value for an HBase region.

### Results

Displays the start key for the region in Splice Machine unencoded format.

### Example

The following call returns the start key for an HBase table region:

```
splice> CALL SYSCS_UTIL.GET_START_KEY('SPLICE', 'myTable', null, '9d427082bedabb796
56369b353e401cc');
START_KEY
-----
{ 2, NULL }

1 row selected
```

The following call returns the start key for for the region that stores index `myIndex` on table `myTable`:

```
splice> CALL SYSCS_UTIL.GET_START_KEY('SPLICE', 'myTable', 'myIndex', 'b35fe82916cddl
d48bb5c43f60a9b8b5');
START_KEY
-----
{ 1996-04-11, 67310, 45983.16, 0.09 }

1 row selected
```

## See Also

- » [SYSCS\\_UTIL.COMPACT\\_REGION](#)
- » [SYSCS\\_UTIL.GET\\_ENCODED\\_REGION\\_NAME](#)
- » [SYSCS\\_UTIL.GET\\_REGIONS](#)
- » [SYSCS\\_UTIL.MAJOR\\_COMPACT\\_REGION](#)
- » [SYSCS\\_UTIL.MERGE\\_REGIONS](#)

# SYSCS\_UTIL.SYSCS\_GET\_VERSION\_INFO

The SYSCS\_UTIL.SYSCS\_GET\_VERSION\_INFO system procedure displays the version of Splice Machine installed on each node in your cluster.

## Syntax

```
SYSCS_UTIL.SYSCS_GET_VERSION_INFO()
```

## Results

This procedure does not return a result.

## Example

```
splice> call SYSCS_UTIL.SYSCS_GET_VERSION_INFO();
HOSTNAME          |RELEASE          | IMPLEMENT& |BUILDTIME          |URL
-----
localhost:52897|2.5.0.1708-SNAPSHOT|85caa07187|2017-02-25 04:56 +0000 |http://www.splicemachine.com

1 row selected
```

# SYSCS\_UTIL.SYSCS\_GET\_WRITE\_INTAKE\_INFO

The SYSCS\_UTIL.SYSCS\_GET\_WRITE\_INTAKE\_INFO system procedure displays information about the number of writes coming into Splice Machine.

You can use this information to know the number of bulk writes currently active on a server. Each bulk write will contain up to 1000 rows; the compaction and flush queue size, plus the reserved ipc thread setting determine how many writes can execute concurrently.

## Syntax

```
SYSCS_UTIL.SYSCS_GET_WRITE_INTAKE_INFO()
```

## Results

The displayed results of calling SYSCS\_UTIL.SYSCS\_GET\_WRITE\_INTAKE\_INFO include these values:

Value	Description
HOSTNAME	The host name.
ACTIVIEWRITETHREADS	The number of active write threads.
COMPACTIONQUEUEUESIZELIMIT	The compaction queue limit at which writes will be blocked.
FLUSHQUEUEELIMIT	The flush queue limit at which writes will be blocked.
IPCRESERVEDPOOL	<div>The number of IPC threads reserved for reads.</div> <div>The maximum number of bulk writes that are allowed currently is equal to the total number of IPC threads minus this value.</div>

Example

```
splice> CALL SYSCS_UTIL.SYSCS_GET_WRITE_INTAKE_INFO();
host          |depThreads |indThreads |depCount      |indCount      |avgThroughpu
t             |oneMinAvgThroughput  |fiveMinAvgThroughput  |fifteenMinAvgThroughp&|tota
lRejected
-----
-----
-----
localhost:55709      |0          |0          |0             |0             |0.01145109332258
9732 |5.472367185912236E-4 |0.007057900046373111 |0.00538845111108858845 |0

1 row selected
```



## SYSCS\_UTIL.IMPORT\_DATA

The `SYSCS_UTIL.IMPORT_DATA` system procedure imports data to a new record in a table. You can choose to import all or a subset of the columns from the input data into your database using the `insertColumnList` parameter.

After a successful import completes, a simple report displays, showing how many files were imported, and how many record imports succeeded or failed.

### Selecting an Import Procedure

Splice Machine provides four system procedures for importing data:

- » This procedure, `SYSCS_UTIL.IMPORT_DATA`, imports each input record into a new record in your database.
- » The [SYSCS\\_UTIL.UPSERT\\_DATA\\_FROM\\_FILE](#) procedure updates existing records and adds new records to your database. It only differs from `SYSCS_UTIL.MERGE_DATA_FROM_FILE` in that upserting **overwrites** the generated or default value of a column that *is not specified* in your `insertColumnList` parameter when updating a record.
- » The [SYSCS\\_UTIL.MERGE\\_DATA\\_FROM\\_FILE](#) procedure updates existing records and adds new records to your database. It only differs from `SYSCS_UTIL.UPSERT_DATA_FROM_FILE` in that merging **does not overwrite** the generated or default value of a column that *is not specified* in your `insertColumnList` parameter when updating a record.
- » The [SYSCS\\_BULK\\_IMPORT\\_HFILE](#) procedure 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 has improved performance for large tables; however, the bulk HFile import requires extra work on your part and lacks constraint checking.

Our [Importing Data Tutorial](#) includes a decision tree and brief discussion to help you determine which procedure best meets your needs.

## Syntax

```
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
);
```

## Parameters

The following table summarizes the parameters used by SYSCS\_UTIL.IMPORT\_DATA and other Splice Machine data importation procedures. Each parameter name links to a more detailed description in our [Importing Data Tutorial](#).

Category	Parameter	Description	Example Value
Table Info	<a href="#">schemaName</a>	The name of the schema of the table in which to import.	SPLICE
	<a href="#">tableName</a>	The name of the table in which to import	playerTeams
Data Location	<a href="#">insertColumnList</a>	The names, in single quotes, of the columns to import. If this is null, all columns are imported.	'ID, TEAM'

Category	Parameter	Description	Example Value
	<u>fileOrDirectoryName</u>	<p>Either a single file or a directory. If this is a single file, that file is imported; if this is a directory, all of the files in that directory are imported. You can import compressed or uncompressed files.</p> <div> <p>The SYSCS_UTIL.MERGE_DATA_FROM_FILE procedure only works with single files; <b>you cannot specify a directory name</b> when calling SYSCS_UTIL.MERGE_DATA_FROM_FILE.</p> </div> <p>On a cluster, the files to be imported <b>MUST</b> be on S3, HDFS (or MapR-FS). If you're using our Database Service product, files can only be imported from S3.</p>	<p>/data/mydata/mytable.csv</p> <p>'s3a://splice-benchmark-data/flat/TPCH/100/region'</p>
Data Formats	<u>oneLineRecords</u>	A Boolean value that specifies whether ( <code>true</code> ) each record in the import file is contained in one input line, or ( <code>false</code> ) if a record can span multiple lines.	<code>true</code>
	<u>charset</u>	The character encoding of the import file. The default value is UTF-8.	<code>null</code>
	<u>columnDelimiter</u>	The character used to separate columns, Specify <code>null</code> if using the comma (,) character as your delimiter.	<code>' '</code>
	<u>characterDelimiter</u>	The character is used to delimit strings in the imported data.	<code>'\"'</code>
	<u>timestampFormat</u>	<p>The format of timestamps stored in the file. You can set this to <code>null</code> if there are no time columns in the file, or if the format of any timestamps in the file match the <code>Java.sql.Timestamp</code> default format, which is: "<code>yyyy-MM-dd HH:mm:ss</code>".</p> <div>  <p>All of the timestamps in the file you are importing must use the same format.</p> </div>	<code>'yyyy-MM-dd HH:mm:ss.SSZ'</code>

Category	Parameter	Description	Example Value
	<u>dateFormat</u>	The format of timestamps stored in the file. You can set this to <code>null</code> if there are no date columns in the file, or if the format of any dates in the file match pattern: " <code>yyyy-MM-dd</code> ".	<code>yyyy-MM-dd</code>
	<u>timeFormat</u>	The format of time values stored in the file. You can set this to <code>null</code> if there are no time columns in the file, or if the format of any times in the file match pattern: " <code>HH:mm:ss</code> ".	<code>HH:mm:ss</code>
<b>Problem Logging</b>	<u>badRecordsAllowed</u>	The number of rejected (bad) records that are tolerated before the import fails. If this count of rejected records is reached, the import fails, and any successful record imports are rolled back. Specify 0 to indicate that no bad records are tolerated, and specify -1 to indicate that all bad records should be logged and allowed.	25
	<u>badRecordDirectory</u>	<p>The directory in which bad record information is logged. Splice Machine logs information to the <code>&lt;import_file_name&gt;.bad</code> file in this directory; for example, bad records in an input file named <code>foo.csv</code> would be logged to a file named <code>badRecordDirectory/foo.csv.bad</code>.</p> <p>On a cluster, this directory <b>MUST be on S3, HDFS (or MapR-FS)</b>. If you're using our Database Service product, files can only be imported from S3.</p>	<code>'importErrsDir'</code>
<b>Bulk HFile Import</b>	<u>bulkImportDirectory (outputDirectory)</u>	<p>For <code>SYSCS_UTIL.BULK_IMPORT_HFILE</code>, this is the name of the directory into which the generated HFiles are written prior to being imported into your database.</p> <p>For the <code>SYSCS_UTIL.COMPUTE_SPLIT_KEY</code> procedure, where it is named <code>outputDirectory</code>, this parameter specifies the directory into which the split keys are written.</p>	<code>hdfs:///tmp/test_hfile_import/</code>
	<u>skipSampling</u>	<p>The <code>skipSampling</code> parameter is a Boolean value that specifies how you want the split keys used for the bulk HFile import to be computed. Set to <code>false</code> to have <code>SYSCS_UTIL.BULK_IMPORT_HFILE</code> automatically determine splits for you.</p> <p>This parameter is only used with the <code>SYSCS_UTIL.BULK_IMPORT_HFILE</code> system procedure.</p>	<code>false</code>

## Results

`SYSCS_UTIL.IMPORT_DATA` displays a summary of the import process results that looks like this:

rowsImported	failedRows	files	dataSize	failedLog
94	0	1	4720	NONE

This procedure also logs rejected record activity into `.bad` files in the `badRecordDirectory` directory; one file for each imported file.

## Importing and Updating Records

What distinguishes `SYSCS_UTIL.IMPORT_DATA` from the similar [SYSCS\\_UTIL.UPSERT\\_DATA\\_FROM\\_FILE](#) and [SYSCS\\_UTIL.SYSCS\\_MERGED\\_DATA\\_FROM\\_FILE](#) procedures is how each works with these specific conditions:

- » You are importing only a subset of data from the input data into your table, either because the table contains less columns than does the input file, or because you've specified a subset of the columns in your `insertColumnList` parameter.
- » Inserting and updating data in a column with generated values.
- » Inserting and updating data in a column with default values.
- » Handling of missing values.

The [Importing Data Tutorial: Input Handling](#) topic describes how each of these conditions is handled by the different system procedures.

## Record Import Failure Reasons

Typical reasons for a row (record) import to fail include:

- » Improper data expected for a column.
- » Improper number of columns of data.
- » A primary key violation: [SYSCS\\_UTIL.IMPORT\\_DATA](#) will only work correctly if the table into which you are inserting/ updating has primary keys.

## Usage Notes

A few important notes:

- » Splice Machine advises you to run a full compaction (with the [SYSCS\\_UTIL.SYSCS\\_PERFORM\\_MAJOR\\_COMPACTION\\_ON\\_TABLE](#) system procedure) after importing large amounts of data into your database.

- » On a cluster, the files to be imported **MUST be on S3, HDFS (or MapR-FS)**, as must the `badRecordDirectory` directory. If you're using our Database Service product, files can only be imported from S3.

In addition, the files must be readable by the `hbase` user, and the `badRecordDirectory` directory must be writable by the `hbase` user, either by setting the user explicitly, or by opening up the permissions; for example:

```
sudo -su hdfs hadoop fs -chmod 777 /badRecordDirectory
```

## Examples

This section presents a couple simple examples.

The [Importing Data Usage Examples](#) topic contains a more extensive set of examples.

### Example 1: Importing our doc examples player data

This example shows the `IMPORT_DATA` call used to import the `Players` table into our documentation examples database:

```
splice> CALL SYSCS_UTIL.IMPORT_DATA('SPICEBALL', 'Players',
  'ID, Team, Name, Position, DisplayName, BirthDate',
  '/Data/DocExamplesDb/Players.csv',
  null, null, null, null, null, 0, null, true, null);
```

rowsImported	failedRows	files	dataSize	failedLo
94	0	1	4720	NONE

1 row selected

### Example 2: Specifying a timestamp format for an entire table

Use a single timestamp format for the entire table by explicitly specifying a single `timeStampFormat`.

```
Mike,2013-04-21 09:21:24.98-05
Mike,2013-04-21 09:15:32.78-04
Mike,2013-03-23 09:45:00.68-05
```

You can then import the data with the following call:

```
splice> CALL SYSCS_UTIL.IMPORT_DATA('app', 'tabx', 'c1,c2',
  '/path/to/ts3.csv',
  ',', '',
  'yyyy-MM-dd HH:mm:ss.SSZ',
  null, null, 0, null, true, null);
```

Note that for any import use case shown above, the time shown in the imported table depends on the timezone setting in the server timestamp. In other words, given the same csv file, if imported on different servers with timestamps set to different time zones, the value in the table shown will be different. Additionally, daylight savings time may account for a 1-hour difference if timezone is specified.

See [Importing Data Usage Examples](#) for more examples.

## See Also

- » [Our Importing Data Tutorial](#)
- » [Importing Data Usage Examples](#)
- » [SYSCS\\_UTIL.UPSERT\\_DATA\\_FROM\\_FILE](#)
- » [SYSCS\\_UTIL.MERGE\\_DATA\\_FROM\\_FILE](#)

# SQLJ.INSTALL\_JAR

The `SQLJ.INSTALL_JAR` system procedure stores a jar file in a database.

**NOTE:** For more information about using JAR files, see the [Using Functions and Stored Procedures](#) section in our *Developer's Guide*.

## Syntax

```
SQLJ.INSTALL_JAR(IN jar_file_path_or-url VARCHAR(32672),
                 IN qualified_jar_name VARCHAR(32672),
                 IN deploy INTEGER)
```

### *jar\_file\_path\_or-url*

The path or URL of the jar file to add. A path includes both the directory and the file name (unless the file is in the current directory, in which case the directory is optional). For example:

```
d:/todays_build/tours.jar
```

### *qualified\_jar\_name*

Splice Machine name of the jar file, qualified by the schema name. Two examples:

```
MYSCHEMA.Sample1
-- a delimited identifier
MYSCHEMA."Sample2"
```

### *deploy*

If this set to 1, it indicates the existence of an SQLJ deployment descriptor file. Splice Machine ignores this argument, so it is normally set to 0.

## Usage Notes

This procedure will not work properly unless you have first added your procedure to the Derby CLASSPATH variable. For example:

```
CALL SYSCS_UTIL.SYSCS_SET_GLOBAL_DATABASE_PROPERTY('derby.database.classpath', 'SPICE.MY_EXAMPLE_APP');
```

For information about storing and updating stored procedures, and the setting of the Derby classpath, see the [Storing and Updating Splice Machine Functions and Stored Procedures](#) topic.

## Results

This procedure does not return a result.



## Execute Privileges

If authentication and SQL authorization are both enabled, only the database owner has execute privileges on this function by default. The database owner can grant access to other users.

## SQL Examples

```
-- Make sure Derby classpath variable is correctly set for our examples
CALL SYCS_UTIL.SYCS_SET_GLOBAL_DATABASE_PROPERTY(
    'derby.database.classpath',
    'SPLICE.SAMPLE1_APP:SPLICE.SAMPLE2');

-- install jar from current directory
splice> CALL SQLJ.INSTALL_JAR('tours.jar', 'SPLICE.SAMPLE1_APP', 0);

-- install jar using full path
splice> CALL SQLJ.INSTALL_JAR('c:\myjarfiles\tours.jar', 'SPLICE.SAMPLE1_APP', 0);

-- install jar from remote location
splice> CALL SQLJ.INSTALL_JAR('http://www.example.com/tours.jar', 'SPLICE.SAMPLE2_APP', 0);

-- install jar using a quoted identifier for the
-- Splice Machine jar name
splice> CALL SQLJ.INSTALL_JAR('tours.jar', 'SPLICE."SAMPLE2"', 0);
```

## See Also

- » [SQLJ\\_REMOVE\\_JAR](#)
- » [SQLJ\\_REPLACE\\_JAR](#)

# SYSCS\_UTIL.SYSCS\_INVALIDATE\_STORED\_STATEMENTS

The `SYSCS_UTIL.SYSCS_INVALIDATE_STORED_STATEMENTS` system procedure invalidates all system prepared statements, and forces the query optimizer to create new execution plans. You can use this to speed up query execution by the data dictionary when performance has become sub-optimal.

If you notice that `ij show` commands have slowed down, you can call `SYSCS_UTIL.SYSCS_INVALIDATE_STORED_STATEMENTS` to refresh the execution plans.

**NOTE:** Splice Machine uses prepared statements known as system procedures to access data in the system tables. These procedures are cached, along with their execution plans, in the data dictionary. The cached execution plans can become sub-optimal after you issue a large number of schema-modifying DDL statements, such as defining and/or modifying a number of tables.

## Syntax

```
SYSCS_UTIL.SYSCS_INVALIDATE_STORED_STATEMENTS ()
```

## Results

This procedure does not return a result.

## Example

```
splice> CALL SYSCS_UTIL.SYSCS_INVALIDATE_STORED_STATEMENTS ();  
Statement executed.
```

## See Also

- » [SYSCS\\_UTIL.SYSCS\\_EMPTY\\_GLOBAL\\_STATEMENT\\_CACHE](#)
- » [SYSCS\\_UTIL.SYSCS\\_EMPTY\\_STATEMENT\\_CACHE](#)

# SYSCS\_UTIL.SYSCS\_KILL\_OPERATION

The SYSCS\_UTIL.SYSCS\_KILL\_OPERATION system procedure terminates an operation that is running on the server to which you are currently connected.

You can use the [SYSCS\\_UTIL.SYSCS\\_GET\\_RUNNING\\_OPERATIONS](#) system procedure. to find the UUID for an operation you want to kill.

## Syntax

```
SYSCS_UTIL.SYSCS_KILL_OPERATION(operationId)
```

*operationId*  
The UUID of the operation that you want to terminate.

This is the same UUID that is shown in the Spark console. You can use the [SYSCS\\_UTIL.SYSCS\\_GET\\_RUNNING\\_OPERATIONS](#) system procedure to discover the UUID for the operation.

## Results

This procedure does not return a result.

## Example

```
splice> call SYSCS_UTIL.SYSCS_GET_RUNNING_OPERATIONS();
UUID                                |USER      |HOSTNAME                |SESSION  |SQL
-----
bf610dea-d33e-4304-bf2e-4f10e667aa98 |SPLICE    |localhost:1528          |2         |cal
1 SYSCS_UTIL.SYSCS_GET_RUNNING_OPERATIONS()
33567e3c-ef33-46dc-8d10-5ceb79348c2e |SPLICE    |localhost:1528          |20        |ins
ert into a select * from a

2 rows selected

splice> call SYSCS_UTIL.SYSCS_KILL_OPERATION('33567e3c-ef33-46dc-8d10-5ceb79348c2e');
Statement executed.
```

# SYSCS\_UTIL.MAJOR\_COMPACT\_REGION

The `SYSCS_UTIL.MAJOR_COMPACT_REGION` system procedure performs a major compaction on a table region or an index region.

Region names must be specified in HBase-encoded format. You can retrieve the encoded name for a region by calling the [SYSCS\\_UTIL.GET\\_ENCODED\\_REGION\\_NAME](#) system procedure.

A common reason for calling this procedure is to improve compaction performance by only compacting recent updates in a table. For example, you might confine any updates to regions of the current month, so older regions need not be re-compacted.

## Syntax

```
SYSCS_UTIL.MAJOR_COMPACT_REGION ( VARCHAR schemaName,
                                   VARCHAR tableName,
                                   VARCHAR indexName,
                                   VARCHAR startKey)
```

### *schemaName*

The name of the schema of the table.

### *tableName*

The name of the table to compact.

### *indexName*

`NULL` or the name of the index.

Specify the name of the index you want to compact; if you are compacting the table, specify `NULL` for this parameter.

### *regionName*

The **encoded** HBase name of the region you want compacted. You can call the [SYSCS\\_UTIL.GET\\_ENCODED\\_REGION\\_NAME](#) procedure to look up the region name for an unencoded Splice Machine table or index key.

## Usage

You can compact a table region by specifying `NULL` for the index name. To compact an index region, specify both the table name and the index name.

Region compaction is asynchronous, which means that when you invoke this procedure from the command line, Splice Machine issues a compaction request to HBase, and returns control to you immediately; HBase will determine when to subsequently run the compaction.

## Results

This procedure does not return a result.

## Examples

The following example will perform a major compaction on the region with encoded key value `8ffc80e3f8ac3b180441371319ea90e2` for table `testTable`. The encoded key value is first retrieved by passing the unencoded key value, `1|2`, into the `SYSCS_UTIL.GET_ENCODED_REGION_NAME` procedure:

```
splice> CALL SYSCS_UTIL.GET_ENCODED_REGION_NAME('SPLICE', 'TESTTABLE',
                                                null, '1|2', '|', null, null, null,
null););
```

ENCODED_REGION_NAME	START_KEY	END_KEY
8ffc80e3f8ac3b180441371319ea90e2	\x81\x00\x82	\x81\x00\x84

1 row selected

```
splice> CALL SYSCS_UTIL.COMPACT_REGION('SPLICE', 'testTable',
                                         NULL, '8ffc80e3f8ac3b180441371319ea90e2');
Statement executed.
```

And this example performs a major compaction on the region with encoded index key value `ff8f9e54519a31e15f264ba6d2b828a4` for index `testIndex` on table `testTable`. The encoded key value is first retrieved by passing the unencoded index key value, `1996-04-12|155190|21168.23|0.04`, into the `SYSCS_UTIL.GET_ENCODED_REGION_NAME` procedure:

```
splice> CALL SYSCS_UTIL.GET_ENCODED_REGION_NAME('SPLICE', 'TESTTABLE',
                                                'SHIP_INDEX', '1996-04-12|155190|21168.23|0.04',
                                                '|', null, null, null, null);
```

ENCODED_REGION_NAME	START_KEY	END_KEY
ff8f9e54519a31e15f264ba6d2b828a4	\xEC\xC1\x15\xAD\xCD\x80\x00\xE1\x06\xEE\x00\xE4V&	

1 row selected

```
splice> CALL SYSCS_UTIL.COMPACT_REGION('SPLICE', 'testTable',
                                         'testIndex', 'ff8f9e54519a31e15f264ba6d2b828a4');
Statement executed.
```

## See Also

- » [SYSCS\\_UTIL.COMPACT\\_REGION](#)
- » [SYSCS\\_UTIL.GET\\_ENCODED\\_REGION\\_NAME](#)
- » [SYSCS\\_UTIL.GET\\_REGIONS](#)
- » [SYSCS\\_UTIL.GET\\_START\\_KEY](#)
- » [SYSCS\\_UTIL.MERGE\\_REGIONS](#)

## SYSCS\_UTIL.MERGE\_DATA\_FROM\_FILE

The `SYSCS_UTIL.MERGE_DATA_FROM_FILE` system procedure imports data to update an existing record or create a new record in your database. You can choose to import all or a subset of the columns from the input data into your database using the `insertColumnList` parameter.

After a successful import completes, a simple report displays, showing how many files were imported, and how many record imports succeeded or failed.

### Selecting an Import Procedure

Splice Machine provides four system procedures for importing data:

- » The [SYSCS\\_UTIL.IMPORT\\_DATA](#) procedure imports each input record into a new record in your database.
- » The [SYSCS\\_UTIL.UPSERT\\_DATA\\_FROM\\_FILE](#) procedure updates existing records and adds new records to your database. It only differs from `SYSCS_UTIL.MERGE_DATA_FROM_FILE` in that upserting **overwrites** the generated or default value of a column that *is not specified* in your `insertColumnList` parameter when updating a record.
- » This procedure, `SYSCS_UTIL.MERGE_DATA_FROM_FILE` procedure updates existing records and adds new records to your database. It only differs from `SYSCS_UTIL.UPSERT_DATA_FROM_FILE` in that merging **does not overwrite** the generated or default value of a column that *is not specified* in your `insertColumnList` parameter when updating a record.
- » The [SYSCS\\_BULK\\_IMPORT\\_HFILE](#) procedure 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 has improved performance for large tables; however, the bulk HFile import requires extra work on your part and lacks constraint checking.

Our [Importing Data Tutorial](#) includes a decision tree and brief discussion to help you determine which procedure best meets your needs.

## Syntax

```
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
);
```

## Parameters

The following table summarizes the parameters used by SYSCS\_UTIL.MERGE\_DATA\_FROM\_FILE and other Splice Machine data importation procedures. Each parameter name links to a more detailed description in our [Importing Data Tutorial](#).

Category	Parameter	Description	Example Value
Table Info	<a href="#">schemaName</a>	The name of the schema of the table in which to import.	SPLICE
	<a href="#">tableName</a>	The name of the table in which to import	playerTeams
Data Location	<a href="#">insertColumnList</a>	The names, in single quotes, of the columns to import. If this is null, all columns are imported.	'ID, TEAM'



Category	Parameter	Description	Example Value
	<u>fileOrDirectoryName</u>	<p>Either a single file or a directory. If this is a single file, that file is imported; if this is a directory, all of the files in that directory are imported. You can import compressed or uncompressed files.</p> <div> <p>The SYSCS_UTIL.MERGE_DATA_FROM_FILE procedure only works with single files; <b>you cannot specify a directory name</b> when calling SYSCS_UTIL.MERGE_DATA_FROM_FILE.</p> </div> <p>On a cluster, the files to be imported <b>MUST</b> be on S3, HDFS (or MapR-FS). If you're using our Database Service product, files can only be imported from S3.</p>	<p>/data/mydata/mytable.csv</p> <p>'s3a://splice-benchmark-data/flat/TPCH/100/region'</p>
Data Formats	<u>oneLineRecords</u>	A Boolean value that specifies whether ( <code>true</code> ) each record in the import file is contained in one input line, or ( <code>false</code> ) if a record can span multiple lines.	<code>true</code>
	<u>charset</u>	The character encoding of the import file. The default value is UTF-8.	<code>null</code>
	<u>columnDelimiter</u>	The character used to separate columns, Specify <code>null</code> if using the comma (,) character as your delimiter.	<code>' '</code>
	<u>characterDelimiter</u>	The character is used to delimit strings in the imported data.	<code>'\"'</code>
	<u>timestampFormat</u>	<p>The format of timestamps stored in the file. You can set this to <code>null</code> if there are no time columns in the file, or if the format of any timestamps in the file match the <code>Java.sql.Timestamp</code> default format, which is: "<code>yyyy-MM-dd HH:mm:ss</code>".</p> <div>  <p>All of the timestamps in the file you are importing must use the same format.</p> </div>	<code>'yyyy-MM-dd HH:mm:ss.SSZ'</code>

Category	Parameter	Description	Example Value
	<u>dateFormat</u>	The format of timestamps stored in the file. You can set this to <code>null</code> if there are no date columns in the file, or if the format of any dates in the file match pattern: " <code>yyyy-MM-dd</code> ".	<code>yyyy-MM-dd</code>
	<u>timeFormat</u>	The format of time values stored in the file. You can set this to <code>null</code> if there are no time columns in the file, or if the format of any times in the file match pattern: " <code>HH:mm:ss</code> ".	<code>HH:mm:ss</code>
<b>Problem Logging</b>	<u>badRecordsAllowed</u>	The number of rejected (bad) records that are tolerated before the import fails. If this count of rejected records is reached, the import fails, and any successful record imports are rolled back. Specify 0 to indicate that no bad records are tolerated, and specify -1 to indicate that all bad records should be logged and allowed.	25
	<u>badRecordDirectory</u>	<p>The directory in which bad record information is logged. Splice Machine logs information to the <code>&lt;import_file_name&gt;.bad</code> file in this directory; for example, bad records in an input file named <code>foo.csv</code> would be logged to a file named <code>badRecordDirectory/foo.csv.bad</code>.</p> <p>On a cluster, this directory <b>MUST be on S3, HDFS (or MapR-FS)</b>. If you're using our Database Service product, files can only be imported from S3.</p>	<code>'importErrsDir'</code>
<b>Bulk HFile Import</b>	<u>bulkImportDirectory (outputDirectory)</u>	<p>For <code>SYSCS_UTIL.BULK_IMPORT_HFILE</code>, this is the name of the directory into which the generated HFiles are written prior to being imported into your database.</p> <p>For the <code>SYSCS_UTIL.COMPUTE_SPLIT_KEY</code> procedure, where it is named <code>outputDirectory</code>, this parameter specifies the directory into which the split keys are written.</p>	<code>hdfs:///tmp/test_hfile_import/</code>
	<u>skipSampling</u>	<p>The <code>skipSampling</code> parameter is a Boolean value that specifies how you want the split keys used for the bulk HFile import to be computed. Set to <code>false</code> to have <code>SYSCS_UTIL.BULK_IMPORT_HFILE</code> automatically determine splits for you.</p> <p>This parameter is only used with the <code>SYSCS_UTIL.BULK_IMPORT_HFILE</code> system procedure.</p>	<code>false</code>

## Notes

- » The `SYSCS_UTIL.MERGE_DATA_FROM_FILE` procedure only imports single files; it does not process directories. This means that the `fileOrDirectoryName` parameter value must be a file name.

