InfiniDB®
SQL Syntax Guide

Release: 4.6
Document Version: 4.6-1

InfiniDB SQL Syntax Guide
July 2014

Contents

1 Introduction

InfiniDB is a high performance, scalable data warehouse storage engine. To enable the use of InfiniDB by end users and applications, it is necessary to integrate InfiniDB with a database front end. This document describes the supported SQL syntax available to users in the combined deployment of InfiniDB and MySQL v5.1.

- Querying the database
- Inserting, updating, and deleting rows
- Altering, creating, and deleting objects

This guide does not document all MySQL SQL statements. This guide specifically lists the datatypes, expressions, and conditions that are allowed using the InfiniDB. In addition, it lists SQL statements that run native on InfiniDB. Statements that run native on InfiniDB deliver optimal performance results that exceed the results of other MySQL Oracle commands not documented in this guide. See the ”Operating Mode” section for more information.

1.1 Audience

This guide is intended for database administrators and business intelligence managers that use InfiniDB to perform queries for business intelligence reporting and updates to the data warehouse.

1.2 List of Documentation

The InfiniDB Database Platform documentation consists of several guides intended for different audiences. The documentation is described in the following table:

Document	Description
InfiniDB Administrator’s Guide	Provides detailed steps for maintaining InfiniDB.
InfiniDB Apache Hadoop^{TM} Configuration Guide	Installation and Administration of an InfiniDB for Apache Hadoop system.
InfiniDB Concepts Guide	Introduction to the InfiniDB analytic database.
InfiniDB Minimum Recommended Technical Specifications	Lists the minimum recommended hardware and software specifications for implementing InfiniDB.
InfiniDB Installation Guide	Contains a summary of steps needed to perform an install of InfiniDB.
InfiniDB Multiple UM Configuration Guide	Provides information for configuring multiple User Modules.
Performance Tuning for the InfiniDB Analytics Database	Provides help for tuning the InfiniDB analytic database for parallelization and scalability.
InfiniDB Windows Installation and Administrator’s Guide	Provides information for installing and maintaining InfiniDB for Windows.

1.3 Obtaining documentation

These guides reside on our http://www.infinidb.co website. Contact support@infinidb.co for any additional assistance.

1.4 Documentation feedback

We encourage feedback, comments, and suggestions so that we can improve our documentation. Send comments to support@infinidb.co along with the document name, version, comments, and page numbers.

1.5 Additional resources

If you need help installing, tuning, or querying your data with InfiniDB, you can contact support@infinidb.co.

2 SQL Overview

This chapter provides information necessary to perform DDL and DML statements using InfiniDB. It also lists the guidelines for naming objects.

2.1 Notes

This section provides details important to running SQL statements. See "SQL Syntax"31 for information about specific SQL statements.

2.2 Naming conventions

The following are the maximum lengths for MySQL objects in InfiniDB:

- Schema, table and column names: 128 characters
- To ensure full compatibility with internal InfiniDB usage, the first character of all table and column names should be a letter (a-z).
- Spaces are not supported in object names.
- UTF-8 characters are not supported in object names.
- Certain words such as SELECT, CHAR and TABLE are reserved words in InfiniDB and cannot be used for object names. InfiniDB will return with a syntax error even if these words are wrapped with the backtick (') identifier.
- The keyword DATE may be used as a column name.

3 Datatypes and Functions

This chapter describes the naming conventions, datatypes, and functions that are native to InfiniDB.

3.1 Datatypes

InfiniDB supports the ANSI-92 datatypes listed in the table below:

Table 1 - Datatypes

Datatypes	Column Size	Description
BIGINT	8-bytes	A large integer. Numeric value with scale 0. Min/Max Signed: -9,223,372,036,854,775,806 to +9,223,372,036,854,775,807 Min/Max Unsigned: 0 to +18,446,744,073,709,551,613
CHAR	1, 2, 4, or 8 bytes	Holds letters and special characters of fixed length. Max length is 255. Default and minimum size is 1 byte.
DATE	4-bytes	Date has year, month, and day. The internal representation of a date is a string of 4 bytes. The first 2 bytes represent the year, .5 bytes the month, and .75 bytes the day in the following format: YYYY-MM-DD. Supported range: 1400-01-01 to 9999-12-31.
DATETIME	8-bytes	A date and time combination. Supported range: 1400-01-01 00:00:00 to 9999-12-31 23:59:59.
DECIMAL/NUMERIC	2, 4, or 8 bytes	A packed fixed-point number that can have a specific total number of digits and with a set number of digits after a decimal. The maximum precision (total number of digits) that can be specified is 18.

DOUBLE/REAL	8 bytes	Stored in 64-bit IEEE-754 floating point format. The MySQL extension to specify precision and scale is not supported. “REAL” is a synonym for “DOUBLE”. Min/Max Signed: approximately -1.797693134862316 +/- e307 to 1.797693134862316 +/- e307. Min/Max Unsigned: 0 to approximately 1.797693134862316 +/- e307.
FLOAT	4 bytes	Stored in 32-bit IEEE-754 floating point format. The MySQL extension to specify precision and scale is not supported. Min/Max Signed: approximately -3.402823466385289 +/- e38 to 3.402823466385289 +/- e38. Min/Max Unsigned: 0 to approximately 3.402823466385289 +/- e38.
INTEGER/INT	4-bytes	A normal-size integer. Numeric value with scale 0. Min/Max Signed: -2,147,483,646 to 2,147,483,647 Min/Max Unsigned: 0 to +4294967293
SMALLINT	2-bytes	A small integer. Min/Max Signed: -32,766 to 32,767 Min/Max Unsigned: 0 to 65533
TINYINT	1-byte	A very small integer. Numeric value with scale 0. Min/Max Signed: -126 to +127 Min/Max Unsigned: 0 to 253
VARCHAR	1, 2, 4, or 8 bytes or 8-byte token	Holds letters, numbers, and special characters of variable length. Max length = 8000 bytes or characters. Minimum length = 1 byte or character.

VARBINARY	8 bytes	Partial support- See Varbinary Note below. Holds binary letters and numbers of variable length. Max length = 8000 bytes. Minimum length = 8 bytes.

Notes to Datatypes:

- The InfiniDB engine, unlike the MyISAM engine, treats a zero-length string as a NULL value.
- The InfiniDB engine, like the MyISAM engine, employs “saturation semantics” on integer values. This means that if a value is inserted into an integer field that is too big/small for it to hold (i.e. it is more negative or more positive than the values indicated above), InfiniDB will “saturate” that value to the min/max value indicated above as appropriate. For example, for a SIGNED SMALLINT column, if 32800 is attempted, the actual value inserted will be 32767.
- InfiniDB’s largest negative number for SIGNED datatypes appears to be 2 less than what MySQL supports. InfiniDB reserves these 2 most-negative numbers for its internal use and cannot be used. For example, if there is a need to store -128 in a column, the TINYINT datatype cannot be used; the SMALLINT datatype must be used instead. If the value -128 is inserted into a TINYINT column, InfiniDB will saturate it to -126 (and issue a warning).
- InfiniDB’s largest positive number for UNSIGNED datatypes appears to be 2 less than what MySQL supports. InfiniDB reserves these 2 highest positive numbers for its internal use and cannot be used.
- For the date and datetime datatypes, a value outside of the supported range (1400-01-01 to 9999-12-31) will be stored as NULL in InfiniDB.
- An optional display width may be added to the BIGINT, INTEGER/INT, SMALLINT & TINYINT columns. As with MyISAM tables, this value does not affect the internal storage requirements of the column nor does it affect the valid value ranges.
- All columns in InfiniDB are nullable and the default value for any column is NULL. You may optionally specify NOT NULL for any column and/or one with a DEFAULT value.
- When creating string columns with UTF, the length is defined in bytes, not characters. To store ten multi-byte characters with UTF may require twenty to thirty bytes, depending on the actual values being stored. Please see “Using UTF-8 Character Sets” 67for more information.
- InfiniDB provides the following feature set for VARBINARY columns. Note that if the behavior is not documented here it may or may not work the same as in normal MySQL. You should always validate the InfiniDB VARBINARY behavior in a test environment before using this datatype in production.

  VARBINARY support is disabled by default in InfiniDB. To enable it, execute the following commands:

  ```
  /usr/local/Calpont/bin/setConfig WriteEngine AllowVarbinary Yes
  /usr/local/Calpont/bin/calpontConsole RestartSystem
  ```

  To create a table with a VARBINARY column the syntax is:

```
CREATE TABLE VAR_TABLE (COL1 INT, COL2 VARBINARY(1024)) ENGINE=INFINIDB;
```

To get VARBINARY data into your table using INSERT the syntax is:

```
INSERT INTO VAR_TABLE VALUES (1, 0xABCDEF01);
```

If you are using cpimport, the import file format would look like:

```
1|ABCDEF01|
```

Normally, if you select a VARBINARY column back to a terminal using the mysql client, you will get binary data printed to the screen. If you want to see this data in hex format execute the following at the SQL prompt before running the select statement:

```
set infinidb_varbin_always_hex=1;
```

Generally speaking, in InfiniDB, you can only put VARBINARY data into a table and select it back out again, as-is. You cannot join on VARBINARY columns nor can you use them as inputs to functions or expressions. Usually you will receive an error message if you try to do this, but there may be ways to avoid the built-in error checking. In this case, the results are undefined.

3.2 Distributed InfiniDB Functions

3.2.1 Distributed Aggregate Functions

InfiniDB supports the following aggregate functions. These functions can be specified in the projection (SELECT), HAVING, and ORDER BY portions of the SQL statement.

Table 2 - Distributed InfiniDB Aggregate Functions

Function	Description
AVG([DISTINCT] column)	Return the average value of a number (INT variations, NUMERIC, DECIMAL) datatype column.
COUNT (*, [DISTINCT] column)	The number of rows returned by a query. All datatypes described above are supported.
MAX ([DISTINCT] column)	The maximum value of a column. All datatypes described above are supported.
MIN ([DISTINCT] column)	The minimum value of a column. All datatypes described above are supported.

STD()
STDDEV()
STDDEV_POP()
	Return the population standard deviation of a number (INT variations, NUMERIC, DECIMAL) datatype column.

STDDEV_SAMP()	Return the sample standard deviation of a number (INT variations, NUMERIC, DECIMAL) datatype column.
SUM([DISTINCT] column)	Return the sum of a number (INT variations, NUMERIC, DECIMAL) datatype column.

VARIANCE()
VAR_POP()
	Return the population standard variance of a number (INT variations, NUMERIC, DECIMAL) datatype column.

VAR_SAMP()	Return the sample standard variance of a number (INT variations, NUMERIC, DECIMAL) datatype column.

3.2.2 Distributed Functions

InfiniDB supports the following functions. These functions can be specified in the projection (SELECT), WHERE, GROUP BY, HAVING and ORDER BY portions of the SQL statement and will be processed in a distributed manner.

Table 3- Distributed InfiniDB Functions

Function	Description
&[1]	Bitwise AND
ABS()[2]	Return the absolute value.
ACOS()	Return the arc cosine.
ADDTIME()	Add time.
ASCII()	Return numeric value of left-most character
ASIN()[3]	Return the arc sine.
ATAN()[3]	Return the arc tangent.
BETWEEN...AND...	Check whether a value is within a range of values.
BIT_AND()	Return bitwise and
BIT_OR()	Return bitwise or.
BIT_XOR()	Return bitwise xor.
CASE()	Case operator.

CAST()
CONVERT()
	Take a value of one type and produce a value of another type.

CEIL()
CEILING()
	Returns the smallest integer value not less than argument.

CHAR()	Return the character for each integer passed.

CHAR_LENGTH()
CHARACTER_LENGTH()
	Return number of characters in argument.

COALESCE()	Return the first non-NULL argument.
CONCAT()	Return concatenated string.
CONCAT_WS()	Return concatenate with separator.
CONV()	Convert numbers between different number bases.
COS()[3]	Return the cosine.
COT()[3]	Return the cotangent.
CRC32()	Compute a cyclic redundancy check value.
DATE()	Extract the date part of a date or datetime expression.

DATE_ADD()
ADDDATE()
	Add time values (intervals) to a date value.

DATE_FORMAT()	Format date as specified.

DATE_SUB()
SUBDATE()
	Subtract two dates.

DATEDIFF()	Return the difference between two dates.

DAY()
DAYOFMONTH()
	Return the day of the month (0-31).

DAYNAME()	Return the name of the weekday.
DAYOFWEEK()	Return the weekday index of the argument
DAYOFYEAR()	Return the day of the year (1-366)
DEGREES()	Convert radians to degrees.
DIV()	Integer division.
ELT()	Return string at index number.
EXP()[3]	Raise to the power of.
EXTRACT()	Extract part of a date.
FIND_IN_SET()	Return the index position of the first argument within the second argument.
FLOOR()	Return the largest integer value not greater than the argument.
FORMAT()	Return a number formatted to specified number of decimal places.
FROM_DAYS()	Convert a day number to a date.
FROM_UNIXTIME()	Format UNIX timestamp as a date.
GET_FORMAT()	Return a date format string.
GREATEST()	Return the largest argument.
GROUP_CONCAT()	Return a concatenated string.
HEX()	Return a hexadecimal representation of a decimal or string value.
HOUR()	Extract the hour.
IF()	If/else construct.
IFNULL()	Null if/else construct.
IN	Check whether a value is within a set of values. Currently support only literal values.
INET_ATON()	Return the numeric value of an IP address.
INET_NTOA()	Retun the IP address of a numeric value.
INSERT()	Insert a substring at the specified position up to a specified number of characters.
INSTR()	Return the index of the first occurrence of substring.
ISNULL()	Test whether the argument is NULL.
LAST_DAY()	Return the last day of the month for the argument.
LCASE()	Synonym for LOWER().
LEAST()	Return the smallest argument.
LEFT()	Return the leftmost number of characters as specified.
LENGTH()	Return the length of a string in bytes.
LIKE	Simple pattern matching.
LN()[3]	Return the natural logarithm of the argument.

