

# Qubik Lava SQL Technical Reference

# Table of Contents

## Lava SQL Reference

Introduction

Distributed Lava Database Operation

Supported Data Types

Data types and sizes

Variable length types

Nulls in the Lava Database

SQL Operators, Functions and Conditions

Functions

Aggregates

Reserved expressions

Comparisons

SQL in the Lava Database

SQL Command Categories

Database Manipulation

Schema Manipulation

User Manipulation

Table Manipulation

Data Extraction and Manipulation

Transaction Statements

Database Information

Date and Time Manipulation

Miscellaneous Statements and Clauses

Future Enhancement

SQL Command Reference

Alter schema

Alter table

Alter session

Alter user

Backup

Column List Clause - Insert

Column List Clause - Select

Column List Clause - Specification

Column List Clause - Drop

Column List Clause - Update

Commit

Connect

Create schema

Create relation

Create sequence

Create synonym / Create alias

[Create table](#)  
[Create user](#)  
[Create view](#)  
[Delete](#)  
[Disable user](#)  
[Enable user](#)  
[Disconnect](#)  
[Dismount](#)  
[Distribute](#)  
[Drop schema](#)  
[Drop sequence](#)  
[Drop relation](#)  
[Drop synonym / Drop alias](#)  
[Drop table](#)  
[Drop user](#)  
[Drop view](#)  
[Grant Role](#)  
[Grant privilege](#)  
[Group by Clause](#)  
[Index](#)  
[Insert](#)  
[Order by Clause](#)  
[Rename schema](#)  
[Rename sequence](#)  
[Rename synonym / Rename alias](#)  
[Rename table](#)  
[Rename user](#)  
[Rename view](#)  
[Restore](#)  
[Revoke privilege](#)  
[Rollback](#)  
[Savepoint](#)  
[Select, \*Select Statement\*](#)  
[Set](#)  
[Subqueries](#)  
[\*Lava\* pseudo-table](#)  
[Table List Clause](#)  
[Truncate](#)  
[Undelete](#)  
[Update](#)  
[Value List Clause](#)  
[Where Clause](#)  
[SQL Syntax Specification](#)

# Lava SQL Reference

## Introduction

Lava SQL is derived from a number of current SQL dialects, as well as the ISO SQL-92 and SQL-99 specifications. In general, the SQL syntax implemented in the Lava SQL engine is very similar to (and in many respects the same as) most of the popular SQL implementations currently in use. In several cases, variants of commands are provided for to allow commonly used syntaxes with slightly different forms to function correctly.

There are, as with any SQL engine, certain limitations and unique characteristics which distinguish Lava SQL from other implementations. See the paragraph [Relations, foreign keys and inter-table joins](#) and specifically the sub-paragraph [SQL Join Syntax](#) for information on the largest deviations from standard or ISO-92 SQL.

In addition to the above, Lava SQL is distinguished from other implementations of SQL in that the Lava database is distributed, implying that the client runs a complete and independent Lava database. This allows for commands to be executed either on the client or on the server, as required by circumstances and application architecture. The following paragraph describes these options.

## Distributed Lava Database Operation

Unlike all other SQL databases, the Lava database is fully distributed. This means that every Lava client is, in fact, a fully operational Lava database, capable of performing any operation that the server can perform - including executing SQL commands.

When executing a SQL command on the client, the distributed kernel will examine the command and decide whether execution on the client is possible. If so, the command will be executed on the client as a matter of preference, as this both reduces the processing load of the server and (generally) allows faster execution and response - especially if the server is connected through a relatively slow Internet line.

It is, however, possible to force execution of a command on either the server or the client, where the programmer has special knowledge. For example, if the command to be executed is very complex and would execute far faster on the server (assuming that the server is a much more powerful machine than the average client workstation) and of course assuming that the greater majority of the client commands execute on the client, thereby freeing up the server to execute exactly these most demanding SQL statements. In order to do this, a command (such as a SELECT statement) can be forced to the server as follows :

```
SELECT [SERVER] * FROM SYSTEM.SYS_OBJECTS
```

The above command will unconditionally be executed on the server, as specified by the [SERVER] directive after the SELECT command.

Similarly, it is possible to force a command to execute on the client.

```
SELECT [CLIENT] * FROM SYSTEM.SYS_OBJECTS
```

The above command will unconditionally execute on the client.

Note that forcing a command to execute on a particular database may cause complications. A conventional SELECT statement may be specified as in the following example :

```
SELECT * FROM TESTSCHEMA.CUSTOMERS
```

In this example, due to the fact that there is no database directive, the client database will attempt to determine

the best location to execute the command. Firstly, the client will check whether the nominated table (Customers) has been distributed to the client. Assuming that this table has not been distributed, it will automatically pass the command to the server for execution. The command will execute correctly, and return the content of the Customers table.

If, however, the command is stated as follows :

```
SELECT [CLIENT] * FROM TESTSCHEMA.CUSTOMERS
```

the [CLIENT] directive will force execution on the client database. Given that the Customers table is not distributed to the client, execution of the command will fail.

The responsibility for ensuring the correct environment for the command falls on the programmer if a database directive is specified. If in doubt, allow the database to evaluate the best location for execution.

## Supported Data Types

Each column defined in a Lava Table has a specified data type, which is numerically coded in the [Sys Table Columns](#) system table which stores all column data for all tables in the database. The supported data types are listed below, with the numeric data types used in the column definition.

### Data types and sizes

Code		Internal Datatype	SQL syntax	Description
1	1H	byte	BYTE	
33	21H	Row status	ROWSTATUS	
8	8H	boolean	BOOLEAN	
2	2H	short integer (2-byte)	SHORTINTEGER	
3	3H	integer (4-byte)	INTEGER	
35	23H	Row ID	ROWID	
14	0EH	Address (32-bit Pointer)	ADDRESS	
4	4H	quad integer (8-byte)	QUADINTEGER	
51	33H	Julian Date	DATE	Only Julian dates (dates represented as an integer number of days from 1 January 300) are natively supported in the Lava database. Although there are built-in conversion facilities to and from text date representations, these are supported only for data entry purposes.

## Nulls in the Lava Database

67	43H	Time	TIME	Time is represented as a fraction of a day, in order to fully support arithmetic on date-time (VDT) variables and columns. In other words, 1 hour into the day (1AM) is represented as 1/24, being 0.04167.
55	37H	(DateTime (VDT))	VDT	A DateTime or VDT (Version Date-Time) is represented in an 8-byte Float variable (or column). The integer portion is a Julian date representation, while the fraction is a time representation as a fraction of a day (24 hours). This allows arithmetic on date-time columns or variables with guaranteed correct results.
6	6H	short float (4-byte)	SHORTFLOAT	
7	7H	float (8-byte)	FLOAT	
15	0FH	IP (Internet Protocol Address)	IP	IP addresses are internally stored as a 4-byte integer.
9	9H	character (1 byte)	CHAR	
10	0AH	string (fixed length)	STRING	
11	0BH	varstring (fixed length base with variable length extension to $2 \times 10^9$ )	VARSTRING	
28	1BH	packed varstring	PACKEDVARSTRING	
42	2AH	unicode string (fixed length)	UNICODESTRING	
43	2BH	varunicode (fixed length base, variable length extension to $2 \times 10^9$ )	VARUNICODE	
13	0DH	varbyte (fixed length base, variable length extension to $2 \times 10^9$ )	VARBYTE	
29	1DH	packed varbyte	PACKEDVARBYTE	

100	64H	Structure (Single-depth record comprised of basic data types)	<i>Available in API - Future provision for SQL</i>	
-	-	Array (1-dimensional array of any fixed length type, including structures)	ARRAY nn OF simpletype	Only arrays of simple datatypes are supported - specifically, any VAR-type (such as VARSTRING) or structure is excluded.

**Variable length types**

Variable length columns are implemented in the Lava database as a fixed-length portion (length defined by the user at time of table creation), stored within the fixed-length row definition, and a variable length portion, unlimited in length, stored separately in the database and referenced to the row by the kernel using internal links.

Variable data in a given row of data can neither be stored nor retrieved using row-level operations (with the exception of the fixed-length portion). To store variable data, a [PutColumn](#) must be executed, and to retrieve variable data, a [GetColumn](#) must be executed. Using these commands, variable data of any length may be accessed.

The currently supported variable length types are :

- varstring**      A variable length ASCII string. Although data containing nulls may be stored and retrieved, most string procedures and functions will regard the first null character as the end of the string.
- varunicode**    A variable length Unicode string. Most unicode string procedures will regard the first double-null entry pair as the terminator to the Unicode string.
- varbyte**        Variable data of any type, represented as an array of byte. Any values are permissible in the stipulated data length, including nulls.

**Unsupported types**

The following types are listed for completeness, but are not supported in the current revision of the database.

- Decimal**        Binary coded decimal type
- Currency**      Decimal type flagged as a particular currency in the default format for the column

**Nulls in the Lava Database**

The current release of the Lava Database does not support null column values. This feature is provided for in the Lava design, and will be implemented by the next major release.

**SQL Operators, Functions and Conditions**

**Functions**

- ABS
- ARCCOS
- ARCSIN
- ARCTAN

COS  
DEG  
EXP  
FORMAT  
INT  
LN  
LOG  
LOWER  
RAD  
ROUND  
SIN  
SLICE  
STRINGPOS  
SQRT  
SOUNDEX  
TAN  
TRUNC  
UPPER

### Aggregates

AVG  
COUNT  
MIN  
MAX  
SUM

### Reserved expressions

PI  
ROWID  
DATE  
TIME  
VDT

### Comparisons

<  
>  
<=  
>=  
=  
#, <>  
LIKE

## SQL in the Lava Database

### General Approach to SQL Syntax

Taking cognisance of the effort expended on the ISO SQL standard, it is the opinion of the Lava system architects that for the majority of users the ISO syntax is somewhat clumsy and unwieldy, and the specification is not in all cases easy to interpret. In particular, the method proposed for specifying joins is, in our opinion, not as elegant or easy to understand, code and interpret as a simple filter clause in the *where* clause of an SQL statement. After careful consideration, and with due respect to the detail in the ISO specification, we have therefore decided to depart from this standard in a number of respects.

### Departures from the ISO syntax

As a general rule, the syntax used in Lava SQL is very similar or in many cases identical to that stipulated by the ISO standards. In all cases, the ISO standard was taken into account before deciding on a particular syntax (as were a number of other de-facto industry standards used by major SQL databases in the past and present).

There are a few minor departures from strict ISO syntax in certain cases, but the only major departure is the specification of joins in the select statement. In the following example, the joins are specified in the *where* clause in the style adopted by Lava SQL. This should be easy to interpret, and equally easy to duplicate.

```
select
  schema_name
from
  sys_objects, sys_schemas
where
  sys_schemas.id = sys_objects.schema_id and
  sys_objects.id < 70
```

SQL-92 style joins are - to a reasonable degree - supported, but are not considered part of the core of the Lava SQL syntax.

### Comments in SQL statements

Comments may be specified at any point in a SQL statement, and comments may be nested to any level. A comment is delimited by */\** and *\*/*

## SQL Command Categories

### Database Manipulation

<a href="#">Backup</a>	Backup system - create a schema backup
<a href="#">Restore</a>	Backup system - restore a schema backup
<a href="#">Connect</a>	Connect to a nominated server
<a href="#">Disconnect</a>	Disconnect from a server

### Schema Manipulation

<a href="#">Create</a>	Create a new schema
<a href="#">Drop</a>	Drop an existing schema
<a href="#">Rename</a>	Rename an existing schema
<a href="#">Alter</a>	Alter attributes of an existing schema

## SQL Command Categories

<a href="#">Distribute</a>	Distribute a schema from the server to the local client database
<a href="#">Restore</a>	Backup system - restore a schema backup
<a href="#">Backup</a>	Backup system - create a schema backup

### User Manipulation

<a href="#">Create</a>	Create a new user account
<a href="#">Drop</a>	Drop an existing user account
<a href="#">Rename</a>	Rename the username for an existing user account
<a href="#">Disable user</a>	Disable a user account
<a href="#">Enable user</a>	Enable a user account which is currently disabled
<a href="#">Grant</a>	Grant privileges to a user account
<a href="#">Revoke</a>	Revoke privileges from a user account
<a href="#">Alter user</a>	Alter attributes of a user account
<a href="#">Alter session</a>	Alter session default attributes
<a href="#">Connect</a>	Connect to a nominated server
<a href="#">Disconnect</a>	Disconnect from a server

### Table Manipulation

<a href="#">Create Table</a>	Create a new data table
<a href="#">Drop</a>	Drop an existing data table
<a href="#">Alter</a>	Alter attributes for a data table
<a href="#">Distribute</a>	Distribute a table to the local client database
<a href="#">Create alias</a>	Create an alias (synonym) for a data table
<a href="#">Drop alias</a>	Drop an alias of a data table
<a href="#">Truncate</a>	Truncate a data table
<a href="#">Create Relation</a>	Create a system relation between two data tables
<a href="#">Drop Relation</a>	Drop a system relation between to data tables
<a href="#">Index</a>	Create an index on a table column
Drop Index	Drop index(es) on a table column

**Data Extraction and Manipulation**

<a href="#">Select</a>	Query the database and produce a result set
<a href="#">Update</a>	Update data rows in a data table
<a href="#">Delete</a>	Delete data rows in a data table
<a href="#">Insert</a>	Insert data rows into a data table
<a href="#">Subqueries</a>	Correlated subqueries to a select statement

**Transaction Statements**

<a href="#">Commit</a>	Commit a pending transaction frame
<a href="#">Rollback</a>	Rollback a pending transaction frame
<a href="#">Savepoint</a>	Create a nested transaction frame

**Database Information**

Show alias	
Show index	
Show schemas	
Show sessions	
Show tables	
Describe table	

**Date and Time Manipulation**

CalendarDate	Convert a Julian date into calendar date format
CalendarDateTime	Convert a Julian date-time (VDT) into calendar / clock date-time format
ClockTime	Convert a numeric Lava time column into clock time
JulianDate	Convert calendar (text) date into a Julian date
JulianTime	Convert clock (text) time into a Lava numeric time
SysDate	Obtain the system (server) date
SysDateTime	Obtain the system (server) date time - UDT.