## Results

`SYSCS_UTIL.MERGE_DATA_FROM_FILE` displays a summary of the import process results that looks like this:

rowsImported	failedRows	files	dataSize	failedLog
94	0	1	4720	NONE

This procedure also logs rejected record activity into `.bad` files in the `badRecordDirectory` directory; one file for each imported file.

## Importing and Updating Records

What distinguishes `SYSCS_UTIL.IMPORT_DATA` from the similar [SYSCS\\_UTIL.UPSERT\\_DATA\\_FROM\\_FILE](#) and [SYSCS\\_UTIL.SYSCS\\_MERGED\\_DATA\\_FROM\\_FILE](#) procedures is how each works with these specific conditions:

- » You are importing only a subset of data from the input data into your table, either because the table contains less columns than does the input file, or because you've specified a subset of the columns in your `insertColumnList` parameter.
- » Inserting and updating data in a column with generated values.
- » Inserting and updating data in a column with default values.
- » Handling of missing values.

The [Importing Data Tutorial: Input Handling](#) topic describes how each of these conditions is handled by the different system procedures.

## Record Import Failure Reasons

When upserting data from a file, the input file you generate must contain:

- » the columns to be changed
- » all `NON_NULL` columns

Typical reasons for a row (record) import to fail include:

- » Improper data expected for a column.
- » Improper number of columns of data.

- » A primary key violation: [SYSCS\\_UTIL.MERGE\\_DATA\\_FROM\\_FILE](#) will only work correctly if the table into which you are inserting/updating has primary keys.

A few important notes:

- » Splice Machine advises you to run a full compaction (with the [SYSCS\\_UTIL.SYSCS\\_PERFORM\\_MAJOR\\_COMPACTION\\_ON\\_TABLE](#) system procedure) after importing large amounts of data into your database.
- » On a cluster, the files to be imported **MUST be on S3, HDFS (or MapR-FS)**, as must the `badRecordDirectory` directory. If you're using our Database Service product, files can only be imported from S3.

In addition, the files must be readable by the `hbase` user, and the `badRecordDirectory` directory must be writable by the `hbase` user, either by setting the user explicitly, or by opening up the permissions; for example:

```
sudo -su hdfs hadoop fs -chmod 777 /badRecordDirectory
```

## Examples

This section presents a couple simple examples.

The [Importing Data Usage Examples](#) topic contains a more extensive set of examples.

### Example 1: Updating our doc examples player data

This example shows the `MERGE_DATA` call used to update the Players in our documentation examples database:

```
splice> CALL SYSCS_UTIL.MERGE_DATA_FROM_FILE('SPLICEBALL', 'Players',
      'ID, Team, Name, Position, DisplayName, BirthDate',
      '/Data/DocExamplesDb/Players.csv',
      null, null, null, null, null, 0, null, true, null);
rowsImported      |failedRows      |files      |dataSize      |failedLo
g-----
---
94                |0               |1          |4720          |NONE
1 row selected
```

### Example 2: Using single quotes to delimit strings

This example uses single quotes instead of double quotes as the character delimiter in the input:

```
1,This field is one line,Able
2,'This field has two lines
This is the second line of the field',Baker
3,This field is also just one line,Charlie
```

Note that you must escape single quotes in SQL, which means that you actually define the character delimiter parameter with four single quotes, as follow

```
SYSCS_UTIL.MERGE_DATA_FROM_FILE('SPLICE','MYTABLE',null,'data.csv','\t','''',null,null,null,0,'BAD', false, null);
```

See [Importing Data Usage Examples](#) for more examples.

## See Also

- » [Our Importing Data Tutorial](#)
- » [Importing Data Usage Examples](#)
- » [SYSCS\\_UTIL.IMPORT\\_DATA](#)
- » [SYSCS\\_UTIL.UPSERT\\_DATA\\_FROM\\_FILE](#)

# SYSCS\_UTIL.MERGE\_REGIONS

The `SYSCS_UTIL.MERGE_REGIONS` system procedure merges two adjacent Splice Machine table regions or two adjacent Splice Machine index regions.

Region names must be specified in HBase-encoded form. You can retrieve the encoded name for a region by calling the [SYSCS\\_UTIL.GET\\_ENCODED\\_REGION\\_NAME](#) system procedure.

You might use this procedure if you want to collect older data into a smaller set of regions to minimize the number of regions required.

## Syntax

```
SYSCS_UTIL.MERGE_REGIONS ( VARCHAR schemaName)
                           VARCHAR tableName,
                           VARCHAR indexName,
                           VARCHAR regionName1,
                           VARCHAR regionName2 )
```

### *schemaName*

The name of the schema of the table.

### *tableName*

The name of the table.

### *indexName*

`NULL` or the name of the index.

Specify the name of the index if you are merging index regions; if you are merging table regions, specify `NULL` for this parameter.

### *regionName1*

The **encoded** HBase name of the first of the two regions you want merged. You can call the [SYSCS\\_UTIL.GET\\_ENCODED\\_REGION\\_NAME](#) procedure to look up the region name for an unencoded Splice Machine table or index key.

### *regionName2*

The **encoded** HBase name of the second of the two regions you want merged. You can call the [SYSCS\\_UTIL.GET\\_ENCODED\\_REGION\\_NAME](#) procedure to look up the region name for an unencoded Splice Machine table or index key.

## Usage

You can merge two adjacent table regions by specifying `NULL` for the index name. To merge two adjacent index regions, specify both the table name and the index name.

## Results

This procedure does not return a result.

If the specified regions are not adjacent, you'll see an error message, and no merging will be performed.

## Examples

The following call will merge two adjacent regions of a table, after you have called `SYSCS_UTIL.GET_ENCODED_REGION_NAME` to retrieve the encoded key values for each region:

```
splice> CALL SYSCS_UTIL.MERGE_REGIONS('SPlice','TESTTABLE', NULL,
                                     'cf0163796bba8666b1183788fc7bc31b',
                                     '4e11260fb5ae106a681574be90709449');

Statement executed.
```

And this call will merge two adjacent regions of an index, after you have called `SYSCS_UTIL.GET_ENCODED_REGION_NAME` to retrieve the encoded key values for each region::

```
splice> CALL SYSCS_UTIL.MERGE_REGIONS('SPlice','TESTTABLE', 'SHIP_INDEX',
                                     '5a59b4a46a8a0a7180a469dbe0b40fad',
                                     '039ba9b2ecdf458b3293bd9e74e88f65');

Statement executed.
```

## See Also

- » [SYSCS\\_UTIL.COMPACT\\_REGION](#)
- » [SYSCS\\_UTIL.GET\\_ENCODED\\_REGION\\_NAME](#)
- » [SYSCS\\_UTIL.GET\\_REGIONS](#)
- » [SYSCS\\_UTIL.GET\\_START\\_KEY](#)
- » [SYSCS\\_UTIL.MAJOR\\_COMPACT\\_REGION](#)

## SYSCS\_UTIL.SYSCS\_MODIFY\_PASSWORD

The `SYSCS_UTIL.SYSCS_MODIFY_PASSWORD` system procedure is called by a user to change that user's own password.

This procedure is used in conjunction with `NATIVE` authentication.

The `derby.authentication.native.passwordLifetimeMillis` property sets the password expiration time, and the `derby.authentication.native.passwordLifetimeThreshold` property sets the time when a user is warned that the password will expire.

### Syntax

```
SYSCS_UTIL.SYSCS_MODIFY_PASSWORD(IN password VARCHAR(32672))
```

*password*

A case-sensitive password.

### Results

This procedure does not return a result.

### Execute Privileges

Any user can execute this procedure.

As of this writing, your administrator must grant a user execute permission on this procedure before that user can successfully modify his or her password.

### JDBC example

```
CallableStatement cs = conn.prepareCall(  
    "CALL SYSCS_UTIL.SYSCS_MODIFY_PASSWORD('baseball!')");  
cs.execute();  
cs.close();
```

### SQL Example

The following example sets the current user's password to `baseball!`:



```
splice> CALL SYSCS_UTIL.SYSCS_MODIFY_PASSWORD('baseball!');  
Statement executed
```

## See Also

» [SYSCS\\_UTIL.SYSCS\\_RESET\\_PASSWORD](#)

## SYSCS\_UTIL.SYSCS\_PEEK\_AT\_SEQUENCE Function

The `SYSCS_UTIL.SYSCS_PEEK_AT_SEQUENCE` function allows users to observe the instantaneous current value of a sequence generator without having to query the [SYSSEQUENCES system table](#).

Querying the `SYSSEQUENCES` table does not actually return the current value; it only returns an upper bound on that value, which is the end of the chunk of sequence values that has been pre-allocated but not yet used.

The `SYSCS_UTIL.SYSCS_PEEK_AT_SEQUENCE` function shows you the very next value that will be returned by a `NEXT VALUE FOR` clause. Users should never directly query the `SYSSEQUENCES` table, because that will cause sequence generator concurrency to slow drastically.

### Syntax

```
BIGINT SYSCS_UTIL.SYSCS_PEEK_AT_SEQUENCE (
    IN SchemaName VARCHAR(128),
    IN SequenceName VARCHAR(128)
)
```

*SchemaName*

The name of the schema.

*SequenceName*

The name of the sequence.

### Results

Returns the next value that will be returned for the sequence.

### Execute Privileges

By default, all users have execute privileges on this function.

### Example

```
splice> VALUES SYSCS_UTIL.SYSCS_PEEK_AT_SEQUENCE('SPLICE', 'PlayerID_seq');
```

### See Also

» [SYSSEQUENCES](#)



# SYSCS\_UTIL.SYSCS\_PERFORM\_MAJOR\_COMPACTION\_ON\_SCHEMA

The `SYSCS_UTIL.SYSCS_PERFORM_MAJOR_COMPACTION_ON_SCHEMA` system procedure performs a major compaction on a schema. The compaction is performed on all of the tables in the schema, and on all of its index and constraint tables for each table in the schema.

## Syntax

```
SYSCS_UTIL.SYSCS_PERFORM_MAJOR_COMPACTION_ON_SCHEMA (schemaName)
```

*schemaName*

A string that specifies the Splice Machine schema name to which the table belongs.

## Usage

Major compaction is synchronous, which means that when you invoke this procedure from the command line, your command line prompt won't be available again until the compaction completes, which can take a little time.

## Results

This procedure does not return a result.

## Example

```
splice> CALL SYSCS_UTIL.SYSCS_PERFORM_MAJOR_COMPACTION_ON_SCHEMA ('SPICE');
Statement executed.
```

## See Also

» [SYSCS\\_UTIL.SYSCS\\_PERFORM\\_MAJOR\\_COMPACTION\\_ON\\_TABLE](#)

# SYSCS\_UTIL.SYSCS\_PERFORM\_MAJOR\_COMPACTION\_ON\_TABLE

The `SYSCS_UTIL.SYSCS_PERFORM_MAJOR_COMPACTION_ON_TABLE` system procedure performs a major compaction on a table. The compaction is performed on the table and on all of its index and constraint tables.

## Syntax

```
SYSCS_UTIL.SYSCS_PERFORM_MAJOR_COMPACTION_ON_TABLE (
    schemaName, tableName)
```

*schemaName*

A string that specifies the Splice Machine schema name to which the table belongs.

*tableName*

A string that specifies name of the Splice Machine table on which to perform the compaction.

## Usage

Major compaction is synchronous, which means that when you invoke this procedure from the command line, your command line prompt won't be available again until the compaction completes, which can take a little time.

## Results

This procedure does not return a result.

## Example

```
splice> CALL SYSCS_UTIL.SYSCS_PERFORM_MAJOR_COMPACTION_ON_TABLE('SPICE','Pitchin
g');
Statement executed.
```

## See Also

» [SYSCS\\_UTIL.SYSCS\\_PERFORM\\_MAJOR\\_COMPACTION\\_ON\\_SCHEMA](#)

# SYSCS\_UTIL.SYSCS\_REFRESH\_EXTERNAL\_TABLE

You call the `SYSCS_UTIL.SYSCS_REFRESH_EXTERNAL_TABLE` system procedure to manually refresh the schema of an external table in Splice Machine that has been modified outside of Spark. 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 `SYSCS_UTIL.SYSCS_REFRESH_EXTERNAL_TABLE`.

## Syntax

```
SYSCS_UTIL.SYSCS_REFRESH_EXTERNAL_TABLE(  
    String schemaName,  
    String tableName )
```

### *schemaName*

Specifies the schema of the table. Passing a `null` or non-existent schema name generates an error.

### *table Name*

The table name.

## Results

This procedure does not return a result.

## Example

This refreshes the schema of the external table named `myTable`:

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

## See Also

- [CREATE EXTERNAL TABLE](#)

# SQLJ.REMOVE\_JAR

The `SQLJ.REMOVE_JAR` system procedure removes a jar file from a database.

**NOTE:** For more information about using JAR files, see the [Using Functions and Stored Procedures](#) section in our *Developer's Guide*.

## Syntax

```
SQLJ.REMOVE_JAR (
    IN qualified_jar_name VARCHAR(32672),
    IN undeploy INTEGER
)
```

### *qualified\_jar\_name*

The Splice Machine name of the jar file, qualified by the schema name. Two examples:

```
MYSCHEMA.Sample1
    -- a delimited identifier.
MYSCHEMA."Sample2"
```

### *undeploy*

If set to 1, this indicates the existence of an SQLJ deployment descriptor file. Splice Machine ignores this argument, so it is normally set to 0.

## Results

This procedure does not return a result.

## Execute Privileges

If authentication and SQL authorization are both enabled, only the database owner has execute privileges on this function by default. The database owner can grant access to other users.

## SQL Example

```
-- SQL statement
CALL SQLJ.REMOVE_JAR('SPLICE.Sample1', 0);
```

## See Also

» [SOLJ\\_INSTALL\\_JAR](#)

» [SOLJ\\_REPLACE\\_JAR](#)



# SQLJ.REPLACE\_JAR

The `SQLJ.REPLACE_JAR` system procedure replaces a jar file in a database.

**NOTE:** For more information about using JAR files, see the [Using Functions and Stored Procedures](#) section in our *Developer's Guide*.

## Syntax

```
SQLJ.REPLACE_JAR (
    IN jar_file_path_or-url VARCHAR(32672),
    IN qualified_jar_name VARCHAR(32672)
)
```

### *jar\_file\_path\_or-url*

The path or URL of the jar file to use as a replacement. A path includes both the directory and the file name (unless the file is in the current directory, in which case the directory is optional). For example:

```
d:/todays_build/tours.jar
```

### *qualified\_jar\_name*

The Splice Machine name of the jar file, qualified by the schema name. Two examples:

```
MYSHEMA.Sample1
-- a delimited identifier.
MYSHEMA."Sample2"
```

## Results

This procedure does not return a result.

## Execute Privileges

If authentication and SQL authorization are both enabled, only the database owner has execute privileges on this function by default. The database owner can grant access to other users.

## SQL Example

```
-- SQL statement
CALL sqlj.replace_jar('c:\myjarfiles\newtours.jar', 'SPICE.Sample1');

-- SQL statement
-- replace jar from remote location
CALL SQLJ.REPLACE_JAR('http://www.example.com/tours.jar', 'SPICE.Sample2');
```

## See Also

» [SQLJ\\_INSTALL\\_JAR](#)

» [SQLJ\\_REMOVE\\_JAR](#)

# SYSCS\_UTIL.SYSCS\_RESET\_PASSWORD

The `SYSCS_UTIL.SYSCS_RESET_PASSWORD` system procedure resets a password for a user whose password has expired or has been forgotten.

This procedure is used in conjunction with NATIVE authentication.

## Syntax

```
SYSCS_UTIL.SYSCS_RESET_PASSWORD(IN userName VARCHAR(128),
                                IN password VARCHAR(32672))
```

### *userName*

A user name that is case-sensitive if you place the name string in double quotes. This user name is an authorization identifier..

### *password*

A case-sensitive password.

## Results

This procedure does not return a result.

## Execute Privileges

If authentication and SQL authorization are both enabled, only the database owner has execute privileges on this function by default. The database owner can grant access to other users.

## JDBC example

Reset the password of a user named FRED:

```
CallableStatement cs = conn.prepareCall
("CALL SYSCS_UTIL.SYSCS_RESET_PASSWORD(?, ?)");
cs.setString(1, "fred");
cs.setString(2, "temppassword");
cs.execute();
cs.close();
```

Reset the password of a user named Fred:

```
CallableStatement cs = conn.prepareCall  
    ("CALL SYSCS_UTIL.SYSCS_RESET_PASSWORD(?, ?)");  
cs.setString(1, "\"Fred\"");  
cs.setString(2, "temppassword");  
cs.execute();  
cs.close();
```

## SQL Example

Reset the password of a user named FRED:

```
splice> CALL SYSCS_UTIL.SYSCS_RESET_PASSWORD('fred', 'temppassword');  
Statement executed.
```

Reset the password of a user named MrBaseball:

```
splice> CALL SYSCS_UTIL.SYSCS_RESET_PASSWORD('MrBaseball', 'baseball!');  
Statement executed.
```

## See Also

» [SYSCS\\_UTIL.SYSCS\\_MODIFY\\_PASSWORD](#)

## SYSCS\_UTIL.SYSCS\_RESTORE\_DATABASE

The `SYSCS_UTIL.SYSCS_RESTORE_DATABASE` system procedure restores your database to the state it was in when a specific backup was performed, using a backup that you previously created using either the [SYSCS\\_UTIL.SYSCS\\_SCHEDULE\\_DAILY\\_BACKUP](#) system procedure.

You can restore your database from any previous full or incremental backup.

There are several important things to know about restoring your database from a previous backup:

- » Restoring a database **wipes out your database** and replaces it with what had been previously backed up.
- » You **cannot use your cluster** while restoring your database.
- » You **must reboot your database** after the restore is complete. See the [Starting Your Database](#) topics in this book for instructions on restarting your database.



When you restore from a backup, Splice Machine automatically determines and runs whatever sequence of restores may be required to accomplish the restoration of your database; this means that when you select an incremental backup from which to restore, Splice Machine will detect that it needs to first restore from the previous full backup and then apply any incremental restorations.

## Syntax

```
SYSCS_UTIL.SYSCS_RESTORE_DATABASE ( VARCHAR backupDir,
                                     BIGINT backupId );
```

### *backupDir*

Specifies the path to the directory containing the backup from which you want to restore your database. This can be a local directory if you're using the standalone version of Splice Machine, or a directory in your cluster's file system (HDFS or MapR-FS).

Relative paths are resolved based on the current user directory. To avoid confusion, we strongly recommend that you use an absolute path when specifying the backup location.

**NOTE:** You must specify the backup's directory when you call this procedure because, if your database has become corrupted and needs to be restored, the data in the `BACKUP.BACKUP` table (which includes the location of each backup) may also be corrupted.

### *backupId*

The ID of the backup job from which you want to restore your database.

The system [Backing Up and Restoring](#) topic for more information.

## Usage

Restoring your database can take a while, and has several major implications:

There are several important things to know about restoring your database from a previous backup:

- » Restoring a database **wipes out your database** and replaces it with what had been previously backed up.
- » You **cannot use your cluster** while restoring your database.
- » You **must reboot your database** after the restore is complete by first [Starting Your Database](#).

As noted at the top of this topic: if you are restoring from an incremental backup, you must first restore from the most recent full backup, and then incrementally restore from each subsequent incremental backup. See [Example 2 below](#).

## Results

This procedure does not return a result.

## Execute Privileges

If authentication and SQL authorization are both enabled, only the database owner has execute privileges on this function by default. The database owner can grant access to other users.

## Examples

The following example first queries the system backup table to find the ID of the backup from which we want to restore, and then initiates the restoration.



Stop using your database before backing up, and keep in mind that restoring a database may take several minutes, depending on the size of your database.

```

splice> SELECT * FROM SYS.SYSBACKUP;
BACKUP_ID |BEGIN_TIMESTAMP      |END_TIMESTAMP          |STATUS  |FILESYSTEM
M         |SCOPE |INCR&|INCREMENTAL_PARENT_&|BACKUP_ITEM
-----
74101     |2015-11-30 17:46:41.431 |2015-11-30 17:46:56.664 |S       |./dbBackup
s/        |D      |true |40975             |30
40975     |2015-11-25 09:32:53.04  |2015-11-25 09:33:09.081 |S       |~/splicemachin
e |D      |false|-1              |93

2 rows selected

splice> CALL SYSCS_UTIL.SYSCS_RESTORE_DATABASE('./dbBackups/', 74101);
Statement executed.

```

Once the restoration is complete, reboot your database by the [Starting Your Database](#).

## See Also

- » [Backing Up and Restoring Databases](#)
- » [SYSCS\\_UTIL.SYSCS\\_BACKUP\\_DATABASE](#)
- » [SYSCS\\_UTIL.SYSCS\\_CANCEL\\_DAILY\\_BACKUP](#)
- » [SYSCS\\_UTIL.SYSCS\\_DELETE\\_BACKUP](#)
- » [SYSCS\\_UTIL.SYSCS\\_DELETE\\_OLD\\_BACKUPS](#)
- » [SYSCS\\_UTIL.SYSCS\\_SCHEDULE\\_DAILY\\_BACKUP](#)
- » [SYSBACKUP](#)
- » [SYSBACKUPITEMS](#)
- » [SYSBACKUPJOBS](#)

# SYSCS\_UTIL.SYSCS\_RESTORE\_SNAPSHOT

The `SYSCS_UTIL.SYSCS_RESTORE_SNAPSHOT` system procedure restores a table or schema to the state it was in at the time the snapshot was created.

**NOTE:** Snapshots include both the data and indexes for tables.

For more information, see the [Using Snapshots](#) topic.

## Syntax

```
SYSCS_UTIL.SYSCS_RESTORE_SNAPSHOT ( VARCHAR(128) snapshotName );
```

*snapshotName*

The name of the snapshot from which you are restoring.

## Results

This procedure does not return a result.

## Example

The following example restores the `mySchema` schema to its state when the named snapshot was created:

```
splice> CALL SYSCS_UTIL.RESTORE_SNAPSHOT( 'snap_myschema_070417a' );  
Statement executed.
```



# SYSCS\_UTIL.SYSCS\_SCHEDULE\_DAILY\_BACKUP

You can use the `SYSCS_UTIL.SYSCS_SCHEDULE_DAILY_BACKUP` to schedule a full or incremental backup job that runs at a specified time each day.

**NOTE:** You specify the scheduled start hour of the backup in Greenwich Mean Time (GMT).

Note that you can subsequently cancel a scheduled backup job with the [Backing Up and Restoring](#) topic.

## Syntax

```
SYSCS_UTIL.SYSCS_SCHEDULE_DAILY_DATABASE (
    VARCHAR backupDir,
    VARCHAR(30) backupType,
    INT startHour
);
```

### *backupDir*

Specifies the path to the directory in which you want the backup stored. This can be a local directory if you're using the standalone version of Splice Machine, or a directory in your cluster's file system (HDFS or MapR-FS).

**NOTE:** You must have permissions set properly to use cloud storage as a backup destination. See [Backing Up to Cloud Storage](#) in the *Administrator's Guide* for information about setting backup permissions properties.

Relative paths are resolved based on the current user directory. To avoid confusion, we strongly recommend that you use an absolute path when specifying the backup destination.

### *backupType*

Specifies the type of backup that you want performed. This must be one of the following values: 'full' or 'incremental'; any other value produces an error and the backup is not run.

Note that if you specify 'incremental', Splice Machine checks the [SYS.SYSBACKUP](#) table to determine if there already is a backup for the system; if not, Splice Machine will perform a full backup, and subsequent backups will be incremental.

### *startHour*

Specifies the hour (0–23) **in GMT** at which you want the backup to run each day. A value less than 0 or greater than 23 produces an error and the backup is not scheduled.

## SQL Examples

The following example schedules a daily incremental backup that runs at 3 am (GMT) and gets stored in the `hdfs://home/backup/` directory:

```
splice> CALL SYSCS_UTIL.SYSCS_SCHEDULE_DAILY_BACKUP('hdfs:///home/backup', 'incremental', 3);
Statement executed;
```

The following example schedules the same backup and stores it on AWS:

```
splice> CALL SYSCS_UTIL.SYSCS_SCHEDULE_DAILY_BACKUP('s3://backup1234', 'incremental', 3);
Statement executed.
```

And this example schedules a daily backup at 6pm (GMT) on a standalone version of Splice Machine:

```
splice> CALL SYSCS_UTIL.SYSCS_SCHEDULE_DAILY_BACKUP('./dbBackups', 'full', 18);
Statement executed.
```

## See Also

- » [\*Backing Up and Restoring Databases\*](#) in the *Administrator's Guide*
- » [SYSCS\\_UTIL.SYSCS\\_BACKUP\\_DATABASE](#) built-in system procedure
- » [SYSCS\\_UTIL.SYSCS\\_CANCEL\\_DAILY\\_BACKUP](#) built-in system procedure
- » [SYSCS\\_UTIL.SYSCS\\_DELETE\\_BACKUP](#) built-in system procedure
- » [SYSCS\\_UTIL.SYSCS\\_DELETE\\_OLD\\_BACKUPS](#) built-in system procedure
- » [SYSCS\\_UTIL.SYSCS\\_RESTORE\\_DATABASE](#) built-in system procedure
- » [SYSBACKUP](#) system table
- » [SYSBACKUPITEMS](#) system table
- » [SYSBACKUPJOBS](#) system table

# SYSCS\_UTIL.SYSCS\_SET\_GLOBAL\_DATABASE\_PROPERTY

Use the `SYSCS_UTIL.SYSCS_SET_GLOBAL_DATABASE_PROPERTY` system procedure to set or delete the value of a property of the database.

## Syntax

```
SYSCS_UTIL.SYSCS_SET_GLOBAL_DATABASE_PROPERTY (
    IN key VARCHAR(128),
    IN value VARCHAR(32672)
)
```

*key*

The property name.

*value*

The new property value. If this is `null`, then the property with key value `key` is deleted from the database property set. If this is not `null`, then this `value` becomes the new value of the property. If this value is not a valid value for the property, Splice Machine uses the default value of the property.

## Results

This procedure does not return a result.

## Execute Privileges

If authentication and SQL authorization are both enabled, only the database owner has execute privileges on this function by default. The database owner can grant access to other users.

## JDBC example

Set the `splicemachine.locks.deadlockTimeout` property to a value of 10:

```
CallableStatement cs = conn.prepareCall
("CALL SYSCS_UTIL.SYSCS_SET_GLOBAL_DATABASE_PROPERTY(?, ?)");
cs.setString(1, "splicemachine.locks.deadlockTimeout");
cs.setString(2, "10");
cs.execute();
cs.close();
```

## SQL Example

Set the `splicemachine.locks.deadlockTimeout` property to a value of 10:

```
splice> CALL SYSCS_UTIL.SYSCS_SET_GLOBAL_DATABASE_PROPERTY( 'splicemachine.locks.dea  
dlockTimeout', '10' );  
Statement executed.
```

## See Also

» [SYSCS\\_UTIL.SYSCS\\_GET\\_GLOBAL\\_DATABASE\\_PROPERTY](#)

# SYSCS\_UTIL.SYSCS\_SET\_LOGGER\_LEVEL

The `SYSCS_UTIL.SYSCS_SET_LOGGER_LEVEL` system procedure changes the logging level of the specified logger.

**NOTE:** You can read more about Splice Machine loggers and logging levels in the [Logging](#) topic.

## Syntax

```
SYSCS_UTIL.SYSCS_SET_LOGGER_LEVEL(loggerName, logLevel)
```

### *loggerName*

A string specifying the name of the logger whose log level you want to find.

### *loggerLevel*

A string specifying the new level to assign to the named logger. This must be one of the following level values, which are described in the [Logging](#) topic:

- » TRACE
- » DEBUG
- » INFO
- » WARN
- » ERROR
- » FATAL

## Results

This procedure does not return a result.

## Usage Notes

You can use the `TRACE` option of the Splice Machine `StatementManager` log to record the execution time of each statement:

```
splice> CALL SYSCS_UTIL.SYSCS_SET_LOGGER_LEVEL ( 'com.splicemachine.utils.SpliceUtil
ities', 'TRACE');
Statement executed
```

You can find all of the available loggers by using the [SYSCS\\_UTIL.SYSCS\\_GET\\_LOGGERS](#) system procedure.