LOCATE()	Return the position of the first occurrence of substring.
LOG()[3]	Return the natural logarithm of the first argument.
LOG2()[3]	Return the base-2 logarithm of the argument.
LOG10()[3]	Return the base-10 logarithm of the argument
LOWER()	Return the argument in lowercase.
LPAD()	Return the string argument, left-padded with the specified string.
LTRIM()	Remove leading spaces.
MAKEDATE()	Return a date from the year and day of year.
MAKETIME()	Return a time calculated from the hour, minute and second arguments.
MD5()	Calculate MD5 checksum.
MICROSECOND()	Return the microseconds from argument.
MID()	Return a substring starting from the specified position.
MINUTE()	Return the minute from the argument.
MOD()	Return the remainder.
MONTH()	Return the month from the date passed.
MONTHNAME()	Retrn the name of the month.
NOW()	Return the current date and time.
NULLIF()	Return NULL if expr1 = expr2.
PERIOD_ADD()	Add a period to a year-month.
PERIOD_DIFF()	Return the number of months between periods.
POSITION()	A synonym for LOCATE().
POW()/POWER()	Return the argument raised to the specified power.
QUARTER()	Return the quarter from a date argument.
RADIANS()	Return argument converted to radians
RAND()	Return a random floating-point value.

REGEXP()
RLIKE()
	Pattern matching using regular expressions.

REPEAT()	Repeat a string the specified number of times.
REPLACE()	Replace occurrences of a specified string.
REVERSE()	Reverse the characters in a string.
RIGHT()	Return the specified rightmost number of characters.
ROUND()	Round the argument.
RPAD()	Append string the specified number of times.
RTRIM()	Remove trailing spaces.
SECOND()	Return the second (0-59).
SEC_TO_TIME()	Converts seconds to ‘HH:MM:SS’ format.

SHA()
SHA1()
	Calculate an SHA-1 160-bit checksum.

SIGN()	Return the sign of the argument.
SIN()[3]	Return the sine of the argument.
SPACE()	Return a string of the specified number of spaces.
SQRT()[3]	Return the square root of the argument.
STR_TO_DATE()	Convert a string to a date.
STRCMP()	Compare two strings.

SUBSTR()
SUBSTRING()
	Return the substring as specified.

SUBSTRING_INDEX()	Return a substring from a string before the specified number of occurrences of the delimiter.
SUBTIME()	Subtract times.
SYSDATE()	Return the time at which funtion executes.
TAN()[3]	Return the tangent of the argument.
TIME()	Return the time part of a time or datetime expression as a string
TIMEDIFF()	Return the difference between two times.
TIME_FORMAT()	Format as time.
TIME_TO_SEC()	Return the argument converted to seconds.
TIMESTAMPADD()	Add an interval to a datetime expression.
TIMESTAMPDIFF()	Subtract an interval from a datetime expression.
TO_DAYS()	Return the date argument converted to days.
TRIM()	Remove leading and trailing spaces.
TRUNCATE()	Truncate the specified number of decimal places.
UCASE()	Synonym for UPPER().
UNIX_TIMESTAMP()	Return a UNIX timestamp.
UPPER()	Convert to uppercase.
WEEK()	Return the week number.
WEEKDAY()	Return the weekday index.
WEEKOFYEAR()	Return the calendar week of the date (0-53).
XOR()	Logical XOR.
YEAR()	Return the year.
YEARWEEK()	Return the year and week.

1. BITWISE & on columns containing negative values or floating point datatypes may produce results different from other storage engines.
2. Consult the MySQL function reference for details on these functions. As these are integer functions, they only produce predictable results on integer columns.
3. Consult the MySQL function reference for details on these functions. As these are floating-point trigonometric functions, they only produce predictable results on FLOAT and DOUBLE columns.

3.2.3 Windowing Functions

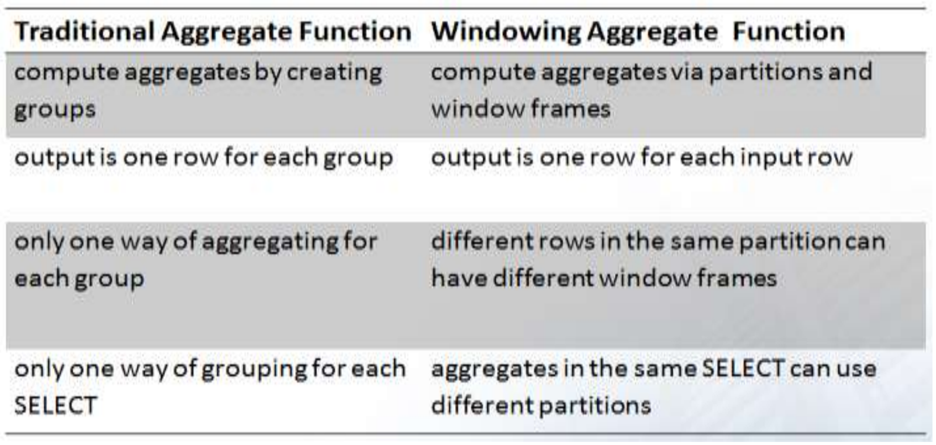
Windowing functions perform a calculation across a set of rows that are somehow related to the current row. They are typically used in analytics to compute cumulative, moving, centered or reporting aggregates.

Additional characteristics of windowing functions:

- Also referred to as analytic functions
- Does not cause rows to become grouped into a single output row — the rows retain their separate identities.
- Unlike traditional aggregate functions using “group by” clause in standard SQL syntax. windowing functions:
- Returns multiple rows for each group
    - Does not require every column in the projection list to be in the group clause.
    - Window functions are the last set of operations performed in a query except for the final ORDER BY clause. All joins and all WHERE, GROUP BY, and HAVING clauses are completed before the window functions are processed. Therefore, window functions can appear only in the SELECT list or ORDER BY clause.

3.2.3.1 Traditional vs Windowing Aggregation

The following chart summarizes some differences between traditional aggregation and windowing aggregation:



Traditional Aggregate Function	Windowing Aggregate Function
compute aggregates by creating groups	compute aggregates via partitions and window frames
output is one row for each group	output is one row for each input row
only one way of aggregating for each group	different rows in the same partition can have different window frames
only one way of grouping for each SELECT	aggregates in the same SELECT can use different partitions

3.2.3.2 Partitioning and Frames

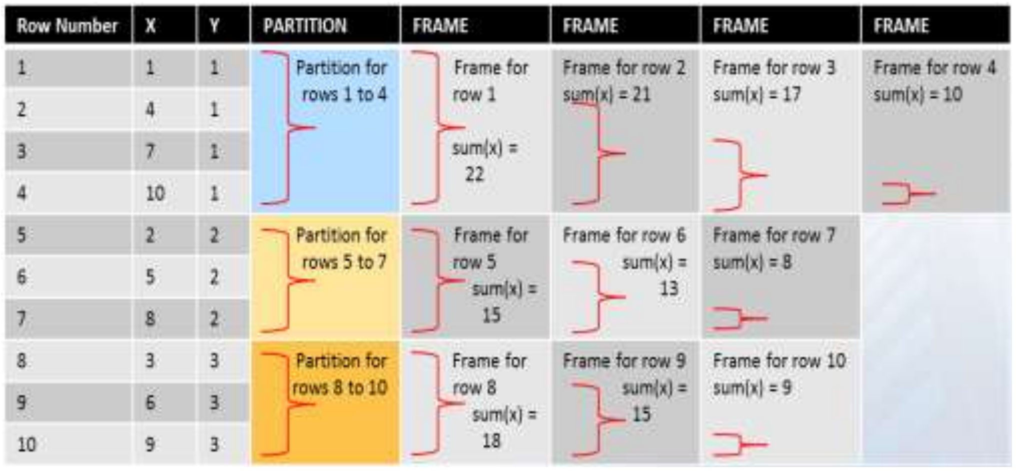
Two key elements with windowing functions are PARTITION and FRAME.

The PARTITION is a group of rows, or window, that has the same value for a specific column that is the same as the current row.

The FRAME for each row is a subset of a PARTITION for the row. For each row, a sliding frame of rows is defined. The frame determines the range of rows used to perform the calculations for the current row.

The following chart illustrates the subset of partitions and frames, along with the result for each frame. For the given window function query,

```
SELECT x,y,sum(x) OVER (PARTITION BY y RANGE BETWEEN CURRENT ROW AND
UNBOUNDED FOLLOWING) FROM a ORDER BY x, y
```



Row Number	X	Y	PARTITION	FRAME	FRAME	FRAME	FRAME
1	1	1	Partition for rows 1 to 4	Frame for row 1 sum(x) = 22	Frame for row 2 sum(x) = 21	Frame for row 3 sum(x) = 17	Frame for row 4 sum(x) = 10
2	4	1					
3	7	1					
4	10	1					
5	2	2	Partition for rows 5 to 7	Frame for row 5 sum(x) = 15	Frame for row 6 sum(x) = 13	Frame for row 7 sum(x) = 8	
6	5	2					
7	8	2					
8	3	3	Partition for rows 8 to 10	Frame for row 8 sum(x) = 18	Frame for row 9 sum(x) = 15	Frame for row 10 sum(x) = 9	
9	6	3					
10	9	3					

3.2.3.3 Supported Windowing Functions

These functions can be specified in the projection (SELECT) or ORDER BY portions of the SQL statement and will be processed in a distributed manner.

Table 4 - Distributed InfiniDB Windowing Functions

Function	Description
AVG()	The average (arithmetic mean) of all input values.
COUNT()	Number of input rows.
CUME_DIST()	Calculates the cumulative distribution, or relative rank, of the current row to other rows in the same partition. Number of peer or preceding rows / number of rows in partition.
DENSE_RANK()	Ranks items in a group leaving no gaps in ranking sequence when there are ties.
FIRST_VALUE()	The value evaluated at the row that is the first row of the window frame (counting from 1); null if no such row.
LAG()	The value evaluated at the row that is offset rows before the current row within the partition; if there is no such row, instead return default. Both offset and default are evaluated with respect to the current row. If omitted, offset defaults to 1 and default to null. LAG provides access to more than one row of a table at the same time without a self-join. Given a series of rows returned from a query and a position of the cursor, LAG provides access to a row at a given physical offset prior to that position.
LAST_VALUE()	The value evaluated at the row that is the last row of the window frame (counting from 1); null if no such row.

LEAD()	Provides access to a row at a given physical offset beyond that position. Returns value evaluated at the row that is offset rows after the current row within the partition; if there is no such row, instead return default. Both offset and default are evaluated with respect to the current row. If omitted, offset defaults to 1 and default to null.
MAX()	Maximum value of expression across all input values.
MEDIAN()	An inverse distribution function that assumes a continuous distribution model. It takes a numeric or datetime value and returns the middle value or an interpolated value that would be the middle value once the values are sorted. Nulls are ignored in the calculation.
MIN()	Minimum value of expression across all input values.
NTH_VALUE()	The value evaluated at the row that is the nth row of the window frame (counting from 1); null if no such row.
NTILE()	Divides an ordered data set into a number of buckets indicated by expr and assigns the appropriate bucket number to each row. The buckets are numbered 1 through expr. The expr value must resolve to a positive constant for each partition. Integer ranging from 1 to the argument value, dividing the partition as equally as possible.
PERCENT_RANK()	relative rank of the current row: (rank - 1) / (total rows - 1).
PERCENTILE_CONT()	An inverse distribution function that assumes a continuous distribution model. It takes a percentile value and a sort specification, and returns an interpolated value that would fall into that percentile value with respect to the sort specification. Nulls are ignored in the calculation.
PERCENTILE_DISC()	An inverse distribution function that assumes a discrete distribution model. It takes a percentile value and a sort specification and returns an element from the set. Nulls are ignored in the calculation.
RANK()	rank of the current row with gaps; same as row_number of its first peer.
ROW_NUMBER()	number of the current row within its partition, counting from 1

STDDEV()
STDDEV_POP()
	Computes the population standard deviation and returns the square root of the population variance.

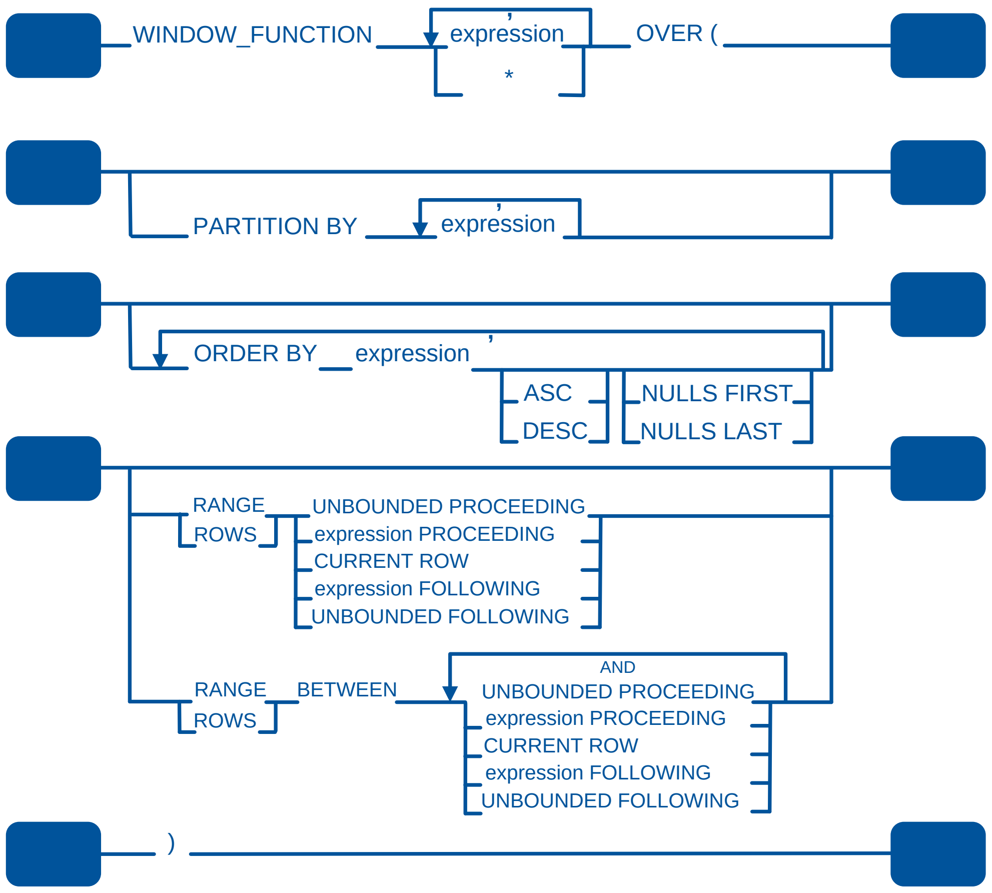
STDDEV_SAMP()	Computes the cumulative sample standard deviation and returns the square root of the sample variance.
SUM()	Sum of expression across all input values.

