Lava Technical Reference

Table of Contents

List of Illustrations

Introduction

Manual Scope and Target Audience

This reference manual covers the mounting, connecting and programming of client applications which use a Lava Database as a storage mechanism. It does not cover the installation and administration of a Lava Database - for information on these topics consult the Lava Installation and Administration Guide.

This manual is targeted at programmers, power users and administrators who wish to :

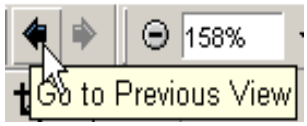
- Obtain detailed information on specific interfaces and mechanisms in the Lava Database
- Learn in-depth technical detail on the operation and design of the Lava Database kernel
- Code client application programs to interface to the Lava Database

Topics covered include :

- SQL usage
- Non-SQL interface using the Lava API
- Lava System schema layout
- Code examples

Using This Reference

This PDF document is extensively hyperlinked. In order to derive the best usage from this document, note that the Acrobat PDF reader has a “back” button (just like a browser) located in the toolbar, which looks like this :



After following a hyperlink (any blue underlined text) by left-clicking on the hyperlink with your mouse, clicking the “back” button will return you to the hyperlink just accessed. The “back” button will work through multiple levels of links.

The “back” option can also be found in the mouse right-click pop-up menu, named “Go to Previous View”.

Reference Manual Structure

The manual is divided into a number of sections, each of which addresses a specific topic.

Lava Features and Constraints

This section should be read by any prospective power user of a Lava Database. It introduces major features, requirements and constraints of the Lava Database, and explains important concepts which distinguish the Lava Database from other SQL databases. All users of Lava Databases who intend to interact with the database through any of the interface mechanisms should at least read the Key Concepts subsection.

Lava SQL Reference

The SQL reference should be consulted by administrators and programmers who intend to use SQL to query, control and update Lava data tables. It specifies the range and limitation of Lava SQL, and provides information on how to optimize and improve SQL statements in the context of the Lava SQL engine.

The Lava Access Privilege System

The Lava access privilege system allows control over user access to groups of tables (schemas) or individual tables, as well as limitation of certain other facilities in a Lava Database. This section describes the privilege mechanism and provides a comprehensive list of privileges available in the Lava Database.

The Lava API

This section covers usage of the Lava API to access low-level procedures for high speed and powerful interaction with the Lava Database. The topics covered are of an advanced nature, and will only be of interest to application designers and programmers, and to some advanced administrators who wish to acquire in-depth understanding of the operation of the Lava Database.

The Lava System Schemas

The system schemas provide information regarding the Lava management mechanisms and system tables which may be used to obtain information on database operation and user objects. This section will be of interest primarily to administrators and power users, but will also be consulted from time to time by application designers and programmers.

API Structures and Constants

The API structures are intended to be used for reference only, and are comprehensively cross-linked and hyperlinked from the Lava API section.

Appendix I : Lava Error Codes

The Lava error codes are presented for information and reference only, and may be consulted when errors are reported by Lava API procedures.

Appendix II : Source Code Examples

The code examples will be primarily of interest to programmers who wish to examine examples of working techniques to achieve specific results using the Lava API. Examples are provided in several programming languages and cover a range of Lava programming techniques.

Appendix III : SQL Examples

The SQL examples provide a number of working examples of SQL statements ranging from simple to advanced, illustrating usage and capabilities of Lava SQL. It will be of interest to especially administrators, but also to programmers and power users who intend to use SQL to interact with and control the Lava Database.

Appendix IV : ODBC Interface

This appendix documents the Lava ODBC driver which provides standard ODBC access to the server database. However, use of this interface is **strongly discouraged** as all the major advances in database technology presented in the Lava Database are rendered void through use of this interface, which still adheres to legacy interface techniques. Please consult the sections Lava SQL Reference and The Lava API for information on the native Lava interfaces, and in particular the section Key Concepts in the Lava Database and the reference for the command LavaCommand which provides direct SQL access into the Lava Database should be consulted to acquire information on the totally revised techniques used in the native interface to the Lava Database. The coded example SQL Execution and Data Extraction may also be consulted for insight into the native mechanisms provided.

Lava Features and Constraints

Database limits

All database objects (schemas, users, tables, indexes, synonyms, sequences...) are unlimited in number, except for limits on disk storage.

Total database size is limited only by disk storage on the server, but the client space is also limited in terms of active (in-use) tables, which are loaded in memory on the client workstation in order to enhance performance.

Table size is limited to $2x10^9$ bytes per table, except for variable data tables (used to store varstring data) which are limited only by available data storage.

Row size in all tables (with the exception of variable length columns, which are dealt with separately) is limited to a maximum of 1500 bytes. This excludes any variable length extensions to varstrings, but includes the varstring base length. Table rows are limited to a maximum of 500 columns.

A row may have any number of variable length columns, provided the sum of the base lengths of these columns (the portion stored within the originating row) does not exceed 1500 bytes less the other non-variable columns. The variable portion of these columns may be any length up to, but not exceeding, $2x10^9$ bytes.

Variable length columns may not be used as filter columns for any SQL or search operation.

Length of an index entry	255
Length of a raw device name	50
Number of partitions in a dbspace	127
Length of a user name	14
Number of log files	64
Number of indexes per table	20
Length of a database name	10
Length of a table name (the same applies to statement IDs, index names, domain names, constraint names, synonym names, view names, document names, column names and cursor names)	18
Number of VARCHAR columns per table	20
Length of a dbspace name	10
Length of a DBS name	10
Number of columns per index	16
Number of extents per data or index area	90
Length of a DECIMAL/MONEY type	32
Length of a CHAR type	32767
Length of a VARCHAR type	254
Length of a BLOB/TEXT type	2 GB
Length of a table record	32767
Number of columns per table	32767
Number of tables per database	unlimited
Number of databases per DBS	unlimited
Number of dbspaces	127
Number of locks per DBS	configurable
Number of buffers per DBS	configurable
Number of concurrently opened tables	configurable

Future enhancement

Number of concurrently active users configurable

Technical Support

Server Requirements
Windows NT, 2000 or XP, or Windows Server 2000 or 2003.