## Example

```
splice> CALL SYSCS_UTIL.SYSCS_SET_LOGGER_LEVEL( 'com.splicemachine.mrio.api', 'DEBUG' );  
Statement executed.
```

## See Also

- » [SYSCS\\_UTIL.SYSCS\\_GET\\_LOGGERS](#)
- » [SYSCS\\_UTIL.SYSCS\\_SET\\_LOGGER\\_LEVEL](#)
- » [Splice Machine Logging](#)

# SYSCS\_UTIL.SET\_PURGE\_DELETED\_ROWS

The `SYSCS_UTIL.SET_PURGE_DELETED_ROWS` system procedure enables (or disables) physical deletion of logically deleted rows from a specific table.

## Syntax

```
SYSCS_UTIL.SET_PURGE_DELETED_ROWS ( VARCHAR schema,  
                                     VARCHAR table,  
                                     VARCHAR enable );
```

*schema*

The name of the schema.

*table*

The name of the table

*enable*

A Boolean specifying whether or not to physically delete rows that have been logically deleted during major compaction.

## Results

This procedure does not return a result.

## Execute Privileges

If authentication and SQL authorization are both enabled, only the database owner has execute privileges on this function by default. The database owner can grant access to other users.

## Example

This specifies that deleted rows from `my_table` will be physically deleted when the next major compaction is run:

```
CALL SYSCS_UTIL.SET_PURGE_DELETED_ROWS('SPLICE', 'my_table', true);
```

## SYSCS\_UTIL.SYSCS\_SNAPSHOT\_SCHEMA

The `SYSCS_UTIL.SYSCS_SNAPSHOT_SCHEMA` system procedure creates a Splice Machine snapshot of the specified schema. These snapshots can subsequently be used to restore the schema to its state at the time that a snapshot was created.

**NOTE:** Snapshots include both the data and indexes for tables.

For more information, see the [Using Snapshots](#) topic.

### Syntax

```
SYSCS_UTIL.SYSCS_SNAPSHOT_SCHEMA ( VARCHAR(128)  schemaName,
                                     VARCHAR(128)  snapshotName );
```

#### *schemaName*

The name of the schema for which you are creating a snapshot.

#### *snapshotName*

The name that you are assigning to this snapshot, which you can subsequently use to restore or delete the snapshot.

### Results

This procedure does not return a result.



Creating a schema snapshot can require several minutes or more to complete, depending on the size of the schema.

### Example

The following example creates a snapshot of the schema named `mySchema`:

```
splice> CALL SYSCS_UTIL.SNAPSHOT_SCHEMA('mySchema', 'snap_myschema_070417a');
Statement executed.
```



## SYSCS\_UTIL.SYSCS\_SNAPSHOT\_TABLE

The `SYSCS_UTIL.SYSCS_SNAPSHOT_TABLE` system procedure creates a Splice Machine snapshot of the specified table. These snapshots can subsequently be used to restore the table to its state at the time that a snapshot was created.

**NOTE:** Snapshots include both the data and indexes for tables.

For more information, see the [Using Snapshots](#) topic.

### Syntax

```
SYSCS_UTIL.SYSCS_SNAPSHOT_TABLE ( VARCHAR(128) schemaName,
                                   VARCHAR(128) tableName,
                                   VARCHAR(128) snapshotName );
```

#### *schemaName*

The name of the table's schema.

#### *tableName*

The name of the table for which you are creating a snapshot.

#### *snapshotName*

The name that you are assigning to this snapshot, which you can subsequently use to restore or delete the snapshot.

### Results

This procedure does not return a result.



Creating a table snapshot can require several minutes or more to complete, depending on the size of the table.

### Example

The following example creates a snapshot of the table named `myTable`:

```
splice> CALL SYSCS_UTIL.SNAPSHOT_SCHEMA('mySchema', 'myTable', 'snap_myschema_070417a');
Statement executed.
```

# SYSCS\_UTIL.SYSCS\_SPLIT\_TABLE\_OR\_INDEX

The SYSCS\_UTIL.SYSCS\_SPLIT\_TABLE\_OR\_INDEX system procedure computes the split keys for a table or index, prior to importing that table in HFile format. You must use this procedure in conjunction with the [SYSCS\\_UTIL.BULK\\_IMPORT\\_HFILE](#) system procedure to import your data in HFile format.

## Syntax

```
call SYSCS_UTIL.SYSCS_SPLIT_TABLE_OR_INDEX (
  schemaName,
  tableName,
  indexName,
  columnList | null,
  fileName,
  columnDelimiter | null,
  characterDelimiter | null,
  timestampFormat | null,
  dateFormat | null,
  timeFormat | null,
  maxBadRecords,
  badRecordDirectory | null,
  oneLineRecords | null,
  charset | null,
);
```

## Parameters

The following table summarizes the parameters used by SYSCS\_UTIL.SYSCS\_SPLIT\_TABLE\_OR\_INDEX and other Splice Machine data importation procedures. Each parameter name links to a more detailed description in our [Importing Data Tutorial](#).



The parameter values that you pass into this procedure should match the values that you use when you subsequently call the [SYSCS\\_UTIL.BULK\\_IMPORT\\_HFILE](#) procedure to perform the import.

Category	Parameter	Description	Example Value
Table Info	<a href="#">schemaName</a>	The name of the schema of the table in which to import.	SPLICE
	<a href="#">tableName</a>	The name of the table in which to import	playerTeams
Data Location	<a href="#">insertColumnList</a>	The names, in single quotes, of the columns to import. If this is null, all columns are imported.	'ID, TEAM'

Category	Parameter	Description	Example Value
	<u>fileOrDirectoryName</u>	<p>Either a single file or a directory. If this is a single file, that file is imported; if this is a directory, all of the files in that directory are imported. You can import compressed or uncompressed files.</p> <div> <p>The SYSCS_UTIL.MERGE_DATA_FROM_FILE procedure only works with single files; <b>you cannot specify a directory name</b> when calling SYSCS_UTIL.MERGE_DATA_FROM_FILE.</p> </div> <p>On a cluster, the files to be imported <b>MUST</b> be on S3, HDFS (or MapR-FS). If you're using our Database Service product, files can only be imported from S3.</p>	<p>/data/mydata/mytable.csv</p> <p>'s3a://splice-benchmark-data/flat/TPCH/100/region'</p>
Data Formats	<u>oneLineRecords</u>	A Boolean value that specifies whether ( <code>true</code> ) each record in the import file is contained in one input line, or ( <code>false</code> ) if a record can span multiple lines.	<code>true</code>
	<u>charset</u>	The character encoding of the import file. The default value is UTF-8.	<code>null</code>
	<u>columnDelimiter</u>	The character used to separate columns, Specify <code>null</code> if using the comma ( , ) character as your delimiter.	<code>' '</code>
	<u>characterDelimiter</u>	The character is used to delimit strings in the imported data.	<code>'\"'</code>
	<u>timestampFormat</u>	<p>The format of timestamps stored in the file. You can set this to <code>null</code> if there are no time columns in the file, or if the format of any timestamps in the file match the <code>Java.sql.Timestamp</code> default format, which is: "<code>yyyy-MM-dd HH:mm:ss</code>".</p> <div>  <p>All of the timestamps in the file you are importing must use the same format.</p> </div>	<code>'yyyy-MM-dd HH:mm:ss.SSZ'</code>

Category	Parameter	Description	Example Value
	<u>dateFormat</u>	The format of timestamps stored in the file. You can set this to <code>null</code> if there are no date columns in the file, or if the format of any dates in the file match pattern: " <code>yyyy-MM-dd</code> ".	<code>yyyy-MM-dd</code>
	<u>timeFormat</u>	The format of time values stored in the file. You can set this to <code>null</code> if there are no time columns in the file, or if the format of any times in the file match pattern: " <code>HH:mm:ss</code> ".	<code>HH:mm:ss</code>
<b>Problem Logging</b>	<u>badRecordsAllowed</u>	The number of rejected (bad) records that are tolerated before the import fails. If this count of rejected records is reached, the import fails, and any successful record imports are rolled back. Specify 0 to indicate that no bad records are tolerated, and specify -1 to indicate that all bad records should be logged and allowed.	25
	<u>badRecordDirectory</u>	<p>The directory in which bad record information is logged. Splice Machine logs information to the <code>&lt;import_file_name&gt;.bad</code> file in this directory; for example, bad records in an input file named <code>foo.csv</code> would be logged to a file named <code>badRecordDirectory/foo.csv.bad</code>.</p> <p>On a cluster, this directory <b>MUST be on S3, HDFS (or MapR-FS)</b>. If you're using our Database Service product, files can only be imported from S3.</p>	<code>'importErrsDir'</code>
<b>Bulk HFile Import</b>	<u>bulkImportDirectory (outputDirectory)</u>	<p>For <code>SYSCS_UTIL.BULK_IMPORT_HFILE</code>, this is the name of the directory into which the generated HFiles are written prior to being imported into your database.</p> <p>For the <code>SYSCS_UTIL.COMPUTE_SPLIT_KEY</code> procedure, where it is named <code>outputDirectory</code>, this parameter specifies the directory into which the split keys are written.</p>	<code>hdfs:///tmp/test_hfile_import/</code>
	<u>skipSampling</u>	<p>The <code>skipSampling</code> parameter is a Boolean value that specifies how you want the split keys used for the bulk HFile import to be computed. Set to <code>false</code> to have <code>SYSCS_UTIL.BULK_IMPORT_HFILE</code> automatically determine splits for you.</p> <p>This parameter is only used with the <code>SYSCS_UTIL.BULK_IMPORT_HFILE</code> system procedure.</p>	<code>false</code>

## Usage

The [SYSCS\\_UTIL.BULK\\_IMPORT\\_HFILE](#) procedure needs the data that you're importing split into multiple HFiles before it actually imports the data into your database. You can achieve these splits in three ways:

- » You can call `SYSCS_UTIL.BULK_IMPORT_HFILE` with the `skipSampling` parameter to `false`. `SYSCS_UTIL.BULK_IMPORT_HFILE` samples the data to determine the splits, then splits the data into multiple HFiles, and then imports the data.
- » You can split the data into HFiles with this procedure, `SYSCS_UTIL.SYSCS_SPLIT_TABLE_OR_INDEX`, which both computes the keys and performs the splits. You then call `SYSCS_UTIL.BULK_IMPORT_HFILE` with the `skipSampling` parameter to `true` to import your data.
- » You can split the data into HFiles by first calling the [SYSCS\\_UTIL.COMPUTE\\_SPLIT\\_KEY](#) procedure and then calling the [SYSCS\\_UTIL.SYSCS\\_SPLIT\\_TABLE\\_OR\\_INDEX\\_AT\\_POINTS](#) procedure to split the table or index. You then call `SYSCS_UTIL.BULK_IMPORT_HFILE` with the `skipSampling` parameter to `true` to import your data.

In all cases, `SYSCS_UTIL.BULK_IMPORT_HFILE` automatically deletes the HFiles after the import process has completed.

The [Bulk HFile Import Examples](#) section of our *Importing Data Tutorial* describes how these methods differ and provides examples of using them to import data.

## Examples

The [Importing Data: Bulk HFile Examples](#) topic walks you through several examples of importing data with bulk HFiles.

## See Also

- » [SYSCS\\_UTIL.BULK\\_IMPORT\\_HFILE](#)
- » [SYSCS\\_UTIL.COMPUTE\\_SPLIT\\_KEY](#)
- » [SYSCS\\_UTIL.SYSCS\\_SPLIT\\_TABLE\\_OR\\_INDEX\\_AT\\_POINTS](#)

## SYSCS\_UTIL.SYSCS\_SPLIT\_TABLE\_OR\_INDEX\_AT\_POINTS

Before using this procedure, `SYSCS_UTIL.SYSCS_SPLIT_TABLE_OR_INDEX_AT_POINTS`, you must first call the [SYSCS\\_UTIL.COMPUTE\\_SPLIT\\_KEY](#) procedure to compute the split points for the data you're importing. After computing the split keys, use this procedure to split the data into HFiles, and then call [SYSCS\\_UTIL.BULK\\_IMPORT\\_HFILE](#) system procedure to import your data in HFile format.

### Syntax

```
SYSCS_UTIL.SYSCS_SPLIT_TABLE_OR_INDEX_AT_POINTS (
    schemaName,
    tableName,
    indexName,
    splitPoints
);
```

#### *schemaName*

The name of the schema of the table or index that you are splitting.

#### *tableName*

The name of the table you are splitting.

#### *indexName*

The name of the index that you are splitting. If this is null, the specified table is split; if this is non-null, the index is split instead.

#### *splitPoints*

A comma-separated list of split points for the table or index.

This is the list of split points computed by a previous call to the [SYSCS\\_UTIL.COMPUTE\\_SPLIT\\_KEY](#) procedure.

### Usage

The [SYSCS\\_UTIL.BULK\\_IMPORT\\_HFILE](#) procedure needs the data that you're importing split into multiple HFiles before it actually imports the data into your database. You can achieve these splits in three ways:

- » You can call `SYSCS_UTIL.BULK_IMPORT_HFILE` with the `skipSampling` parameter to `false`. `SYSCS_UTIL.BULK_IMPORT_HFILE` samples the data to determine the splits, then splits the data into multiple HFiles, and then imports the data.
- » You can split the data into HFiles with the [SYSCS\\_UTIL.SYSCS\\_SPLIT\\_TABLE\\_OR\\_INDEX](#) procedure, which both computes the keys and performs the splits. You then call `SYSCS_UTIL.BULK_IMPORT_HFILE` with the `skipSampling` parameter to `true` to import your data.

- » You can split the data into HFiles by first calling the [SYSCS\\_UTIL.COMPUTE\\_SPLIT\\_KEY](#) procedure to compute the split points, and then call this procedure, `SYSCS_UTIL.SYSCS_SPLIT_TABLE_OR_INDEX_AT_POINTS` procedure to split the table or index. You then call `SYSCS_UTIL.BULK_IMPORT_HFILE` with the `skipSampling` parameter to `true` to import your data.

In all cases, `SYSCS_UTIL.BULK_IMPORT_HFILE` automatically deletes the HFiles after the import process has completed.

The [Bulk HFile Import Examples](#) section of our *Importing Data Tutorial* describes how these methods differ and provides examples of using them to import data.

## See Also

- » [SYSCS\\_UTIL.BULK\\_IMPORT\\_HFILE](#)
- » [SYSCS\\_UTIL.COMPUTE\\_SPLIT\\_KEY](#)
- » [SYSCS\\_UTIL.SYSCS\\_SPLIT\\_TABLE\\_OR\\_INDEX](#)

# SYSCS\_UTIL.SYSCS\_UPDATE\_ALL\_SYSTEM\_PROCEDURES

The `SYSCS_UTIL.SYSCS_UPDATE_ALL_SYSTEM_PROCEDURES` system procedure updates the signatures of all of the system procedures in a database.

You need to call this procedure when you update to a new version of Splice Machine that includes new or updates system procedure signatures.

## About System Procedures

Splice Machine uses prepared statements known as *system procedures* to access data in the system tables. Each system procedure has two parts:

- » An *implementation*, which is compiled Java byte code that is stored in the Splice jar and is included in the CLASSPATH of the Splice server.
- » A *declaration* (or *signature*), which is a CREATE PROCEDURE statement that is stored in the Splice jar file and is synchronized with the data dictionary (in the SYSALIASES table).

The `SYSALIASES` table is synchronized with a database when the database is first created. Thereafter, when you make changes to the system procedures, you need to call a function to keep the `SYSALIASES` table synchronized with the procedures in the Splice jar file.

If you've modified, deleted, or added a system procedure, call the [SYSCS\\_UTIL.SYSCS\\_UPDATE\\_SYSTEM\\_PROCEDURE](#) function, which drops the procedure from the data dictionary, and updates the dictionary with the new version in the Splice jar file.

If you've made multiple modifications to the system procedures, you can call this function, `SYSCS_UTIL.SYSCS_UPDATE_ALL_SYSTEM_PROCEDURES`, to update all of the stored declarations for a database in the data dictionary. This function drops all of the system procedures from the data dictionary and then recreates the system procedures stored in the dictionary from the definitions in the Splice jar file.

## Results

This procedure does not return a result.

## Syntax

```
SYSCS_UTIL.SYSCS_UPDATE_ALL_SYSTEM_PROCEDURES (schemaName)
```

*schemaName*

A string specifying the name of the schema that needs to be updated in the data dictionary.



## Example

```
splice> call SYSCS_UTIL.SYSCS_UPDATE_ALL_SYSTEM_PROCEduRES('SYSCS_UTIL');  
Statement executed.
```

## See Also

» [SYSCS\\_UTIL.SYSCS\\_UPDATE\\_SYSTEM\\_PROCEDURE](#)

# SYSCS\_UTIL.SYSCS\_UPDATE\_METADATA\_STORED\_STATEMENTS

The `SYSCS_UPDATE_METADATA_STORED_STATEMENTS` system procedure updates the execution plan for stored procedures in your database.

## About System Procedures and Metadata

Splice Machine uses prepared statements known as system procedures to access data in the system tables. These procedures are cached, along with their execution plans, in the data dictionary. The cached execution plans can become sub-optimal after you issue a large number of schema-modifying DDL statements, such as defining and/or modifying a number of tables.

You typically need to call this procedure (along with the [SYSCS\\_UTIL.SYSCS\\_EMPTY\\_STATEMENT\\_CACHE](#) procedure) whenever you update your Splice Machine software installation.

If you have called the [SYSCS\\_UTIL.SYSCS\\_INVALIDATE\\_STORED\\_STATEMENTS](#) system procedure to improve query speed, and performance is still sub-optimal, it is probably because the query optimizer needs some manual hints to generate an optimal execution plan.

The manual hints are stored in the `metadata.properties` file, which is external to the database. Versions of this file are typically supplied by Splice Machine consultants or engineers.

Use this function to update the execution plans stored in the data dictionary.

## Syntax

```
SYSCS_UPDATE_METADATA_STORED_STATEMENTS ()
```

## Results

This procedure does not return a result.

## Example

```
splice> CALL SYSCS_UPDATE_METADATA_STORED_STATEMENTS ();
Statement executed.
```

## See Also

» [SYSCS\\_UTIL.SYSCS\\_EMPTY\\_STATEMENT\\_CACHE](#)

# SYSCS\_UTIL.SYSCS\_UPDATE\_SCHEMA\_OWNER

The `SYSCS_UTIL.SYSCS_UPDATE_SCHEMA_OWNER` system procedure changes the owner of a schema.

## Syntax

```
SYSCS_UTIL.SYSCS_UPDATE_SCHEMA_OWNER(  
    schemaName VARCHAR(128),  
    userName VARCHAR(128))
```

*schemaName*

Specifies the name of the schema..

*userName*

Specifies the user ID in the Splice Machine database.

## Results

This procedure does not return a result.

## Execute Privileges

If authentication and SQL authorization are both enabled, only the database owner has execute privileges on this function by default. The database owner can grant access to other users.

## Example

```
splice> CALL SYSCS_UTIL.SYSCS_UPDATE_SCHEMA_OWNER( 'SPICEBALL', 'Walt');  
Statement executed.
```

# SYSCS\_UTIL.SYSCS\_UPDATE\_SYSTEM\_PROCEDURE

The `SYSCS_UTIL.SYSCS_UPDATE_SYSTEM_PROCEDURE` system procedure updates the stored declaration of a specific system procedure in the data dictionary. Call this procedure after adding a new system procedure or modifying an existing system procedure.

## About System Procedures

Splice Machine uses prepared statements known as *system procedures* to access data in the system tables. Each system procedure has two parts:

- » An *implementation*, which is compiled Java byte code that is stored in the Splice jar and is included in the CLASSPATH of the Splice server.
- » A *declaration* (or *signature*), which is a `CREATE PROCEDURE` statement that is stored in the Splice jar file and is synchronized with the data dictionary (in the `SYSALIASES` table).

The `SYSALIASES` table is synchronized with a database when the database is first created. Thereafter, when you make changes to the system procedures, you need to call a function to keep the `SYSALIASES` table synchronized with the procedures in the Splice jar file.

If you've modified, deleted, or added a system procedure, call this function, `SYSCS_UTIL.SYSCS_UPDATE_SYSTEM_PROCEDURE`, which drops the procedure from the data dictionary, and updates the dictionary with the new version in the Splice jar file.

## Syntax

```
SYSCS_UTIL.SYSCS_UPDATE_SYSTEM_PROCEDURE (schemaName, procName)
```

*schemaName*

A string specifying the name of the schema that needs to be updated in the data dictionary.

*procName*

A string specifying the name of the system procedure whose declaration needs to be updated in the data dictionary.

## Results

This procedure does not return a result.

## Example

```
splice> CALL SYSCS_UTIL.SYSCS_UPDATE_SYSTEM_PROCEDURE ('SYSCS_UTIL', 'IMPORT_DATA');
Statement executed.
```

## See Also

» [SYSCS\\_UTIL.SYSCS\\_UPDATE\\_ALL\\_SYSTEM\\_PROCEEDURES](#)

## SYSCS\_UTIL.UPSERT\_DATA\_FROM\_FILE

The `SYSCS_UTIL.UPSERT_DATA_FROM_FILE` system procedure imports data to update an existing record or create a new record in your database. You can choose to import all or a subset of the columns from the input data into your database using the `insertColumnList` parameter.

After a successful import completes, a simple report displays, showing how many files were imported, and how many record imports succeeded or failed.

### Selecting an Import Procedure

Splice Machine provides four system procedures for importing data:

- » The [SYSCS\\_UTIL.IMPORT\\_DATA](#) procedure imports each input record into a new record in your database.
- » This procedure, `SYSCS_UTIL.UPSERT_DATA_FROM_FILE`, updates existing records and adds new records to your database. It only differs from `SYSCS_UTIL.MERGE_DATA_FROM_FILE` in that upserting **overwrites** the generated or default value of a column that *is not specified* in your `insertColumnList` parameter when updating a record.
- » The [SYSCS\\_UTIL.MERGE\\_DATA\\_FROM\\_FILE](#) procedure updates existing records and adds new records to your database. It only differs from `SYSCS_UTIL.UPSERT_DATA_FROM_FILE` in that merging **does not overwrite** the generated or default value of a column that *is not specified* in your `insertColumnList` parameter when updating a record.
- » The [SYSCS\\_BULK\\_IMPORT\\_HFILE](#) procedure 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 has improved performance for large tables; however, the bulk HFile import requires extra work on your part and lacks constraint checking.

Our [Importing Data Tutorial](#) includes a decision tree and brief discussion to help you determine which procedure best meets your needs.

## Syntax

```
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
);
```

## Parameters

The following table summarizes the parameters used by SYSCS\_UTIL.UPSERT\_DATA\_FROM\_FILE and other Splice Machine data importation procedures. Each parameter name links to a more detailed description in our [Importing Data Tutorial](#).

Category	Parameter	Description	Example Value
Table Info	<a href="#">schemaName</a>	The name of the schema of the table in which to import.	SPLICE
	<a href="#">tableName</a>	The name of the table in which to import	playerTeams
Data Location	<a href="#">insertColumnList</a>	The names, in single quotes, of the columns to import. If this is null, all columns are imported.	'ID, TEAM'

Category	Parameter	Description	Example Value
	<u>fileOrDirectoryName</u>	<p>Either a single file or a directory. If this is a single file, that file is imported; if this is a directory, all of the files in that directory are imported. You can import compressed or uncompressed files.</p> <div> <p>The SYSCS_UTIL.MERGE_DATA_FROM_FILE procedure only works with single files; <b>you cannot specify a directory name</b> when calling SYSCS_UTIL.MERGE_DATA_FROM_FILE.</p> </div> <p>On a cluster, the files to be imported <b>MUST</b> be on S3, HDFS (or MapR-FS). If you're using our Database Service product, files can only be imported from S3.</p>	<p>/data/mydata/mytable.csv</p> <p>'s3a://splice-benchmark-data/flat/TPCH/100/region'</p>
Data Formats	<u>oneLineRecords</u>	A Boolean value that specifies whether ( <code>true</code> ) each record in the import file is contained in one input line, or ( <code>false</code> ) if a record can span multiple lines.	<code>true</code>
	<u>charset</u>	The character encoding of the import file. The default value is UTF-8.	<code>null</code>
	<u>columnDelimiter</u>	The character used to separate columns, Specify <code>null</code> if using the comma (,) character as your delimiter.	<code>' '</code>
	<u>characterDelimiter</u>	The character is used to delimit strings in the imported data.	<code>'\"'</code>
	<u>timestampFormat</u>	<p>The format of timestamps stored in the file. You can set this to <code>null</code> if there are no time columns in the file, or if the format of any timestamps in the file match the <code>Java.sql.Timestamp</code> default format, which is: "<code>yyyy-MM-dd HH:mm:ss</code>".</p> <div>  <p>All of the timestamps in the file you are importing must use the same format.</p> </div>	<code>'yyyy-MM-dd HH:mm:ss.SSZ'</code>



Category	Parameter	Description	Example Value
	<u>dateFormat</u>	The format of timestamps stored in the file. You can set this to <code>null</code> if there are no date columns in the file, or if the format of any dates in the file match pattern: " <code>yyyy-MM-dd</code> ".	<code>yyyy-MM-dd</code>
	<u>timeFormat</u>	The format of time values stored in the file. You can set this to <code>null</code> if there are no time columns in the file, or if the format of any times in the file match pattern: " <code>HH:mm:ss</code> ".	<code>HH:mm:ss</code>
<b>Problem Logging</b>	<u>badRecordsAllowed</u>	The number of rejected (bad) records that are tolerated before the import fails. If this count of rejected records is reached, the import fails, and any successful record imports are rolled back. Specify 0 to indicate that no bad records are tolerated, and specify -1 to indicate that all bad records should be logged and allowed.	25
	<u>badRecordDirectory</u>	<p>The directory in which bad record information is logged. Splice Machine logs information to the <code>&lt;import_file_name&gt;.bad</code> file in this directory; for example, bad records in an input file named <code>foo.csv</code> would be logged to a file named <code>badRecordDirectory/foo.csv.bad</code>.</p> <p>On a cluster, this directory <b>MUST be on S3, HDFS (or MapR-FS)</b>. If you're using our Database Service product, files can only be imported from S3.</p>	<code>'importErrsDir'</code>
<b>Bulk HFile Import</b>	<u>bulkImportDirectory (outputDirectory)</u>	<p>For <code>SYSCS_UTIL.BULK_IMPORT_HFILE</code>, this is the name of the directory into which the generated HFiles are written prior to being imported into your database.</p> <p>For the <code>SYSCS_UTIL.COMPUTE_SPLIT_KEY</code> procedure, where it is named <code>outputDirectory</code>, this parameter specifies the directory into which the split keys are written.</p>	<code>hdfs:///tmp/test_hfile_import/</code>
	<u>skipSampling</u>	<p>The <code>skipSampling</code> parameter is a Boolean value that specifies how you want the split keys used for the bulk HFile import to be computed. Set to <code>false</code> to have <code>SYSCS_UTIL.BULK_IMPORT_HFILE</code> automatically determine splits for you.</p> <p>This parameter is only used with the <code>SYSCS_UTIL.BULK_IMPORT_HFILE</code> system procedure.</p>	<code>false</code>

## Results

`SYSCS_UTIL.UPSERT_DATA_FROM_FILE` displays a summary of the import process results that looks like this:

rowsImported	failedRows	files	dataSize	failedLog
94	0	1	4720	NONE

This procedure also logs rejected record activity into `.bad` files in the `badRecordDirectory` directory; one file for each imported file.

## Importing and Updating Records

What distinguishes `SYSCS_UTIL.IMPORT_DATA` from the similar [SYSCS\\_UTIL.UPSERT\\_DATA\\_FROM\\_FILE](#) and [SYSCS\\_UTIL.SYSCS\\_MERGED\\_DATA\\_FROM\\_FILE](#) procedures is how each works with these specific conditions:

- » You are importing only a subset of data from the input data into your table, either because the table contains less columns than does the input file, or because you've specified a subset of the columns in your `insertColumnList` parameter.
- » Inserting and updating data in a column with generated values.
- » Inserting and updating data in a column with default values.
- » Handling of missing values.

The [Importing Data Tutorial: Input Handling](#) topic describes how each of these conditions is handled by the different system procedures.

## Record Import Failure Reasons

When upserting data from a file, the input file you generate must contain:

- » the columns to be changed
- » all `NON_NULL` columns

Typical reasons for a row (record) import to fail include:

- » Improper data expected for a column.
- » Improper number of columns of data.
- » A primary key violation: [SYSCS\\_UTIL.UPSERT\\_DATA\\_FROM\\_FILE](#) will only work correctly if the table into which you are inserting/updating has primary keys.

A few important notes:

- » Splice Machine advises you to run a full compaction (with the [SYSCS\\_UTIL.SYSCS\\_PERFORM\\_MAJOR\\_COMPACTION\\_ON\\_TABLE](#) system procedure) after importing large amounts of data into your database.

- » On a cluster, the files to be imported **MUST be on S3, HDFS (or MapR-FS)**, as must the `badRecordDirectory` directory. If you're using our Database Service product, files can only be imported from S3.