VARIANCE()
VAR_POP()
	Population variance of the input values (square of the population standard deviation).

VAR_SAMP()	Sample variance of the input values (square of the sample standard deviation).

3.2.3.4 Windowing Functions Syntax

The following diagram describes the syntax to enable windowing functions:



PARTITION BY

Characteristics for the PARTITION BY clause:

- Partitions the query result set into groups based on one or more *expression*.
- Analytic functions in the same SELECT can use different partitions.
- Valid values of *expression* are constants, columns, non-analytic functions, function expressions, or expressions involving any of these.
- The columns used in PARTITION BY do not need to be in the SELECT projection list, but need to be available as a result set generated after FROM, WHERE, GROUP BY, and HAVING clauses of the SELECT query have been applied.
- If PARTITION BY is omitted, then the function treats all rows of the query result set as a single group.

ORDER BY

Characteristics for the ORDER BY clause:

- ORDER BY within an analytic clause specifies how data is ordered within a partition
- Values in a partition can be ordered on multiple keys, each defined by a *expression* and each qualified by an ordering sequence.

- Valid values of *expression* are constants, columns, non-analytic functions, function expressions, or expressions involving any of these.
- The columns used in ORDER BY do not need to be in SELECT projection list, but need to be available as a result set generated after FROM, WHERE, GROUP BY, and HAVING clauses of the SELECT query have been applied.
- The column alias cannot be used in ORDER BY.
- The ASC | DESC option specifies the ordering sequence (ascending or descending). ASC is the default.
- The NULLS FIRST | NULLS LAST option specifies whether returned rows containing nulls should appear first or last in the ordering sequence. NULLS FIRST is the default for ASC order, and NULLS LAST is the default for DESC order.

RANGE/ROWS (Windowing Clause)

Characteristics for the RANGE/ROWS clause:

- The windowing clause defines for each row a window (a physical or logical set of rows) used for calculating the function result.
- Function is then applied to all the rows in the window.
- The window moves through the query result set or partition from top to bottom.
- ROWS | RANGE defines the windows:
  - Valid only if the ORDER BY clause is present.
  - ROWS specifies the window in physical units (rows).
    - *expression* is a physical offset. It must be a constant or expression and evaluate to a positive numeric value.
    - If *expression* is part of the start point, then it must evaluate to a row before the end point.
- RANGE specifies the window as a logical offset.
  - *expression* is a logical offset. It must be a constant or expression that evaluates to a positive numeric value or an interval literal.
  - You can specify only one *expression* in the ORDER BY clause as the range operates on the order by column/*expression*.
  - If *expression* evaluates to a numeric value, then the ORDER BY *expression* must be a numeric or DATE data type.
  - If *expression* evaluates to an interval value, then the ORDER BY *expression* must be a DATE data type.
- BETWEEN ... AND defines the start and end point of the window
  - When BETWEEN is omitted in the windowing clause but a single point is specified, this point is the start point and current row is end point.
- UNBOUNDED PRECEDING indicates that the window starts at the first row of the partition. It cannot be used as an end point specification.
- UNBOUNDED FOLLOWING indicates that the window ends at the last row of the partition. If cannot be used as an start point specification.
- CURRENT ROW specifies that the window begins at the current row or value
- When the windowing clause is ommited, then the default is RANGE BETWEEN UNBOUNDED PRECEDING AND UNBOUNDED FOLLOWING.

3.2.3.5 Windowing Function Examples

Example #1 - Partition by a function:

For every employee find the number of other employees hired in the same calendar year

```
SELECT empno, deptno, hiredate,
COUNT(*) OVER (PARTITION BY YEAR(hiredate) ORDER BY hiredate) emp_cnt
FROM emp
ORDER BY hiredate;

EMPNO  DEPTNO  HIREDATE    EMP_CNT
-----  ------  ----------  -------
 7369      20  2008-01-01        2
 7499      30  2008-01-12        2
 7520      30  2009-02-27        9
 7521      30  2009-02-27        9
 7566      20  2009-04-03        9
 7698      30  2009-04-18        9
 7782      10  2009-04-14        9
 7844      30  2009-05-01        9
 7654      30  2009-05-15        9
 7839      10  2009-07-02        9
 7900      30  2009-08-30        9
```

Example #2 - Omitting the PARTITION BY clause:

For each employee find the population standard deviation of the salaries in Department 30, ordered by hire_date:

```
SELECT last_name, salary,
STDDEV(salary) OVER (ORDER BY hire_date) "StdDev"
FROM employees
WHERE dept_id = 30
ORDER BY last_name, salary, "StdDev";

LAST_NAME    SALARY      StdDev
-----------  ----------  ----------
Baida              2900    3891.015
Himuro             2600    3891.015
Raphaely          11000    3891.015
```

Example #3 - Partitioning on more than one column:

Find the total quantity of product sold for every product in every country for each calendar year. Also find the average amount sold for every product in every country over all years:

```
SELECT p.prod_id, country_id, calendar_year,
SUM(s.quantity_sold) OVER (PARTITION BY p.prod_id, p.country_id,
   a.calendar_year) units,
AVG(s.amount_sold)  OVER (PARTITION BY p.prod_id, p.country_id)
   avg_sales
FROM sales s, products p
WHERE p.prod_id = s.prod_id;
```

```
PROD     COUNTRY    YEAR     UNITS    AVG_SALES
------   --------   ------   -------  ----------
   147      52782     1999        29      400.02
   147      52782     2000        71      400.02
   147      52785     1999         1     2023.22
   147      52785     2000       139     2023.22
   147      52786     1999         1       12.01
   147      52786     2000         2       12.01
   148      52782     1999       251     6050.14
   148      52782     2000       308     6050.14
   148      52788     1999         4      175.31
   148      52788     2000         8      175.31
```

Example #4 - ORDER BY on multiple keys:

Rank the employees in department 80 based on their salary and commission:

```
SELECT department_id, last_name, salary, commission_pct,
RANK() OVER (PARTITION BY department_id
  ORDER BY salary DESC, commission_pct) "Rank"
FROM employees WHERE department_id = 80
ORDER BY department_id, last_name, salary, commission_pct, "Rank";

DEPARTMENT_ID  LAST_NAME   SALARY   COMMISSION_PCT  Rank
-------------  ----------  -------  --------------  -----
           80  Abel          11000              .3      1
           80  Ande           6400              .1      4
           80  Banda          6200              .1      5
           80  Bates          7300             .15      3
           80  Bates          9500             .25      2
```

Example #5 - ROWS UNBOUND PROCEEDING:

The name of the employee with lowest salary in department 90;

```
SELECT department_id, last_name, salary,
FIRST_VALUE(last_name)
 OVER (ORDER BY salary ASC ROWS UNBOUNDED PRECEDING) AS lowest_sal
 FROM (SELECT * FROM employees
          WHERE department_id = 90
          ORDER BY employee_id)
 ORDER BY last_name;

DEPARTMENT_ID  LAST_NAME   SALARY    LOWEST_SAL
-------------  ----------  --------  ----------
           90  De Haan        17500     Kochhar
           90  King           24000     Kochhar
           90  Kochhar        17000     Kochhar
```

Example #6 - ROWS BETWEEN...AND:

For each employee in the employees table, find the average salary of the employees reporting to the same manager who were hired in the range just before through just after the employee:

```
SELECT manager_id, last_name, hire_date, salary,
  AVG(salary) OVER (PARTITION BY manager_id ORDER BY hire_date
              ROWS BETWEEN 1 PRECEDING AND 1 FOLLOWING) AS c_mavg
  FROM employees
  ORDER BY manager_id, hire_date, salary;

MANAGER_ID LAST_NAME                 HIRE_DATE     SALARY     C_MAVG
---------- ------------------------- --------- ---------- ----------
       100 De Haan                   13-JAN-01      17000      14000
       100 Raphaely                  07-DEC-02      11000 11966.6667
       100 Kaufling                  01-MAY-03       7900 10633.3333
       100 Hartstein                 17-FEB-04      13000 9633.33333
       100 Weiss                     18-JUL-04       8000 11666.6667
       100 Russell                   01-OCT-04      14000 11833.3333
       100 Partners                  05-JAN-05      13500 13166.6667
       100 Errazuriz                 10-MAR-05      12000 11233.3333
```

3.2.4 Information Functions

InfiniDB Information Functions are selectable pseudo functions that return InfiniDB specific "meta" information to ensure queries can be locally directed to a specific node. These functions can be specified in the projection (SELECT), WHERE, GROUP BY, HAVING and ORDER BY portions of the SQL statement and will be processed in a distributed manner.

Table 5 - InfiniDB Information Functions

Function	Description
idbBlockId(column)	The Logical Block Identifier (LBID) for the block containing the physical row.
idbDBRoot(column)	The DBRoot where the physical row resides.
idbExtentId(column)	The Logical Block Identifier (LBID) for the first block in the extent containing the physical row. This is the same as the first number reported for an extent in the editem utility.
idbExtentMax(column)	The max value from the extent map entry for the extent containing the physical row.
idbExtentMin(column)	The min value from the extent map entry for the extent containing the physical row.
idbExtentRelativeRid(column)	The row id (1 to 8,388,608) within the column's extent.
idbLocalPm()	The PM from which the query was launched. This function will return NULL if the query is launched from a standalone UM.
idbPartition(column)	The three part partition id (Directory.SegmentDBRoot) as used by calshowpartitions, etc.
idbPm(column)	The PM where the physical row resides.
idbSegmentDir(column)	The lowest level directory id for the column file containing the physical row.
idbSegment(column)	The number of the segment file containing the physical row.

Please see "Local PM Query Examples" section in the InfiniDB Multiple UM Configuration Guide for some examples on the use of these functions.

3.3 Distributed User-Defined Functions

InfiniDB supports distributed user-defined functions. These functions can be specified in the projection (SELECT), WHERE, GROUP BY, HAVING and ORDER BY portions of the SQL statement and will be processed in a distributed manner.

You can obtain the User-Defined Functions SDK by downloading the InfiniDB source from https://github.com/infinidb. Once you unpack the source tar file, the InfiniDB UDF SDK, instructions and examples are in the utils/udfsdk directory.

3.4 Non-Distributed Post-Processed Functions

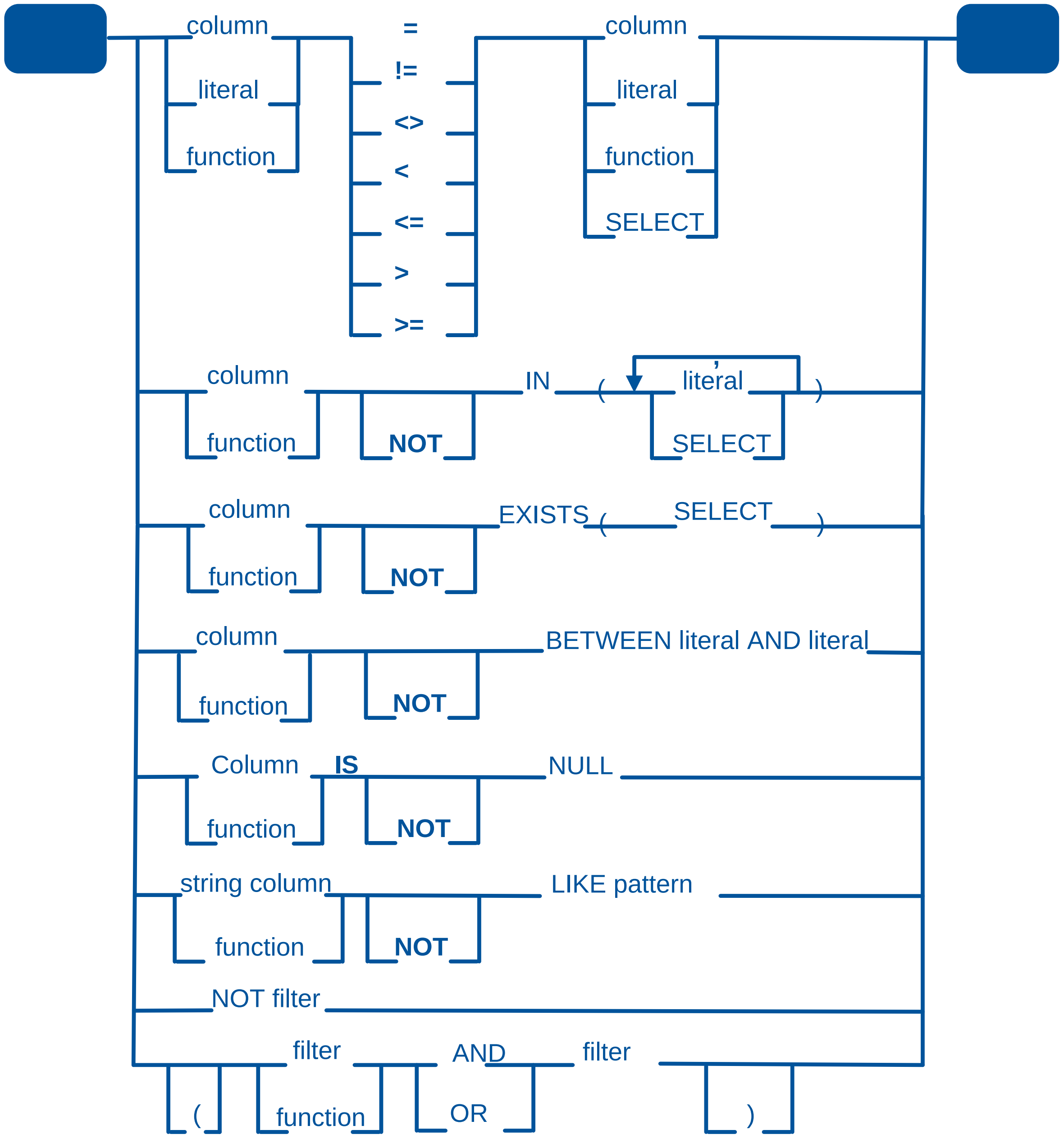
All other MySQL functions can be used in a post-processing manner where data is returned by InfiniDB first and then MySQL executes the function on the data returned. These functions are currently supported only in the projection (SELECT) and ORDER BY portions of the SQL statement.