Client Requirements
Windows XP or Windows Server 2003.

Lava Kernel Releases

The following table lists only major Lava releases. Interim releases with minor upgrades and error corrections will be announced at irregular intervals as the requirement arises.

Lava Release	Planned Release Date
5.0	July 2005
5.1	November 2005
6.0	March - June 2006
6.1	Late 2006

The following table lists currently planned enhancements and the associated Lava release. Although this schedule is subject to change, every effort will be made to adhere to the planned release. Where deviations from this schedule are unavoidable, the revised schedule will be provided to all licenced database users by e-mail at the earliest possible time.

Feature	Planned release	Limitations
Standby server	5.0	
Views	5.0	
Distinct clause	5.0	
Time-Domain	5.0	
Database freeze for Backup	5.0	
Stored procedures and functions	5.1	
Integrity constraints on columns	5.1	
Nested schemas	5.1	
Triggers	5.1	
Internet / HTML support	5.1	

User Roles (access restriction)	5.1	
Undelete	5.1	
Sequences	5.1	
User password expiry	6.0	
Column-level access restrictions	6.0	
Set command	6.0	
Linux Lava Server release	6.1	

Future enhancement

Integrity constraints on columns

uniqueness and ranging

Stored procedures and functions

Triggers

Internet / HTML support

Time-Domain

Column-level access restrictions

disallow update of particular columns to nominated users

Views

Standby server

Nested schemas

Linux Lava Server release

ODBC Interface

The Lava Server database has a functional ODBC driver which provides standard ODBC access to the server database. However, use of this interface is **strongly discouraged** as all the major advances in database technology presented in the Lava Database are rendered void through use of this interface, which still adheres to legacy interface techniques. A reference to this interface is provided in the appendix ODBC Interface.

Key Concepts in the Lava Database

The following concepts are central to the operation of the Lava database. Although much of the design and implementation of the Lava database is similar or, in some cases, identical to other SQL server databases, there are a number of important differences. These concepts illustrate the differences and key issues important to understanding how a Lava database operates.

Array Access to Virtual Tables	Defining and using pointers to arrays of structures to access table data directly
Boolean Variables	Dedicated TRUE / FALSE variable type
Column buffer	Lava equivalent for a table index
Column Sequence	The sequence (index) of a column within a table
Control file	The single most important Lava system table
DataGrid	An instant table display - extended list control
Distribution	Table and data distribution to client databases
Distributed Client	Mechanism for operating a client database with transparent server communication
Foreign keys, relations and joins	Lava approach to inter-table relations and join syntax, including inner and outer joins
Index	Lava indexing through column buffers
Join methodology	Lava approach to inter-table relations and join syntax, including inner and outer joins
Julian Date	A numeric date format used throughout the Lava database
Mount mode	The mode in which the database is mounted - can be Server, Client or Exclusive.
Object / Object ID	The object concept and object types
Primary Key	The Primary Key of every standard table in a Lava Database is predefined as the row ID of the table
Raw Tables	Packed memory tables
Relations and joins	Lava approach to inter-table relations and join syntax, including inner and outer joins
Replicator Tables	Memory replicators for disk tables
Result set	Result tables for SQL queries
Return code (rc)	Error code philosophy, implementation and interpretation
Row ID /ID	Row identifiers : meaning and usage
Row status	Current and deleted rows, the deletion mechanism and row re-use

Schema	Grouping and isolation mechanism for sets of tables

H

Session / Session ID	Database sessions and their identification
SQL Command Execution	Logic behind execution of SQL commands in a Lava database
Stack Tables	A feature in the Lava database which allows tables to be created which behave like stacks, and offer extensible stack operations
Transaction frame	Transactions, commit and rollback, client and server implications
User	User accounts and constraints
VDT	The Version Date Time concept and usage
Virtual Tables	Memory data tables

Array Access to Virtual Tables

The Lava database kernel supports three primary table modes. The first, physical tables, is analogous to the storage method used in all other SQL server databases. The second, Virtual Tables, are unique to Lava. Virtual tables are defined strictly in memory, and the data is of course transitory. The third, Replicator Tables, are also unique to Lava - these are physical tables which are "shadowed" to memory for high-speed access.

For Virtual Tables defined by the user, it is possible to gain access to the table through a pointer to an array of structures, where the structure type coincides exactly with the format definition (column information) for the virtual table. This may be achieved in one of two ways - the first is through the procedure AssertTablePointer, which informs the Lava database kernel that the array pointer is to be maintained whenever the memory allocation for the virtual table changes. The second is during creation of an instance table through use of the CreateTableInstance procedure, at which time an array pointer may be specified.

When either of these techniques is used, the Lava database kernel maintains the pointer to the array whenever the memory allocation for the virtual table in question changes for whatever reason - for example, if multiple AddRow actions are performed, or a SQL "insert into as select" results in many rows being added to the virtual table, the current memory allocation may prove to be inadequate for the new row requirement of the table. In this case, the memory allocation will be enlarged, which, depending on current memory usage, may result in the memory allocation for the table being moved to a different area of memory. In this case, the Lava database kernel will automatically update the nominated array pointer to the new memory address for the virtual table.

Note that although it is permissible to read and write existing rows from and to the virtual table in question, the programmer should take care not to overstep the current row boundaries on the table - this will most likely result in a memory access violation. The onus is on the programmer to ensure at all times that the array indexes in use are limited to the number of actual rows in the table - this can easily be determined through use of the TableRows procedure.