## SQL Command Categories

SysTime	Obtain the system (server) time - UDT.
---------	--

### Miscellaneous Statements and Clauses

<a href="#">System pseudo-table</a>	Table name used in non-table queries
<a href="#">Column list - Select Clause</a>	List of columns for selection
<a href="#">Group by Clause</a>	Group specification in selects
<a href="#">Order by Clause</a>	Sorting in selects
<a href="#">Table List Clause</a>	List of tables for selection
<a href="#">Where Clause</a>	Specification of joins or filters

### Future Enhancement

The following commands are not yet implemented in the current release, but are listed as the keywords are reserved and the command syntax is finalized

Grant role	Establishment and management of roles is not yet supported
Create view	Views are not yet supported
Drop view	Views are not yet supported
Rename view	Views are not yet supported
<a href="#">Undelete</a>	Undelete (recovery of deleted rows) is not yet supported
Create sequence	Sequences are not yet implemented
Drop sequence	Sequences are not yet implemented
Set	Provision for future functionality

## SQL Command Reference

This section documents all the SQL commands implemented in the Lava SQL engine. The commands have been ordered alphabetically, including clauses such as (for example) the [Group By](#) clause. Every attempt has been made to place reference hyperlinks wherever appropriate, but if the reader does not wish to read the entire SQL command reference, the best access point is through the [SQL command category](#) listing which provides a means of locating associated commands in given command domains. The hyperlinks from the individual commands can then be used to access related references.

## Alter schema

The **Alter Schema** command is used to define the allowable access to the schema. This permits certain schemas with reference information only to be rendered read-only, disallowing any change to the schema content whatsoever.

```
ALTER SCHEMA schemaname SET ACCESS accessmode
```

### Prerequisites

The session must have ALTER SCHEMA privilege on the nominated schema.

### Variants

```
ALTER [CLIENT] SCHEMA schemaname SET ACCESS accessmode
```

The [client] qualifier instructs the SQL engine to perform the alteration on the client database only. This will not affect the status of the schema on the server at all.

```
ALTER [SERVER] SCHEMA schemaname SET ACCESS accessmode
```

The [server] qualifier causes the schema to be altered on the server to which the session is connected. This is the default operation.

### Qualifiers and Parameters

<i>schemaname</i>	The name of the schema to be altered.
<i>accessmode</i>	The data access mode of the schema. Available modes are :
READONLY	The schema allows read access only - updates are disallowed.
READWRITE	The schema may be updated by sessions with appropriate privileges.

### Results

The access mode of the nominated schema is modified as stipulated.

### Remarks

If the schema is set to READONLY mode, no modifications may be made to the objects contained by the schema. This includes dropping or creating tables, truncating tables, or any table row modification such as deletion or update.

READONLY schemas will not permit data restore using the [Restore](#) command, as this will require truncation or drop of tables within the schema, which is not permitted in this case. If a schema which is flagged as READONLY is to be restored, the schema must first be set to READWRITE, after which the restore may be performed and the schema can be set back to READONLY.

[Backup](#) operations are permitted on READONLY schemas.

If the schema is set to READWRITE, normal modifications to table content as well as creation and dropping of tables is permitted.

### Examples

```
ALTER SCHEMA fred SET ACCESS READONLY;
ALTER SCHEMA joe SET ACCESS READWRITE;
```

### See also

[Schema Manipulation](#)

## Alter table

The **Alter Table** command is used to alter attributes of the nominated table, including column definitions, the table name or the table schema.

```
ALTER TABLE tablename ADD (column list - spec)
ALTER TABLE tablename DROP (column list - drop)
ALTER TABLE tablename MODIFY (column list - spec)
ALTER TABLE tablename RENAME TO newtablename
ALTER TABLE tablename SCHEMA newschema
ALTER TABLE tablename DISTRIBUTION FULL | ONDEMAND
ALTER TABLE tablename REPLICATION ON | OFF
```

### Prerequisites

The session must have ALTER TABLE privilege on the nominated table.

For the ALTER TABLE ... SCHEMA command, the session must in addition have ALTER TABLE privileges in the specified new schema for the table.

### Variants

There is no permitted [client] variant of the alter table command, as this would render distributed data tables inconsistent with the server. As a result, if the table is distributed this command always acts on both the server and client copies of the table to ensure consistency.

If the table is non-distributed and occurs only on the server, the command acts on the server table.

If the table is defined only on the client, the command acts on the client table.

### Qualifiers and Parameters

<i>tablename</i>	In all versions of this command, the <i>tablename</i> parameter specifies the table to be altered. If the table is not in the session's current schema, a schema prefix is required to fully identify the table.
<i>Column list - spec</i>	The column list specification allows the definition of a list of column names and types to be added or modified for the nominated table. See <a href="#">Column List - Specification</a> for details.
<i>Column list - drop</i>	The column drop list allows nomination of a list of columns to be removed from the table. See <a href="#">Column List - Drop</a> for details.
<i>newtablename</i>	The new table name for the nominated table.
<i>newschema</i>	The schema to which the table is to be moved.

### Results

ALTER TABLE ... ADD	:	The specified columns are added to the table on both the server and the client databases if the table is distributed, or to the server or client table only if the table occurs on only that database.
ALTER TABLE ... DROP	:	The listed columns are dropped from the table on both the server and client databases if the table is distributed, or from the server or client table only if the table occurs on only that database.
ALTER TABLE .. MODIFY	:	The specified columns are modified to new column types on both the server and client databases if the table is distributed, or on the server or client table only if the table occurs on only that database.
ALTER TABLE ... RENAME	:	The table is renamed on both the server and client databases if the table is distributed, or on the server or client database only if the table occurs on only that database.
ALTER TABLE ... SCHEMA	:	The table is moved to the specified new schema on both the server

## SQL Command Reference

and client databases if the table is distributed, or on the server or client database only if the table occurs on only that database.

### Remarks

If the table column layout is modified through the ADD, MODIFY or DROP forms of the command, table backups performed before the alteration will still restore to the modified table, except for added or dropped columns. In other words, if columns are added to a table and a restore is performed, the new columns will be empty after the restore throughout the table. If columns are dropped, those columns will (obviously) not be restored during the restore operation, but all remaining tables that match the column list prior to the restore will be restored correctly. If column types are modified using the MODIFY form of the command, the restore mechanism will attempt to convert the data in the backup to the new column type. If a sensible conversion can be performed, the restored data will comply with the new column type. If no sensible conversion can be performed (such as from non-numeric string data to a numeric column type) the column is left blank by the restore.

If the table name or schema is altered using the RENAME or SCHEMA forms of the command, the restore will no longer successfully identify the table and will create a new copy of the backup table by the original name in the original schema (or in the session's current schema if the backup set does not specify a source schema).

If the table is moved to a different schema using the ALTER TABLE .. SCHEMA command, the table will be moved only if the user has sufficient privilege to alter the table in both the table's current schema and the proposed new schema for the table. This is to avoid the possibility of the user moving the table to a schema where it is no longer accessible.

### Examples

Add two new columns to existing table *fred* :

```
ALTER TABLE fred ADD (column_new_1 INTEGER, column_new_2 STRING(50));
```

Drop two existing columns from table *fred* :

```
ALTER TABLE fred DROP (column_old_1, column_old_2);
```

Change the type of an existing column in table *fred* :

```
ALTER TABLE fred MODIFY (column_old_3 INTEGER);
```

Rename table *fred* to table *joe* :

```
ALTER TABLE fred RENAME TO joe;
```

Move table *fred* from its current schema to schema *different\_schema* :

```
ALTER TABLE fred SCHEMA different_schema;
```

### See also

[Rename Table](#), [Table Manipulation](#)

## Alter session

The **Alter Session** command may be used to modify the current schema or default backup folder for an existing session.

```
ALTER SESSION SCHEMA newschema
ALTER SESSION BACKUP FOLDER newbackupfolder
```

### Prerequisites

To modify the current schema the user must have at least READ access to the nominated schema. There are no prerequisites for modifying the backup folder.

### Variants

None - the command acts on the current session.

### Qualifiers and Parameters

<i>newschema</i>	The name of an existing schema which becomes the current schema for the session.
<i>newbackupfolder</i>	The full path of an existing folder on the workstation

### Results

ALTER SCHEMA	:	The current schema for the session is changed to the nominated schema.
ALTER BACKUP FOLDER	:	The default backup folder for the session is changed to the nominated folder path.

### Remarks

ALTER SCHEMA :  
The nominated schema must exist, and the user must be permitted to read the schema.

Even if the user may see the schema and may change the current schema as nominated, this does not guarantee that any tables will be visible to the user in the new schema - if the user is denied access permission to all tables in the new schema, the schema will appear empty.

The current schema for this session is changed only - the default schema for the user account is not modified, and the next connect will revert to the default schema for the user. See [Alter User](#) for information on how to change the default schema for a user account.

ALTER BACKUP FOLDER :  
The nominated folder must be fully specified (full file path) and the folder must exist.

### Examples

Change the current schema to the EVENT schema :

```
ALTER SESSION SCHEMA EVENT;
```

Change the default backup folder to a new folder :

```
ALTER SESSION BACKUP FOLDER c:\lava\backup
```

### See also

[Alter User](#), [Backup](#), [User Manipulation](#)

**Alter user**

The **Alter User** command may be used to modify the default schema or the password for an existing database user.

```
ALTER USER username SCHEMA newschema
ALTER USER username PASSWORD newpassword
ALTER USER username SCHEMA newschema PASSWORD newpassword
```

**Prerequisites**

The specified user must be the user account for the current session, or the session must have ALTER USER privilege.

**Variants**

None. The user account on the server is always changed - modifying the user on the client would not be sensible, as this modification would be lost on disconnect.

**Qualifiers and Parameters**

<i>username</i>	The username for the account to be modified
<i>newschema</i>	The new default schema for the user account
<i>newpassword</i>	The new password for the user account

**Results**

The default schema and / or the password for the nominated user account is modified.

**Remarks**

The new default schema and / or new password are immediately implemented, but since there may be current sessions using the nominated user account, these sessions will continue to operate using the schema or password which were valid at the time the sessions were established. Any new sessions established on the nominated user account will see the new attributes for the user account.

The password as specified is unencrypted. On execution of the command, the password is encrypted in the system user table.

**Future enhancement**

*Passwords will have an expiry attribute to allow password expiry at synchronous intervals*

**Examples**

```
ALTER USER fred SCHEMA UserSchema_1 PASSWORD jennifer;
```

**See also**

[Alter Session](#), [User Manipulation](#)

## Backup

The **Backup** command performs a backup on a nominated schema.

```
BACKUP SCHEMA schemaname;
BACKUP SCHEMA schemaname KEY encryptionkey;
```

### Prerequisites

The session must have BACKUP privilege on the schema. In addition, any tables within the schema for which the session does not have READ access will not be backed up.

The nominated schema will automatically be distributed if the backup command is executed on a client, regardless of whether the schema was distributed before issuing the command.

A backup folder for the session must have been asserted - see [Alter Session](#). If the command is being issued from the Lava Query interface, it is possible to assert the backup folder in the command parameters.

### Variants

None. The backup is always performed from the client, although the data in the backup is always server data.

### Qualifiers and Parameters

<i>schemaname</i>	The schema on which the backup is to be performed.
<i>encryptionkey</i>	An optional key to be applied to the backup. If an encryption key is specified, the backup can only be restored if the exact key is specified.

### Results

A backup file is created for the nominated schema.

### Remarks

The mount mode must be [Exclusive](#), or the nominated schema must be distributed to the client, in order for the backup to succeed.

The backup process creates a single file backup in the current default backup folder - see [Alter Session](#).

The default file type (file extension) for Lava backup files is *.lbs* (Lava Backup Set).

### Future enhancement

*Currently it is not possible to freeze updates to the database during execution of the backup. A planned enhancement will allow suspension of any **commits** during the backup process to ensure that the backup data is not inconsistent in any way as a result of partial updates.*

### Examples

Assert a backup folder and backup the schema *fred* :

```
ALTER SESSION BACKUP FOLDER "q:\lava\backup";
BACKUP SCHEMA fred KEY "jzX12 owCP";
```

### See also

[Alter Session](#), [Restore](#), [Schema Manipulation](#)

## Column List Clause - Insert

This variant of the column list is used in the insert command, and simply specifies a list of columns from the nominated table which are to be initialized to values stipulated in the [value list](#) (**insert ... values** variant) or the [select column list](#) (**insert ... as select** variant).

The basic form of the column list is as follows :

```
(column_1, column_2 ... column_n)
```

The syntax of the insert column list is :

```
ColumnListInsert ::= (ColumnList)
ColumnList       ::= ColumnSpec [, ColumnList]
```

where *ColumnSpec* is the name of a column in the table nominated for the insert.

### Remarks

Any number of columns from the table may be specified, but each column may only be specified once.

Each column specified in the column list must be matched by a corresponding value in the [value list](#), or a corresponding select column in the [select column list](#), depending on the form of the [insert](#) command used.

The type of the column to be inserted and the corresponding value or select column do not have to be identical. The SQL engine will do a best-case conversion of the value to be inserted into the data type of the column specified for insertion.

Any columns not specified in the insert column list will be left blank (empty string for string types, 0 for numeric types).

The [Row status](#) and [ID](#) columns must not be specified in the column list for standard (non-[Raw](#)) tables. These columns are maintained by the Lava Database kernel, and cannot be set by the user. The [Row\\_status](#) column will be set to CURRENT, and the [ID](#) column will be set to the correct row ID value for the row used when the resulting data row is inserted into the table. The row to be used will depend on whether the table allows re-use of deleted rows, and on whether unused (deleted) rows are found in the table.

### See also

[Insert](#), [Data Extraction and Manipulation](#)

## Column List Clause - Select

This variant of the column list is used in the Select statement.

The basic form of the column list is as follows :

```
column_1, column_2 ... column_n
```

Each of the columns specified may include calculations or functions, as follows :

```
(column_1 + 3) * 4, log(column_2) / 20.5
```

In addition, any of the columns may be a subquery :

```
column_1, (column_2 + 3) * 4, cos(PI),  
(select id from system.sys_objects where object_name = 'mytable'),  
column_3
```

In the above example, the third column to be selected will be the [object ID](#) of the user's table, named *mytable*.