4 Conditions

A condition is a combination of expressions and operators that return TRUE, FALSE or NULL.

4.1 *Filter*

The following diagram shows the conditions available with InfiniDB;



Note: A ‘literal’ may be a constant (e.g. 3) or an expression that evaluates to a constant [e.g. 100 - (27 * 3)]. For date columns, you may use the SQL ‘interval’ syntax to perform date arithmetic, as long as all the components of the expression are constants (e.g. ‘1998-12-01’ - interval ‘1’ year).

4.1.1 String Comparison

InfiniDB engine, unlike the MyISAM engine, is case sensitive for string comparisons used in filters.

For the most accurate results, and to avoid confusing results, make sure string filter constants are no longer than the column width itself.

4.1.2 Pattern Matching

Pattern matching as described with the LIKE condition allows you to use “_” to match any single character and “%” to match an arbitrary number of characters (including zero characters). To test for literal instances of a wildcard character, (“%” or “_”), precede it by the “\” character.

4.1.3 OR Processing

OR Processing has the following restrictions:

- Only column comparisons against a literal are allowed in conjunction with an OR. The following query would be allowed since all comparisons are against literals.

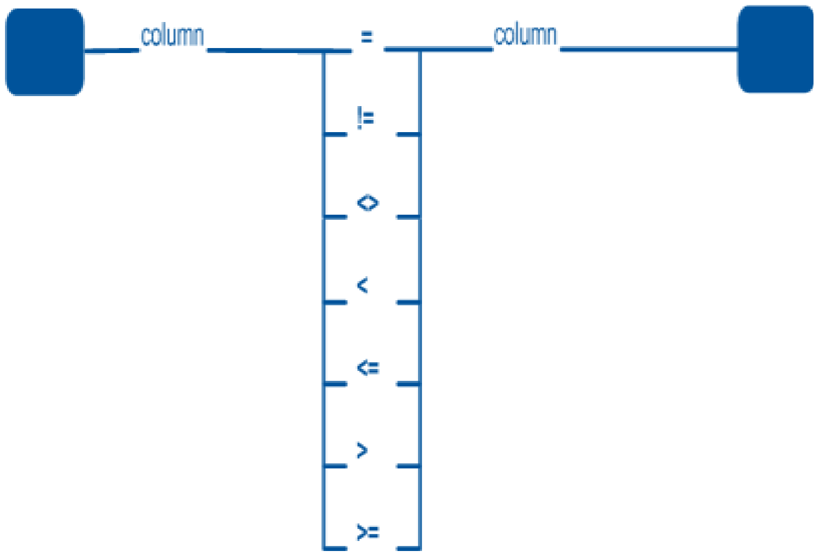
  ```
  SELECT count(*) from lineitem WHERE l_partkey < 100 OR l_linestatus = ‘F‘;
  ```

- InfiniDB binds AND’s more tightly than OR’s, just like any other SQL parser. Therefore you must enclose OR-relations in parentheses, just like in any other SQL parser.

  ```
  SELECT count(*) FROM orders, lineitem WHERE (lineitem.l_orderkey < 100 OR
  lineitem.l_linenumber > 10) AND lineitem.l_orderkey = orders.o_orderkey;
  ```

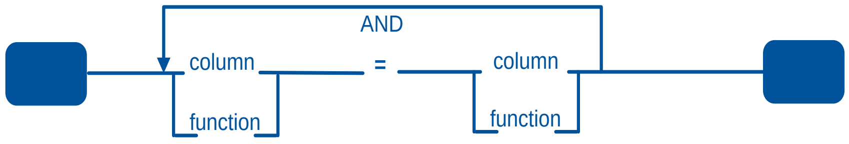
4.2 table_filter

The following diagram show the conditions you can use when executing a condition against two columns. Note that the columns must be from the same table.

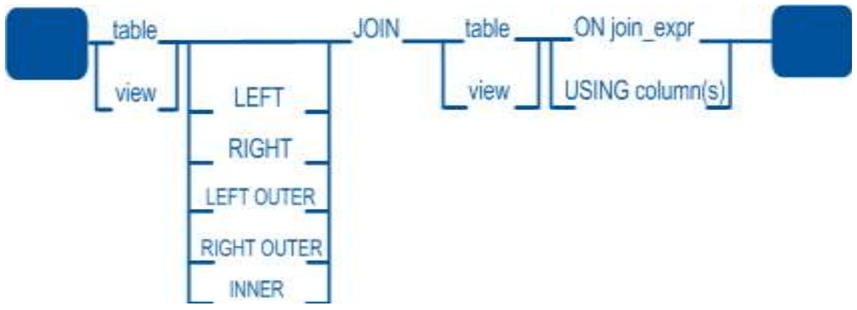


4.3 join

The following diagrams show the conditions you can use when executing a join on two tables:



4.4 join table



Notes for Joins:

- InfiniDB tables can be joined with non-InfiniDB tables (i.e., MyIsam tables). Please see the Cross-Engine Table Access section in the InfiniDB Administrator's Guide for further reference.
- InfiniDB will require a join in the WHERE clause for each set of tables in the FROM clause. No Cartesian product queries will be allowed.
- InfiniDB requires that joins must be on the same datatype. In addition, number datatypes (INT variations, NUMERIC, DECIMAL) may be mixed in the join if they have the same scale.
- Circular joins are not supported in InfiniDB. Please see the Troubleshooting section in the InfiniDB Administrator's Guide for further reference.
- Joins may processed as a disk based join when the join memory limit is exceeded. This option must be enabled. Please see the "Using Disk Based Joins" section below.

5 SQL Syntax

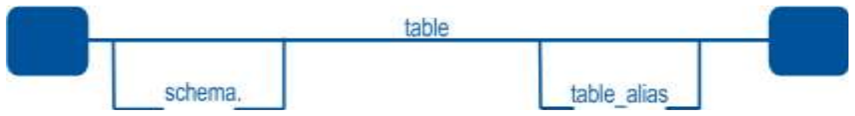
InfiniDB is a high performance SQL engine that supports SQL Syntax Statements. This chapter provides the syntax that must be adhered to when performing INSERT, UPDATE, or DELETE operations.

This guide lists the SELECT syntax that is native to InfiniDB and provides exceptionally fast query executions.

5.1 Table/Column Reference

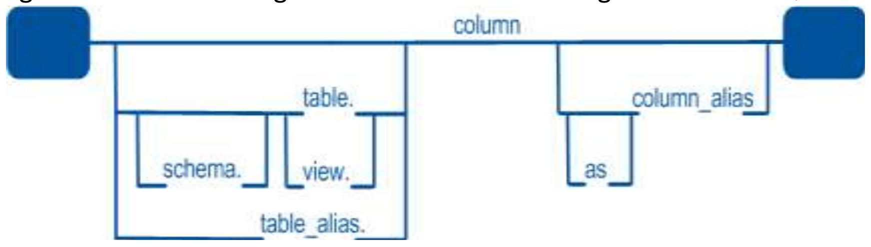
5.1.1 table

The following chart describes the guidelines when referencing a table in SQL statements:



5.1.2 column

The following chart describes the guidelines when referencing a column in SQL statements:

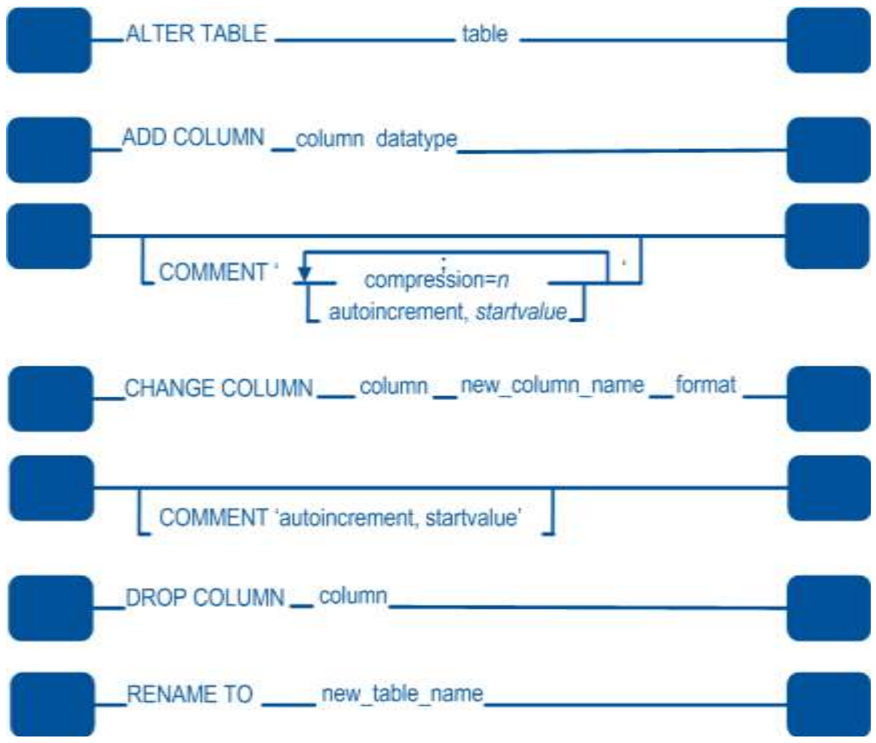


5.2 DDL Statements

DDL statements define database structures. These structures include columns, and tables. Supported DDL statements are listed below in alphabetical order with descriptions and sample syntax statements.

5.2.1 ALTER TABLE

The ALTER TABLE statement modifies existing tables. This includes adding, deleting and renaming columns as well as renaming tables.:



5.2.1.1 ADD

The ADD clause allows you to add columns to a table. You must specify the data type after the column name.

Notes to ALTER TABLE ADD COLUMN**:**

- A compression comment at the column level will override the system, session and table default. Values are:
  0 - do not compress the table/column
  1 - compress the table/column

  If a column is added with no compression comment, the compression default will be taken from the table level compression comment. If no table level compression comment exists, the default will follow the session default and system default, respectively.
- An 'autoincrement' comment on either the column definition or at the table level will create an InfiniDB autoincrement column. Only one autoincrement column may be defined per table. Please

see “Autoincrement Usage in InfiniDB” section for further autoincrement processing. If no *startvalue* is given, the default will be 1.

The following statement adds a **priority** column with an **integer** datatype to the **orders** table:

```
ALTER TABLE orders ADD COLUMN priority INTEGER;
```

Online alter table add column

The InfiniDB engine fully supports online DDL (one session can be adding columns to a table while another session is querying that table). MySQL, unfortunately, does not support this. Until this is fixed in the MySQL server, we have provided the following workaround. This workaround is intended for adding columns to a table, one at a time only. Do not attempt to use it for any other purpose. Follow the example below as closely as possible.

Scenario: add an INT column named col7 to the existing table foo:

```
select calonlinealter('alter table foo add column col7 int;');
alter table foo add column col7 int comment 'schema sync only';
```

The select statement may take several tens of seconds to run, depending on how many rows are currently in the table. Regardless, other sessions can select against the table during this time (but they won’t be able to see the new column yet). The alter table statement will take less than 1 second (depending on how busy MySQL is) and during this brief time interval, other table reads will be held off.

5.2.1.2 CHANGE

The CHANGE clause allows you to rename a column in a table.

Notes to CHANGE COLUMN:

- You cannot use CHANGE COLUMN to change the definition of that column. You cannot change from SIGNED to UNSIGNED or vice versa.
- You can only change a single column at a time.
- An ‘autoincrement’ comment on either the column definition or at the table level will create an InfiniDB autoincrement column. Only one autoincrement column may be defined per table. Please see “Autoincrement Usage in InfiniDB” for further autoincrement processing. A CHANGE COLUMN without the autoincrement comment will remove the autoincrement capability.

The following example renames the **order_qty** field to **quantity** in the **orders** table:

```
ALTER TABLE orders CHANGE COLUMN order_qty quantity INTEGER;

ALTER TABLE tpch ADD (priority INT);
```

5.2.1.3 DROP

The DROP clause allows you to drop columns. All associated data is removed when the column is dropped. You can DROP COLUMN (column_name).

The following example alters the **orders** table to drop the **priority** column:

```
ALTER TABLE orders DROP COLUMN priority;
```

5.2.1.4 RENAME

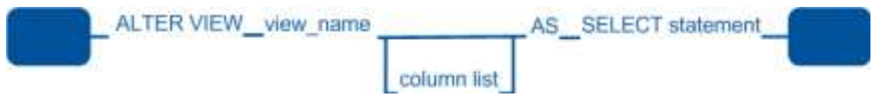
The RENAME clause allows you to rename a table.

The following example renames the **orders** table:

```
ALTER TABLE orders RENAME TO customer_orders;
```

5.2.2 ALTER VIEW

Alters the definition of an InfiniDB view. CREATE OR REPLACE VIEW may also be used to alter the definition of an InfiniDB view.



The following statement alters the definition of the v_cust_orders view to return only open orders:

```
ALTER VIEW v_cust_orders (cust_name, order_number, order_status) as select
c.cust_name, o.ordernum, o.status from customer c, orders o where c.custnum =
o.custnum WHERE o.status = 'O';
```

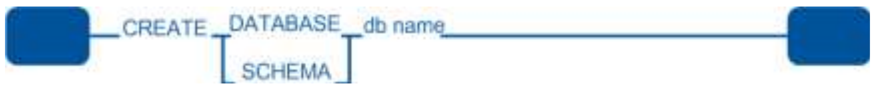
5.2.3 COMMIT

The COMMIT statement makes changes to a table permanent. You should only commit changes after you have verified the integrity of the changed data.

Once data is committed, it cannot be undone with the ROLLBACK statement. To return the database to its former state, you must restore the data from backups. See “ROLLBACK”.