When new rows are to be added to the virtual table, these MUST be added using the AddRow procedure - adding rows via array access will almost certainly end in an access violation, as the Lava database kernel is unaware of the addition of new rows to the table and cannot validate memory allocation.

If the table in question permits row deletion (see DeleteRow) which depends on whether a Row status was defined in the table format, the user should take care when using array access to read data from the table to validate that the row is current.

Example code
For a detailed example of array access to a virtual table, see the source code examples (Virtual table pointers, Instance tables) provided.

Back to Key Concepts

Boolean Variables

In all Pascal language dialects (including Modula 2 and Oberon) the boolean type is separated from conventional numeric variables, and has valid values of TRUE and FALSE only, represented by 1 and 0 numerically. The size of the variable is a **byte**.

Back to Key Concepts

Column buffer

All current SQL databases use indexes (formerly known as indexed-sequential access) to provide random access

(more correctly phrased keyed direct access) to table data within the database.

In contrast, the Lava Database does not use indexes for this purpose. Instead, a new concept in data access is employed, termed column buffers.

Column buffers are sorted memory-based binary-searchable reference arrays which allow ultra-fast access to any row of the table referenced by the column buffer.

Whereas index are disk based, slow to maintain and slow to construct, column buffers are very low cost to maintain and fast to construct. Indexes across two or more columns are needed for complex multi-column filters and joins to the tables represented, whereas column buffers always represent only one column, as range buffers are very fast to construct for special multi-column filter purposes and allow much faster access with much more flexible application than comparable indexes.

Back to Key Concepts

Column Sequence

Similarly to the concept of an object ID referencing tables through a unique numeric reference identifier, column sequences are a numeric reference identifier used throughout the Lava API to uniquely reference a column within a table.

A column sequence is a 1-based index into the columns for a table (i.e. the first column in the table is declared to be column sequence 1) so that a pair of numbers, the object ID and the column sequence, uniquely identify every column in the entire database.

Back to Key Concepts

Control file

The Lava Control File (System_ControlFile, located in the System schema) is very useful to any advanced SQL user, as it presents a single reference providing object (table) identification, many important object attributes, and information ranging from deleted row counts to row size.

Back to Key Concepts

DataGrid

Back to Key Concepts

Distribution

Back to Key Concepts

Distributed Client

Back to Key Concepts

Foreign Key

See Primary Key, row ID and relational integrity for information on foreign keys.

Back to Key Concepts

Index

Back to Key Concepts

Relations, foreign keys and inter-table joins

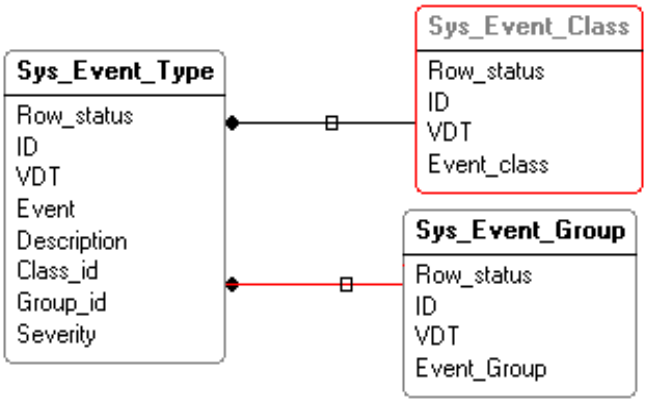
For details related to SQL syntax, see SQL Join Syntax
For details on the relational integrity mechanism in the Lava Database, see Relational Integrity

The Lava approach to joins and inter-table relations is different from the general relational database approach to this topic. Starting from the conception of the Lava database kernel, a new and more rigid approach (explained in detail below) was applied to relations, for two major reasons :

- The first is relational integrity. In order to avoid the problem of relational key columns changing during the database lifecycle, the entire concept of related columns was revised.
- The second was performance. Using the Lava approach, the database becomes far more efficient and significantly faster.
- The third was the issue of relational design. Using the Lava approach to relations, it is more difficult to build inappropriate relations into the database design which fundamentally violate normalization rules.

In order to implement these design goals, a simple yet sufficient and effective strategy for inter-table relations was selected. The mechanism allows only one method for relating two tables, namely a join column in the *many* (child) table which unconditionally links to the row ID in the *one* (parent) table, which is automatically the primary key in the parent table. An example would be :

Consider the three tables in the illustration *Simple join example*. There are two related tables; Sys_Event_Class has a 1:M (one to many) relation with Sys_Event_Type, as does Sys_Event_Group. Each of these is achieved by placing a join column (Class_id and Group_id respectively) in the Sys_Event_Type table. Each of these columns is required to be a longint (dword). These join columns contain, in all cases, the value of a row ID in the related table. For entries in the Sys_Event_Type table where no join entry exists (or is yet linked) the join column is null (0).



Simple join example

Several questions should immediately occur to you in connection with this method. These are answered superficially below, and the more complex ones are treated more comprehensively elsewhere in this reference.

Q. Can I relate more than one source column to single target table?

A. In general the answer is no, if the columns are with respect to a single specific join. As the content of the join column is a row ID in the target table, and this is always unique, having more than one column for a single join is redundant as each would contain the same value. (See below for an example of a

two-column join and the equivalent Lava implementation).

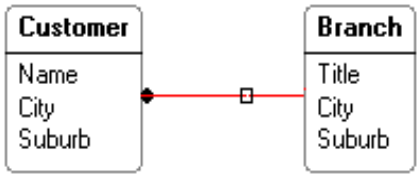
However, if more than one join is in question (for example, an exchange rate table linking both to the source currency and the target currency, source and target currency both being entries in the same currency table) then one should define two columns (say *SourceCurrency_id* and *TargetCurrency_id*) in the rate table, each relating to a specific and separate join to the currency table.