Finally, columns may specify aggregates :

```
sum(column_1), avg(column_2), max(column_3)
```

Where aggregates are used, the results may be grouped by further columns - see the [Group by](#) clause for further details.

The full syntax of the select column list may be found in the section [SQL Syntax Specification](#).

### Remarks

Any number of columns from the nominated tables in the [table list](#) of the select command may be specified, and any column may be specified any number of times.

[Subqueries](#) may be nested to any depth. Note, however, that as subqueries are evaluated separately from the main query, in certain cases it is more efficient to specify a 'flattened' query. The Lava SQL engine will attempt to evaluate queries as economically as possible, but this will not always result in the best possible evaluation of the required results.

When a column is coded as a subquery, that subquery **must** return exactly one row with exactly one column. As the subquery is intended to replace a single column (or calculated value), the result of the subquery must be a single value (any data type is permitted). If the specified subquery results in more than one column or more than one row in its result set, an error will be returned and the select will fail.

### See also

[Select statement](#), [Subquery](#), [Data Extraction and Manipulation](#)

## Column List Clause - Specification

The **Column List Specification** is used in both the **Alter Table** and the **Create Table** commands, and specifies the names and data types of a list of 1 or more columns for addition to (**Alter Table**) or specification of a complete table (**Create Table**).

The general form of the list is :

```
(column_1 datatype_1, column_2 datatype_2, ..., column_n datatype_n)
```

Each entry in the list comprises a pair of specifications separated by one or more space characters; the first is the name of the column, the second is the data type for the column.

The list may contain any number of columns.

For a comprehensive list of allowable data types, see [Supported Data Types](#).

### Remarks

Each column name specified in the list must be unique.

In addition to being unique within the specified list, column names specified in an **Alter Table** command must comply with the following constraints :

- If the **Alter** command specifies the **modify** option, each column name specified must exactly match an existing column in the table to be altered
- If the **Alter** command specifies the **add** option, each column name specified must also be unique considering the existing columns of the table to be altered.

The total number of columns specified for the table in both the **Create** command and the **Alter ... add** command may not exceed 500. In addition, the sum of the column sizes for all columns specified may not exceed 1500 bytes. (This does not include the variable portion of variable length column types - see [Variable length types](#) for more information.)

### See also

[Create Table](#), [Alter Table](#), [Supported Data Types](#), [Table Manipulation](#)

## Column List Clause - Drop

This variant of the column list, identical in syntax to the form used in insert commands (see [Column List Clause - Insert](#)), and simply specifies a list of columns from the nominated table which are to be dropped (deleted).

The basic form of the column list is as follows :

```
(column_1, column_2 ... column_n)
```

The syntax of the insert column list is :

```
ColumnListDrop      ::= (ColumnList)
ColumnList           ::= ColumnSpec [, ColumnList]
```

where *ColumnSpec* is the name of a column which is to be dropped from the nominated table.

### Remarks

Any number of columns from the table may be specified, but each column may only be specified once.

The [Row status](#) and [ID](#) columns may not be specified for standard (non-[Raw](#)) tables. These columns are mandatory, and cannot be dropped by the user.

### See also

[Alter Table](#), [Table Manipulation](#)

## Column List Clause - Update

This form of the column list is used in the **Update** command to specify columns to update and corresponding values.

The general form of the list is :

```
column_1 = value_1, column_2 = value_2, ..., column_n = value_n
```

Each entry in the list comprises a pair of specifications separated by an equals sign; the first is the name of the column, the second is the value to be assigned to the column.

The list may contain any number of columns.

The values specified may contain calculations :

```
column_1 = (column_1 + 3) * 4, column_2 = log(35) / 20.5
```

Any value specification may be derived in terms of a [subquery](#) :

```
column_1 = (select max(amount) from inv_details where ID = Inv_id)
```

### Remarks

Each column name specified to be updated in the list must be unique.

If a subquery is used to specify the value for the column, the subquery must return exactly one row and one column. If more rows or more columns are returned, the query will return an error and the update will fail.

### See also

[Update](#), [Subquery](#), [Data Extraction and Manipulation](#)

## Commit

The **Commit** command commits open transaction frames for the current session.

```
COMMIT
COMMIT subframe
```

### Prerequisites

None. The prerequisites apply to the transactions that comprise the transaction frame on which the commit is to be performed; the commit itself has no prerequisites.

### Variants

XXXXXXXXXXXXXXXXXXXX

[server] / [client]

None. The Commit command always acts on the current session, which determines the appropriate server on which the commit is executed.

### Qualifiers and Parameters

*subframe*                      The name of an existing transaction subframe - see [Savepoint](#)

### Results

The transaction frame is partially committed if a *subframe* is specified, and fully committed if no *subframe* is specified.

### Remarks

If a [Savepoint](#) is executed before any modifications are performed in the session, specifying that savepoint as the *subframe* to be committed is equivalent to specifying commit without a *subframe*.

If a *subframe* is specified which is partway through the current transaction frame, the commit does a partial commit from that savepoint onward - effectively what this does is to remove the subframe and combine it with the root transaction frame for the session.

### Examples

```
DELETE fred WHERE ID = 5;
SAVEPOINT newsubframe;
DELETE fred WHERE ID = 6;
COMMIT newsubframe;
UPDATE fred SET name = 'fred' WHERE ID = 7;
COMMIT;
```

### See also

[Rollback](#), [Transaction Statements](#)

## Connect

The **Connect** command attempts to connect to a Lava Database. Both exclusive and server connections are supported.

```
CONNECT SERVER servername USER username PASSWORD password
```

Connect *ServerClause* user [*password password*]

*ServerClause* : Exclusive | server *servername* | serverip *ipnumber*

### Prerequisites

A valid user account with the stated password on the nominated server.

### Variants

```
CONNECT SERVERIP serverIPaddress USER username PASSWORD password
```

The SERVERIP variant of the connect command may be used to connect to servers visible only via an IP network. This is generally true of servers accessed via the internet.

```
CONNECT EXCLUSIVE USER username PASSWORD password
```

The CONNECT EXCLUSIVE variant of the command can only be used on databases which are mounted [Exclusive](#) - Client mode databases only support SERVER connections.

### Qualifiers and Parameters

<i>servername</i>	The network server name of a server which can be addressed by name on a local Windows network.
<i>username</i>	The name of a valid user account.
<i>password</i>	The password for the user account.
<i>serverIPaddress</i>	The IP address (in conventional <i>a.b.c.d</i> numeric format) for the server

### Results

A connection is established with the specified Lava Server.

### Remarks

Establishing a connection to the Lava Server is required to create a current session to the server. A valid session is required for all Lava commands.

Prior to connecting to a server when mounted in [Client](#) mode, the client database must be mounted - see Mount for more information.

### Examples

```
CONNECT SERVER CentralServer USER fred PASSWORD "long password"
```

### See also

[Database Manipulation](#), [User Manipulation](#)

## Create schema

The **Create Schema** command is used to create a new schema in the database. The schema is normally created on the server, but it is possible to specify creation on the client.

A schema is an encapsulating or grouping entity that allows tables to be accumulated into a coherent set, and user permissions and access to be limited to the schema. Schemas allow for efficient backup strategies, as sets of tables belonging together can be backed up as a single group.

A schema may be used as a means of nominating sets of tables (and in the future, other objects) for purposes such as data distribution, as well as being a useful mechanism in large databases with many tables to divide the database into more manageable sets of objects.

```
CREATE SCHEMA schemaname
```

### Prerequisites

The session must have database wide CREATE privilege

### Variants

```
CREATE [CLIENT] SCHEMA schemaname
```

The [client] qualifier instructs the SQL engine to create the schema on the client database only. This implies that the schema will be transient, existing only as long as this particular client session is mounted. On dismantling of the client database, the schema is dropped, together with any tables that were created within the schema.

```
CREATE [SERVER] SCHEMA schemaname
```

The [server] qualifier causes the schema to be created on the Lava Server to which this client is connected. This is the default operation.

### Qualifiers and Parameters

*schemaname*      The parameter *schemaname* specifies the name of the newly created schema. This name must be unique across all schemas currently existing on the database within which the schema is being created.

### Results

On successful completion, a new schema is created on the selected database.

The schema is initially empty, with the exception of the default system variable length column tables, which are for system use only.

### Remarks

In order to access the new schema, the schema and any objects within the schema can only be accessed by prefixing any object to be accessed by the schema name

### Future enhancement

*In the current release of the Lava Database, schemas are non-hierarchical, although provision has been made in the system design for schema hierarchies. Functionality will be added to allow schemas to be created as subordinates of master schemas, to arbitrary depth. This will allow greater flexibility in controlling sets of tables and other objects both in terms of access privileges and in terms of data management for backup and distribution purposes.*

### Examples

In the following example, a new schema (*fred*) is created. An existing table in the current schema, *existingtable*, is then moved to the new schema. A select is first performed using a specified schema prefix to access the schema which is non-default to the current user. The current schema is then set for the current session (this will not alter the default schema for the user on next connection) and the table is once again queried, this time without the need for the schema prefix.

```
CREATE SCHEMA fred;  
ALTER TABLE existingtable SCHEMA fred;  
SELECT * FROM fred.existingtable;  
ALTER USER myuser SET SCHEMA fred;  
SELECT * FROM existingtable;
```

**See also**

Alter Table, Alter User, [Schema Manipulation](#)

## Create relation

The **Create Relation** command creates a relation between two tables which will be used for relational integrity and (potentially) slice backup / restore purposes.

```
CREATE RELATION FROM PARENT sourcetable TO targettable
  SOURCECOLUMN (sourcecolumn [, sourcecolumn])
  TARGETCOLUMN (targetcolumn [, targetcolumn])
  [SOURCECONDITIONAL ConditionColumn = ConditionValue]
  [TARGETCONDITIONAL ConditionColumn = ConditionValue]
  [DELETE CASCADE | RESTRICT]
```

### Prerequisites

The session must have CREATE RELATION privilege on the schema to which the tables belong, as well as ALTER TABLE privilege on both of the tables.

### Variants

None. The command is always executed on the server.

### Qualifiers and Parameters

<i>sourcetable</i>	The parent (source) table in the relation (the <i>one</i> table in a <i>one : many</i> relationship)
<i>targettable</i>	The child (target) table in the relation (the <i>many</i> table in a <i>one : many</i> relationship)
<i>sourcecolumn</i>	The column in the parent (source) table which stores the value of the master table to which a row entry in the child table is related. If more than one source column is required, multiple columns may be specified (comma-separated)
<i>targetcolumn</i>	The column in the child (target) table which stores the value of the master table to which a row entry in the child table is related. If more than one target column is required, multiple columns may be specified (comma-separated)
<i>ConditionColumn</i>	A source or target column on which a conditional relation is based. The condition column must be an integer (32-bit) column.
<i>ConditionValue</i>	The nominated value for a condition column for which the relation is valid. This will be an integer value.

### Results

A relation is created between the nominated parent and child tables on the nominated child table column.

### Remarks

If multiple source columns are specified, the exact same number of target columns must be specified for the target table.

If source or target conditions are specified, these operate as follows :

For the nominated condition column (source or target as applies), the relation is only regarded as valid for the nominated condition value. In other words, if a conditional relation is specified, that relation will only apply where the nominated column has the nominated condition value. For all other values in that column, the relation is ignored (as if it did not exist).

If DELETE CASCADE is specified, deleting a row in the parent (source) table will cause cascade delete of all rows satisfying the relation in the child (target) table.

If DELETE RESTRICT is specified, an attempt to delete a row in the parent (source) table will be denied if any

rows exist in the child (target) table which satisfy the relation.

In the current release, referential update of target values for update of the value in the parent (source) table is not supported. This is due to the fact that native relations in a Lava database are always to or from the Row ID of either table, and therefore referential updates do not apply. This feature will be added in a future release of the database.

For a complete and detailed description of the implementation of relational integrity in a Lava Database, see [Relational Integrity](#).

### Examples

The following is an actual relation in the EVENT schema - for a graphical depiction of the relation see the illustration under [Event Schema](#).

```
CREATE RELATION FROM PARENT Sys_Event_Type TO Sys_Event_Log
SOURCECOLUMN (Event_Type_id)
TARGETCOLUMN (ID)
```

### See also

[Relational Integrity](#), [Table Manipulation](#)

## Create sequence

The Create Sequence command is stipulated as a future provision, and is not available in the current release of the database. The specification below is preliminary, but should be as implemented in release 5.0 of the Lava SQL engine.

The *Create Sequence* command creates a new sequence by the specified name.

```
CREATE SEQUENCE sequencename STARTVAL startvalue ENDVAL endvalue
INCREMENT incrementval termqualifier
```

If the *CYCLE* option is specified, the sequence wraps to the specified *MINVAL* when the *MAXVAL* is reached.

```
CREATE SEQUENCE sequencename STARTVAL startvalue ENDVAL endvalue
INCREMENT incrementval CYCLE
```

### Prerequisites

The session must have *CREATE SEQUENCE* privilege in the current schema.

### Variants

```
CREATE [CLIENT] SEQUENCE sequenceattributes
```

The *[client]* qualifier instructs the SQL engine to create the sequence on the client database only. This implies that the sequence will be transient, existing only as long as this particular client session is mounted. On dismount of the client database, the sequence is dropped.

```
CREATE [SERVER] SEQUENCE sequenceattributes
```

The *[server]* qualifier causes the sequence to be created on the Lava Server to which this client is connected. This is the default operation.

### Qualifiers and Parameters

<i>sequencename</i>	The name of the sequence to be created. This must be unique within the current schema.
<i>startvalue</i>	The starting value of the sequence. This is the first value returned by the sequence. It may be any numeric value, including fractional values.
<i>endvalue</i>	The final value of the sequence. This is the last value returned by the sequence before either terminating or cycling. It may be any numeric value, including fractional values.
<i>incrementval</i>	The increment amount. This may be negative or fractional.
<i>termqualifier</i>	The method used to terminate the sequence. This method is applied when the maxvalue is reached.
<i>TERMINATE</i>	The sequence expires on reaching the maxvalue. An attempt to use the sequence after this fails.
<i>CYCLE</i>	The sequence cycles back to the minvalue and continues incrementing.