5.2.4 CREATE DATABASE

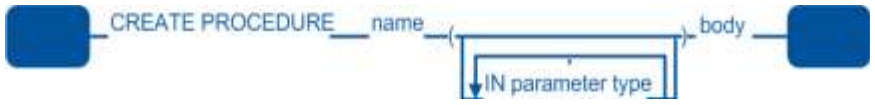
Creates a database/schema in InfiniDB database:



5.2.5 CREATE PROCEDURE

Creates a stored routine in the InfiniDB database.

NOTE: InfiniDB currently accepts definition of stored procedures with only input arguments and a single query while in Operating Mode = 1 (VTABLE mode). However, while in the Operating Mode = 0 (TABLE mode), InfiniDB will allow additional complex definition of stored procedures (i.e., OUT parameter, declare, cursors, etc.) See the "Operating Mode" section for information on Operating Modes.

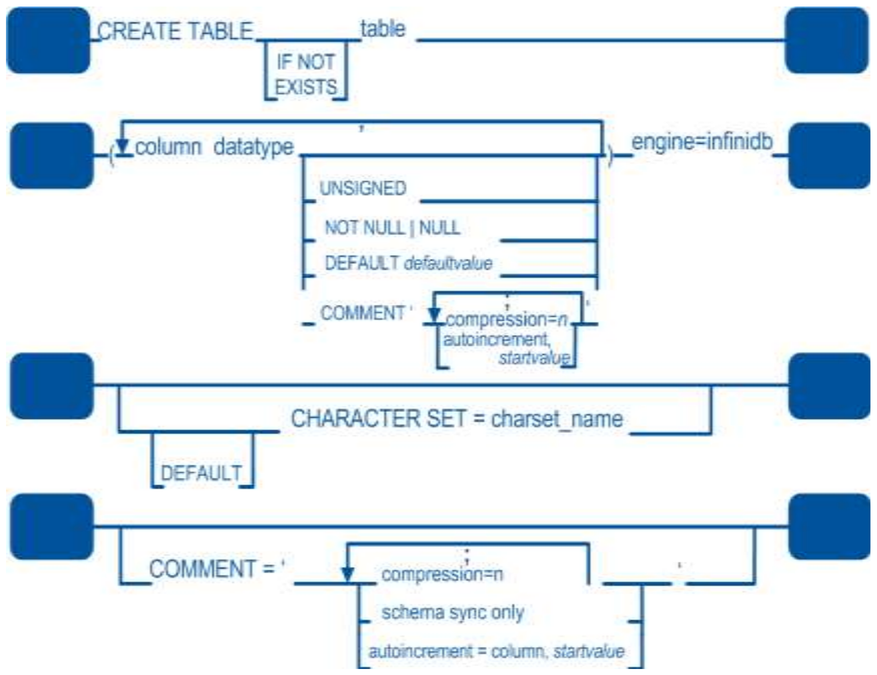


The following statements create and call the sp_complex_variable stored procedure:

```
delimiter $$
CREATE PROCEDURE sp_complex_variable(in arg_key int, in arg_date date)
    begin
         Select *
         from lineitem, orders
         where o_custkey < arg_key
         and     l_partkey < 10000
         and     l_shipdate>arg_date
         and     l_orderkey = o_orderkey
         order by l_orderkey, l_linenumber;
    end
$$

delimiter ;

call sp_complex_variable(1000, '1998-10-10');
```

5.2.6 CREATE TABLE

A database consists of tables that store user data. You can create multiple columns in the create table statement. The datatype follows the column name when adding columns.



Notes to CREATE TABLE:

- InfiniDB tables should not be created in the mysql or information_schema databases.
- All object names are stored in lower case in InfiniDB.
- CREATE TABLE AS SELECT is currently not supported in InfiniDB. You may use this syntax but it will create the table with your current default engine (i.e, MyISAM, etc.)
- A 'compression' comment at the table level will override the system and session default. A 'compression' comment at the column level will override the system, session and table default. Values are:
  0 - do not compress the table/column
  1 - compress the table/column

- A 'schema sync only' comment allows for a table to be created in the front end only. This would be used if, for some reason, the table exists in the back end of the database but not the front. A typical use of this would be in a multiple User Module (UM) scenario where the table has been created on one UM and needs to be synched on the other UMs.
- An 'autoincrement' comment on either the column definition or at the table level will create an InfiniDB autoincrement column. Only one autoincrement column may be defined per table. Please see "Autoincrement Usage in InfiniDB" for further autoincrement processing. If no *startvalue* is given, the default will be 1.

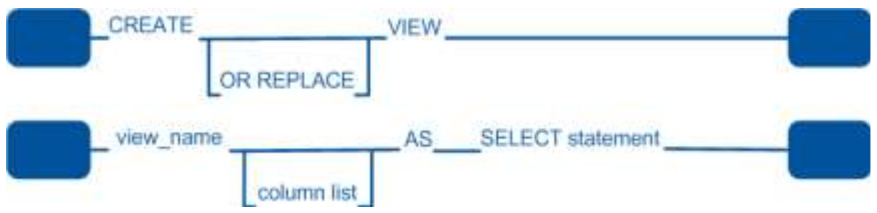
- For maximum compatibility with external tools, InfiniDB will accept the following table declarations; however these are not implemented within InfiniDB.:
  ```
  MIN_ROWS
  MAX_ROWS
  AUTO_INCREMENT
  ```
- A column DEFAULT value is limited to 64 characters in InfiniDB.

The following statement creates a table called **orders** with two columns: **orderkey** with datatype **UNSIGNED integer** defined as NOT NULL and **customer** with datatype **varchar**. The orderkey is also defined to be an auto increment column with a starting value of 1.

```
CREATE TABLE orders
    (orderkey INTEGER UNSIGNED NOT NULL,
     customer VARCHAR(45)
    ) ENGINE=INFINIDB
      COMMENT='autoincrement=orderkey,1';
```

5.2.7 CREATE VIEW

Creates a stored query in the InfiniDB database.



Notes to CREATE VIEW:

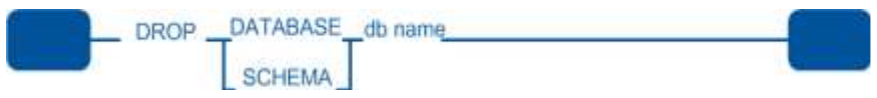
- If you describe a view in InfiniDB, the column types reported may not match the actual column types in the underlying tables. This is normal and can be ignored.

The following statement creates a customer view of orders with status:

```
CREATE VIEW v_cust_orders (cust_name, order_number, order_status) as
select c.cust_name, o.ordernum, o.status from customer c, orders o
where c.custnum = o.custnum;
```

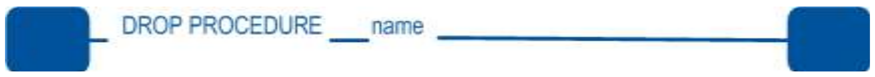
5.2.8 DROP DATABASE

Removes a database/schema from InfiniDB database:



5.2.9 DROP PROCEDURE

The DROP PROCEDURE statement deletes a stored procedure from the InfiniDB database.

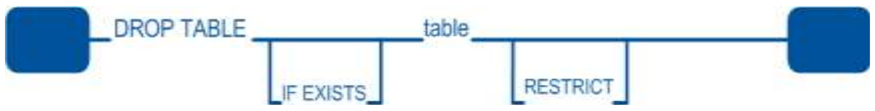


The following statement drops the **sp_complex_variable** procedure:

```
DROP PROCEDURE sp_complex_variable;
```

5.2.10 DROP TABLE

The DROP TABLE statement deletes a table from the InfiniDB database.



Notes to DROP TABLE:

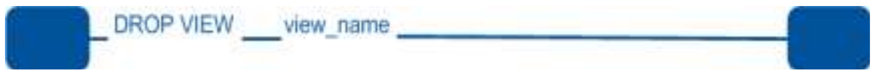
- RESTRICT allows for a table to be dropped in the front end only. This would be used if, for some reason, the table does not exist in the back end of the database but does in the front. A typical use of this would be in a multiple User Module (UM) scenario where the table has been dropped on one UM and needs to be synched on the other UMs.

The following statement drops the **orders** table:

```
DROP TABLE orders;
```

5.2.11 DROP VIEW

The DROP VIEW statement deletes a view from the InfiniDB database.

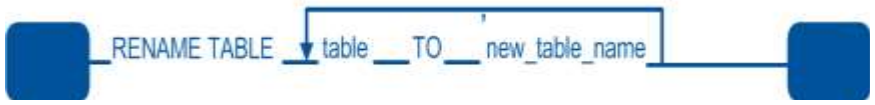


The following statement drops the **v_cust_orders** view:

```
DROP VIEW v_cust_orders;
```

5.2.12 RENAME TABLE

The RENAME TABLE statement renames one or more tables in the InfiniDB database.



Notes to RENAME TABLE:

- You cannot currently use RENAME TABLE to move a table from one database to another.
- See the ALTER TABLE syntax for alternate way to RENAME table.

The following statement renames the **orders** table:

```
RENAME TABLE orders TO customer_order;
```

The following statement renames both the **orders** table and **customer** table:

```
RENAME TABLE orders TO customer_orders,
customer TO customers;
```

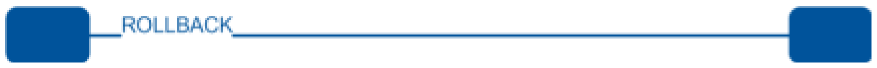
You may also use RENAME TABLE to swap tables. This example swaps the **customer** and **vendor** tables (assuming the **temp_table** does not already exist):

```
RENAME TABLE customer TO temp_table,
vendor TO customer,
temp_table to vendor;
```

5.2.13 ROLLBACK

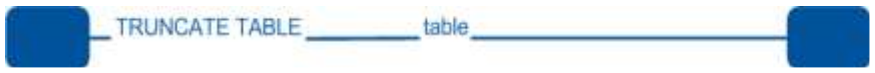
The ROLLBACK statement undoes transactions that have not been permanently saved to the database with the COMMIT statement.

You cannot rollback changes to table properties including ALTER, CREATE, DROP or TRUNCATE TABLE statements.



5.2.14 TRUNCATE TABLE

The TRUNCATE TABLE statement empties a table completely from the InfiniDB database. Logically, TRUNCATE is the same as DROP TABLE and CREATE TABLE statements and allows for a much faster removal of the entire contents of the table.



Notes to TRUNCATE TABLE:

- Users must have the DROP privilege to use this command.

The following statement truncates the **orders** table:

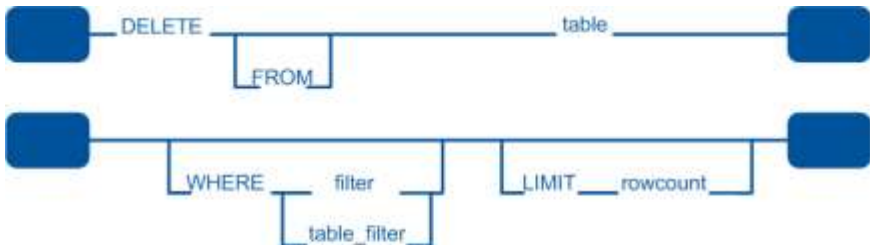
```
TRUNCATE TABLE orders;
```

5.3 DML Statements

The following DML statements are run from the Application User schema. DML statements are used for manipulating data in tables. This includes deleting, inserting, and updating data. DML statements that are native to InfiniDB are listed below in alphabetical order with descriptions and sample syntax statements. Note: LIMIT is currently not supported for DML statements.

5.3.1 DELETE

The DELETE statement is used to remove rows from tables.



Notes on DELETE:

- DELETE does not recover the disk space taken by the rows deleted. If you wish to recover disk space consumed, there are other options to use like TRUNCATE and DROP PARTITION or a combination of reloading the rows (CREATE TABLE and importing/inserting only the rows to be retained, dropping the old table and renaming the new table).
- The LIMIT option to DELETE tells InfiniDB the maximum number of rows to be deleted before control is returned to the client. This can be used to ensure that a given DELETE statement does not take too much time. You can simply repeat the DELETE statement until the number of affected rows is less than the LIMIT value.

The following statement deletes **customer** records with a customer key identification between **1001** and **1999**:

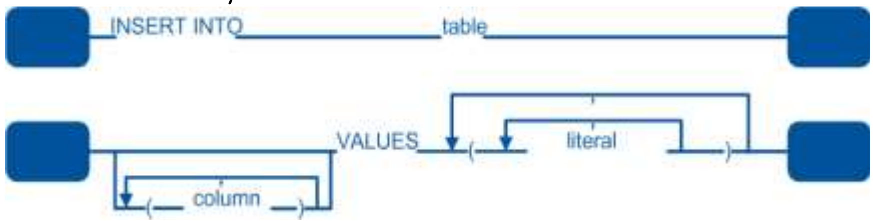
```
DELETE FROM customer WHERE custkey > 1000 and custkey < 2000;
```

The following statement deletes all customers whose name begins with ‘Widgets’:

```
DELETE FROM customer WHERE SUBSTR(name, 1, 7) = ‘Widgets’;
```

5.3.2 INSERT

The INSERT statement allows you to add data to tables.



The following statement inserts a row with all column values into the **customer** table:

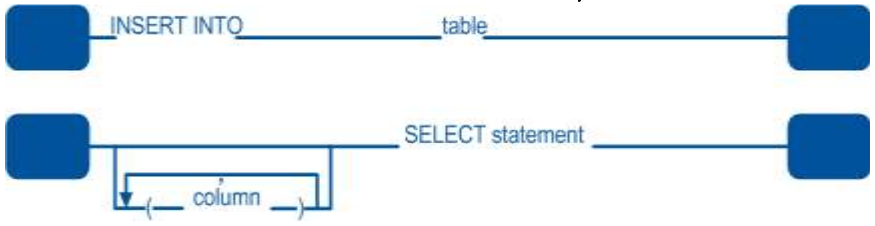
```
INSERT INTO customer (custno, custname, custaddress, phoneno,
cardnumber, comments) VALUES (12, ‘John Smith’, ‘100 First Street,
Dallas’, ‘(214) 555-1212’, 100, ‘On Time’);
```

The following statement inserts two rows with all column values into the **customer** table:

```
INSERT INTO customer (custno, custname, custaddress, phoneno,
cardnumber, comments) VALUES (12, ‘John Smith’, ‘100 First Street,
Dallas’, ‘(214) 555-1212’, 100, ‘On Time’), (13, ‘John Q Public’, ‘200
Second Street, Dallas’, ‘(972) 555-1234’, 200, ‘Late Payment’);
```

5.3.2.1 INSERT...SELECT

The INSERT...SELECT statement allows the insertion of many rows into a table from one or many tables.