Q. How do I implement *many to many* relations?

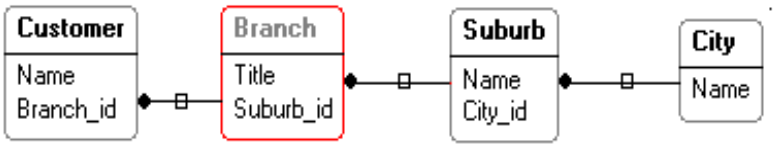
A. By using a join table. If table A has an M:N relation with table B, this is achieved by specifying a join column in table A (say A_J_id) which has a *many to one* relation with a separately defined join table (say J), and similarly table B contains a column (say B_J_id) which also has a *many to one* relation with table J. This successfully implements the M:N relation. Any attributes which belong to the join are now added as columns to the join table, J.

Q. What do I do in cases where my join was formed between two (or more) columns in the source table and two (or more) columns in the target table? (For the case where this implements a M:N join, see the previous example).

A. The easiest way to answer this is by way of an example. Consider the diagram provided, *Two-column join*. Clearly, in this case, both join columns are essential. However, what this join is in fact illustrating is a normalization error. The corrected design is provided below in the diagram *Normalized join*.



Two-column join



Normalized join

In the properly normalized design, not only do we obtain a significant performance enhancement through a forced optimization of the join between *Customer* and *Branch*, but we also are forced to normalize the *Suburb* and *City* tables correctly, which means that any attribute columns to the original non-normalized *Branch* table (in the diagram *Two-column join*) will now be correctly attributed where they belong - in the non-normalized example, unavoidable duplication of attribute values belonging to the *City* table would occur. At the very least, even if a *City* and / or *Suburb* table were implemented in the non-normalized case, duplication of the City name itself would occur, possibly creating erroneous mis-spelt city entries. In addition, maintenance of the original *Branch* table would rapidly become a nightmare, as there would be any number of City columns to correct if an error or change were to have to be made to the data (for example, if a city changes its name - uncommon, but it does happen; consider Leningrad becoming St Petersbug). In the normalized example, no duplication, no redundancy and no maintenance problem can arise.

Q. What about the case where a join absolutely requires more than one column specified between two tables in order to filter or limit the responses?

A. No problem. The above issues relate to database design - in design-relational terms only one join on the row ID of the foreign table is permitted, but in SQL you may use any number of join columns for filtering purposes. In fact, it is not even a requirement that a relation has been formally registered in the database in order to be able to perform a join between the tables in SQL - you may specify any number of any columns of any type in a SQL join between two tables.

Q. Can I do SQL joins on string columns?
A. Absolutely. In the SQL engine, a join is merely a match between the values of two columns in joining tables. These joining columns may have any type except boolean (boolean joins would be too expensive - by definition there are only two boolean values, 0 and 1, therefore a boolean join is half as bad as no join at all) and the SQL engine will simply match the values in the columns to arrive at the join. The only preclusion with string columns is in terms of Lava-maintained relations; these can only be between numeric columns where one is the row ID of the parent table. This having been said, string joins are not recommended as they are slower to process (and less definitive) than numeric joins, but if strings are all you have in your existing data, you can certainly use them.

Relational Integrity

In terms of the method implemented in Lava for inter-table relations, the issue of relational integrity is simplified quite significantly. As it is quite impossible for the join column to "change" (in terms of a renaming or alteration of spelling), there is no **update cascade** integrity (on modification) - if the join column is updated, this is merely a selection of a new related entry in the join table, and no cascaded update is required.

The provided integrity facilities relate strictly and only to deletion of linked entries in the related tables, and therefore are limited to **delete restrict** and **delete cascade**. The first will disallow a deletion of the parent entry if related entries in the child table exist, and the second will delete any related child entries if the parent entry is deleted.

SQL Join Syntax

With due respect to the efforts exerted by the ISO group on SQL syntax, we have decided to stay with the join mechanism conventionally used prior to the announcement of SQL-92. An example is provided below :

```
select
   count(so.id),
   schema_name
from
   sys_objects so, sys_schemas sc
where
   sc.id = so.schema_id and
   so.id < 70
group by
   schema_name
```

The join in the example, `sc.id = so.schema_id`, is stated - as was conventionally the case - in the *where* clause of the select.. As can be seen from the example, filters are also stated in the *where* clause.

The syntax for outer joins is simply an addition of an appended keyword, OJ or OUTERJOIN (both are recognized, and are case insensitive) to that end of the join which may require allowance of a missing join entry. For example, in the case above, the sensible outer join would be `sc.id = so.schema_id oj`, which would include any objects not belonging to (or unlinked to) a schema.

Back to Key Concepts

Mount mode

Client, Server or Exclusive

Back to Key Concepts

Object /Object ID

In order to present a uniform interface to many of the Lava API interfaces which deal with different forms of entities in the Lava database, and to have a consistent way to address various elements within the database, many lava entities are addressed through an object entry in the Sys_Objects system table. These entries can represent tables of various kinds, indexes, stored procedures and functions, and in future releases other object types will be enabled and supported.

In API terms, the unambiguous identifier for an object is the object ID. This is the row ID of the entry for the object in the Sys_Objects table.
Thus, instead of many interfaces which process table requests receiving a table ID, the object ID is used instead. This allows easier and more consistent interpretation of the identifier for various interfaces and various object types, as a single point of departure is used throughout.

Back to Key Concepts

Primary Key

The Primary Key for every non-Raw table in a Lava database is its row ID, which is a mandatory column and is system maintained. See row ID for further information. See also relational integrity as defined and implemented in a Lava database for information on primary and foreign keys and relational joins.