In addition, the files must be readable by the `hbase` user, and the `badRecordDirectory` directory must be writable by the `hbase` user, either by setting the user explicitly, or by opening up the permissions; for example:

```
sudo -su hdfs hadoop fs -chmod 777 /badRecordDirectory
```

## Examples

This section presents a couple simple examples.

The [Importing Data Usage Examples](#) topic contains a more extensive set of examples.

### Example 1: Updating our doc examples player data

This example shows the `UPSERT_DATA` call used to update the `Players` in our documentation examples database:

```
splice> CALL SYSCS_UTIL.UPSERT_DATA_FROM_FILE('SPICEBALL', 'Players',
        'ID, Team, Name, Position, DisplayName, BirthDate',
        '/Data/DocExamplesDb/Players.csv',
        null, null, null, null, null, 0, null, true, null);
rowsImported      |failedRows      |files      |dataSize      |failedLo
g-----
---
94                |0               |1          |4720          |NONE
1 row selected
```

### Example 2: Importing strings with embedded special characters

This example imports a csv file that includes newline (`Ctrl-M`) characters in some of the input strings. We use the default double-quote as our character delimiter to import data such as the following:

```
1,This field is one line,Able
2,"This field has two lines
This is the second line of the field",Baker
3,This field is also just one line,Charlie
```

We then use the following call to import the data:

```
SYSCS_UTIL.UPSERT_DATA_FROM_FILE( 'SPICE', 'MYTABLE', null, 'data.csv',
                                   '\t', null, null, null, null, 0, 'BAD', false, nul
1 );
```

We can also explicitly specify double quotes (or any other character) as our delimiter character for strings:

```
SYSCS_UTIL.UPSERT_DATA_FROM_FILE( 'SPICE','MYTABLE',null,'data.csv',  
                                   '\t', '"', null, null, null, 0,'BAD', false, nul  
1);
```

See [Importing Data Usage Examples](#) for more examples.

## See Also

- » [Our Importing Data Tutorial](#)
- » [Importing Data Usage Examples](#)
- » [SYSCS\\_UTIL.IMPORT\\_DATA](#)
- » [SYSCS\\_UTIL.MERGE\\_DATA\\_FROM\\_FILE](#)

## SYSCS\_UTIL.VACUUM

The `SYSCS_UTIL.VACUUM` system procedure performs the following clean-up operations:

1. Waits for all previous transactions to complete; at this point, it must be all. If it waits past a certain point, the call terminates, and you will need to run it again.
2. Gets all the conglomerates that are seen in `sys.sysconglomerates` (e.g. `select conglomeratenum from sys.sysconglomerates`).
3. Gets a list of all of the HBase tables.
4. If an HBase table is not in the conglomerates list and is not a system table (`conglomeratenum < 1100` or `1168`), then it is deleted. If this does not occur, check the `splice.log`.

You are ready to go when you see the `Ready to accept connections` message.

If you see an exception, but do not see the `Ready to accept connections` message, please retry the command.

### Syntax

```
SYSCS_UTIL.VACUUM()
```

### Example

```
splice> CALL SYSCS_UTIL.VACUUM();  
Ready to accept connections.
```

# System Tables

This section contains the reference documentation for the Splice Machine SQL Statements, in the following subsections:

- » [Database Backups Tables](#)
- » [Database Objects Information Tables](#)
- » [Database Permissions Tables](#)
- » [Database Statistics Tables](#)
- » [System Information Tables](#)

Since the system tables belong to the `SYS` schema, you must preface any inquiries involving these tables with the `SYS.` prefix.

**NOTE:** You can use the Java `java.sql.DatabaseMetaData` class to learn more about these tables.

## Database Backups Tables

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

These are the System Tables with backups information:

System Table	Description
<a href="#">SYSBACKUP</a>	Information about each run of a backup job that has been run for the database. You can query this table to determine status information about a specific backup job.
<a href="#">SYSBACKUPITEMS</a>	Information about the items backed up for each backup job.
<a href="#">SYSBACKUPJOBS</a>	Information about all backup jobs that have been created for the database.

## Database Objects Tables

These are the System Tables with information about database objects:

System Table	Description
<a href="#">SYSALIASES</a>	Describes the procedures, functions, and user-defined types in the database.

System Table	Description
<a href="#">SYSCHECKS</a>	Describes the check constraints within the current database.
<a href="#">SYSCOLUMNS</a>	Describes the columns within all tables in the current database.
<a href="#">SYSCONSTRAINTS</a>	Describes the information common to all types of constraints within the current database.
<a href="#">SYSDEPENDS</a>	Stores the dependency relationships between persistent objects in the database.
<a href="#">SYSFOREIGNKEYS</a>	Describes the information specific to foreign key constraints in the current database.
<a href="#">SYSKEYS</a>	Describes the specific information for primary key and unique constraints within the current database.
<a href="#">SYSROLES</a>	Stores the roles in the database.
<a href="#">SYSSCHEMAS</a>	Describes the schemas within the current database.
<a href="#">SYSSEQUENCES</a>	Describes the sequence generators in the database.
<a href="#">SYSSNAPSHOTS</a>	Stores metadata for a Splice Machine snapshot.
<a href="#">SYSTABLES</a>	Describes the tables and views within the current database.
<a href="#">SYSTRIGGERS</a>	Describes the triggers defined for the database.
<a href="#">SYSVIEWS</a>	Describes the view definitions within the current database.

## Database Permissions Tables

These are the System Tables with database permissions information:

System Table	Description
<a href="#">SYSCOLPERMS</a>	Stores the column permissions that have been granted but not revoked.
<a href="#">SYSPERMS</a>	Describes the usage permissions for sequence generators and user-defined types.
<a href="#">SYSROUTINEPERMS</a>	Stores the permissions that have been granted to routines.
<a href="#">SYSTABLEPERMS</a>	Stores the table permissions that have been granted but not revoked.

## Database Statistics Tables

These are the System Tables with database statistics information:

System Table	Description
<a href="#">SYSCOLUMNSTATISTICS</a>	Statistics gathered for each column in each table.
<a href="#">SYSTABLESTATISTICS</a>	Describes the statistics for each table within the current database.

## System Information Tables

These are the System Tables with system information:

System Table	Description
<a href="#">SYSCONGLOMERATES</a>	Describes the conglomerates within the current database. A conglomerate is a unit of storage and is either a table or an index.
<a href="#">SYSFILES</a>	Describes jar files stored in the database.
<a href="#">SYSSTATEMENTS</a>	Describes the prepared statements in the database.
<a href="#">SYSUSERS</a>	Stores user credentials when NATIVE authentication is enabled.



## SYSALIASES System Table

The `SYSALIASES` table describes the procedures, functions, and user-defined types in the database.

The following table shows the contents of the `SYSALIASES` system table.

**SYSALIASES system table**

Column Name	Type	Length	Nullable	Contents
ALIASID	CHAR	36	NO	Unique identifier for the alias
ALIAS	VARCHAR	128	NO	Alias (in the case of a user-defined type, the name of the user-defined type)
SCHEMAID	CHAR	36	YES	Reserved for future use
JAVACLASSNAME	LONG VARCHAR	2,147,483,647	NO	The Java class name
ALIASTYPE	CHAR	1	NO	'F' (function), 'P' (procedure), 'A' (user-defined type)
NAMESPACE	CHAR	1	NO	'F' (function), 'P' (procedure), 'A' (user-defined type)
SYSTEMALIAS	BOOLEAN	1	NO	YES (system supplied or built-in alias) NO (alias created by a user)
ALIASINFO	<i>org.apache.Splice Machine. catalog.AliasInfo</i>  This class is not part of the public API.	-1	YES	A Java interface that encapsulates the additional information that is specific to an alias
SPECIFICNAME	VARCHAR	128	NO	System-generated identifier

## See Also

» [About System Tables](#)

## SYSBACKUP System Table

The `SYSBACKUP` table maintains information about each database backup. You can query this table to find the ID of and details about a backup that was run at a specific time.

**SYSBACKUP system table**

Column Name	Type	Length	Nullable	Contents
<code>BACKUP_ID</code>	<code>BIGINT</code>	19	NO	The backup ID
<code>BEGIN_TIMESTAMP</code>	<code>TIMESTAMP</code>	29	NO	The start time of the backup
<code>END_TIMESTAMP</code>	<code>TIMESTAMP</code>	29	YES	The end time of the backup
<code>STATUS</code>	<code>VARCHAR</code>	10	NO	The status of the backup
<code>FILESYSTEM</code>	<code>VARCHAR</code>	32642	NO	The backup destination directory
<code>SCOPE</code>	<code>VARCHAR</code>	10	NO	The scope of the backup: database, schemas, tables, etc. The current allowable values are:  » D for the entire database
<code>INCREMENTAL_BACKUP</code>	<code>BOOLEAN</code>	1	NO	YES for incremental backups, NO for full backups  <b>NOTE:</b> Incremental backups are not yet available.
<code>INCREMENTAL_PARENT_BACKUP_ID</code>	<code>BIGINT</code>	19	YES	For an incremental backup, this is the <code>BACKUP_ID</code> of the previous backup on which this incremental backup is based.  For full backups, this is -1.  <b>NOTE:</b> Incremental backups are not yet available.
<code>BACKUP_ITEM</code>	<code>INTEGER</code>	10	YES	The number of tables that were backed up.

# SYSDBACKUPITEMS System Table

The SYSDBACKUPITEMS table maintains information about each item (table) backed up during a backup.

SYSDBACKUPITEMS system table

Column Name	Type	Length	Nullable	Contents
BACKUP_ID	BIGINT	19	NO	The backup ID.
ITEM	VARCHAR	32642	NO	The name of the item.
BEGIN_TIMESTAMP	TIMESTAMP	29	NO	The start time of backing up this item.
END_TIMESTAMP	TIMESTAMP	29	YES	The end time of backing up this item.
SNAPSHOT_NAME	VARCHAR	32642	NO	The name of the snapshot associated with this item.

# SYSDBACKUPJOBS System Table

The SYSDBACKUPJOBS table maintains information about all backup jobs that have been created for the database.

SYSDBACKUPJOBS system table

Column Name	Type	Length	Nullable	Contents
JOB_ID	BIGINT	19	NO	The ID of this backup job.
FILESYSTEM	VARCHAR	4000	NO	The backup destination directory.
TYPE	VARCHAR	32	NO	The backup type; possible values are: incremental or full.
HOURL_OF_DAY	INTEGER	10	NO	The regularly scheduled start time (in GMT hours) of the backup job if it is a daily backup.
BEGIN_TIMESTAMP	TIMESTAMP	29	NO	When this job was submitted.

# SYSCHECKS System Table

The SYSCHECKS table describes the check constraints within the current database.

The following table shows the contents of the SYSCHECKS system table.

SYSCHECKS system table

Column Name	Type	Length	Nullable	Contents
CONSTRAINTID	CHAR	36	NO	Unique identifier for the constraint
CHECKDEFINITION	LONG VARCHAR	32,700	NO	Text of check constraint definition
REFERENCEDCOLUMNS	<i>com.splicemachine. db.catalog. ReferencedColumns</i> This class is not part of the public API.	-1	NO	Description of the columns referenced by the check constraint

## See Also

» [About System Tables](#)

# SYSCOLPERMS System Table

The SYSCOLPERMS table stores the column permissions that have been granted but not revoked.

All of the permissions for one (GRANTEE, TABLEID, TYPE, GRANTOR) combination are specified in a single row in the SYSCOLPERMS table. The keys for the SYSCOLPERMS table are:

- » Primary key (GRANTEE, TABLEID, TYPE, GRANTOR)
- » Unique key (COLPERMSID)
- » Foreign key (TABLEID references SYS.SYSTABLES)

The following table shows the contents of the SYSCOLPERMS system table.

SYSCOLPERMS system table

Column Name	Type	Length	Nullable	Contents
COLPERMSID	CHAR	36	NO	Used by the dependency manager to track the dependency of a view, trigger, or constraint on the column level permissions
GRANTEE	VARCHAR	128	NO	The authorization ID of the user or role to which the privilege was granted
GRANTOR	VARCHAR	128	NO	The authorization ID of the user who granted the privilege. Privileges can be granted only by the object owner
TABLEID	CHAR	36	NO	The unique identifier for the table on which the permissions have been granted
TYPE	CHAR	1	NO	<div>If the privilege is non-grantable, the valid values are:<ul style="list-style-type: none"><li>» 's' for SELECT</li><li>» 'u' for UPDATE</li><li>» 'r' for REFERENCES</li></ul><div>If the privilege is grantable, the valid values are:<ul style="list-style-type: none"><li>» 'S' for SELECT</li><li>» 'U' for UPDATE</li><li>» 'R' for REFERENCES</li></ul></div></div>

Column Name	Type	Length	Nullable	Contents
COLUMNS	<i>org.apache.Splice Machine. iapi.services.io. FormatableBitSet</i>  This class is not part of the public API.	-1	NO	A list of columns to which the privilege applies

See Also

>> [About System Tables](#)

## SYSCOLUMNS System Table

The `SYSCOLUMNS` table describes the columns within all tables in the current database.

The following table shows the contents of the `SYSCOLUMNS` system table.

**SYSCOLUMNS system table**

Column Name	Type	Length	Nullable	Contents
REFERENCEID	CHAR	36	NO	Identifier for table (join with <code>SYSTABLES.TABLEID</code> )
COLUMNNAME	VARCHAR	128	NO	Column or parameter name
COLUMNNUMBER	INTEGER	10	NO	The position of the column within the table
COLUMNDATATYPE	<i>com.splicemachine.db.catalog.TypeDescriptor</i> This class is not part of the public API.	-1	NO	System type that describes precision, length, scale, nullability, type name, and storage type of data. For a user-defined type, this column can hold a <i>TypeDescriptor</i> that refers to the appropriate type alias in <code>SYS.SYSALIASES</code> .
COLUMNDEFAULT	<i>java.io.Serializable</i>	-1	YES	For tables, describes default value of the column. The <i>toString()</i> method on the object stored in the table returns the text of the default value as specified in the <code>CREATE TABLE</code> or <code>ALTER TABLE</code> statement.
COLUMNDEFAULTID	CHAR	36	YES	Unique identifier for the default value



Column Name	Type	Length	Nullable	Contents
AUTOINCREMENTVALUE	BIGINT	19	YES	What the next value for column will be, if the column is an identity column
AUTOINCREMENTSTART	BIGINT	19	YES	Initial value of column (if specified), if it is an identity column
AUTOINCREMENTINC	BIGINT	19	YES	Amount column value is automatically incremented (if specified), if the column is an identity column
COLLECTSTATS	BOOLEAN	1	YES	Whether or not to collect statistics on the table.

See Also

>> [About System Tables](#)

## SYSCOLUMNSTATISTICS System Table

The SYSCOLUMNSTATISTICS table view describes the statistics for a specific table column within the current database.

**NOTE:** SYS.SYSCOLUMNSTATISTICS is a system view.

The following table shows the contents of the SYSCOLUMNSTATISTICS system table.

**SYSCOLUMNSTATISTICS system table**

Column Name	Type	Length	Nullable	Contents
SCHEMANAME	VARCHAR	32672	YES	The name of the schema.
TABLERNAME	VARCHAR	32672	YES	The name of the table.
COLUMNNAME	VARCHAR	32672	YES	The name of the column.
CARDINALITY	BIGINT	19	YES	The estimated number of distinct values for the column.
NULL_COUNT	BIGINT	19	YES	The number of rows in the table that have NULL for the column.
NULL_FRACTION	REAL	23	YES	The ratio of NULL records to all records:  NULL_COUNT / TOTAL_ROW_COUNT
MIN_VALUE	VARCHAR	32672	YES	The minimum value for the column.
MAX_VALUE	VARCHAR	32672	YES	The maximum value for the column.
QUANTILES	VARCHAR	32672	YES	The quantiles statistics sketch for the column.
FREQUENCIES	VARCHAR	32672	YES	The frequencies statistics sketch for the column.
THETA	VARCHAR	32672	YES	The theta statistics sketch for the column.

The QUANTILES, FREQUENCIES, and THETA values are all sketches computed using the Yahoo Data Sketches library, which you can read about here: <https://datasketches.github.io/>

## See Also

- » [About System Tables](#)
- » [SYSTABLESTATISTICS](#)

# SYSCONGLOMERATES System Table

The SYSCONGLOMERATES table describes the conglomerates within the current database. A conglomerate is a unit of storage and is either a table or an index.

The following table shows the contents of the SYSCONGLOMERATES system table.

SYSCONGLOMERATES system table

Column Name	Type	Length	Nullable	Contents
SCHEMAID	CHAR	36	NO	Schema ID for the conglomerate
TABLEID	CHAR	36	NO	Identifier for table (join with SYSTABLES.TABLEID)
CONGLOMERATENUMBER	BIGINT	19	NO	Conglomerate ID for the conglomerate (heap or index)
CONGLOMERATENAME	VARCHAR	128	YES	Index name, if conglomerate is an index, otherwise the table ID
ISINDEX	BOOLEAN	1	NO	Whether or not conglomerate is an index
DESCRIPTOR	<i>org.apache.splicemachine.catalog.IndexDescriptor</i> This class is not part of the public API.	-1	YES	System type describing the index
ISCONSTRAINT	BOOLEAN	1	YES	Whether or not the conglomerate is a system-generated index enforcing a constraint
CONGLOMERATEID	CHAR	36	NO	Unique identifier for the conglomerate

## See Also

» [About System Tables](#)

## SYSCONSTRAINTS System Table

The `SYSCONSTRAINTS` table describes the information common to all types of constraints within the current database (currently, this includes primary key, unique, and check constraints).

The following table shows the contents of the `SYSCONSTRAINTS` system table.

**SYSCONSTRAINTS system table**

Column Name	Type	Length	Nullable	Contents
CONSTRAINTID	CHAR	36	NO	Unique identifier for constraint
TABLEID	CHAR	36	NO	Identifier for table (join with <code>SYSTABLES.TABLEID</code> )
CONSTRAINTNAME	VARCHAR	128	NO	Constraint name (internally generated if not specified by user)
TYPE	CHAR	1	NO	Possible values: >> 'P' for primary key) >> 'U' for unique) >> 'C' for check)
SCHEMAID	CHAR	36	NO	Identifier for schema that the constraint belongs to (join with <code>SYSSCHEMAS.SCHEMAID</code> )
STATE	CHAR	1	NO	Possible values: >> 'E' for enabled >> 'D' for disabled
REFERENCECOUNT	INTEGER	10	NO	The count of the number of foreign key constraints that reference this constraint; this number can be greater than zero only for <code>PRIMARY KEY</code> and <code>UNIQUE</code> constraints

## See Also

>> [About System Tables](#)

## SYSDEPENDS System Table

The `SYSDEPENDS` table stores the dependency relationships between persistent objects in the database.

Persistent objects can be dependents or providers. Dependents are objects that depend on other objects. Providers are objects that other objects depend on.

- » Dependents are views, constraints, or triggers.
- » Providers are tables, conglomerates, constraints, or privileges.

The following table shows the contents of the `SYSDEPENDS` system table.

**SYSDEPENDS system table**

Column Name	Type	Length	Nullable	Contents
DEPENDENTID	CHAR	36	NO	A unique identifier for the dependent
DEPENDENTFINDER	<i>com.splicemachine.db.catalog.TypeDescriptor</i> This class is not part of the public API.	-1	NO	A system type that describes the view, constraint, or trigger that is the dependent
PROVIDERID	CHAR	36	NO	A unique identifier for the provider
PROVIDERFINDER	<i>com.splicemachine.db.catalog.TypeDescriptor</i> This class is not part of the public API.	-1	NO	A system type that describes the table, conglomerate, constraint, and privilege that is the provider

# SYSFILES System Table

The `SYSFILES` table describes jar files stored in the database.

The following table shows the contents of the `SYSFILES` system table.

SYSFILES system table

Column Name	Type	Length	Nullable	Contents
FILEID	CHAR	36	NO	Unique identifier for the jar file
SCHEMAID	CHAR	36	NO	ID of the jar file's schema (join with <code>SYSSCHEMAS . SCHEMAID</code> )
FILENAME	VARCHAR	128	NO	SQL name of the jar file
GENERATIONID	BIGINT	19	NO	Generation number for the file. When jar files are replaced, their generation identifiers are changed.

## See Also

>> [About System Tables](#)



## SYSFOREIGNKEYS System Table

The `SYSFOREIGNKEYS` table describes the information specific to foreign key constraints in the current database.

Splice Machine generates a backing index for each foreign key constraint. The name of this index is the same as `SYSFOREIGNKEYS.CONGLOMERATEID`.

The following table shows the contents of the `SYSFOREIGNKEYS` system table.

**SYSFOREIGNKEYS system table**

Column Name	Type	Length	Nullable	Contents
CONSTRAINTID	CHAR	36	NO	Unique identifier for the foreign key constraint (join with <code>SYSCONSTRAINTS.CONSTRAINTID</code> )
CONGLOMERATEID	CHAR	36	NO	Unique identifier for index backing up the foreign key constraint (join with <code>SYSCONGLOMERATES.CONGLOMERATEID</code> )
KEYCONSTRAINTID	CHAR	36	NO	Unique identifier for the primary key or unique constraint referenced by this foreign key <code>SYSKEYS.CONSTRAINTID</code> or <code>SYSCONSTRAINTS.CONSTRAINTID</code>
DELETERULE	CHAR	1	NO	Possible values:  'R' for NO ACTION (default)  'S' for RESTRICT  'C' for CASCADE  'U' for SET NULL
UPDATERULE	CHAR	1	NO	Possible values:  'R' for NO ACTION (default)  'S' for RESTRICT

# SYSKEYS System Table

The `SYSKEYS` table describes the specific information for primary key and unique constraints within the current database.

Splice Machine generates an index on the table to back up each such constraint. The index name is the same as `SYSKEYS.CONGLOMERATEID`.

The following table shows the contents of the `SYSKEYS` system table.

SYSKEYS system table

Column Name	Type	Length	Nullable	Contents
CONSTRAINTID	CHAR	36	NO	Unique identifier for constraint
CONGLOMERATEID	CHAR	36	NO	Unique identifier for backing index

## See Also

>> [About System Tables](#)

## SYSPERMS System Table

The `SYSPERMS` table describes the `USAGE` permissions for sequence generators and user-defined types.

The following table shows the contents of the `SYSPERMS` system table.

**SYSPERMS system table**

Column Name	Type	Length	Nullable	Contents
UUID	CHAR	36	NO	The unique ID of the permission. This is the primary key.
OBJECTTYPE	VARCHAR	36	NO	The kind of object receiving the permission. The only valid values are:  >> 'SEQUENCE'  >> 'TYPE'
OBJECTID	CHAR	36	NO	The UUID of the object receiving the permission.  For sequence generators, the only valid values are <code>SEQUENCEIDs</code> in the <code>SYS.SYSSEQUENCES</code> table.  For user-defined types, the only valid values are <code>ALIASIDs</code> in the <code>SYS.SYSALIASES</code> table if the <code>SYSALIASES</code> row describes a user-defined type.
PERMISSION	CHAR	36	NO	The type of the permission. The only valid value is 'USAGE'.
GRANTOR	VARCHAR	128	NO	The authorization ID of the user who granted the privilege. Privileges can be granted only by the object owner.
GRANTEE	VARCHAR	128	NO	The authorization ID of the user or role to which the privilege was granted
ISGRANTABLE	CHAR	1	NO	If the <code>GRANTEE</code> is the owner of the sequence generator or user-defined type, this value is 'Y'.  If the <code>GRANTEE</code> is not the owner of the sequence generator or user-defined type, this value is 'N'.

## SYSROLES System Table

The `SYSROLES` table stores the roles in the database.

A row in the `SYSROLES` table represents one of the following:

- » A role definition (the result of a [CREATE ROLE statement](#))
- » A role grant

The keys for the `SYSROLES` table are:

- » Primary key (`GRANTEE`, `ROLEID`, `GRANTOR`)
- » Unique key (`UUID`)

The following table shows the contents of the `SYSROLES` system table.

**SYSROLES system table**

Column Name	Type	Length	Nullable	Contents
UUID	CHAR	36	NO	A unique identifier for this role
ROLEID	VARCHAR	128	NO	The role name, after conversion to case normal form
GRANTEE	VARCHAR	128	NO	If the row represents a role grant, this is the authorization identifier of a user or role to which this role is granted. If the row represents a role definition, this is the database owner's user name.
GRANTOR	VARCHAR	128	NO	This is the authorization identifier of the user that granted this role. If the row represents a role definition, this is the authorization identifier <code>__SYSTEM</code> . If the row represents a role grant, this is the database owner's user name (since only the database owner can create and grant roles).
WITHADMINOPTION	CHAR	1	NO	A role definition is modelled as a grant from <code>__SYSTEM</code> to the database owner, so if the row represents a role definition, the value is always 'Y'.  This means that the creator (the database owner) is always allowed to grant the newly created role. Currently roles cannot be granted <code>WITH ADMIN OPTION</code> , so if the row represents a role grant, the value is always 'N'.
ISDEF	CHAR	1	NO	If the row represents a role definition, this value is 'Y'. If the row represents a role grant, the value is 'N'.

## See Also

- » [About System Tables](#)
- » [CURRENT\\_ROLE](#) function
- » [CREATE\\_ROLE](#) statement
- » [DROP\\_ROLE](#) statement
- » [GRANT](#) statement
- » [REVOKE](#) statement
- » [SET\\_ROLE](#) statement

# SYSROUTINEPERMS System Table

The SYSROUTINEPERMS table stores the permissions that have been granted to routines.

Each routine EXECUTE permission is specified in a row in the SYSROUTINEPERMS table. The keys for the SYSROUTINEPERMS table are:

- » Primary key (GRANTEE, ALIASID, GRANTOR)
- » Unique key (ROUTINEPERMSID)
- » Foreign key (ALIASID references SYS.SYSALIASES)

The following table shows the contents of the SYSROUTINEPERMS system table.

SYSROUTINEPERMS system table

Column Name	Type	Length	Nullable	Contents
ROUTINEPERMSID	CHAR	36	NO	Used by the dependency manager to track the dependency of a view, trigger, or constraint on the routine level permissions
GRANTEE	VARCHAR	128	NO	The authorization ID of the user or role to which the privilege is granted
GRANTOR	VARCHAR	128	NO	The authorization ID of the user who granted the privilege. Privileges can be granted only by the object owner.
ALIASID	CHAR	36	NO	The ID of the object of the required permission.  If PERMTYPE= 'E ', the ALIASID is a reference to the SYS.SYSALIASES table.  Otherwise, the ALIASID is a reference to the SYS.SYSTABLES table.
GRANTOPTION	CHAR	1	NO	Specifies if the GRANTEE is the owner of the routine. Valid values are 'Y' and 'N'.

# SYSSCHEMAS System Table

The SYSSCHEMAS table describes the schemas within the current database.

The following table shows the contents of the SYSSCHEMAS system table.

SYSSCHEMAS system table

Column Name	Type	Length	Nullable	Contents
SCHEMAID	CHAR	36	NO	Unique identifier for the schema
SCHEMANAME	VARCHAR	128	NO	Schema name
AUTHORIZATIONID	VARCHAR	128	NO	The authorization identifier of the owner of the schema

## See Also

>> [About System Tables](#)

# SYSSEQUENCES System Table

The SYSSEQUENCES table describes the sequence generators in the database.

**NOTE:** Users should not directly query the SYSSEQUENCES table, because that will slow down the performance of sequence generators. Instead, users should call the [SYSCS\\_UTIL.SYSCS\\_PEEK\\_AT\\_SEQUENCE system function](#)

The following table shows the contents of the SYSSEQUENCES system table.

SYSSEQUENCES system table

Column Name	Type	Length	Nullable	Contents
SEQUENCEID	CHAR	36	NO	The ID of the sequence generator. This is the primary key.
SEQUENCENAME	VARCHAR	128	NO	The name of the sequence generator. There is a unique index on this column (SEQUENCENAME).
SCHEMAID	CHAR	36	NO	The ID of the schema of the sequence generator. There is a unique index on this column to SYSSCHEMAS.
SEQUENCEDATATYPE	com.splicemachine.db.catalog.TypeDescriptor	-1	NO	System type that describes the sequence generator's length, scale, nullability, and storage type of the sequence generator.
CURRENTVALUE	BIGINT	19	YES	The current value of the sequence generator. This is not the actual value of the sequence generator, but the value obtained by calling SYSCS_UTIL.SYSCS_PEEK_AT_SEQUENCE. This column is NULL if the sequence generator is exhausted and no more numbers are available.
STARTVALUE	BIGINT	19	NO	The initial value of the sequence generator.



Column Name	Type	Length	Nullable	Contents
MINIMUMVALUE	BIGINT	19	NO	The minimum value
MAXIMUMVALUE	BIGINT	19	NO	The maximum value
INCREMENT	BIGINT	19	NO	The step size of the
CYCLEOPTION	CHAR	1	NO	If the sequence gets to the maximum value, it cycles back to the minimum value. If the sequence gets to the minimum value, it cycles back to the maximum value.