Notes on INSERT…SELECT:

- By default, non-transactional INSERT…SELECT is directed to the InfiniDB cpimport tool for significant increase in performance. This can be turned off, if desired. For more information, see “Using cpimport for Batch Insert”.
- Transactional INSERT…SELECT (statements within a START TRANSACTION or statements with AUTOCOMMIT off) are processed through standard DML processes.
- InfiniDB autoincrement columns work as normal.
- The ON DUPLICATE KEY clause will be ignored in InfiniDB.

The following statement inserts on time customers into the customer_ontime table:

```
INSERT INTO customer_ontime (custno, custname, custaddress) SELECT
custno, custname, custaddress from customer where comments = ‘On Time’;
```

5.3.3 LOAD DATA INFILE

The LOAD DATA INFILE statement reads rows from a text file into a table at a very high speed. The file name must be given as a literal string.

LOAD DATA INFILE '*file_name*'
  INTO TABLE *tbl_name*
  [CHARACTER SET charset_name]
  [{FIELDS | COLUMNS}
    [TERMINATED BY '*string*']

```
        [[OPTIONALLY] ENCLOSED BY 'char']
        [ESCAPED BY 'char']
    ]
    [LINES
        [STARTING BY 'string']
        [TERMINATED BY 'string']
    ]
```

Notes on LOAD DATA INFILE:

- By default, Non-transactional LOAD DATA INFILE is directed to the InfiniDB cpimport tool for significant increase in performance. This can be turned off if desired. For more information, see "Using cpimport for Batch Insert".
- Transactional LOAD DATA INFILE (statements within a START TRANSACTION or statements with AUTOCOMMIT off) are processed through standard DML processes.

The following example loads data into the a simple 5 column table:

A file named simpletable.tbl has the following data in it.

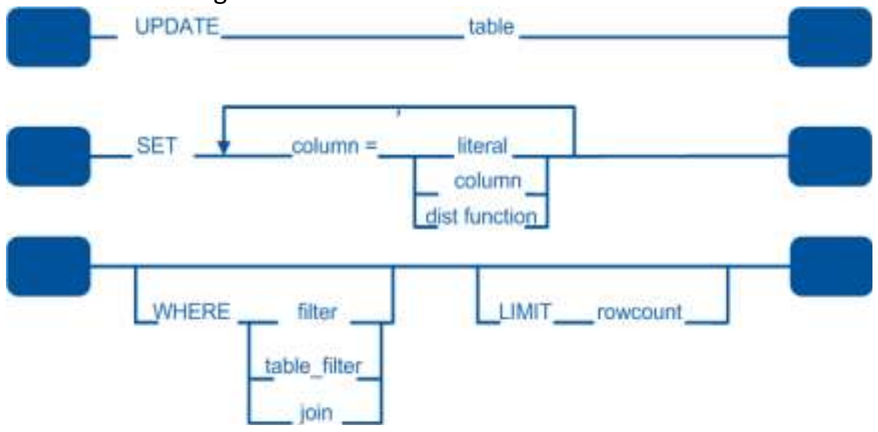
```
1|100|1000|10000|Test Number 1|
2|200|2000|20000|Test Number 2|
3|300|3000|30000|Test Number 3|
```

The data can then be loaded into the simpletable table with the following syntax:

```
LOAD DATA INFILE '/home/source/simpletable.tbl' INTO TABLE simpletable
  FIELDS TERMINATED BY '|';
```

5.3.4 UPDATE

The UPDATE statement changes data stored in rows.



NOTE: The LIMIT option to UPDATE tells InfiniDB the maximum number of rows to be updated before control is returned to the client. This can be used to ensure that a given UPDATE statement does not take too much time. You can simply repeat the UPDATE statement until the number of affected rows is less than the LIMIT value.

The following statement updates the WidgetFactory supplier name in the **supplier** table to WidgetsInc:

```
UPDATE supplier SET name = 'WidgetsInc.' WHERE name = 'WidgetFactory';
```

The following statement updates delivery met flag in the orders table where the item was shipped ahead of the estimated shipping date.

```
UPDATE orders SET delivery_met = 'Y' WHERE shipdate <
estimated_shipdate;
```

The following statement defaults the customer name with the literal 'Customer_' and the customer key.

```
UPDATE customer set name = concat('Customer_', custkey);
```

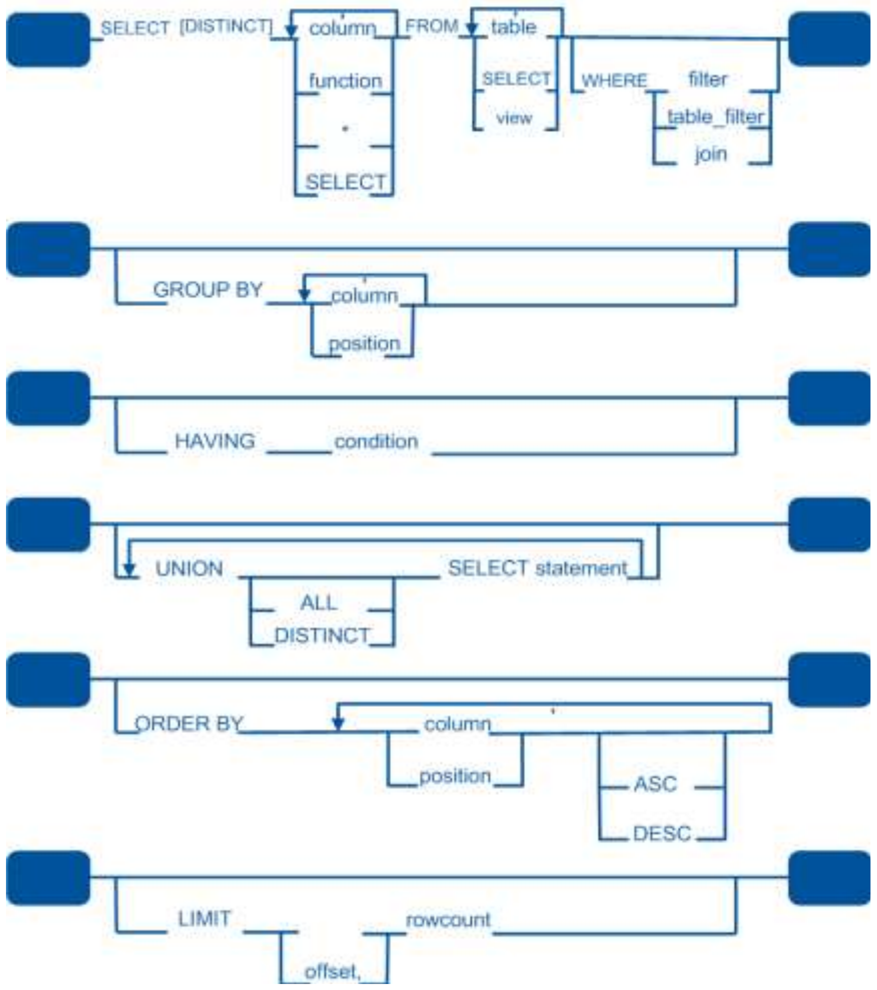
Update with multi-table syntax

The following examples describe an update of a table in a multi-table syntax format:

```
update table2, table1 set table2.name=table1.name where table1.id =
   table2.id;
update table2 set name = (select name from table1 where table1.id =
   table2.id);
update table1 join table2 on table1.id = table2.id set table1.name =
   table2.name where table1.id = table2.id;
```

5.4 SELECT

The SELECT statement is used to query the database and display table data. You can add many clauses to filter the data.



5.4.1 Projection List (SELECT) Notes

If the same column needs to be referenced more than once in the projection list, a unique name is required for each column using a column alias.

The total length of the name of a column, inclusive of length of functions, in the projection list must be 64 characters or less.

5.4.2 WHERE

The WHERE clause filters data retrieval based on criteria. **NOTE:** column_alias cannot be used in the WHERE clause.

The following statement returns rows in the region table where the **region** = ‘ASIA’:

```
SELECT * FROM region WHERE name = ’ASIA’;
```

5.4.3 GROUP BY

GROUP BY groups data based on values in one or more specific columns.

The following statement returns rows from the **lineitem** table where **orderkey** is less than 1000000 and groups them by the quantity.

```
SELECT quantity, count(*) FROM lineitem WHERE orderkey < 1000000 GROUP
BY quantity;
```

The following statement returns the **custkey** rows that are returned from the **orders** table and the **customer** table:

```
SELECT custkey FROM orders INTERSECT SELECT custkey FROM customer;
```

5.4.4 HAVING

HAVING is used in combination with the GROUP BY clause. It can be used in a SELECT statement to filter the records that a GROUP BY returns.

The following statement returns shipping dates, and the respective quantity where the quantity is 2500 or more.

```
SELECT shipdate, count(*) FROM lineitem GROUP BY shipdate HAVING
count(*) >= 2500;
```

The following statement returns unique rows with the **partkey** from the **partsupp** table minus the partkey in the **part** table:

```
SELECT partkey FROM partsupp MINUS SELECT partkey FROM part;
```

5.4.5 ORDER BY

The ORDER BY clause presents results in a specific order.

Note: The ORDER BY clause represents a statement that is post processed by MySQL. Since the ORDER BY is a clause that is post processed by MySQL, it should remain on the outer most portion of the query. Any subqueries containing an ORDER BY clause will return the correct data, but not necessarily be returned in the correct order.

The following statement returns an ordered **quantity** column from the **lineitem** table.

```
SELECT quantity FROM lineitem WHERE orderkey < 1000000 order by quantity;
```

The following statement returns an ordered **shipmode** column from the **lineitem** table.

```
Select shipmode from lineitem where orderkey < 1000000  order by 1;
```

5.4.6 UNION

Used to combine the result from multiple SELECT statements into a single result set.

The UNION or UNION DISTINCT clause returns query results from multiple queries into one display and discards duplicate results. The UNION ALL clause displays query results from multiple queries and does not discard the duplicates.

The following statement returns the p_name rows in the part table and the partno table and discards the duplicate results:

```
SELECT p_name FROM part UNION select p_name FROM partno;
```

The following statement returns **all** the p_name rows in the part table and the partno table:

```
SELECT p_name FROM part UNION ALL select p_name FROM partno;
```

5.4.7 LIMIT

Used to constrain the number of rows returned by the SELECT statement. LIMIT can have up to two arguments. LIMIT must contain a rowcount and may optionally contain an offset of the first row to return (initial row is 0).

The following statement returns 5 customer keys from the customer table:

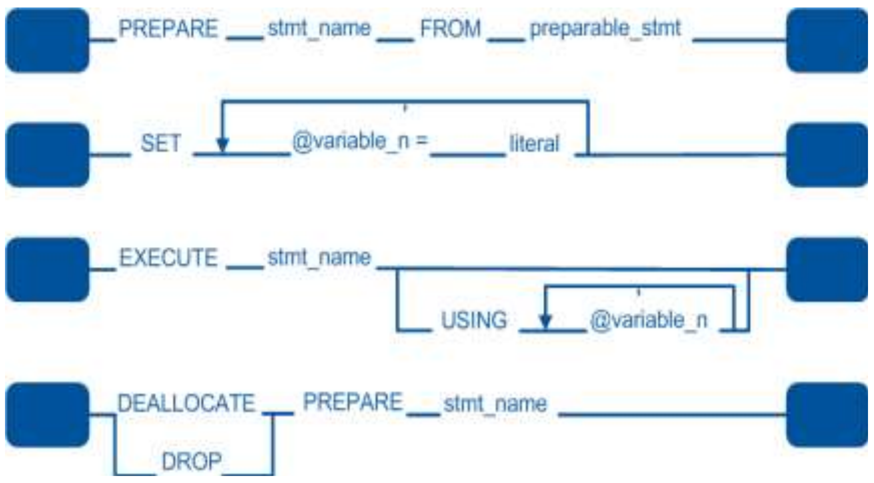
```
SELECT custkey from customer limit 5;
```

The following statement returns 5 customer keys from the customer table beginning at offset 1000:

```
SELECT custkey from customer limit 1000,5;
```

5.5 Prepared Statements

Prepared statements are used to send SQL statements to the server to be executed. SQL syntax for prepared statements consists of 4 SQL statements: PREPARE, SET, EXECUTE and DEALLOCATE/DROP PREPARE.



5.5.1 PREPARE

The PREPARE statement prepares a statement and assigns it a name, stmt_name, by which to refer to the statement later. The text must represent a single SQL statement. Within the statement, "?" characters can be used as parameters to indicate where data values are to be bound to the query later when you execute it. The "?" characters should not be enclosed within quotes, even if you intend to bind them to string values.

```
PREPARE test1 FROM
  "SELECT
     SUM(L_EXTENDEDPRICE*L_DISCOUNT) AS REVENUE
   FROM
     LINEITEM
   WHERE
     L_SHIPDATE >= ? AND
     L_SHIPDATE < ? + interval '1' year AND
     L_DISCOUNT BETWEEN ? - 0.01 AND ? + 0.01 AND
     L_QUANTITY < ?;";
```

5.5.2 SET

The SET statement initializes any variables used in the PREPARE statement.

```
SET @v1=date '1994-01-01';
SET @v2=0.06;
SET @v3=24;
```

5.5.3 EXECUTE

After preparing a statement with PREPARE, you execute it with an EXECUTE statement that refers to the prepared statement name. If the prepared statement contains any parameters, you must supply a USING clause that lists user variables containing the values to be bound to the parameters. Parameter values can be supplied only by user variables, and the USING clause must name exactly as many variables as the number of parameter markers in the statement.