Back to Key Concepts

Raw Tables

Back to Key Concepts

Replicator Tables

Back to Key Concepts

Result set

Back to Key Concepts

Return code (rc)

Throughout the Lava system, return codes are standardized and have a uniform implication wherever encountered. In all cases, a return code of 0 (zero) implies no error, i.e. successful completion.

A return code greater than 0 is an event ID, that is the row ID of an event entry in the system event log (Sys_EventLog). This entry in the event log will specify the error (or potentially the series of errors) which occurred during execution of the request.

A return code less than 0 is an error constant, as specified in the appendix Lava Error Codes

Back to Key Concepts

Row ID /ID

The Lava Database kernel is designed around a number of core table mechanisms, of which one of the most important is the concept of a unique row ID for each row in each table.

The row ID is the ordinal of the row in the table, with the first valid row being row one. Thus, the row ID is simply a numbering of rows in the table, counting all rows including any deleted rows (see Raw tables for information on non-deletable rows in packed tables).

In theory, since the row ID is identified by the database as part of the fundamental row access mechanism, it is not even necessary for the row ID to be represented in the table data. However, the moment the row data is removed from the table (through a GetRow, for example) into a local buffer, the row cannot be identified unless the row ID is stored within the column information.
For this reason, the first two columns of every standard, non-raw table are defined as being the row status and the row ID, as follows :

Column Sequence	Column Name	Column Type	Generic Type
1	Row_status	RowStatus	Byte
2	ID	RowID	longint (dword)

The ID field as depicted above is system maintained (i.e. on adding a new row to a table, either through a SQL insert command or through AddRow, the system asserts the correct value in the ID column).

In addition, the ID column is defined as the automatic and only Primary key for every table. All relations managed by the Lava Database with this table as the parent table, are defined from the ID column of the table as the primary key of the relation. See Relational Integrity for further information on this topic.

Back to Key Concepts

Row status

The Lava database deviates from conventional database practice in the implementation of a user-visible row status column (for tables which permit conventional deletion - see also Raw Tables for further information). The row status (a single byte-wide column, which is always the first column in the table) indicates the status of the row, including unused, current, deleted, locked, and owned. Amongst other advantages, the row status permits undeletion (recovery) of deleted rows provided the row has not yet been re-used. In addition, the row status is critical to the Lava implementation of distributed client operation, which allows the database to maintain the most current row information even on rows which are still within a transaction frame (uncommitted) on the originating client database.

Back to Key Concepts

Schema

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.

Back to Key Concepts

Session / Session ID

In order to access a Lava database, the user must open a valid session. This session defines the nature of the connection to the server (exclusive, client) as well as defining the user account being used to access the database. This, in turn, will define the access privileges accorded the session.

When a valid session has been created (see OpenSession), an entry in the Sys_Sessions table is created for the session which defines the session attributes. Almost all interaction with the database requires specification of a valid session entry; this is achieved by providing the session ID, which is the row ID (or ID) of the session entry in Sys_Sessions.

Back to Key Concepts

SQL Command Execution

client and server execution; mount mode, LavaCommand, virtual table access through Array Access to Virtual Tables, example code

Back to Key Concepts

Stack Tables

see also Lava Stack Tables for information on the Lava stack API.

Back to Key Concepts

Transaction frame

Back to Key Concepts

User

Back to Key Concepts

VDT

The acronym VDT expands to Version Date Time, and is used as the uniform Lava date-time stamp whenever chronology is logged in the Lava database.

The current VDT as defined by the Lava Server may be obtained using the GetServerDateTime procedure.

The VDT is stored in a longreal (8 byte float) in which the integer portion represents a Lava Julian date, and the fractional portion represents the time of day logged to the millisecond.

A Julian date is a numerically coded date, from a (normally arbitrary) starting date, which varies from application to application. In the case of the Lava Julian date, day 1 is January 1st, 300 CE.

The VDT can be decoded through use of several Lava procedures; such as Format_VDT (which returns a string with the date and time coded in text format), and GregorianDate (which returns individual date fields from the integer portion of a VDT). The time fields of a VDT can be decoded using the Extract_VDT_Time procedure

Back to Key Concepts

Virtual Tables

Back to Key Concepts

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.

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.

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	
4	4H	quad integer (8-byte) - limited support	QUADINTEGER	
51	33H	Date	DATE	
67	43H	Time	TIME	
55	37H	VDT (DateTime)	VDT	
7	7H	float (8-byte)	FLOAT	
9	9H	character (1 byte)	CHAR	

10	0AH	string (fixed length)	STRING	
11	0BH	varstring (fixed length base with variable length extension to $2x10^9$)	VARSTRING	
28	1BH	packed varstring	PACKEDVARSTRING	
43	2BH	varunicode	VARUNICODE	
13	0DH	varbyte	VARBYTE	
29	1DH	packed varbyte	PACKEDVARBYTE	
100	64H	Structure (Single-depth record comprised of basic data types)		
-	-	Array (1-dimensional array of any fixed length type, including structures)		

Variable length types

varstring
varunicode
varbyte

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
    random
    concatenate (||)

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 into account 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

Select statement : join specification
Referential integrity constraints
The only major departure from this standard is in terms of the join syntax. For the rest,

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

Create Database	
Mount	
Dismount	
Backup	
Restore	
Connect	
Disconnect	

Schema Manipulation

Create	
Drop	
Rename	
Alter	
Distribute	

Restore	
Backup	

User Manipulation

Create	
Drop	
Rename	

H

Grant	
Revoke	
Alter user(password schema)	
Alter session	
Connect	
Disconnect	

Table Manipulation

Create Table	
Drop	
Alter	
Distribute	
Create alias	
Drop alias	
Truncate	
Create Relation	
Drop Relation	

Data Extraction and Manipulation

Select	

Update	
Delete	
Insert	
Subqueries	

Transaction Statements

Commit	
Rollback	
Savepoint	

Miscellaneous Statements and Clauses

System pseudo-table	
Column list - Select Clause	
Group by Clause	
Order by Clause	
Table List Clause	
Where Clause	

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	
Create view	
Drop view	
Rename view	
Undelete	
Create sequence	
Drop sequence	
Set	

SQL Command Categories

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

schemaname	The name of the schema to be altered.	
accessmode	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*

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

tablename	In all versions of this command, the *tablename* 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.
Column list - spec	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 Column List - Specification for details.
Column list - drop	The column drop list allows nomination of a list of columns to be removed from the table. See Column List - Drop for details.
newtablename	The new table name for the nominated table.
newschema	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 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

newschema	The name of an existing schema which becomes the current schema for the session.
newbackupfolder	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 dismount.