## See Also

» [About System Tables](#)

# SYSSNAPSHOTS System Table

The SYSSSNAPSHOTS table describes the metadata for system snapshots.

The following table shows the contents of the SYSSNAPSHOTS system table.

**NOTE:** Table snapshots both the data and indexes for the table.

SYSSNAPSHOTS system table

Column Name	Type	Length	Nullable	Contents
SNAPSHOTNAME	VARCHAR	128	NO	The name of the snapshot
SCHEMANAME	VARCHAR	128	NO	Schema name
OBJECTNAME	VARCHAR	128	NO	The name of the table or index
CONGLOMERATENUMBER	BIGINT	19	NO	The conglomerate number of the object
CREATIONTIME	TIMESTAMP	29	NO	The time at which the snapshot was taken
LASTRESTORETIME	TIMESTAMP	29	NO	The time at which the snapshot was most recently restored

## See Also

>> [About System Tables](#)

## SYSSTATEMENTS System Table

The `SYSSTATEMENTS` table describes the prepared statements in the database.

The table contains one row per stored prepared statement.

The following table shows the contents of the `SYSSTATEMENTS` system table.

**SYSSTATEMENTS system table**

Column Name	Type	Length	Nullable	Contents
STMTID	CHAR	36	NO	Unique identifier for the statement
STMTNAME	VARCHAR	128	NO	Name of the statement
SCHEMAID	CHAR	36	NO	The schema in which the statement resides
TYPE	CHAR	1	NO	Always 'S'
VALID	BOOLEAN	1	NO	Whether or not the statement is valid
TEXT	LONG VARCHAR	32,700	NO	Text of the statement
LASTCOMPILED	TIMESTAMP	29	YES	Time that the statement was compiled
COMPILATIONSCHEMAID	CHAR	36	NO	ID of the schema containing the statement
USINGTEXT	LONG VARCHAR	32,700	YES	Text of the <code>USING</code> clause of the <code>CREATE STATEMENT</code> and <code>ALTER STATEMENT</code> statements

## See Also

» [About System Tables](#)

## SYSTABLEPERMS System Table

The `SYSTABLEPERMS` table stores the table permissions that have been granted but not revoked.

All of the permissions for one (`GRANTEE`, `TABLEID`, `GRANTOR`) combination are specified in a single row in the `SYSTABLEPERMS` table. The keys for the `SYSTABLEPERMS` table are:

- » Primary key (`GRANTEE`, `TABLEID`, `GRANTOR`)
- » Unique key (`TABLEPERMSID`)
- » Foreign key (`TABLEID` references `SYS.SYSTABLES`)

The following table shows the contents of the `SYSTABLEPERMS` system table.

**SYSTABLEPERMS system table**

Column Name	Type	Length	Nullable	Contents
<code>TABLEPERMSID</code>	<code>CHAR</code>	36	NO	Used by the dependency manager to track the dependency of a view, trigger, or constraint on the table level permissions
<code>GRANTEE</code>	<code>VARCHAR</code>	128	NO	The authorization ID of the user or role to which the privilege is granted
<code>GRANTOR</code>	<code>VARCHAR</code>	128	NO	The authorization ID of the user who granted the privilege. Privileges can be granted only by the object owner
<code>TABLEID</code>	<code>CHAR</code>	36	NO	The unique identifier for the table on which the permissions have been granted
<code>SELECTPRIV</code>	<code>CHAR</code>	1	NO	Specifies if the <code>SELECT</code> permission is granted. The valid values are: <ul style="list-style-type: none"> <li>» 'Y' (non-grantable privilege)</li> <li>» 'Y' (grantable privilege)</li> <li>» 'N' (no privilege)</li> </ul>
<code>DELETEPRIV</code>	<code>CHAR</code>	1	NO	Specifies if the <code>DELETE</code> permission is granted. The valid values are: <ul style="list-style-type: none"> <li>» 'Y' (non-grantable privilege)</li> <li>» 'Y' (grantable privilege)</li> <li>» 'N' (no privilege)</li> </ul>

Column Name	Type	Length	Nullable	Contents
INSERTPRIV	CHAR	1	NO	<p>Specifies if the <code>INSERT</code> permission is granted. The valid values are:</p> <ul style="list-style-type: none"> <li>» 'Y' (non-grantable privilege)</li> <li>» 'Y' (grantable privilege)</li> <li>» 'N' (no privilege)</li> </ul>
UPDATEPRIV	CHAR	1	NO	<p>Specifies if the <code>UPDATE</code> permission is granted. The valid values are:</p> <ul style="list-style-type: none"> <li>» 'Y' (non-grantable privilege)</li> <li>» 'Y' (grantable privilege)</li> <li>» 'N' (no privilege)</li> </ul>
REFERENCESPRIV	CHAR	1	NO	<p>Specifies if the <code>REFERENCE</code> permission is granted. The valid values are:</p> <ul style="list-style-type: none"> <li>» 'Y' (non-grantable privilege)</li> <li>» 'Y' (grantable privilege)</li> <li>» 'N' (no privilege)</li> </ul>
TRIGGERPRIV	CHAR	1	NO	<p>Specifies if the <code>TRIGGER</code> permission is granted. The valid values are:</p> <ul style="list-style-type: none"> <li>» 'Y' (non-grantable privilege)</li> <li>» 'Y' (grantable privilege)</li> <li>» 'N' (no privilege)</li> </ul>

## See Also

» [About System Tables](#)

# SYSTABLES System Table

The SYSTABLES table describes the tables and views within the current database.

The following table shows the contents of the SYSTABLES system table.

SYSTABLES system table

Column Name	Type	Length	Nullable	Contents
TABLEID	CHAR	36	NO	Unique identifier for table or view
TABLERNAME	VARCHAR	128	NO	Table or view name
TABLETYPE	CHAR	1	NO	Possible values are:  » 'S' (system table) » 'T' (user table) » 'A' (synonym) » 'V' (view)
SCHEMAID	CHAR	36	NO	Schema ID for the table or view
LOCKGRANULARITY	CHAR	1	NO	Lock granularity for the table:  » 'T' (table level locking) » 'R' (row level locking, the default)
VERSION	VARCHAR	128	YES	Version ID.

## See Also

» [About System Tables](#)

# SYSTABLESTATISTICS System Table

The SYSTABLESTATISTICS table view describes the statistics for tables within the current database.

**NOTE:** SYS.SYSTABLESTATISTICS is a system view.

The following table shows the contents of the SYSTABLESTATISTICS system table.

SYSTABLESTATISTICS system table

Column Name	Type	Length	Nullable	Contents
SCHEMANAME	VARCHAR	32672	YES	The name of the schema
TABlename	VARCHAR	32672	YES	The name of the table
CONGLOMERATENAME	VARCHAR	32672	YES	The name of the table
TOTAL_ROW_COUNT	BIGINT	19	YES	The total number of rows in the table
AVG_ROW_COUNT	BIGINT	19	YES	The average number of rows in the table
TOTAL_SIZE	BIGINT	19	YES	The total size of the table
NUM_PARTITIONS	BIGINT	19	YES	The number of partitions <sup>1</sup> for which statistics were collected.
AVG_PARTITION_SIZE	BIGINT	19	YES	The average size of a single partition <sup>1</sup> , in bytes.
ROW_WIDTH	BIGINT	19	YES	<p>The <i>maximum average</i> of the widths of rows in the table, across all partitions, in bytes.</p> <p>Each partition records the average width of a single row. This value is the maximum of those averages across all partitions.</p>

Column Name	Type	Length	Nullable	Contents								
STATS_TYPE	INTEGER	10	YES	<div>The type of statistics, which is one of these values:</div> <table><tr><td>0</td><td>Full table (not sampled) statistics that reflect the unmerged partition values.</td></tr><tr><td>1</td><td>Sampled statistics that reflect the unmerged partition values.</td></tr><tr><td>2</td><td>Full table (not sampled) statistics that reflect the table values after all partitions have been merged.</td></tr><tr><td>3</td><td>Sampled statistics that reflect the table values after all partitions have been merged.</td></tr></table> <div>If this value is NULL, 0 is used.</div>	0	Full table (not sampled) statistics that reflect the unmerged partition values.	1	Sampled statistics that reflect the unmerged partition values.	2	Full table (not sampled) statistics that reflect the table values after all partitions have been merged.	3	Sampled statistics that reflect the table values after all partitions have been merged.
0	Full table (not sampled) statistics that reflect the unmerged partition values.											
1	Sampled statistics that reflect the unmerged partition values.											
2	Full table (not sampled) statistics that reflect the table values after all partitions have been merged.											
3	Sampled statistics that reflect the table values after all partitions have been merged.											
SAMPLE_FRACTION	DOUBLE	52	YES	<div>The sampling percentage, expressed as 0.0 to 1.0,</div> <div>&gt;&gt; If statsType=0 (full statistics), this value is not used, and is shown as 0.</div> <div>&gt;&gt; If statsType=1, this value is the percentage or rows to be sampled. A value of 0 means no rows, and a value of 1 means all rows (full statistics).</div>								

<sup>1</sup>Currently, a *partition* is equivalent to a region. In the future, we may use a more finely-grained definition for partition.

See Also

- >> [About System Tables](#)
- >> [SYSCOLUMNSTATISTICS](#) system table



# SYSTRIGGERS System Table

The SYSTRIGGERS table describes the database’s triggers.

The following table shows the contents of the SYSTRIGGERS system table.

SYSTRIGGERS system table

Column Name	Type	Length	Nullable	Contents
TRIGGERID	CHAR	36	NO	Unique identifier for the trigger
TRIGGERNAME	VARCHAR	128	NO	Name of the trigger
SCHEMAID	CHAR	36	NO	ID of the trigger's schema (join with SYSSCHEMAS . SCHEMAID)
CREATIONTIMESTAMP	TIMESTAMP	29	NO	Time the trigger was created
EVENT	CHAR	1	NO	Possible values are:  » 'U' for update » 'D' for delete » 'I' for insert
FIRINGTIME	CHAR	1	NO	Possible values are:  » 'B' for before » 'A' for after
TYPE	CHAR	1	NO	Possible values are:  » 'R' for row » 'S' for statement
STATE	CHAR	1	NO	Possible values are:  » 'E' for enabled » 'D' for disabled

Column Name	Type	Length	Nullable	Contents
TABLEID	CHAR	36	NO	ID of the table on which the trigger is defined
WHENSTMTID	CHAR	36	YES	Used only if there is a WHEN clause (not yet supported)
ACTIONSTMTID	CHAR	36	YES	ID of the stored prepared statement for the triggered-SQL-statement (join with SYSSTATEMENTS . STMTID)
REFERENCEDCOLUMNS	<i>org.apache.Splice Machine .catalog.ReferencedColumns</i> This class is not part of the public API.	-1	YES	Descriptor of the columns to be updated, if this trigger is an update trigger (that is, if the EVENT column contains 'U')
TRIGGERDEFINITION	LONG VARCHAR	2,147,483,647	YES	Text of the action SQL statement
REFERENCINGOLD	BOOLEAN	1	YES	Whether or not the OLDREFERENCINGNAME, if non-null, refers to the OLD row or table
REFERENCINGNEW	BOOLEAN	1	YES	Whether or not the NEWREFERENCINGNAME, if non-null, refers to the NEW row or table
OLDREFERENCINGNAME	VARCHAR	128	YES	Pseudonym as set using the REFERENCING OLD AS clause
NEWREFERENCINGNAME	VARCHAR	128	YES	Pseudonym as set using the REFERENCING NEW AS clause

Any SQL text that is part of a triggered-SQL-statement is compiled and stored in the SYSSTATEMENTS table. ACTIONSTMTID and WHENSTMTID are foreign keys that reference SYSSTATEMENTS . STMTID. The statements for a trigger are always in the same schema as the trigger.

## SYSUSERS System Table

The `SYSUSERS` table stores user credentials when `NATIVE` authentication is enabled.

When SQL authorization is enabled (as it is, for instance, when `NATIVE` authentication is on) only the database owner can `SELECT` from this table, and no one, not even the database owner, can `SELECT` the `PASSWORD` column.

The following table shows the contents of the `SYSUSERS` system table.

**SYSUSERS system table**

Column Name	Type	Length	Nullable	Contents
USERNAME	VARCHAR	128	NO	The user's name, the value of the <code>user</code> attribute on a connection URL.
HASHINGSCHEME	VARCHAR	32672	NO	Describes how the password is hashed.
PASSWORD	VARCHAR	32672	NO	The password after applying the <code>HASHINGSCHEME</code> .
LASTMODIFIED	TIMESTAMP	29	NO	The time when the password was last updated.

## See Also

» [About System Tables](#)

# SYSVIEWS System Table

The `SYSVIEWS` table describes the view definitions within the current database.

The following table shows the contents of the `SYSVIEWS` system table.

SYSVIEWS system table

Column Name	Type	Length	Nullable	Contents
TABLEID	CHAR	36	NO	Unique identifier for the view (join with <code>SYSTABLES.TABLEID</code> )
VIEWDEFINITION	LONG VARCHAR	32,700	NO	Text of view definition
CHECKOPTION	CHAR	1	NO	'N' (check option not supported yet)
COMPILATIONSCHEMAID	CHAR	36	NO	ID of the schema containing the view

## See Also

» [About System Tables](#)

## Error Codes

This section contains descriptions of Splice Machine error codes, in these topics:

Error Class	Description
01	<a href="#">Warning Messages</a>
07	<a href="#">Dynamic SQL Error</a>
08	<a href="#">Connection Exception</a>
0A	<a href="#">Feature not supported</a>
0P	<a href="#">Invalid role specification</a>
21	<a href="#">Cardinality Violation</a>
22	<a href="#">Data Exception</a>
23	<a href="#">Constraint Violation</a>
24	<a href="#">Invalid Cursor State</a>
25	<a href="#">Invalid Transaction State</a>
28	<a href="#">Invalid Authorization Specification</a>
2D	<a href="#">Invalid Transaction Termination</a>
38	<a href="#">External Function Exception</a>
39	<a href="#">External Routine Invocation Exception</a>
3B	<a href="#">Invalid SAVEPOINT</a>
40	<a href="#">Transaction Rollback</a>
42	<a href="#">Syntax Error or Access Rule Violation</a>
57	<a href="#">DRDA Network Protocol - Execution Failure</a>
58	<a href="#">DRDA Network Protocol - Protocol Error</a>
X0	<a href="#">Execution exceptions</a>

XBCA	<a href="#">CacheService</a>
XBCM	<a href="#">ClassManager</a>
XBCX	<a href="#">Cryptography</a>
XBM	<a href="#">Monitor</a>
XCL	<a href="#">Execution exceptions</a>
XCW	<a href="#">Upgrade unsupported</a>
XCX	<a href="#">Internal Utility Errors</a>
XCY	<a href="#">Derby Property Exceptions</a>
XCZ	<a href="#">org.apache.derby.database.UserUtility</a>
XD00	<a href="#">Dependency Manager</a>
XIE	<a href="#">Import/Export Exceptions</a>
XJ	<a href="#">Connectivity Errors</a>
XK	<a href="#">Security Exceptions</a>
XN	<a href="#">Network Client Exceptions</a>
XRE	<a href="#">Replication Exceptions</a>
XSAI	<a href="#">Store - access.protocol.interface</a>
XSAM	<a href="#">Store - AccessManager</a>
XSAS	<a href="#">Store - Sort</a>
XSAX	<a href="#">Store - access.protocol.XA statement</a>
XSCB	<a href="#">Store - BTree</a>
XSCG0	<a href="#">Conglomerate</a>
XSCH	<a href="#">Heap</a>
XSDA	<a href="#">RawStore - Data.Generic statement</a>
XSDB	<a href="#">RawStore - Data.Generic transaction</a>

XSDF	<a href="#">RawStore - Data.Filesystem statement</a>
XSDG	<a href="#">RawStore - Data.Filesystem database</a>
XSLA	<a href="#">RawStore - Log.Generic database exceptions</a>
XSLB	<a href="#">RawStore - Log.Generic statement exceptions</a>
XSRS	<a href="#">RawStore - protocol.Interface statement</a>
XSTA2	<a href="#">XACT_TRANSACTION_ACTIVE</a>
XSTB	<a href="#">RawStore - Transactions.Basic system</a>
XXXXX	<a href="#">No SQLSTATE</a>

## Error Class 01: Warning Messages

### Error Class 01: Warnings

SQLSTATE	Message Text
01001	An attempt to update or delete an already deleted row was made: No row was updated or deleted.
01003	Null values were eliminated from the argument of a column function.
01006	Privilege not revoked from user <authorizationID>.
01007	Role <authorizationID> not revoked from authentication id <authorizationID>.
01008	WITH ADMIN OPTION of role <authorizationID> not revoked from authentication id <authorizationID>.
01009	Generated column <columnName> dropped from table <tableName>.
0100E	XX Attempt to return too many result sets.
01500	The constraint <constraintName> on table <tableName> has been dropped.
01501	The view <viewName> has been dropped.
01502	The trigger <triggerName> on table <tableName> has been dropped.
01503	The column <columnName> on table <tableName> has been modified by adding a not null constraint.
01504	The new index is a duplicate of an existing index: <indexName>.
01505	The value <valueName> may be truncated.
01522	The newly defined synonym '<synonymName>' resolved to the object '<objectName>' which is currently undefined.
01J01	Database '<databaseName>' not created, connection made to existing database instead.
01J02	Scroll sensitive cursors are not currently implemented.
01J04	The class '<className>' for column '<columnName>' does not implement java.io.Serializable or java.sql.SQLData. Instances must implement one of these interfaces to allow them to be stored.
01J05	Database upgrade succeeded. The upgraded database is now ready for use. Revalidating stored prepared statements failed. See next exception for details of failure.
01J06	ResultSet not updatable. Query does not qualify to generate an updatable ResultSet.



SQLSTATE	Message Text
01J07	ResultSetHoldability restricted to ResultSet.CLOSE_CURSORS_AT_COMMIT for a global transaction.
01J08	Unable to open resultSet type <resultSetType>. ResultSet type <resultSetType> opened.
01J10	Scroll sensitive result sets are not supported by server; remapping to forward-only cursor
01J12	Unable to obtain message text from server. See the next exception. The stored procedure SYSIBM.SYSCAMESSAGE is not installed on the server. Please contact your database administrator.
01J13	Number of rows returned (<number>) is too large to fit in an integer; the value returned will be truncated.
01J14	SQL authorization is being used without first enabling authentication.
01J15	Your password will expire in <remainingDays> day(s). Please use the SYSCS_UTIL.SYSCS_MODIFY_PASSWORD procedure to change your password in database '<databaseName>'.
01J16	Your password is stale. To protect the database, you should update your password soon. Please use the SYSCS_UTIL.SYSCS_MODIFY_PASSWORD procedure to change your password in database '<databaseName>'.
01J17	Statistics are unavailable or out of date for one or more tables involved in this query.

# Error Class 07: Dynamic SQL Errors

Error Class 07: Dynamic SQL Errors

SQLSTATE	Message Text
07000	At least one parameter to the current statement is uninitialized.
07004	Parameter <parameterName> is an <procedureName> procedure parameter and must be registered with CallableStatement.registerOutParameter before execution.
07009	No input parameters.

## Error Class 08: Connection Exception

### Error Class 08: Connection Exception

SQLSTATE	Message Text
08000	Connection closed by unknown interrupt.
08001.C.10	A connection could not be established because the security token is larger than the maximum allowed by the network protocol.
08001.C.11	A connection could not be established because the user id has a length of zero or is larger than the maximum allowed by the network protocol.
08001.C.12	A connection could not be established because the password has a length of zero or is larger than the maximum allowed by the network protocol.
08001.C.13	A connection could not be established because the external name (EXTNAM) has a length of zero or is larger than the maximum allowed by the network protocol.
08001.C.14	A connection could not be established because the server name (SRVNAM) has a length of zero or is larger than the maximum allowed by the network protocol.
08001.C.1	Required Splice DataSource property <propertyName> not set.
08001.C.2	<error> : Error connecting to server <serverName> on port <portNumber> with message <messageText>.
08001.C.3	SocketException: '<error>'
08001.C.4	Unable to open stream on socket: '<error>'.
08001.C.5	User id length (<number>) is outside the range of 1 to <number>.
08001.C.6	Password length (<value>) is outside the range of 1 to <number>.
08001.C.7	User id can not be null.
08001.C.8	Password can not be null.
08001.C.9	A connection could not be established because the database name '<databaseName>' is larger than the maximum length allowed by the network protocol.
08003	No current connection.
08003.C.1	getConnection() is not valid on a closed PooledConnection.

SQLSTATE	Message Text
08003.C.2	Lob method called after connection was closed
08003.C.3	The underlying physical connection is stale or closed.
08004	Connection refused : <connectionName>
08004.C.1	Connection authentication failure occurred. Reason: <reasonText>.
08004.C.2	The connection was refused because the database <databaseName> was not found.
08004.C.3	Database connection refused.
08004.C.4	User '<authorizationID>' cannot shut down database '<databaseName>'. Only the database owner can perform this operation.
08004.C.5	User '<authorizationID>' cannot (re)encrypt database '<databaseName>'. Only the database owner can perform this operation.
08004.C.6	User '<authorizationID>' cannot hard upgrade database '<databaseName>'. Only the database owner can perform this operation.
08004.C.7	Connection refused to database '<databaseName>' because it is in replication slave mode.
08004.C.8	User '<authorizationID>' cannot issue a replication operation on database '<databaseName>'. Only the database owner can perform this operation.
08004.C.9	Missing permission for user '<authorizationID>' to shutdown system [<exceptionMsg>].
08004.C.10	Cannot check system permission to create database '<databaseName>' [<exceptionMsg>].
08004.C.11	Missing permission for user '<authorizationID>' to create database '<databaseName>' [<exceptionMsg>].
08004.C.12	Connection authentication failure occurred. Either the supplied credentials were invalid, or the database uses a password encryption scheme not compatible with the strong password substitution security mechanism. If this error started after upgrade, refer to the release note for DERBY-4483 for options.
08004.C.13	Username or password is null or 0 length.
08006.C	A network protocol error was encountered and the connection has been terminated: <error>
08006.C.1	An error occurred during connect reset and the connection has been terminated. See chained exceptions for details.
08006.C.2	SocketException: '<error>'

SQLSTATE	Message Text
08006.C.3	A communications error has been detected: <error>.
08006.C.4	An error occurred during a deferred connect reset and the connection has been terminated. See chained exceptions for details.
08006.C.5	Insufficient data while reading from the network - expected a minimum of <number> bytes and received only <number> bytes. The connection has been terminated.
08006.C.6	Attempt to fully materialize lob data that is too large for the JVM. The connection has been terminated.
08006.C.8	com.splicemachine.db.jdbc.EmbeddedDriver is not registered with the JDBC driver manager
08006.C.9	Can't execute statement while in Restore Mode. Reboot database after restore operation is finished.
08006.D	Database '<databaseName>' shutdown.
08006.D.1	Database '<databaseName>' dropped.

# Error Class 0A: Feature Not Supported

Error Class 0A: Feature Not Supported

SQLSTATE	Message Text
0A000.S	Feature not implemented: <featureName>.
0A000.SP	Feature not yet implemented in Splice Machine, but available soon: <featureName>.
0A000.C.6	The DRDA command <commandName> is not currently implemented. The connection has been terminated.
0A000.S.1	JDBC method is not yet implemented.
0A000.S.2	JDBC method <methodName> is not supported by the server. Please upgrade the server.
0A000.S.3	resultSetHoldability property <propertyName> not supported
0A000.S.4	cancel() not supported by the server.
0A000.S.5	Security mechanism '<mechanismName>' is not supported.
0A000.S.7	The data type '<datatypeName>' is not supported.

# Error Class 0P: Invalid Role Specification

Error Class 0P: Invalid Role Specification

SQLSTATE	Message Text
0P000	Invalid role specification, role does not exist: '<roleName>'.
0P000.S.1	Invalid role specification, role not granted to current user or PUBLIC: '<roleName>'.

# Error Class 21: Cardinality Violation

Error Class 21: Cardinality Violation

SQLSTATE	Message Text
21000	Scalar subquery is only allowed to return a single row.



## Error Class 22: Data Exception

### Error Class 22: Data Exception

SQLSTATE	Message Text
22001	A truncation error was encountered trying to shrink <value> '<value>' to length <value>.
22003	The resulting value is outside the range for the data type <datatypeName>.
22003.S.0	The modified row count was larger than can be held in an integer which is required by the JDBC spec. The real modified row count was <modifiedRowCount>.
22003.S.1	Year (<value>) exceeds the maximum '<value>'.
22003.S.2	Decimal may only be up to 31 digits.
22003.S.3	Overflow occurred during numeric data type conversion of '<datatypeName>' to <datatypeName>.
22003.S.4	The length (<number>) exceeds the maximum length (<datatypeName>) for the data type.
22005.S.1	Unable to convert a value of type '<typeName>' to type '<typeName>': the encoding is not supported.
22005.S.2	The required character converter is not available.
22005.S.3	Unicode string cannot convert to Ebcdic string
22005.S.4	Unrecognized JDBC type. Type: <typeName>, columnCount: <value>, columnIndex: <value>.
22005.S.5	Invalid JDBC type for parameter <parameterName>.
22005.S.6	Unrecognized Java SQL type <datatypeName>.
22005.S.7	Unicode string cannot convert to UTF-8 string
22005	An attempt was made to get a data value of type '<datatypeName>' from a data value of type '<datatypeName>'.
22007.S.180	The string representation of a datetime value is out of range.
22007.S.181	The syntax of the string representation of a datetime value is incorrect.
22008.S	'<argument>' is an invalid argument to the <functionName> function.
2200H.S	Sequence generator '<schemaName>.<sequenceName>' does not cycle. No more values can be obtained from this sequence generator.

SQLSTATE	Message Text
2200L	Values assigned to XML columns must be well-formed DOCUMENT nodes.
2200M	Invalid XML DOCUMENT: <parserError>
2200V	Invalid context item for <operatorName> operator; context items must be well-formed DOCUMENT nodes.
2200W	XQuery serialization error: Attempted to serialize one or more top-level Attribute nodes.
22011	The second or third argument of the SUBSTR function is out of range.
22011.S.1	The range specified for the substring with offset <operatorName> and len <len> is out of range for the String: <str>.
22012	Attempt to divide by zero.
22013	Attempt to take the square root of a negative number, '<value>'.
22014	The start position for LOCATE is invalid; it must be a positive integer. The index to start the search from is '<startIndex>'. The string to search for is '<searchString>'. The string to search from is '<fromString>'.
22015	The '<functionName>' function is not allowed on the following set of types. First operand is of type '<typeName>'. Second operand is of type '<typeName>'. Third operand (start position) is of type '<typeName>'.
22018	Invalid character string format for type <typeName>.
22019	Invalid escape sequence, '<sequenceName>'. The escape string must be exactly one character. It cannot be a null or more than one character.
22020	Invalid trim string, '<string>'. The trim string must be exactly one character or NULL. It cannot be more than one character.
22021	Unknown character encoding '<typeName>'.
22025	Escape character must be followed by escape character, '_', or '%'. It cannot be followed by any other character or be at the end of the pattern.
22027	The built-in TRIM() function only supports a single trim character. The LTRIM() and RTRIM() built-in functions support multiple trim characters.
22028	The string exceeds the maximum length of <number>.
22501	An ESCAPE clause of NULL returns undefined results and is not allowed.
2201X	Invalid row count for OFFSET, must be >= 0.

SQLSTATE	Message Text
2201Y	Invalid LEAD, LAG for OFFSET, must be greater or equal to 0 and less than Integer.MAX_VALUE. Got '<value>'.
2202A	Missing argument for first(), last() function.
2202B	Missing argument for lead(), lag() function.
2202C	"default" argument for lead(), lag() function is not implemented.
2202D	NULL value for data type <string> not supported.
2202E	A <string> column cannot be aggregated.
2201W	Row count for FIRST/NEXT/TOP must be >= 1 and row count for LIMIT must be >= 0.
2201Z	NULL value not allowed for <string> argument.

# Error Class 23: Constraint Violation

Error Class 23: Constraint Violation

SQLSTATE	Message Text
23502	Column '<columnName>' cannot accept a NULL value.
23503	<value> on table '<tableName>' caused a violation of foreign key constraint '<constraintName>' for key <keyName>. The statement has been rolled back.
23505	The statement was aborted because it would have caused a duplicate key value in a unique or primary key constraint or unique index identified by '<value>' defined on '<value>'.
23513	The check constraint '<constraintName>' was violated while performing an INSERT or UPDATE on table '<tableName>'.

# Error Class 24: Invalid Cursor State

Error Class 24: Invalid Cursor State

SQLSTATE	Message Text
24000	Invalid cursor state - no current row.
24501.S	The identified cursor is not open.

# Error Class 25: Invalid Transaction State

Error Class 25: Invalid Transaction State

SQLSTATE	Message Text
25001	Cannot close a connection while a transaction is still active.
25001.S.1	Invalid transaction state: active SQL transaction.
25501	Unable to set the connection read-only property in an active transaction.
25502	An SQL data change is not permitted for a read-only connection, user or database.
25503	DDL is not permitted for a read-only connection, user or database.
25505	A read-only user or a user in a read-only database is not permitted to disable read-only mode on a connection.