### Results

A sequence with the specified attributes is created in the current schema.

### Remarks

The sequence is always created in the current schema.

## SQL Command Reference

*The name of the sequence must be unique within the sequences defined in the schema.*

*Sequences are updated immediately on access, and do not revert on execution of a Rollback command.*

*If the incrementval is to be negative, the startvalue should be higher than the endvalue.*

### **Future enhancement**

The Create Sequence command is specified for future enhancement of the Lava Database, and will only be available in release 5.0 of the kernel and SQL engine.

### **Examples**

```
CREATE SEQUENCE integersequence STARTVAL 1 ENDVAL 20
      INCREMENT 2 CYCLE;
CREATE SEQUENCE floatsequence STARTVAL 5.7 ENDVAL 2.3
      INCREMENT -0.05 TERMINATE;
```

### **See also**

[Miscellaneous Statements and Clauses](#)

## Create synonym / Create alias

The Create Synonym and Create Alias commands are equivalent alternatives of the command to create an alias (synonym) on a table. A synonym or alias is an alternative name for the table which will identify the table equivalently to the proper table name.

```
CREATE SYNONYM tablealias FOR tablename
CREATE ALIAS tablealias FOR tablename
```

### Prerequisites

The session must have CREATE privilege on the schema to which the nominated table belongs.

### Variants

None. The **Create Synonym** command always executes on the server.

### Qualifiers and Parameters

<i>tablealias</i>	A new alias for the nominated table.
<i>tablename</i>	The table for which the alias is to be created. If the table is not in the current schema, the schema name must be prefixed to the table name.

### Results

An alias as specified is created for the nominated table. The alias is persistent, and will exist until explicitly dropped.

### Remarks

After creation of the alias, the table may be referred to by either the original table name or by the alias name.

The specified alias must be unique within the schema. This uniqueness requirement includes any tables defined within the schema.

The alias belongs to the same schema as the table, and if the alias is referred to from another current schema, the schema name for the alias / table must be prefixed to the alias as would be the case for the table.

### Examples

Create an alias for the event log :

```
CREATE ALIAS errorlog FOR event.sys_event_log;
```

Select from the new alias :

```
SELECT * FROM event.errorlog;
```

### See also

[Table Manipulation](#)

## Create table

The **Create Table** command creates a new table in the current or specified schema.

```
CREATE TABLE tablename
(Column List - Spec)
    [tableattributes]
CREATE TABLE tablename
AS SELECT Select Statement
    [tableattributes]
```

A variant of the **Create Table** command allows creation of a table (client or server) from an ODBC source :

```
CREATE TABLE tablename FROM ODBC {odbc select}
```

### Prerequisites

The session must have CREATE TABLE privilege on the schema within which the table is to be created.

### Variants

```
CREATE [CLIENT] TABLE tablecreationstatement;
```

The nominated table is created on the client database. **Note** that in contrast with many SQL client / server options, in the case of **Create Table**, the **Client** mode is the **default** mode.

If the table is created on the client database, it is always VIRTUAL (no other qualification is permitted) and is always transitory - it will be dropped on dismount. Both physical and persistent virtual tables can only be created on the server.

```
CREATE [SERVER] TABLE tablecreationstatement;
```

The nominated table is created on the server database. The table is not distributed by default; in order to distribute the table an explicit [Distribute Table](#) command must be issued. Note that in order to create the table on the server, the [SERVER] option must be **explicitly** stated, as client mode is the default for this command.

### Qualifiers and Parameters

<i>tableattributes</i>	An optional qualification of the type of table to be created. If omitted, a virtual table will be created (in the default mode, Client mode) or a physical table if server mode was specified. Valid type qualifications are :
PHYSICAL	This is the default for server creation. A physical (conventional) table is created, without replication. Physical tables can only be created on the server.
VIRTUAL	A virtual table is created. This is the default for client creation, which is also the default mode for the <b>Create Table</b> command. The table exists purely in memory, and the content of the table is transitory; all data in the table is lost on dismount / mount. See <a href="#">Virtual Tables</a> for further information on this type of table.
PERSISTENT	Indicates a non-transitory table - this qualifier is only valid for tables of type VIRTUAL. If not specified, virtual tables are dropped on dismount. Even for PERSISTENT tables the content of the table is lost on dismount, but the table itself still exists on re-mount.

## SQL Command Reference

**REPLICATE** The table is a physical, replicated table (specification of **PHYSICAL** is redundant if the **REPLICATE** qualification is specified). Replicator tables can only be created on the server. See [Replicator Tables](#) for further information on this type of table.

Additional attributes may be specified depending on the type of table being created :

**RAW** If the table being created is a [Raw](#) table (which is only permitted for virtual tables) this attribute may be specified, which prevents the system from adding the standard [version columns](#) to the column specification for the table. Note that Raw tables are very restricted in use and should only be created where absolutely required.

**RECLAIM** This is the default, and allows deleted rows to be re-used when row insertions are performed.

**NORECLAIM** Under special circumstances, it may be necessary to prevent the kernel from re-using rows which have been deleted, for example to use block references to associated rows of data in the table. The **NORECLAIM** attribute prevents the kernel from re-using deleted (free) rows when inserting new data rows.

**INITIALSIZE** Only valid for **VIRTUAL** tables, this attribute allows specification of the initial memory allocation for the table on creation or mount.

```
CREATE TABLE tablecreationstatement VIRTUAL INITIALSIZE (100) ;
```

**RESERVEROWS** Only valid for server tables, this attribute may be used to specify a non-standard number of rows to be reserved for addition when a session distributes the table.

```
CREATE [SERVER] TABLE tablecreationstatement RESERVEROWS (200) ;
```

**FRAMED** Specifies that transactions acting on the table will be framed, i.e. will not be autocommit. This is the default creation mode, and can only be disabled by specifying the attribute **RAW** as described above. All **FRAMED** tables have the standard [version columns](#) added (prepended) to the user-specified column list.

**TIMEDOMAIN** Future provision. Not enabled in the current release of the Lava database.

**SCHEMA** Allows creation of a table in a schema other than the default schema for the current session.

```
CREATE TABLE tablecreationstatement SCHEMA schemaname ;
```

*tablename* The name of the table to be created. If the table is to be created in a schema other than the current schema, the required schema name must be prefixed to the table name.

Column List - Spec

The list of columns to be specified for the new table. See the specification of the [column list](#) for further details.

Select Statement

The table to be created is defined in terms of a select statement (of arbitrary complexity - any variant of the select statement is permitted) performed on existing tables. See the specification of the [select statement](#) for further details.

`odbc select`

A valid select statement for the target ODBC database. The select statement will be transmitted to the target database, and the results of the select used in exactly the same way as a AS SELECT clause.

**Results**

A new table is created in the specified or current schema.

**Remarks**

The table name must be unique within the nominated or implied schema. This uniqueness requirement includes any aliases defined in the schema.

The nominated schema (if different from the current schema) must already exist.

Note that in the default mode for this command, a virtual table is created on the client database, which is transitory both in terms of content and the definition of the table itself - in this case the table will be dropped on dismount of the client database.

If the user wishes to create a physical table on the server, as would be the case with a conventional SQL server database, the following form of the command must be used :

```
CREATE [SERVER] TABLE tablename (Column List - Spec);
```

Note that in the above example, the PHYSICAL qualifier is omitted because this is the default table type for server creation.

For all standard tables (created without the RAW attribute) the system will add the standard [version columns](#) to the front of the user-specified column list - this is mandatory for distributed operation, and these columns are manipulated by the Lava Database exclusively - they may, however, be accessed by the user for reference purposes (strictly read-only).

In the CREATE AS SELECT form of the command, the individual columns of the table created are of the same data type as the selected columns in the select statement. In cases where operations are performed on data columns, the columns created will comply with the data type resulting from the relevant operation.

In the case of a CREATE AS SELECT, if the source table is FRAMED (the default) and therefore already has the standard [version columns](#) in its column list, these columns will be inherited by the newly created table and the database kernel will therefore not add these onto the column list.

**Future enhancement**

*In a future release of the Lava kernel, the option TIME-DOMAIN will be permitted, which creates a Time-Domain table set with advanced auditing and timeslicing features. For more information on the Time-Domain mechanism, see [Time-Domain](#). See [Lava Kernel Releases](#) for the planned release schedule.*

**Examples**

Note that although upper and lower case are used in the examples below for clarity, all SQL commands are case insensitive.

Create a new physical replicator table named *fred* on the server in the schema *testschema* :

```
CREATE [SERVER] TABLE testschema.fred (
    Name          STRING(50),
    Ownership     FLOAT
) REPLICATOR;
```

Create a new virtual table (the only allowable type) on the client, using a select. This example will create an identical copy of the system event log table in the schema *clientschema*.

```
CREATE TABLE clientschema.fred
AS SELECT * FROM EVENT.Sys_Event_Log;
```

Create a virtual table on the client and specify an initial memory allocation :

```
CREATE TABLE fred (
    Name          STRING(50),
    Ownership     FLOAT
) VIRTUAL INITIALSIZE(3M);
```

Create a new physical replicator table named *fred* on the server and specify no re-use of deleted rows and a non-standard row reservation :

```
CREATE [SERVER] TABLE fred (
    Name          STRING(50),
    Ownership     FLOAT
) REPLICATOR NORECLAIM RESERVEROWS(200);
```

Create a virtual raw table on the client (note that in this case the system will not add any columns to the user specified list) :

```
CREATE TABLE fred (
    Name          STRING(50),
    Ownership     FLOAT
) VIRTUAL RAW;
```

In all of the above cases except the last (in which the table is specified as RAW), the column list for the resultant table will include 5 additional columns (as described in [version columns](#)) :

```
Row_status    ROWSTATUS,
ID            ROWID,
VDT          VDT,
User_id      LONGINT,
Cache_id     LONGINT
```

These are unconditional, and should be taken into account for any operations (add, put, get) performed on the table through API calls.

**See also**

[Table Manipulation](#)

**Create user**

The **Create User** command creates a new user account on the server.

```
CREATE USER username PASSWORD password SCHEMA schema
```

**Prerequisites**

The session must have database wide CREATE privilege.

**Variants**

None. The Create User command always executes on the server.

**Qualifiers and Parameters**

<i>username</i>	The new user account name.
<i>password</i>	The initial password for the user account.
<i>schema</i>	The initial default schema for the user account.

**Results**

A new user account is created on the server.

**Remarks**

The new user name must be unique across the server database.

The password may be null; this will result in a user account which may be used without a password. This mode is not recommended, as it permits access to the database without proper security; however, if the user account is limited to non-hazardous access and privileges, this can be a useful form for guest users.

The schema specified is the schema which is automatically set as the current schema for any session logged in to the new user account.

Initially, the user account is allocated only READ privileges on the default schema - all other privileges on the default or any other schemas must be explicitly allocated.

**Future enhancement**

*User passwords will have an expiry attribute to allow password expiry at synchronous intervals*

**Examples**

The following example creates a new schema, creates a user account with the default schema set to the schema just created, and allocates full privileges on the schema to the account, with the exception of the right to drop objects in the schema or drop the schema itself :

```
CREATE SCHEMA fredschema;  
CREATE USER fred PASSWORD fredpassword SCHEMA fredschema;  
GRANT ALL ON SCHEMA fredschema TO fred;  
REVOKE DROP ON SCHEMA fredschema FROM fred;
```

**See also**

[User Manipulation](#)

## Create view

The **Create View** command is provision for future enhancement; the command is not available in the current release of the database.

### Prerequisites

*None - future provision*

### Variants

*None - future provision*

### Qualifiers and Parameters

*None - future provision*

### Results

*None - future provision*

### Remarks

*None - future provision*

### Future enhancement

This command will be available in release 5.0 of the Lava SQL engine. See [Lava Kernel Releases](#) for the planned release schedule.

### Examples

*None - future provision*

### See also

[Table Manipulation](#)

## Delete

The **Delete** command deletes rows in a data table according to the filters specified.

```
DELETE tablename WHERE Where Clause
```

### Prerequisites

The session must have DELETE privilege on the table.

### Variants

In general the answer is none. As a rule, the command is executed on the specified table, which must be unique, and therefore fully determines how and where the delete takes place. However, there are two cases where specification of the Server for execution can be legal.

The first is where the administrator knows that the table exists on the server, but is not distributed to the client. In this case, the command would fail validation on the client and would not execute. If the command is forced to execute on the server, provided that the command is syntactically correct it will execute.

The second case is where the table is a system table, and would execute on the client but the administrator wishes to specifically execute the command on the server, which would imply a different result - probably the persistence of the change, versus a transient change if executed on the client.

In these cases, the following variant may be used :

```
DELETE [SERVER] tablename WHERE Where Clause
```

### Qualifiers and Parameters

<i>tablename</i>	The name of the table within which rows are to be deleted. If the table is not in the current schema, the correct schema name must be prefixed to the table name.
<a href="#">whereclause</a>	The filter clause (optional) which limits the rows deleted in terms of one or more filter specifications. See the syntax for the <a href="#">where clause</a> for further information.

### Results

The required (filtered) rows in the nominated table are deleted.

### Remarks

The deletion occurs within a transaction frame which must be explicitly committed or rolled back.

The WHERE clause is optional, and if omitted will cause every row in the table to be deleted. A faster equivalent for this option is the [Truncate](#) command.

### Examples

The following command deletes all rows in the table with row ID less than 10, and commits the results :

```
DELETE fred WHERE ID < 10;
COMMIT;
```

### See also

[Data Extraction and Manipulation](#)

## Disable user

The **Disable User** command disables a given user account until re-enabled

```
DISABLE USER username
```

### Prerequisites

The session must have database wide UPDATE privilege.

### Variants

None. The Disable User command always executes on the server.

### Qualifiers and Parameters

*username*      The user account to be disabled.

### Results

The nominated user account is disabled - connections with this user account are no longer permitted on the server.

### Remarks

If the user account is already disabled at the time of execution, the command has no effect.

### Examples

```
DISABLE USER fred
```

### See also

[User Manipulation](#)

## Enable user

The **Enable User** command disables a given user account until re-enabled

```
ENABLE USER username
```

### Prerequisites

The session must have database wide UPDATE privilege.