Qualifiers and Parameters

username	The username for the account to be modified
newschema	The new default schema for the user account
newpassword	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;
```

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 must be distributed if the backup command is executed on a client - this is unnecessary if the mount mode is Exclusive.

A backup folder for the session must have been asserted - see Alter Session.

Variants
None. The backup is always performed from the client, although the data in the backup is server data as the schema must be distributed for the backup to proceed.

Qualifiers and Parameters
schemaname The schema on which the backup is to be performed.

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 ***commit****s 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;
```

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 he **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
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

servername	The network server name of a server which can be addressed by name on a local Windows network.
username	The name of a valid user account.
password	The password for the user account.
serverIPaddress	The IP address (in conventional *a.b.c.d* 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 Database

The **Create Database** command creates a new Lava database.

```
CREATE DATABASE FOLDER databasefolder
```

Prerequisites
There should be no existing database at the nominated folder.

Variants
None.

Qualifiers and Parameters
databasefolder A full filepath specifying the location of the new Lava Database.

Results
A new Lava Database, containing no user tables or user accounts.

Remarks
The Create Database command is not intended for client databases. These are created automatically when a connect to a server database is performed.

The system account for the new database is SYSTEM with password MANAGER. The administrator should change the password to this account before placing the database in use.

The default database path is set to the specified *databasefolder*. Subsequent commands which require a database path, such as CONNECT EXCLUSIVE, will refer to the database at the folder specified.

Examples

```
CREATE DATABASE FILEPATH s:\lava\primary;
CONNECT EXCLUSIVE USER system PASSWORD manager;
ALTER USER system PASSWORD newsystempassword;
CREATE USER fred
DISMOUNT;
```

See also
Database 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 dismount 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. See Lava Kernel Releases for the planned release schedule.

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 purposes.

```
CREATE RELATION FROM PARENT mastertable TO childtable
      COLUMN master_id CONSTRAINT relationconstraint
```

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

mastertable	The parent table in the relation (the *one* table in a *one : many* relationship)	
childtable	The child table in the relation (the *many* table in a *one : many* relationship)	
master_id	The column in the child table which stores the row ID of the master table to which a row entry in the child table is related.	
relationconstraint	The constraint to be applied to the relation :	
	CASCADE	If an entry in the parent table is deleted which has related child entries, all related child entries are deleted as well.
	RESTRICT	If an attempt is made to delete an entry in a parent table which has related child entries, the deletion fails.
	NONE	The relation does not restrict or cascade deletions performed on the parent table.

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

Remarks
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
      COLUMN Event_Type_id CONSTRAINT RESTRICT
```

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

sequencename	*The name of the sequence to be created. This must be unique within the current schema.*
startvalue	*The starting value of the sequence. This is the first value returned by the sequence. It may be any numeric value, including fractional values.*
endvalue	*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.*
incrementval	*The increment amount. This may be negative or fractional.*
termqualifier	*The method used to terminate the sequence. This method is applied when the maxvalue is reached.*
	TERMINATE — *The sequence expires on reaching the maxvalue. An attempt to use the sequence after this fails.*
	CYCLE — *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.

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. See Lava Kernel Releases for the planned release schedule.

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

tablealias	A new alias for the nominated table.
tablename	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
```

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

tableattributes 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 **Create Table** 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 Virtual Tables 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.

REPLICATOR — The table is a physical, replicated table (specification of PHYSICAL is redundant if the REPLICATOR qualification is specified). Replicator tables can only be created on the server. See Replicator Tables for futher 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 row status and row ID 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.
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.

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.

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.

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.

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 (
                                    Row_status  RowStatus;
                                    ID          RowID;
                                    Name        STRING(50);
                                    Ownership   LONGREAL;
                            ) 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 (
                    Row_status  RowStatus;
                    ID          RowID;
                    Name        STRING(50);
                    Ownership   LONGREAL;
                    ) 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 (
                          Row_status  RowStatus;
                          ID          RowID;
                          Name        STRING(50);
                          Ownership   LONGREAL;
                          ) REPLICATOR NORECLAIM RESERVEROWS(200);
```

Create a virtual raw table on the client (note the absence of Row_status and ID columns) :

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

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

username	The new user account name.
password	The initial password for the user account.
schema	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 fredspassword 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 *whereclause*

Prerequisites
The session must have DELETE privilege on the table.

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

Qualifiers and Parameters

tablename	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.
whereclause	The filter clause (optional) which limits the rows deleted in terms of one or more filter specifications. See the syntax for the where clause 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

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.

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

schemaname	The name of the schema to be distributed.
tablename	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

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. See Lava Kernel Releases for the planned release schedule.

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

parentname	The name of the parent table for the relation
childname	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
None. Since the table name must be unique within a schema, the table is identified in terms of origin from its definition and is dropped either on the client or the server database depending on point of definition and distribution status.

Qualifiers and Parameters

tablename	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 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

privilege 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 Lava Privileges.

targettype The target of the privilege may be :

- 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.

targetname The name of the target entity. If the target type was SCHEMA, the *targetname* must specify a valid schema on the server. If the target type was TABLE, the *targetname* 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.

username 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.

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 fred;
REVOKE DROP ON SCHEMA fredschema FROM fred;
REVOKE UPDATE ON TABLE products FROM 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. See Lava Kernel Releases for the planned release schedule.

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

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

tablename	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.
columnlist	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 *Column List*.
valuelist	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 Value List Clause. In this form of the command, only one row is inserted into the nominated table.
selectstatement	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 *Select Statement*. 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.

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.

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 unused row, i.e. the next 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

Mount

The **Mount** command mounts an existing Lava Database.