# Error Class 28: Invalid Authorization Specification

Error Class 28: Invalid Authorization Specification

SQLSTATE	Message Text
28502	The user name '<authorizationID>' is not valid.

# Error Class 2D: Invalid Transaction Termination

Error Class 2D: Invalid Transaction Termination

SQLSTATE	Message Text
2D521.S.1	setAutoCommit(true) invalid during global transaction.
2D521.S.2	COMMIT or ROLLBACK invalid for application execution environment.



## Error Class 38: External Function Exception

**Error Class 38: External Function Exception**

SQLSTATE	Message Text
38000	The exception '<exception>' was thrown while evaluating an expression.
38001	The external routine is not allowed to execute SQL statements.
38002	The routine attempted to modify data, but the routine was not defined as MODIFIES SQL DATA.
38004	The routine attempted to read data, but the routine was not defined as READS SQL DATA.

# Error Class 39: External Routine Invocation Exception

Error Class 39: External Routine Invocation Exception

SQLSTATE	Message Text
39004	A NULL value cannot be passed to a method which takes a parameter of primitive type '<type>'.

## Error Class 3B: Invalid SAVEPOINT

### Error Class 3B: Invalid SAVEPOINT

SQLSTATE	Message Text
3B001.S	Savepoint <savepointName> does not exist or is not active in the current transaction.
3B002.S	The maximum number of savepoints has been reached.
3B501.S	A SAVEPOINT with the passed name already exists in the current transaction.
3B502.S	A RELEASE or ROLLBACK TO SAVEPOINT was specified, but the savepoint does not exist.

## Error Class 40: Transaction Rollback

**Error Class 40: Transaction Rollback**

SQLSTATE	Message Text
40001	A lock could not be obtained due to a deadlock, cycle of locks and waiters is: <lockCycle>. The selected victim is XID : <transactionID>.
40XC0	Dead statement. This may be caused by catching a transaction severity error inside this statement.
40XD0	Container has been closed.
40XD1	Container was opened in read-only mode.
40XD2	Container <containerName> cannot be opened; it either has been dropped or does not exist.
40XL1	A lock could not be obtained within the time requested
40XL1.T.1	A lock could not be obtained within the time requested. The lockTable dump is: <tableDump>
40XT0	An internal error was identified by RawStore module.
40XT1	An exception was thrown during transaction commit.
40XT2	An exception was thrown during rollback of a SAVEPOINT.
40XT4	An attempt was made to close a transaction that was still active. The transaction has been aborted.
40XT5	Exception thrown during an internal transaction.
40XT6	Database is in quiescent state, cannot activate transaction. Please wait for a moment till it exits the quiescent state.
40XT7	Operation is not supported in an internal transaction.

## Error Class 42: Syntax Error or Access Rule Violation

Error Class 42: Syntax Error or Access Rule Violation

SQLSTATE	Message Text
42000	Syntax error or access rule violation; see additional errors for details.
42500	User '<authorizationID>' does not have <permissionType> permission on table '<schemaName>'.<tableName>'.
42501	User '<authorizationID>' does not have <permissionType> permission on table '<schemaName>'.<tableName>' for grant.
42502	User '<authorizationID>' does not have <permissionType> permission on column '<columnName>' of table '<schemaName>'.<tableName>'.
42503	User '<authorizationID>' does not have <permissionType> permission on column '<columnName>' of table '<schemaName>'.<tableName>' for grant.
42504	User '<authorizationID>' does not have <permissionType> permission on <objectName> '<schemaName>'.<tableName>'.
42505	User '<authorizationID>' does not have <permissionType> permission on <objectName> '<schemaName>'.<tableName>' for grant.
42506	User '<authorizationID>' is not the owner of <objectName> '<schemaName>'.<tableName>'.
42507	User '<authorizationID>' can not perform the operation in schema '<schemaName>'.
42508	User '<authorizationID>' can not create schema '<schemaName>'. Only database owner could issue this statement.
42509	Specified grant or revoke operation is not allowed on object '<objectName>'.
4250A	User '<authorizationID>' does not have <permissionName> permission on object '<schemaName>'.<objectName>'.
4250B	Invalid database authorization property '<value>=<value>'.
4250C	User(s) '<authorizationID>' must not be in both read-only and full-access authorization lists.
4250D	Repeated user(s) '<authorizationID>' in access list '<listName>';
4250E	Internal Error: invalid <authorizationID> id in statement permission list.
4251A	Statement <value> can only be issued by database owner.

SQLSTATE	Message Text
4251B	PUBLIC is reserved and cannot be used as a user identifier or role name.
4251C	Role <authorizationID> cannot be granted to <authorizationID> because this would create a circularity.
4251D	Only the database owner can perform this operation.
4251E	No one can view the '<tableName>'.<columnName>' column.
4251F	You cannot drop the credentials of the database owner.
4251G	Please set derby.authentication.builtin.algorithm to a valid message digest algorithm. The current authentication scheme is too weak to be used by NATIVE authentication.
4251H	Invalid NATIVE authentication specification. Please set derby.authentication.provider to a value of the form NATIVE:\$credentialsDB or NATIVE:\$credentialsDB:LOCAL (at the system level).
4251I	Authentication cannot be performed because the credentials database '<databaseName>' does not exist.
4251J	The value for the property '<propertyName>' is formatted badly.
4251K	The first credentials created must be those of the DBO.
4251L	The derby.authentication.provider property specifies '<dbName>' as the name of the credentials database. This is not a valid name for a database.
4251M	User '<authorizationID>' does not have <permissionType> permission to analyze table '<schemaName>'.<tableName>'.
42601	In an ALTER TABLE statement, the column '<columnName>' has been specified as NOT NULL and either the DEFAULT clause was not specified or was specified as DEFAULT NULL.
42601.S.372	ALTER TABLE statement cannot add an IDENTITY column to a table.
42605	The number of arguments for function '<functionName>' is incorrect.
42606	An invalid hexadecimal constant starting with '<number>' has been detected.
42610	All the arguments to the COALESCE/VALUE function cannot be parameters. The function needs at least one argument that is not a parameter.
42611	The length, precision, or scale attribute for column, or type mapping '<value>' is not valid.
42613	Multiple or conflicting keywords involving the '<clause>' clause are present.
42621	A check constraint or generated column that is defined with '<value>' is invalid.

SQLSTATE	Message Text
42622	The name '<name>' is too long. The maximum length is '<number>'.
42734	Name '<name>' specified in context '<context>' is not unique.
42802	The number of values assigned is not the same as the number of specified or implied columns.
42803	An expression containing the column '<columnName>' appears in the SELECT list and is not part of a GROUP BY clause.
42815.S.713	The replacement value for '<value>' is invalid.
42815.S.171	The data type, length or value of arguments '<value>' and '<value>' is incompatible.
42818	Comparisons between '<type>' and '<type>' are not supported. Types must be comparable. String types must also have matching collation. If collation does not match, a possible solution is to cast operands to force them to the default collation (e.g. SELECT tablename FROM sys.systables WHERE CAST(tablename AS VARCHAR(128)) = 'T1')
42820	The floating point literal '<string>' contains more than 30 characters.
42821	Columns of type '<type>' cannot hold values of type '<type>'.
42824	An operand of LIKE is not a string, or the first operand is not a column.
42831	'<columnName>' cannot be a column of a primary key or unique key because it can contain null values.
42831.S.1	'<columnName>' cannot be a column of a primary key because it can contain null values.
42834	SET NULL cannot be specified because FOREIGN KEY '<key>' cannot contain null values.
42837	ALTER TABLE '<tableName>' specified attributes for column '<columnName>' that are not compatible with the existing column.
42846	Cannot convert types '<type>' to '<type>'.
42877	A qualified column name '<columnName>' is not allowed in the ORDER BY clause.
42878	The ORDER BY clause of a SELECT UNION statement only supports unqualified column references and column position numbers. Other expressions are not currently supported.
42879	The ORDER BY clause may not contain column '<columnName>', since the query specifies DISTINCT and that column does not appear in the query result.
4287A	The ORDER BY clause may not specify an expression, since the query specifies DISTINCT.
4287B	In this context, the ORDER BY clause may only specify a column number.

SQLSTATE	Message Text
42884	No authorized routine named '<routineName>' of type '<type>' having compatible arguments was found.
42886	'<value>' parameter '<value>' requires a parameter marker '?'.
42894	DEFAULT value or IDENTITY attribute value is not valid for column '<columnName>'.
428C1	Only one identity column is allowed in a table.
428EK	The qualifier for a declared global temporary table name must be SESSION.
428C2	DELETE ROWS is not supported for ON '<txnMode>'.
428C3	Temporary table columns cannot be referenced by foreign keys.
428C4	Attempt to add temporary table, '<txnMode>', as a view dependency.
42903	Invalid use of an aggregate function.
42908	The CREATE VIEW statement does not include a column list.
42909	The CREATE TABLE statement does not include a column list.
42915	Foreign Key '<key>' is invalid because '<value>'.
42916	Synonym '<synonym2>' cannot be created for '<synonym1>' as it would result in a circular synonym chain.
42939	An object cannot be created with the schema name '<schemaName>'.
4293A	A role cannot be created with the name '<authorizationID>', the SYS prefix is reserved.
42962	Long column type column or parameter '<columnName>' not permitted in declared global temporary tables or procedure definitions.
42995	The requested function does not apply to global temporary tables.
42X01	Syntax error: <error>.
42X02	<value>.
42X03	Column name '<columnName>' is in more than one table in the FROM list.
42X04	Column '<columnName>' is either not in any table in the FROM list or appears within a join specification and is outside the scope of the join specification or appears in a HAVING clause and is not in the GROUP BY list. If this is a CREATE or ALTER TABLE statement then '<columnName>' is not a column in the target table.



SQLSTATE	Message Text
42X05	Table/View '<objectName>' does not exist.
42X06	Too many result columns specified for table '<tableName>'.
42X07	Null is only allowed in a VALUES clause within an INSERT statement.
42X08	The constructor for class '<className>' cannot be used as an external virtual table because the class does not implement '<constructorName>'.
42X09	The table or alias name '<tableName>' is used more than once in the FROM list.
42X10	'<tableName>' is not an exposed table name in the scope in which it appears.
42X12	Column name '<columnName>' appears more than once in the CREATE TABLE statement.
42X13	Column name '<columnName>' appears more than once times in the column list of an INSERT statement.
42X14	'<columnName>' is not a column in table or VTI '<value>'.
42X15	Column name '<columnName>' appears in a statement without a FROM list.
42X16	Column name '<columnName>' appears multiple times in the SET clause of an UPDATE statement.
42X17	In the Properties list of a FROM clause, the value '<value>' is not valid as a joinOrder specification. Only the values FIXED and UNFIXED are valid.
42X19	The WHERE or HAVING clause or CHECK CONSTRAINT definition is a '<value>' expression. It must be a BOOLEAN expression.
42X20	Syntax error; integer literal expected.
42X23	Cursor <cursorName> is not updatable.
42X24	Column <columnName> is referenced in the HAVING clause but is not in the GROUP BY list.
42X25	The '<functionName>' function is not allowed on the type.
42X26	The class '<className>' for column '<columnName>' does not exist or is inaccessible. This can happen if the class is not public.
42X28	Delete table '<tableName>' is not target of cursor '<cursorName>'.
42X29	Update table '<tableName>' is not the target of cursor '<cursorName>'.
42X30	Cursor '<cursorName>' not found. Verify that autocommit is OFF.

SQLSTATE	Message Text
42X31	Column '<columnName>' is not in the FOR UPDATE list of cursor '<cursorName>'.
42X32	The number of columns in the derived column list must match the number of columns in table '<tableName>'.
42X33	The derived column list contains a duplicate column name '<columnName>'.
42X34	There is a ? parameter in the select list. This is not allowed.
42X35	It is not allowed for both operands of '<value>' to be ? parameters.
42X36	The '<operator>' operator is not allowed to take a ? parameter as an operand.
42X37	The unary '<operator>' operator is not allowed on the '<type>' type.
42X38	'SELECT *' only allowed in EXISTS and NOT EXISTS subqueries.
42X39	Subquery is only allowed to return a single column.
42X40	A NOT statement has an operand that is not boolean . The operand of NOT must evaluate to TRUE, FALSE, or UNKNOWN.
42X41	In the Properties clause of a FROM list, the property '<propertyName>' is not valid (the property was being set to '<value>').
42X42	Correlation name not allowed for column '<columnName>' because it is part of the FOR UPDATE list.
42X43	The ResultSetMetaData returned for the class/object '<className>' was null. In order to use this class as an external virtual table, the ResultSetMetaData cannot be null.
42X44	Invalid length '<number>' in column specification.
42X45	<type> is an invalid type for argument number <value> of <value>.
42X46	There are multiple functions named '<functionName>'. Use the full signature or the specific name.
42X47	There are multiple procedures named '<procedureName>'. Use the full signature or the specific name.
42X48	Value '<value>' is not a valid precision for <value>.
42X49	Value '<value>' is not a valid integer literal.

SQLSTATE	Message Text
42X50	No method was found that matched the method call <methodName>.<value>(<value>), tried all combinations of object and primitive types and any possible type conversion for any parameters the method call may have. The method might exist but it is not public and/or static, or the parameter types are not method invocation convertible.
42X51	The class '<className>' does not exist or is inaccessible. This can happen if the class is not public.
42X52	Calling method ('<methodName>') using a receiver of the Java primitive type '<type>' is not allowed.
42X53	The LIKE predicate can only have 'CHAR' or 'VARCHAR' operands. Type '<type>' is not permitted.
42X54	The Java method '<methodName>' has a ? as a receiver. This is not allowed.
42X55	Table name '<tableName>' should be the same as '<value>'.
42X56	The number of columns in the view column list does not match the number of columns in the underlying query expression in the view definition for '<value>'.
42X57	The getColumnCount() for external virtual table '<tableName>' returned an invalid value '<value>'. Valid values are greater than or equal to 1.
42X58	The number of columns on the left and right sides of the <tableName> must be the same.
42X59	The number of columns in each VALUES constructor must be the same.
42X60	Invalid value '<value>' for insertMode property specified for table '<tableName>'.
42X61	Types '<type>' and '<type>' are not <value> compatible.
42X62	'<value>' is not allowed in the '<schemaName>' schema.
42X63	The USING clause did not return any results. No parameters can be set.
42X64	In the Properties list, the invalid value '<value>' was specified for the useStatistics property. The only valid values are TRUE or FALSE.
42X65	Index '<index>' does not exist.
42X66	Column name '<columnName>' appears more than once in the CREATE INDEX statement.
42X68	No field '<fieldName>' was found belonging to class '<className>'. It may be that the field exists, but it is not public, or that the class does not exist or is not public.
42X69	It is not allowed to reference a field ('<fieldName>') using a referencing expression of the Java primitive type '<type>'.

SQLSTATE	Message Text
42X70	The number of columns in the table column list does not match the number of columns in the underlying query expression in the table definition for '<value>'.
42X71	Invalid data type '<datatypeName>' for column '<columnName>'.
42X72	No static field '<fieldName>' was found belonging to class '<className>'. The field might exist, but it is not public and/or static, or the class does not exist or the class is not public.
42X73	Method resolution for signature <value>.<value>(<value>) was ambiguous. (No single maximally specific method.)
42X74	Invalid CALL statement syntax.
42X75	No constructor was found with the signature <value>(<value>). It may be that the parameter types are not method invocation convertible.
42X76	At least one column, '<columnName>', in the primary key being added is nullable. All columns in a primary key must be non-nullable.
42X77	Column position '<columnPosition>' is out of range for the query expression.
42X78	Column '<columnName>' is not in the result of the query expression.
42X79	Column name '<columnName>' appears more than once in the result of the query expression.
42X80	VALUES clause must contain at least one element. Empty elements are not allowed.
42X81	A query expression must return at least one column.
42X82	The USING clause returned more than one row. Only single-row ResultSets are permissible.
42X83	The constraints on column '<columnName>' require that it be both nullable and not nullable.
42X84	Index '<index>' was created to enforce constraint '<constraintName>'. It can only be dropped by dropping the constraint.
42X85	Constraint '<constraintName>' is required to be in the same schema as table '<tableName>'.
42X86	ALTER TABLE failed. There is no constraint '<constraintName>' on table '<tableName>'.
42X87	At least one result expression (THEN or ELSE) of the '<expression>' expression must not be a '?'.
42X88	A conditional has a non-Boolean operand. The operand of a conditional must evaluate to TRUE, FALSE, or UNKNOWN.
42X89	Types '<type>' and '<type>' are not type compatible. Neither type is assignable to the other type.

SQLSTATE	Message Text
42X90	More than one primary key constraint specified for table '<tableName>'.
42X91	Constraint name '<constraintName>' appears more than once in the CREATE TABLE statement.
42X92	Column name '<columnName>' appears more than once in a constraint's column list.
42X93	Table '<tableName>' contains a constraint definition with column '<columnName>' which is not in the table.
42X94	<value> '<value>' does not exist.
42X96	The database class path contains an unknown jar file '<fileName>'.
42X98	Parameters are not allowed in a VIEW definition.
42X99	Parameters are not allowed in a TABLE definition.
42XA0	The generation clause for column '<columnName>' has data type '<datatypeName>', which cannot be assigned to the column's declared data type.
42XA1	The generation clause for column '<columnName>' contains an aggregate. This is not allowed.
42XA2	'<value>' cannot appear in a GENERATION CLAUSE because it may return unreliable results.
42XA3	You may not override the value of generated column '<columnName>'.
42XA4	The generation clause for column '<columnName>' references other generated columns. This is not allowed.
42XA5	Routine '<routineName>' may issue SQL and therefore cannot appear in a GENERATION CLAUSE.
42XA6	'<columnName>' is a generated column. It cannot be part of a foreign key whose referential action for DELETE is SET NULL or SET DEFAULT, or whose referential action for UPDATE is CASCADE.
42XA7	'<columnName>' is a generated column. You cannot change its default value.
42XA8	You cannot rename '<columnName>' because it is referenced by the generation clause of column '<columnName>'.
42XA9	Column '<columnName>' needs an explicit datatype. The datatype can be omitted only for columns with generation clauses.
42XAA	The NEW value of generated column '<columnName>' is mentioned in the BEFORE action of a trigger. This is not allowed.

SQLSTATE	Message Text
42XAB	NOT NULL is allowed only if you explicitly declare a datatype.
42XAC	'INCREMENT BY' value can not be zero.
42XAE	'<argName>' value out of range of datatype '<datatypeName>'. Must be between '<minValue>' and '<maxValue>'.
42XAF	Invalid 'MINVALUE' value '<minValue>'. Must be smaller than 'MAXVALUE: <maxValue>'.
42XAG	Invalid 'START WITH' value '<startValue>'. Must be between '<minValue>' and '<maxValue>'.
42XAH	A NEXT VALUE FOR expression may not appear in many contexts, including WHERE, ON, HAVING, ORDER BY, DISTINCT, CASE, GENERATION, and AGGREGATE clauses as well as WINDOW functions and CHECK constraints.
42XAI	The statement references the following sequence more than once: '<sequenceName>'.
42XAJ	The CREATE SEQUENCE statement has a redundant '<clauseName>' clause.
42Y00	Class '<className>' does not implement com.splicemachine.db.iapi.db.AggregateDefinition and thus cannot be used as an aggregate expression.
42Y01	Constraint '<constraintName>' is invalid.
42Y03.S.0	'<statement>' is not recognized as a function or procedure.
42Y03.S.1	'<statement>' is not recognized as a procedure.
42Y03.S.2	'<statement>' is not recognized as a function.
42Y04	Cannot create a procedure or function with EXTERNAL NAME '<name>' because it is not a list separated by periods. The expected format is <full java path>.<method name>.
42Y05	There is no Foreign Key named '<key>'.
42Y07	Schema '<schemaName>' does not exist
42Y08	Foreign key constraints are not allowed on system tables.
42Y09	Void methods are only allowed within a CALL statement.
42Y10	A table constructor that is not in an INSERT statement has all ? parameters in one of its columns. For each column, at least one of the rows must have a non-parameter.
42Y11	A join specification is required with the '<clauseName>' clause.
42Y12	The ON clause of a JOIN is a '<expressionType>' expression. It must be a BOOLEAN expression.

SQLSTATE	Message Text
42Y13	Column name '<columnName>' appears more than once in the CREATE VIEW statement.
42Y16	No public static method '<methodName>' was found in class '<className>'. The method might exist, but it is not public, or it is not static.
42Y22	Aggregate <aggregateType> cannot operate on type <type>.
42Y23	Incorrect JDBC type info returned for column <columnName>.
42Y24	View '<viewName>' is not updatable. (Views are currently not updatable.)
42Y25	'<tableName>' is a system table. Users are not allowed to modify the contents of this table.
42Y26	Aggregates are not allowed in the GROUP BY list.
42Y27	Parameters are not allowed in the trigger action.
42Y29	The SELECT list of a non-grouped query contains at least one invalid expression. When the SELECT list contains at least one aggregate then all entries must be valid aggregate expressions.
42Y30	The SELECT list of a grouped query contains at least one invalid expression. If a SELECT list has a GROUP BY, the list may only contain valid grouping expressions and valid aggregate expressions.
42Y32	Aggregator class '<className>' for aggregate '<aggregateName>' on type <type> does not implement com.splicemachine.db.ijapi.sql.execute.ExecAggregator.
42Y33	Aggregate <aggregateName> contains one or more aggregates.
42Y34	Column name '<columnName>' matches more than one result column in table '<tableName>'.
42Y35	Column reference '<reference>' is invalid. When the SELECT list contains at least one aggregate then all entries must be valid aggregate expressions.
42Y36	Column reference '<reference>' is invalid, or is part of an invalid expression. For a SELECT list with a GROUP BY, the columns and expressions being selected may only contain valid grouping expressions and valid aggregate expressions.
42Y37	'<value>' is a Java primitive and cannot be used with this operator.
42Y38	insertMode = replace is not permitted on an insert where the target table, '<tableName>', is referenced in the SELECT.
42Y39	'<value>' may not appear in a CHECK CONSTRAINT definition because it may return non-deterministic results.
42Y40	'<value>' appears multiple times in the UPDATE OF column list for trigger '<triggerName>'.

SQLSTATE	Message Text
42Y41	'<value>' cannot be directly invoked via EXECUTE STATEMENT because it is part of a trigger.
42Y42	Scale '<scaleValue>' is not a valid scale for a <value>.
42Y43	Scale '<scaleValue>' is not a valid scale with precision of '<precision>'.
42Y44	Invalid key '<key>' specified in the Properties list of a FROM list. The case-sensitive keys that are currently supported are '<key>'.
42Y45	VTI '<value>' cannot be bound because it is a special trigger VTI and this statement is not part of a trigger action or WHEN clause.
42Y46	Invalid Properties list in FROM list. There is no index '<index>' on table '<tableName>'.
42Y47	Invalid Properties list in FROM list. The hint useSpark needs (true/false) and does not support '<value>'.
42Y48	Invalid Properties list in FROM list. Either there is no named constraint '<constraintName>' on table '<tableName>' or the constraint does not have a backing index.
42Y49	Multiple values specified for property key '<key>'.
42Y50	Properties list for table '<tableName>' may contain values for index or for constraint but not both.
42Y55	'<value>' cannot be performed on '<value>' because it does not exist.
42Y56	Invalid join strategy '<strategyValue>' specified in Properties list on table '<tableName>'. The currently supported values for a join strategy are: <supportedStrategyNames>.
42Y58	NumberFormatException occurred when converting value '<value>' for optimizer override '<value>'.
42Y59	Invalid value, '<value>', specified for hashInitialCapacity override. Value must be greater than 0.
42Y60	Invalid value, '<value>', specified for hashLoadFactor override. Value must be greater than 0.0 and less than or equal to 1.0.
42Y61	Invalid value, '<value>', specified for hashMaxCapacity override. Value must be greater than 0.
42Y62	'<statement>' is not allowed on '<viewName>' because it is a view.
42Y63	Hash join requires an optimizable equijoin predicate on a column in the selected index or heap. An optimizable equijoin predicate does not exist on any column in table or index '<index>'. Use the 'index' optimizer override to specify such an index or the heap on table '<tableName>'.
42Y64	bulkFetch value of '<value>' is invalid. The minimum value for bulkFetch is 1.



SQLSTATE	Message Text
42Y65	bulkFetch is not permitted on '<joinType>' joins.
42Y66	bulkFetch is not permitted on updatable cursors.
42Y67	Schema '<schemaName>' cannot be dropped.
42Y69	No valid execution plan was found for this statement. This is usually because an infeasible join strategy was chosen, or because an index was chosen which prevents the chosen join strategy from being used.
42Y70	The user specified an illegal join order. This could be caused by a join column from an inner table being passed as a parameter to an external virtual table.
42Y71	System function or procedure '<procedureName>' cannot be dropped.
42Y82	System generated stored prepared statement '<statement>' that cannot be dropped using DROP STATEMENT. It is part of a trigger.
42Y83	An untyped null is not permitted as an argument to aggregate <aggregateName>. Please cast the null to a suitable type.
42Y84	'<value>' may not appear in a DEFAULT definition.
42Y85	The DEFAULT keyword is only allowed in a VALUES clause when the VALUES clause appears within an INSERT statement.
42Y90	FOR UPDATE is not permitted in this type of statement.
42Y91	The USING clause is not permitted in an EXECUTE STATEMENT for a trigger action.
42Y92	<triggerName> triggers may only reference <value> transition variables/tables.
42Y93	Illegal REFERENCING clause: only one name is permitted for each type of transition variable/table.
42Y94	An AND or OR has a non-boolean operand. The operands of AND and OR must evaluate to TRUE, FALSE, or UNKNOWN.
42Y95	The '<operatorName>' operator with a left operand type of '<operandType>' and a right operand type of '<operandType>' is not supported.
42Y96	Invalid Sort Strategy: '<sortStrategy>'.
42Y97	Invalid escape character at line '<lineNumber>', column '<columnName>'.
42Z02	Multiple DISTINCT aggregates are not supported at this time.

SQLSTATE	Message Text
42Z07	Aggregates are not permitted in the ON clause.
42Z08	Bulk insert replace is not permitted on '<value>' because it has an enabled trigger (<value>).
42Z15	Invalid type specified for column '<columnName>'. The type of a column may not be changed.
42Z16	Only columns of type VARCHAR, CLOB, and BLOB may have their length altered.
42Z17	Invalid length specified for column '<columnName>'. Length must be greater than the current column length.
42Z18	Column '<columnName>' is part of a foreign key constraint '<constraintName>'. To alter the length of this column, you should drop the constraint first, perform the ALTER TABLE, and then recreate the constraint.
42Z19	Column '<columnName>' is being referenced by at least one foreign key constraint '<constraintName>'. To alter the length of this column, you should drop referencing constraints, perform the ALTER TABLE and then recreate the constraints.
42Z20	Column '<columnName>' cannot be made nullable. It is part of a primary key or unique constraint, which cannot have any nullable columns.
42Z20.S.1	Column '<columnName>' cannot be made nullable. It is part of a primary key, which cannot have any nullable columns.
42Z21	Invalid increment specified for identity focr column '<columnName>'. Increment cannot be zero.
42Z22	Invalid type specified for identity column '<columnName>'. The only valid types for identity columns are BIGINT, INT and SMALLINT.
42Z23	Attempt to modify an identity column '<columnName>'.
42Z24	Overflow occurred in identity value for column '<columnName>' in table '<tableName>'.
42Z25	INTERNAL ERROR identity counter. Update was called without arguments with current value \= NULL.
42Z26	A column, '<columnName>', with an identity default cannot be made nullable.
42Z27	A nullable column, '<columnName>', cannot be modified to have identity default.
42Z50	INTERNAL ERROR: Unable to generate code for <value>.
42Z53	INTERNAL ERROR: Type of activation to generate for node choice <value> is unknown.
42Z60	<value> not allowed unless database property <propertyName> has value '<value>'.

SQLSTATE	Message Text
42Z70	Binding directly to an XML value is not allowed; try using XMLPARSE.
42Z71	XML values are not allowed in top-level result sets; try using XMLSERIALIZE.
42Z72	Missing SQL/XML keyword(s) '<keywords>' at line <lineNumber>, column <columnNumber>.
42Z73	Invalid target type for XMLSERIALIZE: '<typeName>'.
42Z74	XML feature not supported: '<featureName>'.
42Z75	XML query expression must be a string literal.
42Z76	Multiple XML context items are not allowed.
42Z77	Context item must have type 'XML'; '<value>' is not allowed.
42Z79	Unable to determine the parameter type for XMLPARSE; try using a CAST.
42Z90	Class '<className>' does not return an updatable ResultSet.
42Z91	subquery
42Z92	repeatable read
42Z93	Constraints '<constraintName>' and '<constraintName>' have the same set of columns, which is not allowed.
42Z97	Renaming column '<columnName>' will cause check constraint '<constraintName>' to break.
42Z99	String or Hex literal cannot exceed 64K.
42Z9A	read uncommitted
42Z9B	The external virtual table interface does not support BLOB or CLOB columns. '<value>' column '<value>'.
42Z9D.S.1	Procedures that modify SQL data are not allowed in BEFORE triggers.
42Z9D	'<statement>' statements are not allowed in '<triggerName>' triggers.
42Z9E	Constraint '<constraintName>' is not a <value> constraint.
42Z9F	Too many indexes (<index>) on the table <tableName>. The limit is <number>.