### Variants

None. The Enable User command always executes on the server.

### Qualifiers and Parameters

*username*      The user account to be enabled.

### Results

The nominated user account is enabled - connections with this user account are once again permitted on the server.

### Remarks

If the user account is already enabled at the time of execution, the command has no effect.

### Examples

```
ENABLE USER fred
```

### See also

[User Manipulation](#)

**Disconnect**

The **Disconnect** command disconnects the current session - the session becomes immediately unusable.

```
DISCONNECT
```

**Prerequisites**

A valid session must exist.

**Variants**

None.

**Qualifiers and Parameters**

None.

**Results**

The session is disconnected.

The current database remains mounted, and a connect can immediately be issued to any valid Lava Server.

While disconnected, no SQL commands other than [Connect](#) may be issued.

**Remarks**

Any open transaction frames are automatically rolled back.

**See also**

[User Manipulation](#)

## Dismount

The **Dismount** command dismounts the database.

```
DISMOUNT
```

### Prerequisites

The database must currently be mounted.

### Variants

There are no variants to the command; the database on which the command acts is implied in terms of the database operational rules. See [Mount Modes](#) for further information on Lava Database operational modes.

### Qualifiers and Parameters

None.

### Results

The database is dismounted.

### Remarks

If in client mode (the default mode of operation), the client database is dismounted and discarded. A client database cannot be re-mounted.

If in Exclusive mode, the database currently mounted is dismounted, and placed in a mountable state. The database can subsequently be re-mounted in either exclusive or server mode.

### See also

[Database Manipulation](#)

## Distribute

The **Distribute** command distributes a schema or table from the server. For information on distributed tables, see [Distributed Client Operation](#).

```
DISTRIBUTE SCHEMA schemaname
DISTRIBUTE TABLE tablename
```

### Prerequisites

The session must have UPDATE privilege on any server tables nominated or implied.

### Variants

None. The **Distribute** always executes on the server.

### Qualifiers and Parameters

<i>schemaname</i>	The name of the schema to be distributed.
<i>tablename</i>	The name of a single table to be distributed. If the table does not belong to the current schema, the appropriate schema name must be prefixed to the table name.

### Results

All tables in the nominated schema are distributed (DISTRIBUTE SCHEMA) or the single table nominated is distributed (DISTRIBUTE TABLE).

### Remarks

In both the case of the DISTRIBUTE SCHEMA and the DISTRIBUTE TABLE options, if the stated schema does not exist on the client, it is created prior to distributing the table.

If a table nominated or implied by the **Distribute** command has already been distributed to the client database from which the current session is operating, the distribution request is ignored.

The command is indivisible - on returning from the command the distribution request has been fully completed, and the nominated tables are present in distributed form on the client database.

Once the distribution request has been completed, all nominated tables are present on the local client database and any data requested from these tables is retrieved locally without reference to the server. Updates to the tables occurs locally and are then distributed to the server.

For more detailed information on the operation of distributed tables, see [Distributed Client Operation](#).

### Examples

Distribute a schema from the server :

```
DISTRIBUTE SCHEMA fredschema;
```

Perform a select on one of the tables contained in the schema :

```
SELECT * FROM fred;
```

The data in the above select is retrieved locally; no communication to the server occurs.

### See also

[Schema Manipulation](#), [Table Manipulation](#)

## SQL Command Reference

## Drop schema

The **Drop Schema** command drops an entire schema, including all objects belonging to the schema.

```
DROP SCHEMA schemaname
```

### Prerequisites

The session must have DROP privilege on the nominated schema.

There may be no locks on any tables belonging to the schema.

### Variants

None. The location of the schema can always be uniquely determined as schema names are required to be unique across the database. Therefore, if the schema is located only on the client database, the command is executed on the client. If the schema is located only on the server database or is local in distributed form, the command is executed on the server. For information on distributed schemas, see [Distributed Client Operation](#).

### Qualifiers and Parameters

*schemaname*      The name of the schema to be dropped.

### Results

The nominated schema is dropped.

### Remarks

If any tables in the nominated schema currently have locks, the drop request will be denied.

If the schema is distributed, the drop request will effectively execute on both server and client - the request will first be executed on the server to determine validity, and if the drop succeeds on the server it will be executed on the client. The result is that the nominated schema ceases to exist both on the server and on the client.

### Examples

Drop the schema *fredschema* :

```
DROP SCHEMA fredschema;
```

### See also

[Schema Manipulation](#)

## Drop sequence

The **Drop Sequence** command is listed as future provision, and is not available in the current release of the Lava SQL engine.

*The **Drop Sequence** command drops a sequence in the current schema.*

```
DROP SEQUENCE sequencename
```

### Prerequisites

*The session must have **DROP** privilege on the schema to which the sequence belongs.*

### Variants

*None. The location of the sequence is determined in terms of the fact that sequence names are unique within a schema, and the command is executed on either client or server depending on the origin of the sequence.*

### Qualifiers and Parameters

*sequencename*    *The name of the sequence to be dropped.*

### Results

*The sequence is dropped.*

### Remarks

*If the sequence originates on the server, it is dropped on both server and client.*

### Future enhancement

The Drop Sequence command is specified for future enhancement of the Lava Database, and will only be available in release 5.0 of the kernel and SQL engine.

### Examples

```
DROP SEQUENCE fredsequence;
```

### See also

[Miscellaneous Statements and Clauses](#)

## Drop relation

The **Drop Relation** command drops a relation.

```
DROP RELATION BETWEEN PARENT parentname AND childname
```

### Prerequisites

The session must have DROP privilege on the schema to which the relation belongs.

### Variants

None. Relations can only be defined on the server.

### Qualifiers and Parameters

<i>parentname</i>	The name of the parent table for the relation
<i>childname</i>	The name of the child table for the relation

### Results

The relation is dropped.

### Remarks

The parent (*one* table in a *one : many* relation) and child (*many* table in a *one : many* relation) tables must be correctly specified in order for the relation to be correctly identified - this is in order to ensure that the correct relation is specified in all cases.

Once the relation is dropped, any constraints implied on table updates by the relation no longer apply.

No table data is affected by the dropping of the relation.

### Examples

In the example below, the constraint relation between Sys\_Event\_Type and Sys\_Event\_Log is dropped, following which a particular row in the Sys\_Event\_Type table can be deleted regardless of implied links to this entry from the Sys\_Event\_Log table. The relation is then re-established.

```
ALTER SESSION SCHEMA event;  
DROP RELATION BETWEEN PARENT Sys_Event_Type AND Sys_Event_Log;  
DELETE Sys_Event_Type WHERE ID = 101;  
CREATE RELATION FROM PARENT Sys_Event_Type TO Sys_Event_Log  
COLUMN Event_Type_id CONSTRAINT RESTRICT;
```

### See also

[Table Manipulation](#)

## Drop synonym / Drop alias

The **Drop Synonym** and **Drop Alias** commands are equivalent alternatives of the command to drop the alias for a table.

```
DROP SYNONYM tablealias
DROP ALIAS tablealias
```

### Prerequisites

The session must have DROP ALIAS privilege on the schema in which the alias is defined.

### Variants

None. The **Drop Synonym** command always executes on the server.

### Qualifiers and Parameters

*tablealias*      The alias to be dropped.

### Results

The nominated alias is dropped.

### Remarks

The **Drop Alias** command has no effects except that the alias is no longer available.

### Examples

```
DROP ALIAS errorlog;
```

### See also

[Table Manipulation](#)

## Drop table

The **Drop Table** command drops a Lava table.

```
DROP TABLE tablename;
```

### Prerequisites

The session must have DROP privileges on the nominated table.

There may be no active locks on the table to be dropped.

### Variants

In general the answer is none. As a rule, the command is executed on the specified table, which must be unique, and therefore fully determines how and where the drop takes place. However, there is one case where specification of the Server for execution can be legal.

The administrator may know that the table exists on the server, but is not distributed to the client. In this case, the command would fail validation on the client and would not execute. If the command is forced to execute on the server, provided that the command is syntactically correct it will execute.

In this cases, the following variant may be used :

```
DROP [SERVER] tablename
```

### Qualifiers and Parameters

<i>tablename</i>	The name of the table to be dropped. If the table does not belong to the current schema, the appropriate schema name must be prefixed to the table name.
------------------	--

### Results

The nominated table is dropped, together with any data contained in the table.

Any relations to the nominated table are dropped as a result of a mandatory component of the relation no longer existing.

### Remarks

The table is identified from the nominated table name. If the table is a client table, the drop is executed on the client. If the table is either located only on the server, or is distributed to the client, the command is executed on the server. If the server drop succeeds, the distributed copy of the table on the client is dropped on the client database.

All data present in the table at the time of the drop is irretrievably lost.

Dropping a table does not result in transaction frame data - no rollback option to recover the table or table data is available subsequent to dropping a table (the drop action does not affect current transaction frames in any way, as any pending updates to the table in question would have caused active locks on the table, preventing it from being dropped).

### Examples

```
DROP TABLE fredschema.fred;
```

### See also

[Table Manipulation](#)

## Drop user

The **Drop User** command drops a current user account.

```
DROP USER username
```

### Prerequisites

The session must have DROP USER privilege.

The account to be dropped may not be the account on which the current session is formed.

The default system account ([SYSTEM](#); created on creating the database) cannot be dropped.

### Variants

None. The **Drop User** command is always executed on the server.

### Qualifiers and Parameters

*username*      The name of the user account to be dropped.

### Results

The nominated user account is dropped.

### Remarks

After dropping a user account all information contained in the user account (password, privileges) is irretrievably lost.

If the nominated account is the only account (with the exception of the SYSTEM account) able to access any particular schema(s), those schemas will be inaccessible until another account is created with appropriate privilege or a currently existent account is granted appropriate privileges from the SYSTEM account.

### Examples

```
DROP USER fred;
```

### See also

[User Manipulation](#)

**Drop view**

The **Drop View** command is provision for future enhancement; the command is not available in the current release of the database.

**Prerequisites**

*None - future provision*

**Variants**

*None - future provision*

**Qualifiers and Parameters**

*None - future provision*

**Results**

*None - future provision*

**Remarks**

*None - future provision*

**Future enhancement**

This command will be available in release 5.0 of the Lava SQL engine. See [Lava Kernel Releases](#) for the planned release schedule.

**Examples**

*None - future provision*

**See also**

[Table Manipulation](#)

## Grant Role

The **Grant Role** command is provision for future enhancement; the command is not available in the current release of the database.

### Prerequisites

*None - future provision*

### Variants

*None - future provision*

### Qualifiers and Parameters

*None - future provision*

### Results

*None - future provision*

### Remarks

*None - future provision*

### Future enhancement

This command will be available in release 5.0 of the Lava SQL engine. See [Lava Kernel Releases](#) for the planned release schedule.

### Examples

*None - future provision*

### See also

[Grant Privilege](#), [User Manipulation](#)

## Grant privilege

The **Grant Privilege** command grants a particular privilege to a nominated user account.

```
GRANT privilege / all ON targettype targetname TO USER username;
```

### Prerequisites

The session must have GRANT privilege on the schema named or implied in the **Grant** action.

### Variants

None. Grant commands are always executed on the server.

### Qualifiers and Parameters

<i>privilege</i>	The privilege to be granted. This may be a specific privilege, or the ALL qualifier grants all available privileges on the nominated schema or table. For a list of available privileges, see <a href="#">Lava Privileges</a> . ALL is not a valid option for the DATABASE target type.
<i>targettype</i>	The target of the privilege may be : DATABASE     The privilege applies database-wide. This is only valid for certain privileges. SCHEMA        The nominated target is a schema - all objects contained in the schema are included in the privilege granted. TABLE        The nominated target is a table - only the specified table is affected by the privilege granted.
<i>targetname</i>	The name of the target entity. If the target type was DATABASE, the <i>targetname</i> is always LAVA. If the target type was SCHEMA, the <i>targetname</i> must specify a valid schema on the server. If the target type was TABLE, the <i>targetname</i> must specify a valid table on the server. If this table does not belong to the current schema, the appropriate schema name must be prefixed to the table name.
<i>username</i>	The name of the user account which is to be granted the specified privilege.

### Results

The nominated user account is granted the privilege(s) stated in the command.

### Remarks

The privilege will only be accorded the user account on the next connection to the server after the grant has been executed.

If the **Grant** is executed on a SCHEMA, all objects (table, alias, sequence) are included in the grant. In addition, any future objects added to the schema of whatever kind are automatically included in the grant regardless of when they are added.

If a user is to be granted access to the greater number of objects in a particular schema, and the lesser number of objects are to be withheld, it may be more efficient to Grant access to the entire schema, then [Revoke](#) access to only those objects for which permission is to be withheld.

The *all* option, when applied to a schema or a database, includes all privileges except the GRANT and REVOKE privileges - these must be granted explicitly.

### Future enhancement

*The ability to define Roles and allocate Roles to user accounts rather than individual privileges will be added in a forthcoming release of the Lava Database. See [Lava Kernel Releases](#) for the planned release schedule.*

### Examples

```
GRANT ALL ON SCHEMA fredschema TO USER fred;  
REVOKE DROP ON SCHEMA fredschema FROM USER fred;  
REVOKE UPDATE ON TABLE products FROM USER fred;
```

**See also**

[Lava Privileges](#), [User Manipulation](#)

## Group by Clause

The **Group By** clause is used in [Select](#) statements to group aggregate results. The **Group by** clause is optional - if omitted, an aggregate clause provides the aggregate information across the entire select range. If specified, results are grouped according to the specified group columns.

The general syntax for the group by clause is

```
GROUP BY column_1, column_2, ..., column_n
```

where *column\_1*, *column\_2*, ..., *column\_n* are the set of columns which define the grouping criterion.

For an illustration of grouped aggregates results, see the examples at the end of this clause, or the more elaborate examples in [Appendix III : SQL Examples](#).

The following aggregate functions are currently supported :

COUNT	Calculates the number of entries (result rows) scanned in processing the Select statement
SUM	Calculates the numeric sum of the column across all entries scanned
AVG	Calculates the arithmetic average of the column across all entries scanned
MIN	Finds the smallest arithmetic or alphabetic value across all entries scanned
MAX	Finds the largest arithmetic or alphabetic value across all entries scanned

### Remarks

The grouping columns do not have to be selected in the column list of the Select statement.