```
MOUNT mountmode FOLDER databasefolder;
```

Prerequisites
A valid, current Lava Database must exist at the folder specified.

The nominated database may not currently be mounted.

Variants
None.

Qualifiers and Parameters

mountmode	The only currently supported mount mode is EXCLUSIVE. See Mount modes for further information.
databasefolder	A full filepath specifying the location of an existing, unmounted Lava Database.

Results
The database at the nominated folder is mounted.

Remarks
In order to operate in exclusive mode, which allows restore of backups, the server must be dismounted and mounted in exclusive mode. See Restore for information on the restoration of backups.

On conclusion of the restore operation, the database should be dismounted, after which the Lava Server can be re-mounted and normal operation resumed.

Future enhancement
The ability to mount a database in StandbyServer mode, to be included by release 5.0 of the Lava kernel, will provide for a hot standby database to be linked to the Lava Server in order to shadow every transaction performed to the server. See Lava Kernel Releases for the planned release schedule.

Examples

```
MOUNT EXCLUSIVE FOLDER s:\lava\primary;
```

See also
Database 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).

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 a near- in place algorithm 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

schemaname	The name of an existing schema.
newname	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

sequencename	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.
newname	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

aliasname	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.
newname	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

username	The name of an existing user account.
newname	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. See Lava Kernel Releases for the planned release schedule.

Examples
None - future provision

See also
Table Manipulation

Restore

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

```
RESTORE BACKUP backupfile
```

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 backup is always performed to an Exclusive-mount database.

Qualifiers and Parameters

backupfile	The backup file to be restored. This must be located in the current backup folder - see Alter Session for information on setting the current backup folder.

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;
```

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 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

privilege 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 Lava Privileges.

targettype The target of the privilege may be :

- 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.

targetname The name of the target entity. If the target type was SCHEMA, the *targetname* must specify a valid schema on the server. If the target type was TABLE, the *targetname* 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.

username 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 fred;
REVOKE DROP ON SCHEMA fredschema FROM fred;
REVOKE UPDATE ON TABLE products FROM fred;
```

See also
Lava Privileges, User Manipulation

Rollback

The **Rollback** command performs a rollback action on open transaction frames for the current session.

```
ROLLBACK
ROLLBACK subframe
```

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

Variants
None. The **Rollback** command always acts on the current session.

Qualifiers and Parameters

subframe	The name of an existing transaction subframe - see Savepoint

Results
The transaction frame is partially rolled back if a *subframe* is specified, and fully rolled back 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 rolled back is equivalent to specifying rollback without a *subframe*.

If a *subframe* is specified which is partway through the current transaction frame, the rollback does a partial rollback from that savepoint onward - effectively what this does is to remove the subframe from the currently pending transaction frame.

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;
```

In the above sequence, the transactions that will be committed to the table are :
1. The initial delete (for ID = 5)
2. The final update (for ID = 7)

The second delete is rolled back, and is not committed to the table.

See also
Commit, Transaction Statements

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 and Create Table statements to substitute for coded value or column lists. 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) 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.

```
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 cannot 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.

Column list	A list of columns from any of the tables in the table list. Computations, aggregate functions and subqueries are permitted.
Table list	A list of tables to be queried. Tables constructed from subqueries are permitted.
Where Clause	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.
Order by Clause	An optional clause providing for single- or multi-column sorts on the result set.
Group by Clause	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 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 phase 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 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

```
SELECT expression FROM Lava
```

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
```

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

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. See Lava Kernel Releases for the planned release schedule.

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 **outer join**, 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 *italics* - all keywords listed in this way are a future provision, and the majority are planned for release in revision 5.0 of the Lava SQL engine.

```
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]

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
                   |  EXISTS subquery
                   |  expression [NOT] LIKE expression
filter         ::= [NOT] condition
filterlist     ::= filter [ { AND | OR } filterlist]

query          ::=   SELECT
                        columnlist
                     FROM
                        tablelist
                     WHERE
                        filterlist
                     ORDER BY
                        orderlist
```

```
GROUP BY
   grouplist
HAVING
   havinglist
```

The Lava Access Privilege System

Lava Privileges

The following table is a comprehensive list of privileges available in the Lava Database.

GRANT REVOKE
CREATE
ALTER
DROP
TRUNCATE
DELETE
UPDATE INSERT
SELECT

SelectRow	00001H
InsertRow	00002H
UpdateRow	00004H
DeleteRow	00008H
TruncateTable	00010H
DropObject	00020H
CreateObject	00040H
AlterObject	00080H
BackupObject	00100H
RestoreObject	00200H

SelectRow through RestoreObject inclusive for schema; equate as above but object_id = 0

AlterSchema
DropSchema
CreateSchema

AlterDatabase
DropDatabase
CreateDatabase

Revoke
Grant

CREATE (table) = CreateObject + DROP
DROP (table) = DropObject + TRUNCATE
TRUNCATE = TruncateTable + DELETE
DELETE (table) = DeleteRow + UPDATE + INSERT
UPDATE = UpdateRow + SELECT
INSERT = InsertRow + SELECT
SELECT {table} = SelectRow

The Lava API

The following interface documentation is divided into a set of categories, commencing with indispensable and mandatory interface calls, through important calls which will be used by the majority applications, to more exotic and seldom used categories of calls which will seldom be used by the majority of programmers. In order to provide the most usable documentation of the total interface, some calls are referenced from more than one category in order to allow logical sets of calls to be as complete as possible. In this case, the primary documentation for the call (typically in the earliest category documenting the call) is referenced to avoid duplication of technical detail.