SQLSTATE	Message Text
42ZA0	Statement too complex. Try rewriting the query to remove complexity. Eliminating many duplicate expressions or breaking up the query and storing interim results in a temporary table can often help resolve this error.
42ZA1	Invalid SQL in Batch: '<batch>'.
42ZA2	Operand of LIKE predicate with type <type> and collation <value> is not compatible with LIKE pattern operand with type <type> and collation <value>.
42ZA3	The table will have collation type <type> which is different than the collation of the schema <type> hence this operation is not supported .
42ZB1	Parameter style DERBY_JDBC_RESULT_SET is only allowed for table functions.
42ZB2	Table functions can only have parameter style DERBY_JDBC_RESULT_SET.
42ZB3	XML is not allowed as the datatype of a user-defined aggregate or of a column returned by a table function.
42ZB4	'<schemaName>.<functionName>' does not identify a table function.
42ZB5	Class '<className>' implements VTICosting but does not provide a public, no-arg constructor.
42ZB6	A scalar value is expected, not a row set returned by a table function.
42ZC0	Window '<windowName>' is not defined.
42ZC1	Only one window is supported.
42ZC2	Window function is illegal in this context: '<clauseName>' clause
42ZC3	A user defined aggregate may not have the name of an aggregate defined by the SQL Standard or the name of a builtin Derby function having one argument: '<aggregateName>'
42ZC4	User defined aggregate '<schemaName>.<aggregateName>' is bound to external class '<className>'. The parameter types of that class could not be resolved.
42ZC6	User defined aggregate '<schemaName>.<aggregateName>' was declared to have this input Java type: '<javaDataType>'. This does not extend the following actual bounding input Java type: '<javaDataType>'.
42ZC7	User defined aggregate '<schemaName>.<aggregateName>' was declared to have this return Java type: '<javaDataType>'. This does not extend the following actual bounding return Java type: '<javaDataType>'.

SQLSTATE	Message Text
42ZC8	Implementing class '<className>' for user defined aggregate '<schemaName>.<aggregateName>' could not be instantiated or was malformed. Detailed message follows: <detailedMessage>
43001	The truncate function was provided a null operand.
43002	The truncate function was provided an operand which it does not know how to handle: '<operand>'. It requires a DATE, TIMESTAMP, INTEGER or DECIMAL type.
43003	The truncate function expects a right-side argument of type CHAR for an operand of type DATE or TIMESTAMP but got: '<truncValue>'.
43004	The truncate function expects a right-side argument of type INTEGER for an operand of type DECIMAL but got: '<truncValue>'.
43005	The truncate function got an invalid right-side trunc value for operand type DATE: '<truncValue>'.
43006	The truncate function got an unknown right-side trunc value for operand type '<operand>': '<truncValue>'. Acceptable values are: '<acceptableValues>'.
44001	<dateOrTimestamp>s cannot be multiplied or divided. The operation is undefined.
44002	<dateOrTimestamp>s cannot be added. The operation is undefined.
44003	Timestamp '<dateOrTimestamp>' is out of range (~ from 21 Sep 1677 00:12:44 GMT to 11 Apr 2262 23:47:16 GMT).

## Error Class 57: DRDA Network Protocol: Execution Failure

### Error Class 57: DRDA Network Protocol: Execution Failure

SQLSTATE	Message Text
57017.C	There is no available conversion for the source code page, <codePage>, to the target code page, <codePage>. The connection has been terminated.

## Error Class 58: DRDA Network Protocol: Protocol Error

### Error Class 58: DRDA Network Protocol: Protocol Error

SQLSTATE	Message Text
58009.C.10	Network protocol exception: only one of the VCM, VCS length can be greater than 0. The connection has been terminated.
58009.C.11	The connection was terminated because the encoding is not supported.
58009.C.12	Network protocol exception: actual code point, <value>, does not match expected code point, <value>. The connection has been terminated.
58009.C.13	Network protocol exception: DDM collection contains less than 4 bytes of data. The connection has been terminated.
58009.C.14	Network protocol exception: collection stack not empty at end of same id chain parse. The connection has been terminated.
58009.C.15	Network protocol exception: DSS length not 0 at end of same id chain parse. The connection has been terminated.
58009.C.16	Network protocol exception: DSS chained with same id at end of same id chain parse. The connection has been terminated.
58009.C.17	Network protocol exception: end of stream prematurely reached while reading InputStream, parameter #<value>. The connection has been terminated.
58009.C.18	Network protocol exception: invalid FDOCA LID. The connection has been terminated.
58009.C.19	Network protocol exception: SECTKN was not returned. The connection has been terminated.
58009.C.20	Network protocol exception: only one of NVCM, NVCS can be non-null. The connection has been terminated.
58009.C.21	Network protocol exception: SCLDTA length, <length>, is invalid for RDBNAM. The connection has been terminated.
58009.C.7	Network protocol exception: SCLDTA length, <length>, is invalid for RDBCOLID. The connection has been terminated.
58009.C.8	Network protocol exception: SCLDTA length, <length>, is invalid for PKGID. The connection has been terminated.
58009.C.9	Network protocol exception: PKGNAMCSN length, <length>, is invalid at SQLAM <value>. The connection has been terminated.

SQLSTATE	Message Text
58010.C	A network protocol error was encountered. A connection could not be established because the manager <value> at level <value> is not supported by the server.
58014.C	The DDM command 0x<value> is not supported. The connection has been terminated.
58015.C	The DDM object 0x<value> is not supported. The connection has been terminated.
58016.C	The DDM parameter 0x<value> is not supported. The connection has been terminated.
58017.C	The DDM parameter value 0x<value> is not supported. An input host variable may not be within the range the server supports. The connection has been terminated.



# Error Class XBCA: CacheService

Error Class XBCA: CacheService

SQLSTATE	Message Text
XBCA0 . S	Cannot create new object with key <key> in <cache> cache. The object already exists in the cache.

# Error Class XBCM: ClassManager

Error Class XBCM: ClassManager

SQLSTATE	Message Text
XBCM1 . S	Java linkage error thrown during load of generated class <className>.
XBCM2 . S	Cannot create an instance of generated class <className>.
XBCM3 . S	Method <methodName>() does not exist in generated class <className>.
XBCM4 . S	Java class file format limit(s) exceeded: <value> in generated class <className>.
XBCM5 . S	This operation requires that the JVM level be at least <vmLevel>.

## Error Class XBCX: Cryptography

### Error Class XBCX: Cryptography

SQLSTATE	Message Text
XBCX0 . S	Exception from Cryptography provider. See next exception for details.
XBCX1 . S	Initializing cipher with illegal mode, must be either ENCRYPT or DECRYPT.
XBCX2 . S	Initializing cipher with a boot password that is too short. The password must be at least <number> characters long.
XBCX5 . S	Cannot change boot password to null.
XBCX6 . S	Cannot change boot password to a non-string serializable type.
XBCX7 . S	Wrong format for changing boot password. Format must be : old_boot_password, new_boot_password.
XBCX8 . S	Cannot change boot password for a non-encrypted database.
XBCX9 . S	Cannot change boot password for a read-only database.
XBCXA . S	Wrong boot password.
XBCXB . S	Bad encryption padding '<value>' or padding not specified. 'NoPadding' must be used.
XBCXC . S	Encryption algorithm '<algorithmName>' does not exist. Please check that the chosen provider '<providerName>' supports this algorithm.
XBCXD . S	The encryption algorithm cannot be changed after the database is created.
XBCXE . S	The encryption provider cannot be changed after the database is created.
XBCXF . S	The class '<className>' representing the encryption provider cannot be found.
XBCXG . S	The encryption provider '<providerName>' does not exist.
XBCXH . S	The encryptionAlgorithm '<algorithmName>' is not in the correct format. The correct format is algorithm/feedbackMode/NoPadding.
XBCXI . S	The feedback mode '<mode>' is not supported. Supported feedback modes are CBC, CFB, OFB and ECB.
XBCXJ . S	The application is using a version of the Java Cryptography Extension (JCE) earlier than 1.2.1. Please upgrade to JCE 1.2.1 and try the operation again.

SQLSTATE	Message Text
XBCXK.S	The given encryption key does not match the encryption key used when creating the database. Please ensure that you are using the correct encryption key and try again.
XBCXL.S	The verification process for the encryption key was not successful. This could have been caused by an error when accessing the appropriate file to do the verification process. See next exception for details.
XBCXM.S	The length of the external encryption key must be an even number.
XBCXN.S	The external encryption key contains one or more illegal characters. Allowed characters for a hexadecimal number are 0-9, a-f and A-F.
XBCXO.S	Cannot encrypt the database when there is a global transaction in the prepared state.
XBCXP.S	Cannot re-encrypt the database with a new boot password or an external encryption key when there is a global transaction in the prepared state.
XBCXQ.S	Cannot configure a read-only database for encryption.
XBCXR.S	Cannot re-encrypt a read-only database with a new boot password or an external encryption key .
XBCXS.S	Cannot configure a database for encryption, when database is in the log archive mode.
XBCXT.S	Cannot re-encrypt a database with a new boot password or an external encryption key, when database is in the log archive mode.
XBCXU.S	Encryption of an un-encrypted database failed: <failureMessage>
XBCXV.S	Encryption of an encrypted database with a new key or a new password failed: <failureMessage>
XBCXW.S	The message digest algorithm '<algorithmName>' is not supported by any of the available cryptography providers. Please install a cryptography provider that supports that algorithm, or specify another algorithm in the derby.authentication.builtin.algorithm property.

## Error Class XBM: Monitor

### Error Class XBM: Monitor

SQLSTATE	Message Text
XBM01.D	Startup failed due to an exception. See next exception for details.
XBM02.D	Startup failed due to missing functionality for <value>. Please ensure your classpath includes the correct Splice software.
XBM05.D	Startup failed due to missing product version information for <value>.
XBM06.D	Startup failed. An encrypted database cannot be accessed without the correct boot password.
XBM07.D	Startup failed. Boot password must be at least 8 bytes long.
XBM08.D	Could not instantiate <value> StorageFactory class <value>.
XBM0A.D	The database directory '<directoryName>' exists. However, it does not contain the expected '<servicePropertiesName>' file. Perhaps Splice was brought down in the middle of creating this database. You may want to delete this directory and try creating the database again.
XBM0B.D	Failed to edit/write service properties file: <errorMessage>
XBM0C.D	Missing privilege for operation '<operation>' on file '<path>': <errorMessage>
XBM0G.D	Failed to start encryption engine. Please make sure you are running Java 2 and have downloaded an encryption provider such as jce and put it in your class path.
XBM0H.D	Directory <directoryName> cannot be created.
XBM0I.D	Directory <directoryName> cannot be removed.
XBM0J.D	Directory <directoryName> already exists.
XBM0K.D	Unknown sub-protocol for database name <databaseName>.
XBM0L.D	Specified authentication scheme class <className> does implement the authentication interface <interfaceName>.
XBM0M.D	Error creating an instance of a class named '<className>'. This class name was the value of the derby.authentication.provider property and was expected to be the name of an application-supplied implementation of com.splicemachine.db.authentication.UserAuthenticator. The underlying problem was: <detail>
XBM0N.D	JDBC Driver registration with java.sql.DriverManager failed. See next exception for details.

SQLSTATE	Message Text
XBM0P.D	Service provider is read-only. Operation not permitted.
XBM0Q.D	File <fileName> not found. Please make sure that backup copy is the correct one and it is not corrupted.
XBM0R.D	Unable to remove File <fileName>.
XBM0S.D	Unable to rename file '<fileName>' to '<fileName>'
XBM0T.D	Ambiguous sub-protocol for database name <databaseName>.
XBM0U.S	No class was registered for identifier <identifierName>.
XBM0V.S	An exception was thrown while loading class <className> registered for identifier <identifierName>.
XBM0W.S	An exception was thrown while creating an instance of class <className3> registered for identifier <identifierName>.
XBM0X.D	Supplied territory description '<value>' is invalid, expecting In[_CO[_variant]] In=lower-case two-letter ISO-639 language code, CO=upper-case two-letter ISO-3166 country codes, see java.util.Locale.
XBM03.D	Supplied value '<value>' for collation attribute is invalid, expecting UCS_BASIC or TERRITORY_BASED.
XBM04.D	Collator support not available from the JVM for the database's locale '<value>'.
XBM0Y.D	Backup database directory <directoryName> not found. Please make sure that the specified backup path is right.
XBM0Z.D	Unable to copy file '<fileName>' to '<fileName>'. Please make sure that there is enough space and permissions are correct.

## Error Class XCL: Execution exceptions

### Error Class XCL: Execution exceptions

SQLSTATE	Message Text
XCL01.S	Result set does not return rows. Operation <operationName> not permitted.
XCL05.S	Activation closed, operation <operationName> not permitted.
XCL07.S	Cursor '<cursorName>' is closed. Verify that autocommit is OFF.
XCL08.S	Cursor '<cursorName>' is not on a row.
XCL09.S	An Activation was passed to the '<methodName>' method that does not match the PreparedStatement.
XCL10.S	A PreparedStatement has been recompiled and the parameters have changed. If you are using JDBC you must prepare the statement again.
XCL12.S	An attempt was made to put a data value of type '<datatypeName>' into a data value of type '<datatypeName>'.
XCL13.S	The parameter position '<parameterPosition>' is out of range. The number of parameters for this prepared statement is '<number>'.
XCL14.S	The column position '<columnPosition>' is out of range. The number of columns for this ResultSet is '<number>'.
XCL15.S	A ClassCastException occurred when calling the compareTo() method on an object '<object>'. The parameter to compareTo() is of class '<className>'.
XCL16.S	ResultSet not open. Operation '<operation>' not permitted. Verify that autocommit is OFF.
XCL18.S	Stream or LOB value cannot be retrieved more than once
XCL19.S	Missing row in table '<tableName>' for key '<key>'.
XCL20.S	Catalogs at version level '<versionNumber>' cannot be upgraded to version level '<versionNumber>'.
XCL21.S	You are trying to execute a Data Definition statement (CREATE, DROP, or ALTER) while preparing a different statement. This is not allowed. It can happen if you execute a Data Definition statement from within a static initializer of a Java class that is being used from within a SQL statement.
XCL22.S	Parameter <parameterName> cannot be registered as an OUT parameter because it is an IN parameter.

SQLSTATE	Message Text
XCL23.S	SQL type number '<type>' is not a supported type by registerOutParameter().
XCL24.S	Parameter <parameterName> appears to be an output parameter, but it has not been so designated by registerOutParameter(). If it is not an output parameter, then it has to be set to type <type>.
XCL25.S	Parameter <parameterName> cannot be registered to be of type <type> because it maps to type <type> and they are incompatible.
XCL26.S	Parameter <parameterName> is not an output parameter.
XCL27.S	Return output parameters cannot be set.
XCL30.S	An IOException was thrown when reading a '<value>' from an InputStream.
XCL31.S	Statement closed.
XCL33.S	The table cannot be defined as a dependent of table <tableName> because of delete rule restrictions. (The relationship is self-referencing and a self-referencing relationship already exists with the SET NULL delete rule.)
XCL34.S	The table cannot be defined as a dependent of table <tableName> because of delete rule restrictions. (The relationship forms a cycle of two or more tables that cause the table to be delete-connected to itself (all other delete rules in the cycle would be CASCADE)).
XCL35.S	The table cannot be defined as a dependent of table <tableName> because of delete rule restrictions. (The relationship causes the table to be delete-connected to the indicated table through multiple relationships and the delete rule of the existing relationship is SET NULL.).
XCL36.S	The delete rule of foreign key must be <value>. (The referential constraint is self-referencing and an existing self-referencing constraint has the indicated delete rule (NO ACTION, RESTRICT or CASCADE).)
XCL37.S	The delete rule of foreign key must be <value>. (The referential constraint is self-referencing and the table is dependent in a relationship with a delete rule of CASCADE.)
XCL38.S	the delete rule of foreign key must be <ruleName>. (The relationship would cause the table to be delete-connected to the same table through multiple relationships and such relationships must have the same delete rule (NO ACTION, RESTRICT or CASCADE).)
XCL39.S	The delete rule of foreign key cannot be CASCADE. (A self-referencing constraint exists with a delete rule of SET NULL, NO ACTION or RESTRICT.)
XCL40.S	The delete rule of foreign key cannot be CASCADE. (The relationship would form a cycle that would cause a table to be delete-connected to itself. One of the existing delete rules in the cycle is not CASCADE, so this relationship may be definable if the delete rule is not CASCADE.)



SQLSTATE	Message Text
XCL41.S	the delete rule of foreign key can not be CASCADE. (The relationship would cause another table to be delete-connected to the same table through multiple paths with different delete rules or with delete rule equal to SET NULL.)
XCL42.S	CASCADE
XCL43.S	SET NULL
XCL44.S	RESTRICT
XCL45.S	NO ACTION
XCL46.S	SET DEFAULT
XCL47.S	Use of '<value>' requires database to be upgraded from version <versionNumber> to version <versionNumber> or later.
XCL48.S	TRUNCATE TABLE is not permitted on '<value>' because unique/primary key constraints on this table are referenced by enabled foreign key constraints from other tables.
XCL49.S	TRUNCATE TABLE is not permitted on '<value>' because it has an enabled DELETE trigger (<value>).
XCL50.S	Upgrading the database from a previous version is not supported. The database being accessed is at version level '<versionNumber>', this software is at version level '<versionNumber>'.
XCL51.S	The requested function can not reference tables in SESSION schema.
XCL52.S	The statement has been cancelled or timed out.

# Error Class XCW: Upgrade unsupported

Error Class XCW: Upgrade unsupported

SQLSTATE	Message Text
XCW00.D	Unsupported upgrade from '<value>' to '<value>'.

# Error Class XCX: Internal Utility Errors

Error Class XCX: Internal Utility Errors

SQLSTATE	Message Text
XCXA0 .S	Invalid identifier: '<value>'.
XCXB0 .S	Invalid database classpath: '<classpath>'.
XCXC0 .S	Invalid id list.
XCXE0 .S	You are trying to do an operation that uses the territory of the database, but the database does not have a territory.

## Error Class XCY: Splice Property Exceptions

Error Class XCY: Splice Property Exceptions

SQLSTATE	Message Text
XCY00.S	Invalid value for property '<value>'='<value>'.
XCY02.S	The requested property change is not supported '<value>'='<value>'.
XCY03.S	Required property '<propertyName>' has not been set.
XCY04.S	Invalid syntax for optimizer overrides. The syntax should be -- SPLICE-PROPERTIES propertyName = value [, propertyName = value]*
XCY05.S.2	Invalid setting of the derby.authentication.provider property. This property is already set to enable NATIVE authentication and cannot be changed.
XCY05.S.3	Invalid setting of the derby.authentication.provider property. To enable NATIVE authentication, use the SYSCS_UTIL.SYSCS_CREATE_USER procedure to store credentials for the database owner.

# Error Class XCZ: com.splicemachine.db.database.UserUtility

Error Class XCZ:  
com.splicemachine.db.database.UserUtility

SQLSTATE	Message Text
XCZ00.S	Unknown permission '<permissionName>'.
XCZ01.S	Unknown user '<authorizationName>'.
XCZ02.S	Invalid parameter '<value>'='<value>'.

# Error Class XD00: Dependency Manager

Error Class XD00: Dependency Manager

SQLSTATE	Message Text
XD003.S	Unable to restore dependency from disk. DependableFinder = '<value>'. Further information: '<value>'.
XD004.S	Unable to store dependencies.

## Error Class XIE: Import/Export Exceptions

Error Class XIE: Import/Export Exceptions

SQLSTATE	Message Text
XIE01.S	Connection was null.
XIE03.S	Data found on line <lineNumber> for column <columnName> after the stop delimiter.
XIE04.S	Data file not found: <fileName>
XIE05.S	Data file cannot be null.
XIE06.S	Entity name was null.
XIE07.S	Field and record separators cannot be substrings of each other.
XIE08.S	There is no column named: <columnName>.
XIE09.S	The total number of columns in the row is: <number>.
XIE0A.S	Number of columns in column definition, <columnName>, differ from those found in import file <type>.
XIE0B.S	Column '<columnName>' in the table is of type <type>, it is not supported by the import/export feature.
XIE0C.S	Illegal <delimiter> delimiter character '<character>'.
XIE0D.S	Cannot find the record separator on line <lineNumber>.
XIE0E.S	Read endOfFile at unexpected place on line <lineNumber>.
XIE0F.S	Character delimiter cannot be the same as the column delimiter.
XIE0I.S	An IOException occurred while writing data to the file.
XIE0J.S	A delimiter is not valid or is used more than once.
XIE0K.S	The period was specified as a character string delimiter.
XIE0M.S	Table '<tableName>' does not exist.
XIE0N.S	An invalid hexadecimal string '<hexString>' detected in the import file.
XIE0P.S	Lob data file <fileName> referenced in the import file not found.

SQLSTATE	Message Text
XIE0Q.S	Lob data file name cannot be null.
XIE0R.S	Import error on line <lineNumber> of file <fileName>: <details>
XIE10.S	Import error during reading source file <fileName> : <details>
XIE11.S	SuperCSVReader error during Import : <details>
XIE12.S	There was <details> RegionServer failures during a write with WAL disabled, the transaction has to rollback to avoid data loss.
XIE0S.S	The export operation was not performed, because the specified output file (<fileName>) already exists. Export processing will not overwrite an existing file, even if the process has permissions to write to that file, due to security concerns, and to avoid accidental file damage. Please either change the output file name in the export procedure arguments to specify a file which does not exist, or delete the existing file, then retry the export operation.
XIE0T.S	The export operation was not performed, because the specified large object auxiliary file (<fileName>) already exists. Export processing will not overwrite an existing file, even if the process has permissions to write to that file, due to security concerns, and to avoid accidental file damage. Please either change the large object auxiliary file name in the export procedure arguments to specify a file which does not exist, or delete the existing file, then retry the export operation.
XIE0U.S	The export operation was not performed, because the specified parameter (replicationCount) is less than or equal to zero.
XIE0X.S	The export operation was not performed, because value of the specified parameter (<paramName>) is wrong.



## Error Class XJ: Connectivity Errors

Error Class XJ: Connectivity Errors

SQLSTATE	Message Text
XJ004.C	Database '<databaseName>' not found.
XJ008.S	Cannot rollback or release a savepoint when in auto-commit mode.
XJ009.S	Use of CallableStatement required for stored procedure call or use of output parameters: <value>
XJ010.S	Cannot issue savepoint when autoCommit is on.
XJ011.S	Cannot pass null for savepoint name.
XJ012.S	'<value>' already closed.
XJ013.S	No ID for named savepoints.
XJ014.S	No name for un-named savepoints.
XJ015.M	Splice system shutdown.
XJ016.S	Method '<methodName>' not allowed on prepared statement.
XJ017.S	No savepoint command allowed inside the trigger code.
XJ018.S	Column name cannot be null.
XJ020.S	Object type not convertible to TYPE '<typeName>', invalid java.sql.Types value, or object was null.
XJ021.S	Type is not supported.
XJ022.S	Unable to set stream: '<name>'.
XJ023.S	Input stream did not have exact amount of data as the requested length.
XJ025.S	Input stream cannot have negative length.
XJ028.C	The URL '<urlValue>' is not properly formed.
XJ030.S	Cannot set AUTOCOMMIT ON when in a nested connection.
XJ040.C	Failed to start database '<databaseName>' with class loader <classLoader>, see the next exception for details.

SQLSTATE	Message Text
XJ041.C	Failed to create database '<databaseName>', see the next exception for details.
XJ042.S	'<value>' is not a valid value for property '<propertyName>'.
XJ044.S	'<value>' is an invalid scale.
XJ045.S	Invalid or (currently) unsupported isolation level, '<levelName>', passed to Connection.setTransactionIsolation(). The currently supported values are java.sql.Connection.TRANSACTION_SERIALIZABLE, java.sql.Connection.TRANSACTION_REPEATABLE_READ, java.sql.Connection.TRANSACTION_READ_COMMITTED, and java.sql.Connection.TRANSACTION_READ_UNCOMMITTED.
XJ048.C	Conflicting boot attributes specified: <attributes>
XJ049.C	Conflicting create attributes specified.
XJ04B.S	Batch cannot contain a command that attempts to return a result set.
XJ04C.S	CallableStatement batch cannot contain output parameters.
XJ056.S	Cannot set AUTOCOMMIT ON when in an XA connection.
XJ057.S	Cannot commit a global transaction using the Connection, commit processing must go thru XAResource interface.
XJ058.S	Cannot rollback a global transaction using the Connection, commit processing must go thru XAResource interface.
XJ059.S	Cannot close a connection while a global transaction is still active.
XJ05B.C	JDBC attribute '<attributeName>' has an invalid value '<value>', valid values are '<value>'.
XJ05C.S	Cannot set holdability ResultSet.HOLD_CURSORS_OVER_COMMIT for a global transaction.
XJ061.S	The '<methodName>' method is only allowed on scroll cursors.
XJ062.S	Invalid parameter value '<value>' for ResultSet.setFetchSize(int rows).
XJ063.S	Invalid parameter value '<value>' for Statement.setMaxRows(int maxRows). Parameter value must be >= 0.
XJ064.S	Invalid parameter value '<value>' for setFetchDirection(int direction).
XJ065.S	Invalid parameter value '<value>' for Statement.setFetchSize(int rows).
XJ066.S	Invalid parameter value '<value>' for Statement.setMaxFieldSize(int max).

SQLSTATE	Message Text
XJ067.S	SQL text pointer is null.
XJ068.S	Only executeBatch and clearBatch allowed in the middle of a batch.
XJ069.S	No SetXXX methods allowed in case of USING execute statement.
XJ070.S	Negative or zero position argument '<argument>' passed in a Blob or Clob method.
XJ071.S	Negative length argument '<argument>' passed in a BLOB or CLOB method.
XJ072.S	Null pattern or searchStr passed in to a BLOB or CLOB position method.
XJ073.S	The data in this BLOB or CLOB is no longer available. The BLOB/CLOB's transaction may be committed, its connection closed or it has been freed.
XJ074.S	Invalid parameter value '<value>' for Statement.setQueryTimeout(int seconds).
XJ076.S	The position argument '<positionArgument>' exceeds the size of the BLOB/CLOB.
XJ077.S	Got an exception when trying to read the first byte/character of the BLOB/CLOB pattern using getBytes/getSubString.
XJ078.S	Offset '<value>' is either less than zero or is too large for the current BLOB/CLOB.
XJ079.S	The length specified '<number>' exceeds the size of the BLOB/CLOB.
XJ080.S	USING execute statement passed <number> parameters rather than <number>.
XJ081.C	Conflicting create/restore/recovery attributes specified.
XJ081.S	Invalid value '<value>' passed as parameter '<parameterName>' to method '<methodName>'
XJ085.S	Stream has already been read and end-of-file reached and cannot be re-used.
XJ086.S	This method cannot be invoked while the cursor is not on the insert row or if the concurrency of this ResultSet object is CONCUR_READ_ONLY.
XJ087.S	Sum of position('<pos>') and length('<length>') is greater than the size of the LOB plus one.
XJ088.S	Invalid operation: wasNull() called with no data retrieved.
XJ090.S	Invalid parameter: calendar is null.
XJ091.S	Invalid argument: parameter index <indexNumber> is not an OUT or INOUT parameter.
XJ093.S	Length of BLOB/CLOB, <number>, is too large. The length cannot exceed <number>.

SQLSTATE	Message Text
XJ095.S	An attempt to execute a privileged action failed.
XJ096.S	A resource bundle could not be found in the <packageName> package for <value>
XJ097.S	Cannot rollback or release a savepoint that was not created by this connection.
XJ098.S	The auto-generated keys value <value> is invalid
XJ099.S	The Reader/Stream object does not contain length characters
XJ100.S	The scale supplied by the registerOutParameter method does not match with the setter method. Possible loss of precision!
XJ103.S	Table name can not be null
XJ104.S	Shared key length is invalid: <value>.
XJ105.S	DES key has the wrong length, expected length <number>, got length <number>.
XJ106.S	No such padding
XJ107.S	Bad Padding
XJ108.S	Illegal Block Size
XJ110.S	Primary table name can not be null
XJ111.S	Foreign table name can not be null
XJ112.S	Security exception encountered, see next exception for details.
XJ113.S	Unable to open file <fileName> : <error>
XJ114.S	Invalid cursor name '<cursorName>'
XJ115.S	Unable to open resultSet with requested holdability <value>.
XJ116.S	No more than <number> commands may be added to a single batch.
XJ117.S	Batching of queries not allowed by J2EE compliance.
XJ118.S	Query batch requested on a non-query statement.
XJ121.S	Invalid operation at current cursor position.
XJ122.S	No updateXXX methods were called on this row.