If, however, a column is selected without an aggregate function, that column should be specified in the Group By list - see the simple example below for an illustration of this point.

An [Order By](#) clause may be added to sort the aggregate results in a desired sequence.

### Future enhancement

*The **DISTINCT** clause will be supported in release 5.0 of the Lava SQL engine.*

### Examples

The following simple example will calculate the sum of invoiced amounts on an invoice detail table, and group the results by customer :

```
SELECT SUM(amount), Customer FROM InvDetail GROUP BY Customer;
```

For a more detailed worked example which lists a grouped aggregate, see [Grouping aggregates, subqueries](#) in the SQL Example appendix.

### See also

[Select statement](#), [Data Extraction and Manipulation](#)

## Index

The **Index** command may be used to create a single-column index on a nominated column for a table.

Indexes in a Lava database are not completely identical to conventional indexes in a SQL database. The largest difference is the limit of one column per index - functionality normally provided by multi-column indexes is derived through alternate means in a Lava database.

```
INDEX COLUMN column ON TABLE tablename
```

## Insert

The **Insert** command allows insertion of new rows into an existing table.

```
INSERT INTO tablename (columnlist) VALUES (valuelist)
INSERT INTO tablename (columnlist) selectstatement
```

### Prerequisites

The session must have INSERT privilege on the nominated table.

In the SELECT form of the command, the session requires READ privilege on any tables accessed in the select statement.

### Variants

None. The location (client or server) for the nominated table may be identified from the fact that table names are unique within a schema. The command is either executed in distributed mode on the client if the table is distributed, or on server if the table is not distributed. If the table is a client table, the command is executed on the client.

### Qualifiers and Parameters

<i>tablename</i>	The name of the table into which rows are to be inserted. If the table does not belong to the current schema, the appropriate schema name must be prefixed to the table name.
<i>columnlist</i>	A list of columns to be set in each row added to the nominated table, separated by commas. For further information on the column list, see <a href="#">Column List</a> .
<i>valuelist</i>	In the VALUES form of the command, a list of values corresponding to the list of columns above, separated by commas. For further information on the value list, see <a href="#">Value List Clause</a> . In this form of the command, only one row is inserted into the nominated table.
<i>selectstatement</i>	In the SELECT form of the command, a select statement of arbitrary complexity specifying the values to be set for the columns specified in the column list. For further information on this clause, see <a href="#">Select Statement</a> . In this form of the command, any number of rows from 0 through many thousands of rows may be inserted into the nominated table with a single command.

### Results

A number of rows (0 or more, depending on the form of the command) are inserted into the nominated table with specified columns set to stipulated or implied values. Non-specified columns are left blank or 0.

### Remarks

The number of columns in the column list and the value list must be equal. Similarly, for the SELECT form, the number of columns in the select column list must be equal to the insert column list.

## SQL Command Reference

The specified columns and the stipulated (value list) values or implied (select) values are associated one-for-one in the appropriate list; in other words the first entry in the insert column list matches with the first value in the value list and so on.

The data types of the insert column and the stipulated or implied value column do not have to be the same. The SQL engine will perform the best possible data conversion between the value and the insert column type where a conversion is required. Should data conversion be impossible (for example a string column which contains non-numeric characters is inserted into a numeric column) the target column is left null or blank.

The inserted rows will be contained in a transaction frame, and will only be written to the nominated table on execution of a Commit command. If a Rollback is executed or the session is disconnected before a Commit is issued, the data will be discarded.

For non-[Raw](#) tables, the Lava kernel will automatically assert the ID column of each inserted (added) row to the correct row ID value - the user may not set this column. If rows are to be placed at a particular predetermined row in the nominated table, the [Update](#) command should be used instead.

The actual rows used for the inserted data depend on (1) whether the nominated table was created with the RECLAIM attribute set, in which case any deleted rows found in the table will first be used before the table extended, or (2) if RECLAIM is not specified for the nominated table or there are no free (deleted) rows, the row used will be the next available row, which would be (on the client) the next reserved row, or (on the server) the next non-reserved row after the last used row in the table. See [Create Table](#) for more information on table attributes.

### Examples

```
INSERT INTO fred (name, ownership) VALUES ('Bloggs', 23.56);
```

```
INSERT INTO fred (name, ownership)
SELECT
    a.name, b.ownership
FROM
    customer a, ownership b
WHERE
    a.id = b.customer_id AND
    b.ownership BETWEEN 30 AND 45;
```

### See also

[Data Extraction and Manipulation](#)

## Order by Clause

The **Order by** clause is an optional clause which may be specified for any [Select statement](#) to sort the results by one or more columns, ascending or descending.

The simplest form of the **Order by** clause is :

```
ORDER BY Column_1, Column_2, ..., Column_3
```

Greater control over the result may be obtained by using the ASC and DESC qualifiers in order to achieve ascending or descending order on individual columns, as follows :

```
ORDER BY Column_1 ASC, Column_2 DESC
```

If the ASC and DESC qualifiers are omitted for any column, the default is Ascending (ASC).

In order to perform a case sensitive sort, the qualifier CASESENSITIVE (abbreviation CS) may be appended, as follows :

```
ORDER BY Column_1 CS  
ORDER BY Column_1 DESC CS
```

The default sort is case insensitive.

### Remarks

Although it is permissible to specify an Order By clause on a subquery, in the general case this will only have the effect of slowing down the encapsulating command as an extra sorting phase will be executed for each subquery execution, which in almost all cases will not affect the overall result. It will certainly not change the content of the result, and in most cases a specific final result order can more effectively be achieved by specifying an Order By clause on the outermost Select.

Sort columns may be of any datatype. There is, however, a limitation to functionality with [variable length](#) datatypes - in this case, the sort is performed taking into account the base portion of the column only; the variable portion is not considered for sort purposes.

Any number of sort columns may be specified, and the number of columns in the sort specification will have very little affect on the sort speed.

The sort algorithm used is an in place algorithm (with some stack requirement) which is order  $n \log n$ , i.e. about as fast as you can go.

### Examples

```
SELECT CustomerName FROM Customers ORDER BY CustomerName
```

### See also

[Group By](#), [Select statement](#), [Data Extraction and Manipulation](#)

## Rename schema

The **Rename Schema** command renames the nominated schema.

```
RENAME SCHEMA schemaname TO newname;
```

### Prerequisites

The session must have ALTER privilege on the nominated schema.

### Variants

None. The schema can be uniquely identified and is modified on the appropriate database.

### Qualifiers and Parameters

<i>schemaname</i>	The name of an existing schema.
<i>newname</i>	The new name for the schema.

### Results

The schema is renamed to *newname*.

### Remarks

The schema is immediately renamed. There is no rollback option.

### See also

[Schema Manipulation](#)

## Rename sequence

The **Rename Sequence** command renames the nominated sequence.

```
RENAME SEQUENCE sequencename TO newname;
```

### Prerequisites

The session must have ALTER privilege on the schema to which the sequence belongs.

### Variants

None. The sequence can be uniquely identified and is modified on the appropriate database.

### Qualifiers and Parameters

<i>sequencename</i>	The name of an existing sequence. If the sequence does not belong to the current schema, the appropriate schema name must be prefixed to the sequence name.
<i>newname</i>	The new name for the sequence.

### Results

The sequence is renamed to *newname*.

### Remarks

The sequence is immediately renamed. There is no rollback option.

### See also

[Miscellaneous Statements and Clauses](#)

## Rename synonym / Rename alias

**Rename Synonym** and **Rename Alias** are equivalent alternative commands which renames the nominated alias.

```
RENAME SYNONYM aliasname TO newname;  
RENAME ALIAS aliasname TO newname;
```

### Prerequisites

The session must have ALTER privilege on the schema to which the alias belongs.

### Variants

None. The alias can be uniquely identified and is modified on the appropriate database.

### Qualifiers and Parameters

<i>aliasname</i>	The name of an existing alias. If the alias does not belong to the current schema, the appropriate schema name must be prefixed to the alias name.
<i>newname</i>	The new name for the alias.

### Results

The alias is renamed to *newname*. The alias will continue to refer to the original table for which the alias was defined.

### Remarks

The alias is immediately renamed. There is no rollback option.

### See also

[Table Manipulation](#)

## Rename table

The **Rename Table** command is an alternative syntax to the ALTER TABLE ... RENAME command. See [Alter Table](#) for information on this command.

```
RENAME TABLE tablename TO newtablename
```

### Qualifiers and Parameters

*tablename*

The *tablename* parameter specifies the table to be renamed. If the table is not in the session's current schema, a schema prefix is required to fully identify the table.

*newtablename*

The new table name for the nominated table.

### Results

The nominated table is renamed to *newtablename*. See [Alter Table](#) for more information on this command.

### See also

[Alter Table](#), [Table Manipulation](#)

## Rename user

The **Rename User** command renames the nominated user account.

```
RENAME USER username TO newname;
```

### Prerequisites

The session must database wide ALTER privilege.

### Variants

None. The user account can only exist on the server.

### Qualifiers and Parameters

<i>username</i>	The name of an existing user account.
<i>newname</i>	The new name for the user account.

### Results

The user account is renamed to *newname*.

### Remarks

The user account is immediately renamed. There is no rollback option. The user account will retain all current attributes and privileges.

### See also

[User Manipulation](#)

**Rename view**

The **Rename View** command is provision for future enhancement; the command is not available in the current release of the database.

**Prerequisites**

*None - future provision*

**Variants**

*None - future provision*

**Qualifiers and Parameters**

*None - future provision*

**Results**

*None - future provision*

**Remarks**

*None - future provision*

**Future enhancement**

This command will be available in release 5.0 of the Lava SQL engine.

**Examples**

*None - future provision*

See also

[Table Manipulation](#)

## Restore

The **Restore** command restores an existing backup file.

```
RESTORE BACKUP backupfile
RESTORE BACKUP backupfile KEY encryptionkey
```

### Prerequisites

The session must have RESTORE privilege on the schema to be restored. In addition, the session must have DROP privilege on every table to be restored - any table for which the session does not have DROP privilege will not be restored. If the schema to be restored does not yet exist in the mounted database, the session must have database wide CREATE privilege.

The database must be mounted in Exclusive mode - see [Mount mode](#) for information on Exclusive mount.

A backup folder for the session must have been asserted - see [Alter Session](#).

### Variants

None. The restore is always performed to an Exclusive-mount database.

### Qualifiers and Parameters

<i>backupfile</i>	The backup file to be restored. This must be located in the current backup folder - see <a href="#">Alter Session</a> for information on setting the current backup folder.
<i>encryptionkey</i>	The encryption key is required only if the backup was encrypted; i.e. a key was specified in the backup. In this case, the key must match the backup key exactly, and is case sensitive.

### Results

The schema nominated in the backup file is restored from the backup.

### Remarks

The mount mode must be [Exclusive](#).

The default file type (file extension) for Lava backup files is *.lbs* (Lava Backup Set).

All tables nominated in the backup file will be dropped before being restored.

Once the restore action is complete, the database may be dismounted and re-mounted in Server mode.

### Examples

Assert a backup folder and restore the backup *fredbackup* :

```
ALTER SESSION BACKUP FOLDER q:\lava\backup;
RESTORE BACKUP fredbackup KEY "jzX12 owCP";
```

### See also

[Alter Session](#), [Backup](#), [Schema Manipulation](#)

## Revoke privilege

The **Revoke Privilege** command revokes a particular privilege from a nominated user account.

```
REVOKE privilege / all ON targettype targetname FROM USER username;
```

### Prerequisites

The session must have GRANT privilege on the schema named or implied in the **Revoke** action.

### Variants

None. Revoke commands are always executed on the server.

### Qualifiers and Parameters

<i>privilege</i>	The privilege to be revoked. This may be a specific privilege, or the ALL qualifier revokes any privileges currently granted the user on the nominated schema or table. For a list of available privileges, see <a href="#">Lava Privileges</a> .
<i>targettype</i>	The target of the privilege may be : DATABASE     The privilege applies database-wide. This is only valid for certain privileges. SCHEMA        The nominated target is a schema - all objects contained in the schema are included in the privilege revoked. TABLE        The nominated target is a table - only the specified table is affected by the privilege revoked.
<i>targetname</i>	The name of the target entity. If the target type was SCHEMA, the <i>targetname</i> must specify a valid schema on the server. If the target type was TABLE, the <i>targetname</i> must specify a valid table on the server. If this table does not belong to the current schema, the appropriate schema name must be prefixed to the table name.
<i>username</i>	The name of the user account from which the specified privilege is to be revoked.

### Results

The privilege(s) stated in the command are revoked from the nominated user account.

### Remarks

The privileges will only be revoked from the user account on the next connection to the server after the revoke has been executed.

If the **Revoke** is executed on a SCHEMA, the privilege is revoked on all objects (table, alias, sequence) within the schema. In addition, the privilege is automatically revoked on any future objects of whatever kind added to the schema regardless of when they are added.

If a user is to be granted access to the greater number of objects in a particular schema, and the lesser number of objects are to be withheld, it may be more efficient to [Grant](#) access to the entire schema, then Revoke access to only those objects for which permission is to be withheld.

### Future enhancement

*The ability to define Roles and allocate Roles to user accounts rather than individual privileges will be added in a forthcoming release of the Lava Database. See [Lava Kernel Releases](#) for the planned release schedule.*

### Examples

```
GRANT ALL ON SCHEMA fredschema TO USER fred;  
REVOKE DROP ON SCHEMA fredschema FROM USER fred;  
REVOKE UPDATE ON TABLE products FROM USER fred;
```

**See also**

[Lava Privileges](#), [User Manipulation](#)



## Savepoint

The **Savepoint** command creates a transaction subframe which may be used to perform partial [commit](#) or [rollback](#) actions.

```
SAVEPOINT subframe
```

### Prerequisites

None. The prerequisites apply to the transactions that comprise the transaction frame within which the savepoint is to be defined; the savepoint action itself has no prerequisites.

### Variants

None. The **Savepoint** command always acts on the current session.

### Qualifiers and Parameters

*subframe*            The name of the transaction subframe (savepoint) to be created.