You can execute a given prepared statement multiple times. Use the SET statement to change any parameter values before each execution, otherwise the previously set parameter values will be used.

```
EXECUTE test1 using @v1, @v1, @v2, @v2, @v3;
```

5.5.4 DEALLOCATE/DROP PREPARE

Removes the prepared statement.

```
DROP PREPARE test1;
```

6 Operating Mode

InfiniDB has the ability to support full MySQL query syntax through an operating mode. This operating mode may be set as a default for the instance or set at the session level.

Please refer to the InfiniDB Administrator's Guide for setting the default operating mode.

To set the operating mode at the session level, the following command is used. Once the session has ended, any subsequent session will return to the default for the instance.

```
set infinidb_vtable_mode = n
```

where *n* is:

- 0) a generic, highly compatible row-by-row processing mode. Some WHERE clause components can be processed by InfiniDB, but joins are processed entirely by mysqld using a nested-loop join mechanism
- 1) (the default) query syntax is evaluated by InfiniDB for compatibility with distributed execution and incompatible queries are rejected. Queries executed in this mode take advantage of distributed execution and typically result in higher performance.
- 2) auto-switch mode: InfiniDB will attempt to process the query internally, if it cannot, it will automatically switch the query to run in row-by-row mode.

NOTE: For more information on supported query syntax above and beyond the supported InfiniDB syntax for modes 0 and 2, please refer to the MySQL 5.1 Reference Manual.

7 Decimal to Double Math

InfiniDB has the ability to change intermediate decimal mathematical results from decimal type to double. The decimal type has approximately 17-18 digits of precision, but a smaller maximum range whereas the double type has approximately 15-16 digits of precision, but a much larger maximum range (refer to the Datatypes section above for details). Therefore, the proper setting depends on the intended usage. In typical mathematical and scientific applications, the ability to avoid overflow in intermediate results with double math is likely more beneficial than the additional two digits of precisions. In financial applications, however, it may be more appropriate to leave in the default decimal setting to ensure accuracy to the least significant digit.

The `infinidb_double_for_decimal_math` variable is used by InfiniDB to control the data type for intermediate decimal results. This decimal for double math may be set as a default for the instance, set at the session level, or at the statement level by toggling this variable on and off.

Please refer to the InfiniDB Administrator's Guide for setting the default decimal to double math at the instance level.

7.1 Enable/Disable Decimal to Double Math

To enable/disable the use of the decimal to double math at the session level, the following command is used. Once the session has ended, any subsequent session will return to the default for the instance.

```
set infinidb_double_for_decimal_math = n
```

where *n* is:

- 0 (disabled)
- 1 (enabled)

8 Decimal Scale

InfiniDB has the ability to support varied internal precision on decimal calculations.

infinidb_decimal_scale is used internally by the InfiniDB engine to control how many significant digits to the right of the decimal point are carried through in suboperations on calculated columns. If, while running a query, you receive the message 'aggregate overflow', try reducing infinidb_decimal_scale and running the query again. Note that, as you decrease infinidb_decimal_scale, you may see reduced accuracy in the least significant digit(s) of a returned calculated column.

infinidb_use_decimal_scale is used internally by the InfiniDB engine to turn on and off the use if this internal precision.

These two system variables may be set as a default for the instance or set at the session level.

Please refer to the InfiniDB Administrator's Guide for setting the default decimal scale.

8.1 Enable/Disable Decimal Scale

To enable/disable the use of the decimal scale at the session level, the following command is used. Once the session has ended, any subsequent session will return to the default for the instance.

```
set infinidb_use_decimal_scale = n
```

where *n* is:

- 0 (disabled)
- 1 (enabled)

8.2 Set Decimal Scale Level

To set the decimal scale at the session level, the following command is used. Once the session has ended, any subsequent session will return to the default for the instance.

```
set infinidb_decimal_scale = n
```

where *n* is the amount of precision desired for calculations.

9 Using cpimport for Batch Insert

InfiniDB has the ability to utilize the cpimport fast data import tool for non-transactional "LOAD DATA INFILE' and "INSERT INTO SELECT FROM" SQL statements. Using this method results in a significant increase in performance in loading data through these two SQL statements. There are two variables used to control this ability:

- The `infinidb_use_import_for_batchinsert` variable is used by InfiniDB to control if cpimport is used for these statements. This use import for batch insert may be set as a default for the instance, set at the session level, or at the statement level by toggling this variable on and off.
- The `infinidb_import_for_batchinsert_delimiter` variable is used internally by InfiniDB on a non-transactional INSERT INTO SELECT FROM statement as the default delimiter passed to the cpimport tool. With a default value ascii 7, there should be no need to change this value unless your data contains ascii 7 values.

Please refer to the InfiniDB Administrator's Guide for setting the default use cpimport for batch insert at the instance level.

9.1 Enable/Disable Using cpimport for Batch Insert

To enable/disable the use of the use cpimport for batch insert at the session level, the following command is used. Once the session has ended, any subsequent session will return to the default for the instance.

```
set infinidb_use_import_for_batchinsert = n
```

where *n* is:
- 0 (disabled)
- 1 (enabled)

9.2 Changing Default Delimiter for INSERT SELECT

The current default of ascii 7 should be sufficient for most cases. But if your data contains ascii 7 values, you will need to change this default to another ascii value. To change this value at the at the session level, the following command is used. Once the session has ended, any subsequent session will return to the default for the instance.

```
set  infinidb_import_for_batchinsert_delimiter = ascii_value
```

where *ascii_value* is an ascii value representation of the delimiter desired.

10 Using Disk Based Joins

InfiniDB performs in-memory joins on the UM node and when a join operation exceeds the memory allocated on the UM for query joins and aggregation, the query is aborted with an error code. The option is available to have such queries using disk for storing intermediate join data when the memory required for the join exceeds the UM memory limit. While such query performance will be slower than the joins that completely fit in memory and still bound by the temp space availability – it increases the probability of completion of those joins that have large cardinality and memory needs.

- Currently excluded from disk based join processing are:
  - Aggregation, including count(distinct)
  - DML

There are new configuration variables that allow you to manage this capability at the instance level. These variables reside in `HashJoin` element in the `Calpont.xml` configuration file (residing in the `etc` directory for your InfiniDB installation):

- `AllowDiskBasedJoin` – Controls the option to use disk Based joins or not. Valid values are Y (enabled) or N (disabled). By default, this option is disabled.
- `TempFileCompression` – Controls whether the disk join files are compressed or non-compressed. Valid values are Y (use compressed files) or N (use non-compressed files).
- `TempFilePath` – The directory path used for the disk joins. By default, this path is the `tmp` directory for your InfiniDB installation (i.e., `/usr/local/Calpont/tmp`). Files (named `infinidb-join-data*`) in this directory will be created and cleaned on an as needed basis. The entire directory is removed and recreated by ExeMgr at startup.)

  Note: When using disk based joins, it is strongly recommended that the `TempFilePath` reside on its own partition because the partition may fill up as queries are executed.

In addition to the above instance wide variables, a session level variable exists to limit the amount of memory used by a user before the join is switched over to a disk based join. It can be set either at the instance level or session level (refer to the InfiniDB Administrator's Guide for instance level modification).

- `infinidb_um_mem_limit` - Memory limit in MB per user (i.e. switch to disk based join if this limit is exceeded). By default, this limit is not set (value of 0).

  For modification at the session level:

  ```
  set infinidb_um_mem_limit = value;
  ```

  where *value* is the value in Mb for in memory limitation per user.

Other session level variables exist that are used internally by InfiniDB for the processing of these disk based joins. **These should not be changed unless instructed to do so by InfiniDB Support.** If instructed to do so by InfiniDB Support, they can be set either at the instance level or session level (refer to the InfiniDB Administrator's Guide for instance level modification).

- `infinidb_diskjoin_smallsidelimit` - Max disk space allowed in MB per user for small side results.
- `infinidb_diskjoin_largesidelimit` - Max disk space allowed in MB per user for large side results.
- `infinidb_diskjoin_bucketsize` - Max bucket size in MB. Bucket will split into multiple smaller buckets when it reaches this size.

  For modification at the session level:

```
set infinidb_diskjoin...  = value
```

  where *value* is the value in Mb to use as instructed by InfiniDB Support.

11 Compression Mode

InfiniDB has the ability to compress data and is controlled through a compression mode. This compression mode may be set as a default for the instance or set at the session level.

Please refer to the InfiniDB Administrator's Guide for setting the default compression mode at the instance level.

To set the compression mode at the session level, the following command is used. Once the session has ended, any subsequent session will return to the default for the instance.

```
set infinidb_compression_type = n
```

where *n* is:

- 0) compression is turned off. Any subsequent table create statements run will have compression turned off for that table unless any statement overrides have been performed. Any alter statements run to add a column will have compression turned off for that column unless any statement override has been performed.
- (1 or 2) compression is turned on. Any subsequent table create statements run will have compression turned on for that table unless any statement overrides have been performed. Any alter statements run to add a column will have compression turned on for that column unless any statement override has been performed. SEE NOTE BELOW.

Compression Value Note: Starting with InfiniDB versions 3.0.6, 3.5.1 (and for all future releases), we have retired compression type 1. You cannot compress data using that algorithm. In order to minimize the effect of this change on our customers, InfiniDB will treat a request to use compression type 1 as a request for type 2 compression. Existing data compressed with type 1 will still be readable, but if you make changes to it, it will be recompressed using type 2 compression. All of this is regardless of the infinidb_compression_type variable setting and the value stored in calpontsys.syscolumn. You can still choose to disable compression by setting infinidb_compression_type to 0, but you cannot select to compress new data using type 1 compression. If you try to use type 1 compression, InfiniDB will silently treat it as a request to use type 2 compression.

In other words, you can turn compression off by setting infinidb_compression_type to 0. You can turn compression on by setting infinidb_compression_type to 1 or 2, but InfiniDB will always use the type 2 algorithm to compress new data regardless of whether you set infinidb_compression_type to 1 or 2.

This is not fundamentally different from previous InfiniDB versions where you only had the choice of compression being on or off: you still only have the choice of compression being on or off, but InfiniDB will be using a new algorithm (which we call "type 2").

12 Local PM Query

InfiniDB has the ability to query data from just a single PM instead of the whole database thru the UM. In order to accomplish this, the `infinidb_local_query` variable in the my.cnf configuration file is used and maybe set as a default for the instance or set at the session level. This variable applies only to executing a query on an individual PM and will error if executed on the UM. The PM must be set up with the local query option during installation.

Please refer to the InfiniDB Administrator's Guide for setting the default decimal to double math at the instance level.

12.1 Enable/Disable Local PM Query

To enable/disable the use of the local PM Query at the session level, the following command is used. Once the session has ended, any subsequent session will return to the default for the instance.

```
set infinidb_local_query = n
```

where *n* is:

- 0 (disabled)
- 1 (enabled)

13 Partition Management

InfiniDB has the ability to better manage the removal of data by managing the disablement and drop of partitions.   For more information on partitions in InfiniDB, please see InfiniDB Storage Concepts in the InfiniDB Concepts Guide.

Caution should be used with these commands as they are destructive.

There are 2 ways to manage the removal or disablement of partitions:

- Column Value
- Partition Number

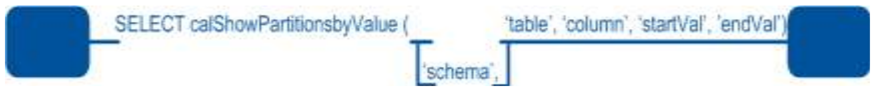
13.1 Partition Management by Column Value

The following functions allow the management of partitions through the use of specific values in a column:

13.1.1 Show Partition Information by Column Value

To display the partition information for a particular column, the calShowPartitionsbyValue function would be used. The information from this function would be used to reveal the partitions whose minimum and maximum values fall completely within the entered values well as the status of that partition. This information would then be used in deciding what, if any, partitions would be disabled, re-enabled and/or removed.

- If the column's min/max value of an extent completely falls within the startVal and endVal, the appropriate action will be taken for that extent.
- Only casual partition column types (INTEGER, DECIMAL, DATE, DATETIME, CHAR up to 8 bytes and VARCHAR up to 7 bytes) are supported with these 'value' partition functions.


SELECT calShowPartitionsbyValue (
'schema',
'table', 'column', 'startVal', 'endVal')

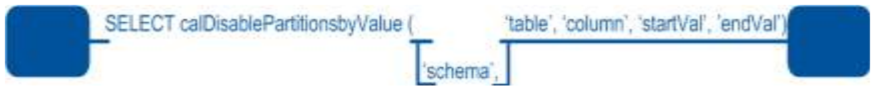
The following execution of calShowPartitions displays 3 partitions of data for the orderdate column that fall into the entered range.

```
mysql> select calShowPartitionsByValue('orders','o_orderdate', '1992-01-01', '20
10-10-31');
+--------------------------------------------------------------------------------
--------------------------------------------------------------------------------+

| calShowPartitionsByValue('orders','o_orderdate', '1992-01-01', '2010-10-31')
                                                                               |

mysql> select calShowPartitionsByValue('orders','orderdate', '1992-01-01', '2010-07-24');
+-----------------------------------------------------------------+
|calShowPartitionsbyvalue('orders','orderdate', '1992-01-02', '2010-07-24')|
+-----------------------------------------------------------------+
| Part#      Min             Max               Status
  0.0.1      1992-01-01      1998-08-02        Enabled
  0.1.2      1998-08-03      2004-05-15        Enabled
  0.2.3      2004-05-16      2010-07-24        Enabled          |
+-----------------------------------------------------------------+
1 row in set (0.05 sec)
```

13.1.2 Disable Partition by Value

After analysis of the partition information from calShowPartitionsbyValue, if a decision is made to disable the partition instead of dropping it, the calDisablePartitionsbyValue function would be used. A disabled partition still exists on the file system but will not participate in any query, DML or import activity.

- If the column's min/max value of an extent completely falls within the startVal and endVal, the appropriate action will be taken for that extent.
- Only casual partition column types (INTEGER, DECIMAL, DATE, DATETIME, CHAR up to 8 bytes and VARCHAR up to 7 bytes) are supported with these 'value' partition functions.