SQLSTATE	Message Text
XJ123.S	This method must be called to update values in the current row or the insert row.
XJ124.S	Column not updatable.
XJ125.S	This method should only be called on ResultSet objects that are scrollable (type TYPE_SCROLL_INSENSITIVE).
XJ126.S	This method should not be called on sensitive dynamic cursors.
XJ128.S	Unable to unwrap for '<value>'
XJ200.S	Exceeded maximum number of sections <value>
XJ202.S	Invalid cursor name '<cursorName>'.
XJ203.S	Cursor name '<cursorName>' is already in use
XJ204.S	Unable to open result set with requested holdability <holdValue>.
XJ206.S	SQL text '<value>' has no tokens.
XJ207.S	executeQuery method can not be used for update.
XJ208.S	Non-atomic batch failure. The batch was submitted, but at least one exception occurred on an individual member of the batch. Use getNextException() to retrieve the exceptions for specific batched elements.
XJ209.S	The required stored procedure is not installed on the server.
XJ210.S	The load module name for the stored procedure on the server is not found.
XJ211.S	Non-recoverable chain-breaking exception occurred during batch processing. The batch is terminated non-atomically.
XJ212.S	Invalid attribute syntax: <attributeSyntax>
XJ213.C	The traceLevel connection property does not have a valid format for a number.
XJ214.S	An IO Error occurred when calling free() on a CLOB or BLOB.
XJ215.S	You cannot invoke other java.sql.Clob/java.sql.Blob methods after calling the free() method or after the Blob/Clob's transaction has been committed or rolled back.
XJ216.S	The length of this BLOB/CLOB is not available yet. When a BLOB or CLOB is accessed as a stream, the length is not available until the entire stream has been processed.
XJ217.S	The locator that was supplied for this CLOB/BLOB is invalid



# Error Class XK: Security Exceptions

Error Class XK: Security Exceptions

SQLSTATE	Message Text
XK000.S	The security policy could not be reloaded: <reason>
XK001.S	Username not found in SYS.SYSUSERS.

## Error Class XN: Network Client Exceptions

Error Class XN: Network Client Exceptions

SQLSTATE	Message Text
XN001.S	Connection reset is not allowed when inside a unit of work.
XN008.S	Query processing has been terminated due to an error on the server.
XN009.S	Error obtaining length of BLOB/CLOB object, exception follows.
XN010.S	Procedure name can not be null.
XN011.S	Procedure name length <number> is not within the valid range of 1 to <number>.
XN012.S	On <operatingSystemName> platforms, XA supports version <versionNumber> and above, this is version <versionNumber>
XN013.S	Invalid scroll orientation.
XN014.S	Encountered an Exception while reading from the stream specified by parameter #<value>. The remaining data expected by the server has been filled with 0x0. The Exception had this message: <messageText>.
XN015.S	Network protocol error: the specified size of the InputStream, parameter #<value>, is less than the actual InputStream length.
XN016.S	Encountered an Exception while trying to verify the length of the stream specified by parameter #<value>. The Exception had this message: <messageText>.
XN017.S	End of stream prematurely reached while reading the stream specified by parameter #<value>. The remaining data expected by the server has been filled with 0x0.
XN018.S	Network protocol error: the specified size of the Reader, parameter #<value>, is less than the actual InputStream length.
XN019.S	Error executing a <value>, server returned <value>.
XN020.S	Error marshalling or unmarshalling a user defined type: <messageDetail>
XN021.S	An object of type <sourceClassName> cannot be cast to an object of type <targetClassName>.



## Error Class XRE: Replication Exceptions

Error Class XRE: Replication Exceptions

SQLSTATE	Message Text
XRE00	This LogFactory module does not support replication.
XRE01	The log received from the master is corrupted.
XRE02	Master and Slave at different versions. Unable to proceed with Replication.
XRE03	Unexpected replication error. See derby.log for details.
XRE04.C.1	Could not establish a connection to the peer of the replicated database '<dbname>' on address '<hostname>:<portname>'.
XRE04.C.2	Connection lost for replicated database '<dbname>'.
XRE05.C	The log files on the master and slave are not in synch for replicated database '<dbname>'. The master log instant is <masterfile>:<masteroffset>, whereas the slave log instant is <slavefile>:<slaveoffset>. This is FATAL for replication - replication will be stopped.
XRE06	The connection attempts to the replication slave for the database <dbname> exceeded the specified timeout period.
XRE07	Could not perform operation because the database is not in replication master mode.
XRE08	Replication slave mode started successfully for database '<dbname>'. Connection refused because the database is in replication slave mode.
XRE09.C	Cannot start replication slave mode for database '<dbname>'. The database has already been booted.
XRE10	Conflicting attributes specified. See reference manual for attributes allowed in combination with replication attribute '<attribute>'.
XRE11.C	Could not perform operation '<command>' because the database '<dbname>' has not been booted.
XRE12	Replication network protocol error for database '<dbname>'. Expected message type '<expectedtype>', but received type '<expectedtype>'.
XRE20.D	Failover performed successfully for database '<dbname>', the database has been shutdown.
XRE21.C	Error occurred while performing failover for database '<dbname>', Failover attempt was aborted.
XRE22.C	Replication master has already been booted for database '<dbname>'

SQLSTATE	Message Text
XRE23	Replication master cannot be started since unlogged operations are in progress, unfreeze to allow unlogged operations to complete and restart replication
XRE40	Could not perform operation because the database is not in replication slave mode.
XRE41.C	Replication operation 'failover' or 'stopSlave' refused on the slave database because the connection with the master is working. Issue the 'failover' or 'stopMaster' operation on the master database instead.
XRE42.C	Replicated database '<dbname>' shutdown.
XRE43	Unexpected error when trying to stop replication slave mode. To stop replication slave mode, use operation 'stopSlave' or 'failover'.

# Error Class XSAI: Store - access.protocol.interface

Error Class XSAI: Store - access.protocol.interface

SQLSTATE	Message Text
XSAI2.S	The conglomerate (<value>) requested does not exist.
XSAI3.S	Feature not implemented.

## Error Class XSAM: Store - AccessManager

Error Class XSAM: Store - AccessManager

SQLSTATE	Message Text
XSAM0 .S	Exception encountered while trying to boot module for '<value>'.
XSAM2 .S	There is no index or conglomerate with conglom id '<conglomID>' to drop.
XSAM3 .S	There is no index or conglomerate with conglom id '<conglomID>'.
XSAM4 .S	There is no sort called '<sortName>'.
XSAM5 .S	Scan must be opened and positioned by calling next() before making other calls.
XSAM6 .S	Record <recordNumber> on page <pageNumber> in container <containerName> not found.

# Error Class XSAS: Store - Sort

Error Class XSAS: Store - Sort

SQLSTATE	Message Text
XSAS0.S	A scan controller interface method was called which is not appropriate for a scan on a sort.
XSAS1.S	An attempt was made to fetch a row before the beginning of a sort or after the end of a sort.
XSAS3.S	The type of a row inserted into a sort does not match the sort's template.
XSAS6.S	Could not acquire resources for sort.

# Error Class XSAX: Store - access.protocol.XA statement

Error Class XSAX: Store - access.protocol.XA statement

SQLSTATE	Message Text
XSAX0 . S	XA protocol violation.
XSAX1 . S	An attempt was made to start a global transaction with an Xid of an existing global transaction.

## Error Class XSCB: Store - BTree

### Error Class XSCB: Store - BTree

SQLSTATE	Message Text
XSCB0 .S	Could not create container.
XSCB1 .S	Container <containerName> not found.
XSCB2 .S	The required property <propertyName> not found in the property list given to createConglomerate() for a btree secondary index.
XSCB3 .S	Unimplemented feature.
XSCB4 .S	A method on a btree open scan has been called prior to positioning the scan on the first row (i.e. no next() call has been made yet). The current state of the scan is (<value>).
XSCB5 .S	During logical undo of a btree insert or delete the row could not be found in the tree.
XSCB6 .S	Limitation: Record of a btree secondary index cannot be updated or inserted due to lack of space on the page. Use the parameters derby.storage.pageSize and/or derby.storage.pageReservedSpace to work around this limitation.
XSCB7 .S	An internal error was encountered during a btree scan - current_rh is null = <value>, position key is null = <value>.
XSCB8 .S	The btree conglomerate <value> is closed.
XSCB9 .S	Reserved for testing.

# Error Class XSCG0: Conglomerate

Error Class XSCG0: Conglomerate

SQLSTATE	Message Text
XSCG0 .S	Could not create a template.



## Error Class XSCH: Heap

Error Class XSCH: Heap

SQLSTATE	Message Text
XSCH0.S	Could not create container.
XSCH1.S	Container <containerName> not found.
XSCH4.S	Conglomerate could not be created.
XSCH5.S	In a base table there was a mismatch between the requested column number <number> and the maximum number of columns <number>.
XSCH6.S	The heap container with container id <containerID> is closed.
XSCH7.S	The scan is not positioned.
XSCH8.S	The feature is not implemented.

## Error Class XSDA: RawStore - Data.Generic statement

Error Class XSDA: RawStore - Data.Generic statement

SQLSTATE	Message Text
XSDA1 . S	An attempt was made to access an out of range slot on a page
XSDA2 . S	An attempt was made to update a deleted record
XSDA3 . S	Limitation: Record cannot be updated or inserted due to lack of space on the page. Use the parameters derby.storage.pageSize and/or derby.storage.pageReservedSpace to work around this limitation.
XSDA4 . S	An unexpected exception was thrown
XSDA5 . S	An attempt was made to undelete a record that is not deleted
XSDA6 . S	Column <columnName> of row is null, it needs to be set to point to an object.
XSDA7 . S	Restore of a serializable or SQLData object of class <className>, attempted to read more data than was originally stored
XSDA8 . S	Exception during restore of a serializable or SQLData object of class <className>
XSDA9 . S	Class not found during restore of a serializable or SQLData object of class <className>
XSDAA . S	Illegal time stamp <value>, either time stamp is from a different page or of incompatible implementation
XSDAB . S	cannot set a null time stamp
XSDAC . S	Attempt to move either rows or pages from one container to another.
XSDAD . S	Attempt to move zero rows from one page to another.
XSDAE . S	Can only make a record handle for special record handle id.
XSDAF . S	Using special record handle as if it were a normal record handle.
XSDAG . S	The allocation nested top transaction cannot open the container.
XSDAI . S	Page <page> being removed is already locked for deallocation.
XSDAJ . S	Exception during write of a serializable or SQLData object
XSDAK . S	Wrong page is gotten for record handle <value>.

SQLSTATE	Message Text
XSDAL.S	Record handle <value> unexpectedly points to overflow page.
XSDAM.S	Exception during restore of a SQLData object of class <className>. The specified class cannot be instantiated.
XSDAN.S	Exception during restore of a SQLData object of class <className>. The specified class encountered an illegal access exception.
XSDAO.S	Internal error: page <pageNumber> attempted latched twice.

## Error Class XSDB: RawStore - Data.Generic transaction

### Error Class XSDB: RawStore - Data.Generic transaction

SQLSTATE	Message Text
XSDB0.D	Unexpected exception on in-memory page <page>
XSDB1.D	Unknown page format at page <page>
XSDB2.D	Unknown container format at container <containerName> : <value>
XSDB3.D	Container information cannot change once written: was <value>, now <value>
XSDB4.D	Page <page> is at version <versionNumber>, the log file contains change version <versionNumber>, either there are log records of this page missing, or this page did not get written out to disk properly.
XSDB5.D	Log has change record on page <page>, which is beyond the end of the container.
XSDB6.D	Another instance of Splice may have already booted the database <databaseName>.
XSDB7.D	WARNING: Splice (instance <value>) is attempting to boot the database <databaseName> even though Splice (instance <value>) may still be active. Only one instance of Splice should boot a database at a time. Severe and non-recoverable corruption can result and may have already occurred.
XSDB8.D	WARNING: Splice (instance <value>) is attempting to boot the database <databaseName> even though Splice (instance <value>) may still be active. Only one instance of Splice should boot a database at a time. Severe and non-recoverable corruption can result if 2 instances of Splice boot on the same database at the same time. The derby.database.forceDatabaseLock=true property has been set, so the database will not boot until the db.lck is no longer present. Normally this file is removed when the first instance of Splice to boot on the database exits, but it may be left behind in some shutdowns. It will be necessary to remove the file by hand in that case. It is important to verify that no other VM is accessing the database before deleting the db.lck file by hand.
XSDB9.D	Stream container <containerName> is corrupt.
XSDBA.D	Attempt to allocate object <object> failed.
XSDBB.D	Unknown page format at page <page>, page dump follows: <value>
XSDBC.D	Write of container information to page 0 of container <container> failed. See nested error for more information.

## Error Class XSDF: RawStore - Data.Filesystem statement

Error Class XSDF: RawStore - Data.Filesystem statement

SQLSTATE	Message Text
XSDF0.S	Could not create file <fileName> as it already exists.
XSDF1.S	Exception during creation of file <fileName> for container
XSDF2.S	Exception during creation of file <fileName> for container, file could not be removed. The exception was: <value>.
XSDF3.S	Cannot create segment <segmentName>.
XSDF4.S	Exception during remove of file <fileName> for dropped container, file could not be removed <value>.
XSDF6.S	Cannot find the allocation page <page>.
XSDF7.S	Newly created page failed to be latched <value>
XSDF8.S	Cannot find page <page> to reuse.
XSDFB.S	Operation not supported by a read only database
XSDFD.S	Different page image read on 2 I/Os on Page <page>, first image has incorrect checksum, second image has correct checksum. Page images follows: <value> <value>
XSDFE.S	The requested operation failed due to an unexpected exception.
XSDFH.S	Cannot backup the database, got an I/O Exception while writing to the backup container file <fileName>.
XSDFI.S	Error encountered while trying to write data to disk during database recovery. Check that the database disk is not full. If it is then delete unnecessary files, and retry connecting to the database. It is also possible that the file system is read only, or the disk has failed, or some other problem with the media. System encountered error while processing page <page>.

## Error Class XSDG: RawStore - Data.Filesystem database

### Error Class XSDG: RawStore - Data.Filesystem database

SQLSTATE	Message Text
XSDG0.D	Page <page> could not be read from disk.
XSDG1.D	Page <page> could not be written to disk, please check if the disk is full, or if a file system limit, such as a quota or a maximum file size, has been reached.
XSDG2.D	Invalid checksum on Page <page>, expected=<value>, on-disk version=<value>, page dump follows: <value>
XSDG3.D	Meta-data for <containerName> could not be accessed to <type> <file>
XSDG5.D	Database is not in create mode when createFinished is called.
XSDG6.D	Data segment directory not found in <value> backup during restore. Please make sure that backup copy is the right one and it is not corrupted.
XSDG7.D	Directory <directoryName> could not be removed during restore. Please make sure that permissions are correct.
XSDG8.D	Unable to copy directory '<directoryName>' to '<value>' during restore. Please make sure that there is enough space and permissions are correct.
XSDG9.D	Splice thread received an interrupt during a disk I/O operation, please check your application for the source of the interrupt.

## Error Class XSLA: RawStore - Log.Generic database exceptions

### Error Class XSLA: RawStore - Log.Generic database exceptions

SQLSTATE	Message Text
XSLA0.D	Cannot flush the log file to disk <value>.
XSLA1.D	Log Record has been sent to the stream, but it cannot be applied to the store (Object <object>). This may cause recovery problems also.
XSLA2.D	System will shutdown, got I/O Exception while accessing log file.
XSLA3.D	Log Corrupted, has invalid data in the log stream.
XSLA4.D	Cannot write to the log, most likely the log is full. Please delete unnecessary files. It is also possible that the file system is read only, or the disk has failed, or some other problems with the media.
XSLA5.D	Cannot read log stream for some reason to rollback transaction <transactionID>.
XSLA6.D	Cannot recover the database.
XSLA7.D	Cannot redo operation <operation> in the log.
XSLA8.D	Cannot rollback transaction <value>, trying to compensate <value> operation with <value>
XSLAA.D	The store has been marked for shutdown by an earlier exception.
XSLAB.D	Cannot find log file <logfileName>, please make sure your logDevice property is properly set with the correct path separator for your platform.
XSLAC.D	Database at <value> have incompatible format with the current version of software, it may have been created by or upgraded by a later version.
XSLAD.D	log Record at instant <value> in log file <logfileName> corrupted. Expected log record length <value>, real length <value>.
XSLAE.D	Control file at <value> cannot be written or updated.
XSLAF.D	A Read Only database was created with dirty data buffers.
XSLAH.D	A Read Only database is being updated.
XSLAI.D	Cannot log the checkpoint log record
XSLAJ.D	The logging system has been marked to shut down due to an earlier problem and will not allow any more operations until the system shuts down and restarts.

SQLSTATE	Message Text
XSLAK.D	Database has exceeded largest log file number <value>.
XSLAL.D	log record size <value> exceeded the maximum allowable log file size <number>. Error encountered in log file <logfileName>, position <value>.
XSLAM.D	Cannot verify database format at <value> due to IOException.
XSLAN.D	Database at <value> has an incompatible format with the current version of the software. The database was created by or upgraded by version <versionNumber>.
XSLAO.D	Recovery failed unexpected problem <value>.
XSLAP.D	Database at <value> is at version <versionNumber>. Beta databases cannot be upgraded,
XSLAQ.D	cannot create log file at directory <directoryName>.
XSLAR.D	Unable to copy log file '<logfileName>' to '<value>' during restore. Please make sure that there is enough space and permissions are correct.
XSLAS.D	Log directory <directoryName> not found in backup during restore. Please make sure that backup copy is the correct one and it is not corrupted.
XSLAT.D	The log directory '<directoryName>' exists. The directory might belong to another database. Check that the location specified for the logDevice attribute is correct.



## Error Class XSLB: RawStore - Log.Generic statement exceptions

Error Class XSLB: RawStore - Log.Generic statement exceptions

SQLSTATE	Message Text
XSLB1.S	Log operation <logOperation> encounters error writing itself out to the log stream, this could be caused by an errant log operation or internal log buffer full due to excessively large log operation.
XSLB2.S	Log operation <logOperation> logging excessive data, it filled up the internal log buffer.
XSLB4.S	Cannot find truncationLWM <value>.
XSLB5.S	Illegal truncationLWM instant <value> for truncation point <value>. Legal range is from <value> to <value>.
XSLB6.S	Trying to log a 0 or -ve length log Record.
XSLB8.S	Trying to reset a scan to <value>, beyond its limit of <value>.
XSLB9.S	Cannot issue any more change, log factory has been stopped.

## Error Class XSRS: RawStore - protocol.Interface statement

Error Class XSRS: RawStore - protocol.Interface statement

SQLSTATE	Message Text
XSRS0 . S	Cannot freeze the database after it is already frozen.
XSRS1 . S	Cannot backup the database to <value>, which is not a directory.
XSRS4 . S	Error renaming file (during backup) from <value> to <value>.
XSRS5 . S	Error copying file (during backup) from <path> to <path>.
XSRS6 . S	Cannot create backup directory <directoryName>.
XSRS7 . S	Backup caught unexpected exception.
XSRS8 . S	Log Device can only be set during database creation time, it cannot be changed on the fly.
XSRS9 . S	Record <recordName> no longer exists
XSRSA . S	Cannot backup the database when unlogged operations are uncommitted. Please commit the transactions with backup blocking operations.
XSRSB . S	Backup cannot be performed in a transaction with uncommitted unlogged operations.
XSRSC . S	Cannot backup the database to <directoryLocation>, it is a database directory.
XSRSD . S	Database backup is disabled. Contact your Splice Machine representative to enable.
XSRSE . S	Unable to enable the enterprise Manager. Enterprise services are disabled. Contact your Splice Machine representative to enable.
XSRSF . S	LDAP authentication is disabled. Contact your Splice Machine representative to enable.
XSRSG . S	SpliceMachine Enterprise services are disabled and so will not run on an encrypted host. Contact your Splice Machine representative to enable.

# Error Class XSTA2: XACT\_TRANSACTION\_ACTIVE

Error Class XSTA2: XACT\_TRANSACTION\_ACTIVE

SQLSTATE	Message Text
XSTA2 . S	A transaction was already active, when attempt was made to make another transaction active.

## Error Class XSTB: RawStore - Transactions.Basic system

### Error Class XSTB: RawStore - Transactions.Basic system

SQLSTATE	Message Text
XSTB0.M	An exception was thrown during transaction abort.
XSTB2.M	Cannot log transaction changes, maybe trying to write to a read only database.
XSTB3.M	Cannot abort transaction because the log manager is null, probably due to an earlier error.
XSTB5.M	Creating database with logging disabled encountered unexpected problem.
XSTB6.M	Cannot substitute a transaction table with another while one is already in use.

# Error Class XXXXX: No SQLSTATE

Error Class XXXXX: No SQLSTATE

SQLSTATE	Message Text
XXXXX	Normal database session close.

## Error Class X0 - Execution exceptions

### Error Class X0: Execution exceptions

SQLSTATE	Message Text
X0A00.S	The select list mentions column '<columnName>' twice. This is not allowed in queries with GROUP BY or HAVING clauses. Try aliasing one of the conflicting columns to a unique name.
X0X02.S	Table '<tableName>' cannot be locked in '<mode>' mode.
X0X03.S	Invalid transaction state - held cursor requires same isolation level
X0X05.S	Table/View '<tableName>' does not exist.
X0X07.S	Cannot remove jar file '<fileName>' because it is on your derby.database.classpath '<classpath>'.
X0X0D.S	Invalid column array length '<columnArrayLength>'. To return generated keys, column array must be of length 1 and contain only the identity column.
X0X0E.S	Table '<columnName>' does not have an auto-generated column at column position '<tableName>'.
X0X0F.S	Table '<columnName>' does not have an auto-generated column named '<tableName>'.
X0X10.S	The USING clause returned more than one row; only single-row ResultSets are permissible.
X0X11.S	The USING clause did not return any results so no parameters can be set.
X0X13.S	Jar file '<fileName>' does not exist in schema '<schemaName>'.
X0X14.S	The file '<fileName>' does not exist.
X0X57.S	An attempt was made to put a Java value of type '<type>' into a SQL value, but there is no corresponding SQL type. The Java value is probably the result of a method call or field access.
X0X60.S	A cursor with name '<cursorName>' already exists.
X0X61.S	The values for column '<columnName>' in index '<indexName>' and table '<schemaName>.<tableName>' do not match for row location <location>. The value in the index is '<value>', while the value in the base table is '<value>'. The full index key, including the row location, is '<indexKey>'. The suggested corrective action is to recreate the index.
X0X62.S	Inconsistency found between table '<tableName>' and index '<indexName>'. Error when trying to retrieve row location '<rowLocation>' from the table. The full index key, including the row location, is '<indexKey>'. The suggested corrective action is to recreate the index.
X0X63.S	Got IOException '<value>'.

SQLSTATE	Message Text
X0X67.S	Columns of type '<type>' may not be used in CREATE INDEX, ORDER BY, GROUP BY, UNION, INTERSECT, EXCEPT or DISTINCT statements because comparisons are not supported for that type.
X0X81.S	<value> '<value>' does not exist.
X0X85.S	Index '<indexName>' was not created because '<indexType>' is not a valid index type.
X0X86.S	0 is an invalid parameter value for ResultSet.absolute(int row).
X0X87.S	ResultSet.relative(int row) cannot be called when the cursor is not positioned on a row.
X0X95.S	Operation '<operationName>' cannot be performed on object '<objectName>' because there is an open ResultSet dependent on that object.
X0X99.S	Index '<indexName>' does not exist.
X0Y16.S	'<value>' is not a view. If it is a table, then use DROP TABLE instead.
X0Y23.S	Operation '<operationName>' cannot be performed on object '<objectName>' because VIEW '<viewName>' is dependent on that object.
X0Y24.S	Operation '<operationName>' cannot be performed on object '<objectName>' because STATEMENT '<statement>' is dependent on that object.
X0Y25.S	Operation '<operationName>' cannot be performed on object '<objectName>' because <value> '<value>' is dependent on that object.
X0Y26.S	Index '<indexName>' is required to be in the same schema as table '<tableName>'.
X0Y28.S	Index '<indexName>' cannot be created on system table '<tableName>'. Users cannot create indexes on system tables.
X0Y29.S	Operation '<operationName>' cannot be performed on object '<objectName>' because TABLE '<tableName>' is dependent on that object.
X0Y30.S	Operation '<operationName>' cannot be performed on object '<objectName>' because ROUTINE '<routineName>' is dependent on that object.
X0Y32.S	<value> '<value>' already exists in <value> '<value>'.
X0Y38.S	Cannot create index '<indexName>' because table '<tableName>' does not exist.
X0Y41.S	Constraint '<constraintName>' is invalid because the referenced table <tableName> has no primary key. Either add a primary key to <tableName> or explicitly specify the columns of a unique constraint that this foreign key references.

SQLSTATE	Message Text
X0Y42.S	Constraint '<constraintName>' is invalid: the types of the foreign key columns do not match the types of the referenced columns.
X0Y43.S	Constraint '<constraintName>' is invalid: the number of columns in <value> (<value>) does not match the number of columns in the referenced key (<value>).
X0Y44.S	Constraint '<constraintName>' is invalid: there is no unique or primary key constraint on table '<tableName>' that matches the number and types of the columns in the foreign key.
X0Y45.S	Foreign key constraint '<constraintName>' cannot be added to or enabled on table <tableName> because one or more foreign keys do not have matching referenced keys.
X0Y46.S	Constraint '<constraintName>' is invalid: referenced table <tableName> does not exist.
X0Y54.S	Schema '<schemaName>' cannot be dropped because it is not empty.
X0Y55.S	The number of rows in the base table does not match the number of rows in at least 1 of the indexes on the table. Index '<indexName>' on table '<schemaName>.<tableName>' has <number> rows, but the base table has <number> rows. The suggested corrective action is to recreate the index.
X0Y56.S	'<value>' is not allowed on the System table '<tableName>'.
X0Y57.S	A non-nullable column cannot be added to table '<tableName>' because the table contains at least one row. Non-nullable columns can only be added to empty tables.
X0Y58.S	Attempt to add a primary key constraint to table '<tableName>' failed because the table already has a constraint of that type. A table can only have a single primary key constraint.
X0Y59.S	Attempt to add or enable constraint(s) on table '<tableName>' failed because the table contains <rowName> row(s) that violate the following check constraint(s): <constraintName>.
X0Y63.S	The command on table '<tableName>' failed because null data was found in the primary key or unique constraint/index column(s). All columns in a primary or unique index key must not be null.
X0Y63.S.1	The command on table '<tableName>' failed because null data was found in the primary key/index column(s). All columns in a primary key must not be null.
X0Y66.S	Cannot issue commit in a nested connection when there is a pending operation in the parent connection.
X0Y67.S	Cannot issue rollback in a nested connection when there is a pending operation in the parent connection.
X0Y68.S	<value> '<value>' already exists.
X0Y69.S	<value> is not supported in trigger <triggerName>.



SQLSTATE	Message Text
X0Y70.S	INSERT, UPDATE and DELETE are not permitted on table <tableName> because trigger <triggerName> is active.
X0Y71.S	Transaction manipulation such as SET ISOLATION is not permitted because trigger <triggerName> is active.
X0Y72.S	Bulk insert replace is not permitted on '<value>' because it has an enabled trigger (<value>).
X0Y77.S	Cannot issue set transaction isolation statement on a global transaction that is in progress because it would have implicitly committed the global transaction.
X0Y78.S	Statement.executeQuery() cannot be called with a statement that returns a row count.
X0Y78.S.1	<value>.executeQuery() cannot be called because multiple result sets were returned. Use <value>.execute() to obtain multiple results.
X0Y78.S.2	<value>.executeQuery() was called but no result set was returned. Use <value>.executeUpdate() for non-queries.
X0Y79.S	Statement.executeUpdate() cannot be called with a statement that returns a ResultSet.
X0Y80.S	ALTER table '<tableName>' failed. Null data found in column '<columnName>'.
X0Y83.S	WARNING: While deleting a row from a table the index row for base table row <rowName> was not found in index with conglomerate id <id>. This problem has automatically been corrected as part of the delete operation.
X0Y84.T	Too much contention on sequence <sequenceName>. This is probably caused by an uncommitted scan of the SYS.SYSSEQUENCES catalog. Do not query this catalog directly. Instead, use the SYSCS_UTIL.SYSCS_PEEK_AT_SEQUENCE function to view the current value of a query generator.
X0Y85.S	The Splice property '<propertyName>' identifies a class which cannot be instantiated: '<className>'. See the next exception for details.
X0Y86.S	Splice could not obtain the locks needed to release the unused, preallocated values for the sequence '<schemaName>'.<sequenceName>'. As a result, unexpected gaps may appear in this sequence.
X0Y87.S	There is already an aggregate or function with one argument whose name is '<schemaName>'.<aggregateOrFunctionName>'.