### Results

A transaction subframe is created (a nested transaction frame) which allows nested commit or rollback actions.

### Remarks

It is permissible to create a savepoint before any transactions (updates) have been issued, but this is not necessary. On executing the first update (delete, update, insert) command, the database kernel will automatically create a transaction frame, which may be used for complete commit or rollback actions.

It is only necessary to create a nested transaction frame (subframe) if portions of the transaction sequence may have to be rolled back individually in the case of a validation failure or other reason.

Regardless of how many savepoints have been created within a given transaction frame, executing a commit or rollback without nominating any of these savepoint names will commit or roll back the entire transaction frame.

If subframes are created sequentially without performing any interim subframe commits, this will result in deeply nested transaction frames which will result in a slower complete commit when the entire transaction is committed. However, the difference is not particularly significant and although it is good practice to close off each subframe once the conditions for the subframe have been validated, it is not essential to do so.

### Examples

In the following example, a savepoint is created partway through a sequence of updates. A rollback is performed specifying that subframe, subsequent to which further updates are performed and a complete commit is executed.

```
DELETE fred WHERE ID = 5;
SAVEPOINT newsubframe;
DELETE fred WHERE ID = 6;
ROLLBACK newsubframe;
UPDATE fred SET name = 'fred' WHERE ID = 7;
COMMIT;
```

### See also

[Commit](#), [Rollback](#), [Transaction Statements](#)

## Select, *Select Statement*

The **Select** statement is used primarily to retrieve data in the form of a result set from one or more tables.

In general, the **Select** statement may be used in a number of instances. It can be used as a standalone select, in which case the select creates a result set which is returned to the user. Select statements can also be used in the form of [subqueries](#) within Insert, Update and Create Table statements to substitute for coded value or column lists. It is also possible to use select statements in the *where* clause of Delete statements, most commonly in an *exists* clause to limit the scope of the delete. Finally, select statements may be used as [subqueries](#) within select statements in certain instances where otherwise specification of a column, a table or a value would be used.

The simplest form of the select statement is as follows :

```
SELECT
    Column list - Select
FROM
    Table list
```

In most cases, this will be augmented by a Where clause, as follows :

```
SELECT
    Column list - Select
FROM
    Table list
WHERE
    Where Clause
```

In addition, the following clauses may be appended to a select statement :

```
ORDER BY
    Order by Clause
```

The **Order by** clause sorts the result set in terms of the specified columns. See the [Order by Clause](#) for further information.

```
GROUP BY
    Group by Clause
```

The **Group by** clause allows the compilation of aggregate values, grouped in terms of the columns specified. See the [Group by Clause](#) for further information.

The final two clauses, **Order by** and **Group by**, may be used in conjunction to order an aggregate result set.

### Prerequisites

The session must have SELECT privilege on any table used within the select or any subqueries to the select.

### Variants

The select command is executed by default on the appropriate location (client or server) depending on the tables included in the select, and the distribution status of these tables.

If all the tables nominated in the select (including any subqueries used in the select statement) have been distributed to the client, the select statement is executed on the client by default.

If any tables (1 or more) used in the select have not been distributed to the client, the select will be executed on the server - regardless of any tables which have been distributed to the client.

As the system tables (with the exception of the user table, `Sys_Users`, and the parameter table, `Sys_Parameter`) are not distributed to the client, any select performed on a system table will be executed on the server. In this one case, it is possible to direct the SQL engine to execute the query on the client, where the desired result from one or more system tables is to be determined from the client database.

```
SELECT [CLIENT] * FROM SYS_OBJECTS
```

In general, however, it is not possible to force the SQL engine to execute a statement on the client which it has determined should be executed on the server. The reason for this is that if even one table nominated in the select has not been distributed to the client, formulating the result set is not possible on the client, and the select must be executed on the server.

It is possible to force a select statement to execute on the server, as follows :

```
SELECT [SERVER] * FROM fred
```

Since the location of the tables in the above case will not be validated on the client, it is possible to force a select to be evaluated on the server which then fails as a result of one or more of the nominated tables not existing on the server (but only on the client). The onus is on the user to ensure that selects forced to server execution will validate correctly in terms of table location.

### Qualifiers and Parameters

The following items are all documented comprehensively in individual clause descriptions. The brief introductions to the clauses may be expanded by following the hyperlinks.

<a href="#">Column list</a>	A list of columns from any of the tables in the table list. Computations, aggregate functions and subqueries are permitted.
<a href="#">Table list</a>	A list of tables to be queried. Tables constructed from subqueries are permitted.
<a href="#">Where Clause</a>	An optional clause in the case of single-table queries, the Where clause is mandatory for queries on more than one table. It specifies joins between tables in the table list, as well as any filters to be applied to the results.
<a href="#">Order by Clause</a>	An optional clause providing for single- or multi-column sorts on the result set.
<a href="#">Group by Clause</a>	An optional clause used only in the case of aggregate functions applied to the select column list, to group the aggregate results.

### Results

The Select query builds a result set in accordance with the select column list and the conditions in the Where clause. The result set is returned in the form of a [Virtual Lava Table](#), placed in the current default schema.

### Remarks

The table is named *SQLresult*, and can be used in further Select queries. As with any other Lava Table, the *SQLresult* table can be renamed or dropped. Due to the fact that this table is unconditionally a [Raw](#) table, rows cannot be deleted from the table, but all other table data operations are supported.

If the result table is not renamed, it will be overwritten on execution of the next Select statement. The result is always returned in a table named *SQLresult*, with the implication that if the last result table has not been renamed, it will be dropped and the new result table will take its place.

The Lava select requires, for selects involving more than one table, that every table be specified in a valid join with another table in the select. In other words, if a table is included in the select, the table must be joined to at least one other table in the select. If any table in the select is found to have no joins to the other tables in the select, the select will be disallowed and will return an error.

### Examples

A simple select statement is provided below :

```
SELECT * FROM SYS_OBJECTS WHERE ID < 20;
```

For further examples, see the section [Appendix III : SQL Examples](#)

### See also

[Group by](#), [Order by](#), [Column list - Select](#), [Where Clause](#), [Subqueries](#), [Data Extraction and Manipulation](#)

**Set**

The **Set** command is provision for future enhancement; the command is not available in the current release of the database.

**Prerequisites**

*None - future provision*

**Variants**

*None - future provision*

**Qualifiers and Parameters**

*None - future provision*

**Results**

*None - future provision*

**Remarks**

*None - future provision*

**Future enhancement**

This command will be available in release 5.0 of the Lava SQL engine. See [Lava Kernel Releases](#) for the planned release schedule.

**Examples**

*None - future provision*

## Subqueries

A *subquery* is a form of the SELECT statement which is nested within another SQL statement, the encapsulating statement, and bounded by parentheses. The row(s) returned by the subquery are used by the encapsulating statement in the place of a value, column or data table.

Subqueries may be used for the following purposes:

- to define the set of rows to be inserted into the target table of an INSERT or CREATE TABLE statement
- to define a value to be assigned to a column in an UPDATE statement
- to provide values for conditions in WHERE clauses of SELECT, UPDATE, and DELETE statements
- to define a table to be operated on by an encapsulating query. In this case the subquery replaces the conventional table specification in the FROM clause of a Select statement.

The following statement prototypes illustrate typical uses of subqueries.

```
CREATE TABLE
    table_1 (columnlist)
AS SELECT
    columnlist
FROM
    table_2
```

```
UPDATE
    table_1 alias_1
SET
    column = (
        SELECT
            expression
        FROM
            table_2
        WHERE
            alias_1.column = table_2.column
    )
```

```
SELECT
    columnlist
FROM
    table_1 alias_1
WHERE
    column_1 = (
        SELECT
            expression
        FROM
            table_2
        WHERE
            alias_1.column = table_2.column
    )
```

```
DELETE
    table_1 alias_1
WHERE
    column = (
        SELECT
            expression
```

```
FROM
    table_2
WHERE
    alias_1.column = table_2.column
)
```

In principle, a subquery is used to obtain a result set where it is easier - or essential - to phrase a select independently of the encapsulating statement. In some cases, such as an Update statement, subqueries are the only way to access values not directly related to the row of the table being updated. In other cases, such as complex SQL statements, subqueries are often a way to simplify the overall statement.

**Remarks**

Subqueries used to substitute for tables in the FROM clause of a Select statement may not use correlation variables linked to the encapsulating statement, as these queries are executed first of all in evaluating a Select statement - therefore, correlation variables cannot be evaluated since none of the other tables in the encapsulating statement are current when the subquery is being evaluated.

Broadly speaking, subqueries fall in two categories : correlated and non-correlated. Non-correlated queries are executed once only (as in the above case, with replacement of a FROM table entry), whereas correlated queries are executed once for every row processed by the encapsulating statement.

A *correlated subquery* is a subquery that is evaluated once for each row processed by the parent statement. The parent statement can be a SELECT, UPDATE, or DELETE statement. The following examples show the general syntax of a correlated subquery:

A subquery can itself contain one or more subqueries. There is no explicit limit on the number or depth of subqueries, but it is worth remembering that deeply nested correlated subqueries do perform slower than the equivalent flattened query.

**See also**

[Appendix III : SQL Examples](#), [Data Extraction and Manipulation](#)

***Lava* pseudo-table**

The *Lava* pseudo-table is used where a clause requires reference to a table but the specific instance does not require an actual table. In addition, the *Lava* pseudo-table may be used to obtain information on the current session from the SQL database.

```
SELECT expression FROM Lava
```

**Qualifiers and Parameters**

TIME

DATE

VDT

CURRENTUSER

CURRENTSESSION

Numeric expressions

**Remarks**

The *Lava* pseudo-table may be used with any non-table related expression, including all the reserved expressions and functions.

**Example**

```
SELECT PI * 4 FROM Lava
```

```
SELECT TIME FROM Lava
```

```
SELECT CURRENTUSER FROM Lava
```

**See also**

[Reserved expressions](#)

**Table List Clause**

*Table Spec*

*[schema.]tablename*

The basic form of the table list is as follows :

```
table_1, table_2 ... table_n
```

Any of the tables may be a subquery :

```
table_1,  
(SELECT * FROM system.sys_objects WHERE ID BETWEEN 20 AND 30),  
table_2
```

In the above example, the second table is constructed from the Sys\_Objects table from the rows 20 through 30.

The full syntax of the select table list may be found in the section [SQL Syntax Specification](#).

**Remarks**

The subquery table may be specified in terms of any valid select.

The table subquery may not be a correlated subquery as it is always evaluated first, rendering any form of correlation impossible.

A table query may return any number of rows, as the result is treated as for any other table.

[Subqueries](#) may be nested to any depth. As table subqueries are evaluated first, and only once, there is relatively little penalty to complex and deeply nested queries in this position.

**See also**

[Select statement](#), [Subquery](#), [Data Extraction and Manipulation](#)

## Truncate

The **Truncate** command truncates a Lava table, effectively deleting all rows in the table.

```
TRUNCATE TABLE tablename
```

### Prerequisites

The session must have TRUNCATE privilege on the nominated table.

### Variants

None. The command is executed on the specified table, which must be unique, and therefore fully determines how and where the truncation takes place.

### Qualifiers and Parameters

*tablename*      The name of the table to be truncated. If the table is not in the current schema, the appropriate schema name must be prefixed to the table name.

### Results

The nominated table is truncated - i.e. all data rows in the table are deleted.

### Remarks

The **Truncate** command does not result in transaction frame data - the loss of data is of immediate effect and no rollback action is possible on the deleted rows.

### Examples

```
TRUNCATE TABLE fred;
```

### See also

[Table Manipulation](#)

## Undelete

The **Undelete** command is provision for future enhancement; the command is not available in the current release of the database.

### Prerequisites

*None - future provision*

### Variants

*None - future provision*

### Qualifiers and Parameters

*None - future provision*

### Results

*None - future provision*

### Remarks

*None - future provision*

### Future enhancement

This command will be available in release 5.0 of the Lava SQL engine. See [Lava Kernel Releases](#) for the planned release schedule.

### Examples

*None - future provision*

### See also

[Data Extraction and Manipulation](#)

## Update

The **Update** command updates rows as specified in the nominated table, optionally filtered using a Where clause.

```
UPDATE tablename SET columnvaluelist WHERE Where Clause
```

### Prerequisites

The session must have UPDATE privilege on the nominated table.

### Variants

In general the answer is none. As a rule, the command is executed on the specified table, which must be unique, and therefore fully determines how and where the update takes place. However, there are two cases where specification of the Server for execution can be legal.

The first is where the administrator knows that the table exists on the server, but is not distributed to the client. In this case, the command would fail validation on the client and would not execute. If the command is forced to execute on the server, provided that the command is syntactically correct it will execute.

The second case is where the table is a system table, and would execute on the client but the administrator wishes to specifically execute the command on the server, which would imply a different result - probably the persistence of the change, versus a transient change if executed on the client.

In these cases, the following variant may be used :

```
UPDATE [SERVER] tablename SET columnvaluelist WHERE Where Clause
```

### Qualifiers and Parameters

*tablename*

The name of the table to be updated. If the table is not in the current schema, the appropriate schema name must be prefixed to the table name.

The following items are all documented comprehensively in individual clause descriptions. The brief introductions to the clauses may be expanded by following the hyperlinks.

[columnvaluelist](#) A list of entry pairs comprising a column name and a value to which the column is to be set. Constant values, computations and subqueries are permitted in the value clause.

[Where Clause](#) The Where clause is optional, and if specified limits the rows to be updated through use of one or more filters.

### Results

The rows in the nominated table, optionally limited by a filter clause, are updated in accordance with a column / value list.

### Remarks

The updated rows are placed in a transaction frame, allowing conditional commit or rollback of the table.

### Examples

```
UPDATE fred SET ownership = 33.5 WHERE ID = 25;
```

### See also

[Table Manipulation](#)



## Value List Clause

The **Value List** clause provides for the specification of a list of values to be used in a single-row insert (see [Insert](#) for the encapsulating command syntax)

The general syntax for the value list is as follows :

```
(value_1, value_2, ..., value_n)
```

Each value may be numeric or string (depending on the corresponding column in the insert clause - see [Column List Clause - Insert](#)) and each value may use operators to arbitrary complexity.