SELECT calDisablePartitionsbyValue (
'table', 'column', 'startVal', 'endVal')
'schema',

The following execution of calDisablePartitionsbyValue disables the first partition:

```
mysql> select calDisablePartitionsByValue('orders','orderdate', '1992-01-01', '2004-05-
15');
+----------------------------------------------------------------------------+
| caldisablepartitionsbyvalue ('orders', 'o_orderdate','1992-01-01','1998-08-02') |
+----------------------------------------------------------------------------+
| Partitions are disabled successfully                                       |
+----------------------------------------------------------------------------+
1 row in set (0.28 sec)
```

The result showing the fist partition has been disabled:

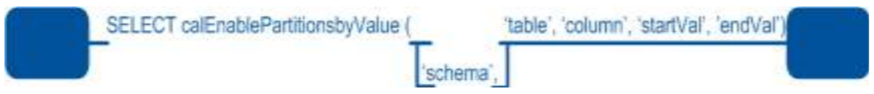
```
mysql> select calShowPartitionsByValue('orders','orderdate', '1992-01-01', '2010-07-24');
+-----------------------------------------------------------------+
|calShowPartitionsbyvalue('orders','orderdate', '1992-01-02','2010-07-24’ )|
+-----------------------------------------------------------------+
| Part#      Min            Max                Status
  0.0.1      1992-01-01     1998-08-02         Disabled
  0.1.2      1998-08-03     2004-05-15         Enabled
  0.2.3      2004-05-16     2010-07-24         Enabled          |
+-----------------------------------------------------------------+
1 row in set (0.05 sec)
```

13.1.3 Enable Partition by Value

After analysis of the partition information from calShowPartitionsbyValue, if a decision is made to enable the partition, the calEnablePartitionsbyValue function would be used.

- If the column's min/max value of an extent completely falls within the startVal and endVal, the appropriate action will be taken for that extent.

- Only casual partition column types (INTEGER, DECIMAL, DATE, DATETIME, CHAR up to 8 bytes and VARCHAR up to 7 bytes) are supported with these 'value' partition functions.


SELECT calEnablePartitionsbyValue (
'schema',
'table', 'column', 'startVal', 'endVal')

The following execution of calDisablePartitionsbyValue disables the first partition:

```
mysql> select calEnablePartitionsByValue('orders','orderdate', '1992-01-01', '2004-05-
15');
+----------------------------------------------------------------------------+
| calenablepartitionsbyvalue ('orders', 'o_orderdate','1992-01-01','1998-08-02')|
+----------------------------------------------------------------------------+
| Partitions are enabled successfully                                        |
+----------------------------------------------------------------------------+
1 row in set (0.28 sec)
```

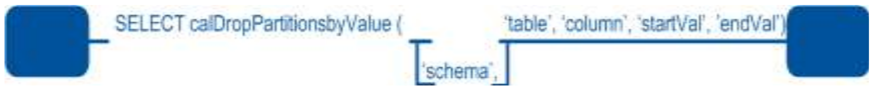
The result showing the first partition has been disabled:

```
mysql> select calShowPartitionsByValue('orders','orderdate', '1992-01-01', '2010-07-
24');
+-----------------------------------------------------------------+
|calShowPartitionsbyvalue('orders','orderdate', '1992-01-02','2010-07-24' )|
+-----------------------------------------------------------------+
| Part#      Min              Max                Status
  0.0.1      1992-01-01       1998-08-02         Enabled
  0.1.2      1998-08-03       2004-05-15         Enabled
  0.2.3      2004-05-16       2010-07-24         Enabled          |
+-----------------------------------------------------------------+
2 rows in set (0.05 sec)
```

13.1.4 Drop Partition by Value

After analysis of the partition information from calShowPartitionsbyValue, if a decision is made to drop the partition, the calDropPartitionsbyValue function would be used. A partition may be dropped from both an Enabled or Disabled state.

- If the column's min/max value of an extent completely falls within the startVal and endVal, the appropriate action will be taken for that extent.
- Only casual partition column types (INTEGER, DECIMAL, DATE, DATETIME, CHAR up to 8 bytes and VARCHAR up to 7 bytes) are supported with these 'value' partition functions.


SELECT calDropPartitionsbyValue (
'schema',
'table', 'column', 'startVal', 'endVal')

The following execution of calDropPartitionsbyValue drops the first partition:

```
mysql> select calDropPartitionsByValue('orders','orderdate', '1992-01-01', '2004-05-
15');
+-----------------------------------------------------------------------------+
| caldroppartitionsbyvalue ('orders', 'o_orderdate','1992-01-01','1998-08-02') |
+-----------------------------------------------------------------------------+
| Partitions are enabled successfully                                        |
+-----------------------------------------------------------------------------+
1 row in set (0.28 sec)
```

The result showing the first partition has been dropped:

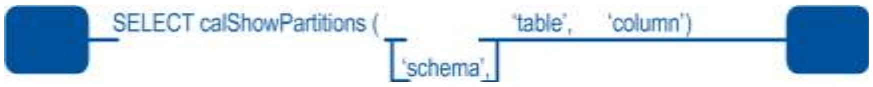
```
mysql> select calShowPartitionsByValue('orders','orderdate', '1992-01-01', '2010-07-
24');
+-----------------------------------------------------------------+
|calShowPartitionsbyvalue('orders','orderdate', '1992-01-02','2010-07-24' )|
+-----------------------------------------------------------------+
| Part#      Min             Max               Status
  0.1.2      1998-08-03      2004-05-15        Enabled
  0.2.3      2004-05-16      2010-07-24        Enabled          |
+-----------------------------------------------------------------+
1 row in set (0.05 sec)
```

13.2 Partition Management by Partition Number

The following functions allow the management of partitions through the use of partition number for a column:

13.2.1 Show Partition Information by Partition Number

To display the partition information for a particular column, the calShowPartitions function would be used. The information from this function would be used to reveal the minimum and maximum values for that column as well as the status of that partition. The minimum and maximum values would then be used in deciding what, if any, partitions would be disabled, re-enabled and/or removed.

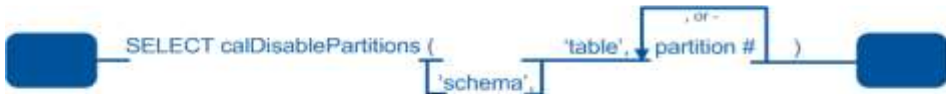


The following execution of calShowPartitions displays 3 partitions of data for the orderdate column:

```
mysql> select calShowPartitions('orders','orderdate');
+--------------------------------------------------------------+
| calShowPartitions('orders','orderdate')                      |
+--------------------------------------------------------------+
| Part#      Min            Max               Status
  0.0.1      1992-01-01     1998-08-02        Enabled
  0.1.2      1998-08-03     2004-05-15        Enabled
  0.2.3      2004-05-16     2010-07-24        Enabled     |
+--------------------------------------------------------------+
1 row in set (0.05 sec)
```

13.2.2 Disable Partition by Partition Number

After analysis of the partition information from calShowPartitions, if a decision is made to disable the partition instead of dropping it, the calDisablePartitions function would be used. A disabled partition still exists on the file system but will not participate in any query, DML or import activity.



- partition # may be a single partition (‘0.0.1’) or a list of partitions (‘0.0.1,0.1.2’)

The following execution of calDisablePartitions disables partition 0.0.1 and 0.1.2:

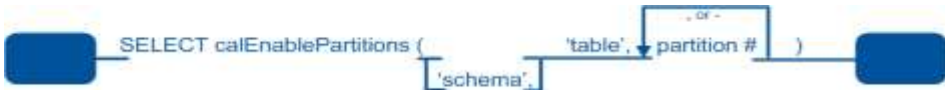
```
mysql> select calDisablePartitions ('mydb','orders', '0.0.1, 0.1.2');
+-------------------------------------------------------+
| calDisablePartitions('mydb','orders', '0.0.1, 0.1.2')|
+-------------------------------------------------------+
| Partitions are disabled.                              |
+-------------------------------------------------------+
1 row in set (0.28 sec)
```

The result showing partitions 0.0.1 and 0.1.2 have been disabled:

```
mysql> select calShowPartitions ('orders','orderdate');
+---------------------------------------------------------------+
| calShowPartitions('orders','orderdate')                       |
+---------------------------------------------------------------+
| Part#      Min             Max                Status
  0.0.1      1992-01-01      1998-08-02         Disabled
  0.1.2      1998-08-03      2004-05-15         Disabled
  0.2.3      2004-05-16      2010-07-24         Enabled     |
+---------------------------------------------------------------+
3 rows in set (0.05 sec)
```

13.2.3 Enable Partition by Partition Number

After analysis of the partition information from calShowPartitions, if a decision is made to enable the partition, the calEnablePartitions function would be used.



- partition # may be a single partition (‘0.0.1’) or a list of partitions (‘0.0.1,0.1.2’)

The following execution of calEnablePartitions enables partition 0.0.1 and 0.1.2:

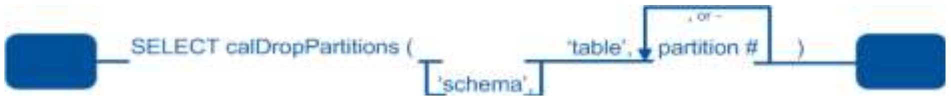
```
mysql> select calEnablePartitions ('mydb','orders', '0.0.1,0.1.2');
+--------------------------------------------------+
| calEnablePartitions('mydb','orders','0.0.1,0.1.2'|
+--------------------------------------------------+
| Partitions are enabled.                          |
+--------------------------------------------------+
1 row in set (0.28 sec)
```

The result showing partitions 0.0.1 and 0.1.2 have been enabled:

```
mysql> select calShowPartitions('orders','orderdate');
+--------------------------------------------------------------+
| calShowPartitions('orders','orderdate')                      |
+--------------------------------------------------------------+
| Part#      Min             Max                 Status
  0.0.1      1992-01-01      1998-08-02          Enabled
  0.1.2      1998-08-03      2004-05-15          Enabled
  0.2.3      2004-05-16      2010-07-24          Enabled     |
+--------------------------------------------------------------+
3 rows in set (0.05 sec)
```

13.2.4 Drop Partition by Partition Number

After analysis of the partition information from calShowPartitions, if a decision is made to drop the partition, the calDropPartitions function would be used. A partition may be dropped from both an Enabled or Disabled state.



- partition # may be a single partition (‘0.0.1’) or a list of partitions (‘0.0.1,0.1.2’)

The following execution of calDropPartitions drops partitions 0.0.1 and 0.1.2:

```
mysql> select calDropPartitions ('orders', '0.0.1,0.1.2');
+---------------------------------------------+
| calDropPartitions('orders', '0.0.1,0.1.2')  |
+---------------------------------------------+
| Partitions are dropped.                     |
+---------------------------------------------+
1 row in set (0.28 sec)
```

The result showing partitions 0.0.1 and 0.1.2 have been dropped from the database:

```
mysql> select calShowPartitions('orders','orderdate');
+-------------------------------------------------------------+
| calShowPartitions('orders','orderdate')                     |
+-------------------------------------------------------------+
| Part#      Min              Max                Status
  0.2.3      2004-05-16       2010-07-24         Enabled     |
+-------------------------------------------------------------+
3 rows in set (0.05 sec)
```

14 Autoincrement Usage in InfiniDB

While the InfiniDB autoincrement column attribute shares its name with a similar MySQL column attribute, it is not identical in behavior to MySQL's implementation. You must carefully read and understand how autoincrement columns work in InfiniDB or you will almost certainly get unexpected results from using this attribute.

You may define a single column in a table as autoincrement. The column must be an integer type. InfiniDB will assign this column a unique value for each row that:

- Is inserted with an INSERT statement, a LOAD DATA INFILE statement, or cpimport and has a value coded, either explicitly or implicitly, as NULL, or is coded explicitly as 0 (zero).
- Is updated to the explicit values of NULL or 0 (zero). Note that if you update a column with the value of another column or with the result of an expression, the column will be updated to that value, regardless of whether the ultimate evaluation is NULL or 0 (zero).

If you never manually update an autoincrement column, InfiniDB will always use a unique value. There is no guarantee that these values are consecutive or even increasing, only that they are unique. However, autoincrement columns are not constrained to unique values. If you insert or update a row and explicitly code a value [other than NULL or 0 (zero)], InfiniDB will insert that value (provided it meets all other requirements for that datatype), even if it causes duplicate values for the column, and no error or warning will be issued.

If you change the table-level starting autoincrement value to a value higher than any value in the table, InfiniDB will begin using that new value on the next insert or update (and possibly introduce a gap in the numbers). If you change the value to a value lower than any value in the table, InfiniDB will still begin using that new value on subsequent inserts and updates, and thus duplicates will be created. No error or warning will be issued in either case.

If the InfiniDB system is unable to generate sequence numbers because they will not fit into the destination datatype (e.g. if the autoincrement column is a TINYINT datatype and you attempt to insert more than about 127 rows), the statement or job will fail. At this point you must either reset the autoincrement settings or drop the autoincrement column and add a new autoincrement column with a wider datatype.

LOAD DATA INFILE will yield warnings from MySQL for NULL values in the source file on the autoincrement columns. To avoid these warnings, use 0 (zero) instead of NULL for the autoincrement column.

15 Using UTF-8 Character Sets

15.1 UTF-8 Character Set

InfiniDB has the ability to support UTF-8 character sets. This profile may be set as a default for the instance or set at the session level.

Please refer to the InfiniDB Administrator's Guide for setting the UTF-8 profile at the instance level.

To set the UTF-8 profile at the session level, the following command is used. Once the session has ended, any subsequent session will return to the default for the instance.

```
set names 'utf8' collate value;
```

where *value* is a valid Unicode character set. Please see the following MySQL reference of valid values:

http://dev.mysql.com/doc/refman/5.1/en/charset-unicode-sets.html

15.2 UTF-8 Object Names

When creating string columns with UTF, the length is defined in bytes, not characters. To store ten multi-byte characters with UTF may require twenty to forty bytes, depending on the actual values being stored.

15.3 Known Issues and Limitations

- UTF-8 must be declared at the table level if the instance has been set up with a UTF-8 profile. Tables created with a non-matching character set will yield indeterminate results.
- Viewing SQL output should be done using client software that supports UTF-8 character sets.
- UTF-8 characters are not supported in object names.