### Remarks

Although it is good practice to match the data type of the value specification to the type of the column in the column list clause, this is not required. If the data type does not match (for example, a numeric value is specified for a string column) the SQL engine will perform the best possible conversion to comply with the column data type.

### Future enhancement

*In release 5.0 of the Lava SQL engine, subqueries will be permitted in any value position to derive the value from existing table data.*

### Examples

```
INSERT INTO fred
      (name, ownership, stock_count)
VALUES
      ('joe bloggs', 23.565, 433)
```

```
INSERT INTO fred
      (name, ownership, stock_count)
VALUES
      ('joe' || ' bloggs', ((3.575 + 1.1) ^ 0.12) * 24, 433 / 5)
```

### See also

[Insert](#), [Column List Clause - Insert](#), [Data Extraction and Manipulation](#)

## Where Clause

The **Where clause** is an optional clause used in [delete](#), [update](#) and [select](#) statements (with only one table) to limit the number of rows processed by the statement. If this clause is omitted, the statement processes all rows in the nominated table(s).

In the case of Select statements where the table list comprises more than one table, the **Where clause** is no longer optional. In this case, **join** conditions are required between all of the tables listed in the table list. This case is covered separately after the filter cases - see [Join conditions](#) below.

### Filter conditions

In its simplest form, the Where clause specifies two expressions and a comparison operator, as follows :

```
expr_1 comparison expr_2
```

The Where clause may list any number of conditions, separated by a boolean logical :

```
expr_1 comparison expr_2 AND
expr_3 comparison expr_4 OR
expr_5 comparison expr_6
```

In the above syntax, as the AND operator has higher precedence than the OR operator, the first two conditions will be evaluated first, the results will be ANDed together, and the final condition will be evaluated and ORed to yield the overall result

In order to force a particular logical evaluation, parentheses (arbitrarily nested) may be used :

```
expr_1 comparison expr_2 AND
(
  expr_3 comparison expr_4 OR
  expr_5 comparison expr_6
)
```

In the above case, the segment in parentheses will be evaluated first, then the result of the first condition will be ANDed with this, yielding the overall result.

[Subqueries](#) may be used in several ways within the where clause. The first is simply as a replacement for an expression :

```
expr_1 comparison (SELECT expr_2 FROM table_2 WHERE where_clause)
```

An example based on this form could be :

```
UPDATE fred
SET
  bValid = TRUE
WHERE
  stock_count = (
    SELECT
      SUM(stockcount)
    FROM
      stocklist
    WHERE
      owner_id = fred.ID
  )
```

Note the use of the correlated subquery (*owner\_id = fred.ID*) in the above example.

The second option for a subquery condition is the EXISTS condition :

```
EXISTS (SELECT expr_2 FROM table_2 WHERE where_clause)
```

In this form, the condition evaluates as TRUE if at least one row is returned by the subquery, and FALSE otherwise.

The Exists condition can also be inverted :

```
NOT EXISTS (SELECT expr_2 FROM table_2 WHERE where_clause)
```

In this form, the condition evaluates as TRUE if the subquery returns an empty result set, and FALSE otherwise.

Finally, the subquery may be specified as the operand of an IN condition :

```
expr IN (SELECT expr_2 FROM table_2 WHERE where_clause)
```

The IN condition is satisfied (returns TRUE) if the value resulting from the specified expression *expr* is found in the result set of the subquery. This subquery must return a result set comprising only one column (but may return any number of rows).

The IN condition can also be inverted :

```
expr NOT IN (SELECT expr_2 FROM table_2 WHERE where_clause)
```

in which case the condition returns TRUE if the value resulting from expression *expr* is **not** found in the result set of the subquery. As for the IN case, the subquery must return a result set comprising only one column (but may return any number of rows).

### Join conditions

If the Where clause forms part of a Select statement, and the Select references more than one table in the [Table list](#), mandatory join conditions are required for each of the tables referenced. This is to avoid the requirement for Cartesian product joins, which in the greater majority of cases yield a result set which is meaningless (and extremely large), and can take extremely long to execute.

If a multi-table select does not specify at least one join condition (to another table in the table list) for each table, the SQL engine will reject the statement and return an error.

In those cases where some form of Cartesian product is required, (and assuming that the result set will be manageable both in size and execution time) this may be achieved by specifying an 'open' join, which includes all the rows of the target table through an appropriately stated inequality (such as BETWEEN). Should this option be exercised, the onus is on the user to ensure that execution of the statement will yield a sensible result - the Lava SQL engine will allow the statement as a join is in place between the tables (although the join will not eliminate any of the Cartesian product rows).

In the general case, a join is phrased as follows (see the section on [Relational Integrity](#) for more information on standard Lava relations) :

```
table_1.ref_column = table_2.ID
```

As with filter conditions, multiple joins are typically connected using AND conditions, yielding the following form :

```
table_1.ref_column = table_2.ID AND
table_2.ref_column = table_3.ID
```

Note the **chaining** effect in the above form, where each table in the table list is successively joined to the next table in order to form a complete **join chain**.

In certain cases, relations between certain tables are non-mandatory - typically the link to a parent table from the child table is not required in all rows. An example would be an attribute in a child table for which the attribute “not applicable” is coded in terms of the absence of a link to the parent (attribute) table.

In this case, the join may be phrased as an **outerjoin**, or **OJ**. This is done as follows :

```
table_1.ref_column OJ = table_2.ID
```

Note that the outer join indicator (**OJ**) is specified for the child table, implying that the reference column (*ref\_column*) is allowed to be null, i.e. no link to the parent (attribute) table is specified.

The default join is **inner join**, which requires no specification.

### Where clause syntax

The full syntax of the Where clause may be found in the section [SQL Syntax Specification](#).

### Remarks

The complexity of the Where clause can contribute significantly to the execution cost of a SQL statement, especially if multiple subqueries are used. Although the Lava SQL engine attempts to optimise the execution of the statement as much as possible, subqueries are still executed individually and can extend execution considerably where large numbers of rows are processed.

The most expensive subquery conditions are **IN** and **NOT IN** comparisons - these should be avoided wherever possible; in the majority of cases (in fact, almost all) it is possible to re-phrase these as **EXISTS** and **NOT EXISTS** conditions, which evaluate considerably faster.

### Examples

A very simple Where clause is provided below :

```
SELECT * FROM system.sys_objects WHERE ID = 4
```

A more complex Where clause including join conditions and a correlated subquery follows :

```
SELECT
    ide_node.rowid, ide_node.nodename
FROM
    design.ide_node, design.ide_workspace
WHERE
    ide_node.nodetype_id = 1 AND
    ide_node.expandworkspace_id = ide_workspace.rowid AND
    ide_workspace.design_id = 1 AND
    EXISTS (
        SELECT
            ide_membernode.id
        FROM
            design.ide_membernode, design.ide_ws_2_node
        WHERE
            ide_workspace.rowid = ide_ws_2_node.workspace_id
            AND
            ide_ws_2_node.node_id = ide_membernode.rowid AND
            ide_membernode.nodetype_id = 2
    )
```

**See also**

[Select](#), [Update](#), [Delete](#), [SQL Syntax Specification](#), [Relational Integrity](#), [Data Extraction and Manipulation](#)

## SQL Syntax Specification

Note that in the following syntax specification, several elements and keywords specified are not yet supported. These elements are indicated through the use of surrounding marks (\*\*) - all keywords or syntactical elements listed in this way are a future provision, and the majority are planned for release in revision 5.0 of the Lava SQL engine.

```

Legend :
|           : Or (alternative)
{}          : Optional element. Also automatically designates a
group
" or '     : Literal character [sequence]
Uppercase lexeme : Literal lexeme (keyword)
!          : Comment (to end of line)
{* .. *}   : Provision for future enhancement

```

```

function ::= ABS | ARCCOS | ARCSIN | ARCTAN
          | COS | DEG | EXP | FORMAT | INT
          | LN | LOG | LOWER | RAD | ROUND
          | SIN | SLICE | STRINGPOS | SQRT
          | SOUNDEX | TAN | TRUNC | UPPER

```

```

aggregate ::= AVG | COUNT | MIN | MAX | SUM
operator  ::= + | - | * | / | MOD | DIV | ^
reservedexpr ::= PI | ROWID | DATE | TIME | VDT
columnident ::= {schemaident.}{tableident
                | tablealias}.columnlabel
                | columnalias
columnexpr ::= columnident | reservedexpr |
aggregate(columnexpr)
                | function(columnexpr {, parm})
                | expression { operator columnexpr }
column ::= columnexpr | subquery
columnlist ::= column {{AS} columnalias} {, columnlist}
columnspecilist ::= columnlabel columntype {, columnspecilist}
simplecolumnlist ::= columnlabel {, simplecolumnlist}
columnvaluespec ::= columnident = columnexpr | (query)
columnvaluelist ::= columnvaluespec {, columnvaluelist }
table ::= tableident | subquery
tablelist ::= table {{AS} tablealias} {, tablelist}
orderlist ::= table {, orderlist}

```

```

grouplist ::= columnident {, grouplist}

```

```

havinglist ::= filterlist

```

```

comparison ::= = | # | <> | > | < | >= | <= | LIKE
wildcard   ::= * | ?
wildcardstring ::= {char | wildcard} {wildcardstring}
expression ::= columnexpr | number | 'wildcardstring'
subquery   ::= ( query )
exprquery  ::= expression | subquery
exprlist   ::= expression {, exprlist}
exprlistset ::= ( exprlist ) | subquery
condition  ::= expression comparison exprquery
                | expression {NOT} IN exprlistset
                | expression {NOT} BETWEEN
                | expression AND expression

```

## SQL Syntax Specification

	EXISTS subquery
	expression {NOT} LIKE expression
filter	::= {NOT} condition
filterlist	::= filter { { AND   OR } filterlist }
accessmode	::= READONLY   READWRITE
alterschema	::= ALTER SCHEMA { [CLIENT]   [SERVER] } SET ACCESS accessmode
schemaspec	::= SCHEMA schemaident
altersession	::= schemaspec
	BACKUP FOLDER folderpath
tablemodspec	::= ADD columnspeclist   DROP simplecolumnlist
	MODIFY columnspeclist
	RENAME TO tableident
	schemaspec
altertable	::= ALTER TABLE tableident tablemodspec
alteruser	::= ALTER USER schemaspec   PASSWORD password
alter	::= alterschema   altersession   altertable   alteruser
backup	::= BACKUP schemaspec {KEY encryptkey}
commit	::= COMMIT {transactionident}
connect	::= CONNECT SERVER   SERVERIP serverident USER userident PASSWORD password
createschema	::= CREATE { [CLIENT]   [SERVER] } schemaspec
relationconstraint	::= CASCADE   RESTRICT   NONE
createrelation	::= CREATE RELATION FROM PARENT tableident TO tableident COLUMN columnident CONSTRAINT relationconstraint
createalias	::= CREATE ALIAS   SYNONYM tablealias FOR tableident
asselectspec	::= AS query
tableattribute	::= PHYSICAL   VIRTUAL   PERSISTENT   REPLICATE   RAW   RECLAIM   NORECLAIM   INITIALSIZE   RESERVEROWS (rowcount)   FRAMED   (* TIMEDOMAIN *)
	schemaspec
tableattributelist	::= tableattribute {tableattributelist}
fromodbc	::= FROM ODBC query
createtable	::= CREATE { [CLIENT]   [SERVER] } TABLE tableident (columnspeclist)   asselectspec   fromodbc   tableattributelist
createuser	::= CREATE USER userident PASSWORD password schemaspec
create	::= createschema   createrelation   (* createsequence *)   createalias   createtable   createuser   (* createview *)

## SQL Syntax Specification

delete	::=	DELETE { [SERVER] } tableident { WHERE filterlist }
disable	::=	DISABLE USER userident
disconnect	::=	DISCONNECT
dismount	::=	DISMOUNT
distribute	::=	DISTRIBUTE schemaspec   TABLE tableident
dropschema	::=	DROP schemaspec
droprelation	::=	DROP RELATION BETWEEN PARENT tableident AND tableident
dropalias	::=	DROP ALIAS   SYNONYM tablealias
droptable	::=	DROP { [SERVER] } tableident
dropuser	::=	DROP USER userident
drop	::=	dropschema   (* dropsequence *)   droprelation   dropalias   droptable   dropuser   (* dropview *)
enable	::=	ENABLE USER userident
privilege	::=	GRANT   REVOKE   CREATE   ALTER   DROP   TRUNCATE   DELETE   UPDATE   INSERT   SELECT   ALL
targetspec	::=	DATABASE   schemaspec   TABLE tableident
grant	::=	GRANT privilege ON targetspec TO USER userident
valueexpression	::=	value   value operator valueexpression   function(valueexpression)
value	::=	number   string   valueexpression
valuelist	::=	value {, valuelist}
valuestatement	::=	VALUES (valuelist)
insert	::=	INSERT INTO tableident (simplecolumnlist) valuestatement   query
renameschema	::=	RENAME SCHEMA schemaident TO newname
renamealias	::=	RENAME ALIAS   SYNONYM tablealias TO newname
renametable	::=	RENAME TABLE tableident TO newname
renameuser	::=	RENAME USER userident TO newname
rename	::=	renameschema   (* renamesequence *)   renamealias   renametable   renameuser   (* renameview *)
restore	::=	RESTORE BACKUP backupfile { KEY encryptkey }
revoke	::=	REVOKE privilege ON targetspec FROM USER userident
rollback	::=	ROLLBACK {transactionident}
savepoint	::=	SAVEPOINT transactionident
truncate	::=	TRUNCATE TABLE tableident

## SQL Syntax Specification

update	::=	UPDATE { [SEVER] } tableident SET columnvaluelist { WHERE filterlist }
transaction	::=	commit   rollback   savepoint
query	::=	SELECT columnlist FROM tablelist WHERE filterlist ORDER BY orderlist GROUP BY grouplist HAVING havinglist
modstatement	::=	delete   insert   update
connectstatement	::=	connect   disconnect   dismount   distribute
backupstatement	::=	backup   restore
adminstatement	::=	alter   create   disable   drop   enable   grant   rename   revoke   truncate
command	::=	adminstatement   backupstatement   connectstatement   modstatement   query   transaction