API Categories

Distributed Client Operation	Operating and manipulating a distributed client database
Lava Backup System	Low-level backup procedures
Lava Compression	Compression, decompression and encryption
Lava DataGrid Control	The API for the Lava DataGrid control
Lava Editor Control	The API for the Lava built-in text editor control
Lava Entry ID Functions	Identifying Lava entities
Lava Private Memory Management	Controlled memory management procedures
Lava Raw Table Interface	The Raw table interface procedures
Lava Replicator Table Functions	Functions related to replicator tables
Lava Row-level Table Interface	Interfaces to individual rows and columns in data tables
Lava Schema Manipulation	Creating and dropping user schemas
Lava Stack Tables	Special-purpose stack tables and stack operations
Lava Table Search Functions	Single and multi-column search facilities on data tables
Lava Table Manipulation	Manipulating, creating and dropping Lava tables
Lava Thread Support	Controlled threading interface
Mandatory Interfaces	Interfaces essential to interacting with Lava databases
Miscellaneous Interfaces	Utility procedures provided by the Lava Runtime Library
SQL Interface	Interface to the SQL engine
Transaction Frames	Operation of transaction frames in Lava
User Manipulation	Procedures to create and manipulate user accounts

Mandatory Interfaces

This category of calls includes all procedures which no client application to the database can function without.

CloseSession	Close the specified session
CreateDatabase	Create a client or server database
Dismount	Dismount the database
Mount	Mount an existing database
OpenSession	Open a new database session

See also
API Categories

Dismount

The Dismount procedure dismounts the database. The database must be in a mounted state in order to be operable.

```
PROCEDURE [PASCAL] Dismount (       pSession_id    : LONGINT;
                                    pShutdownMode  : LONGINT
                                ) :   LONGINT;
```

Parameters

pSession_id	:	The current session ID.
pShutdownMode	:	The shutdown mode (Normal, Immediate, Abort).

Return values

A standard rc (return code).

Remarks

The database must be successfully mounted in order for the dismount to execute.

Any open transaction frames are rolled back.

If the current mount mode is client, any connections to the server are terminated.

For a Normal shutdown, the dismount pends on any running critical threads (primarily those threads which control communication to the server) before shutting down the database. For an Abort shutdown, the database shutdown is performed immediately regardless of active threads.

See also

Mount, OpenSession, CloseSession, CreateDatabase, Mandatory Interfaces

Mount

The Mount procedure mounts an existing database. It is an absolute requirement to mount the database in the required mode in order to use any database facilities.

```
PROCEDURE [PASCAL] Mount (          pPath        : ARRAY OF CHAR;
                                    pStartupMode : LONGINT
                         )  :   LONGINT;
```

Parameters

pPath	:	The full base path of the database to be mounted.
pStartupMode	:	Required mount mode (Client, Exclusive, Server, StandbyServer).

Return values :
A standard rc (return code).

Remarks

The database must exist and be unmounted for the mount procedure to succeed.

The only valid mount mode for a conventional client application is Exclusive - Client mode is invoked through CreateDatabase, and Server and StandbyServer modes are used only in database server applications.

The mount will only succeed if all necessary system tables both exist and are completely valid to the correct revision of the database - this also implies correct relational integrity with all other system tables.

See also
Dismount, OpenSession, CloseSession, CreateDatabase

CreateDatabase

The CreateDatabase procedure creates a new, empty database or creates and mounts a (temporary) client database when connecting to a server. For client operation, CreateDatabase is a mandatory requirement.

```
PROCEDURE  [PASCAL] CreateDatabase (      pPath        : ARRAY OF CHAR;
                                          pStartupMode : LONGINT
                                   )  :   LONGINT;
```

Parameters

pPath	:	The full base path of the database to be created.
pStartupMode	:	Required create mode (Client, Exclusive, Server, StandbyServer).

Return values
A standard rc (return code).

Remarks
Any existing database at the nominated base path will be cleared. If successful, the procedure creates an empty new database.

In both Client and Exclusive mode, the CreateDatabase procedure both creates and mounts the database. In Client mode, however, the database is temporary and cannot be re-mounted after the client application has dismounted the database.

Server and StandbyServer mode are treated as for exclusive mode.

See also
Dismount, Mount, OpenSession, CloseSession

OpenSession

The OpenSession procedure creates a new session on a lava database. The database must have been successfully mounted (or created, for client operation).

```
PROCEDURE [PASCAL] OpenSession(     pUser                :  ARRAY OF CHAR;
                                    pPassword            :  ARRAY OF CHAR;
                                    pServer              :  ARRAY OF CHAR;
                                    pClientIP            :  LONGINT;
                                    pServerIP            :  LONGINT;
                                    pMasterSession_id    :  LONGINT
                                ) :  LONGINT;
```

Parameters

```
      pUser                :  The username for the session
      pPassword            :  The password for the nominated user
      pServer              :  The server name for client-server sessions
      pClientIP            :  The IP address of the client - used only by
                              the server
      pServerIP            :  The server IP - used if the server cannot be
                              addressed by name
      pMasterSession_id    :  The Session ID of the master session, if this
                              is a slave session
```

Return values

The procedure returns the new session ID. If the request fails, the return is 0.

Remarks

The nominated user account and password must be valid on the target database - if the mount mode is Client, the user account must exist on the nominated Lava Server. If the mount mode is Exclusive, the user account must exist on the local database.

The user account may not be disabled - in the disabled state, the only valid operation on the account is the EnableUser procedure.

The pClientIP parameter is not used in application programs - this is strictly for use by the Lava Server.

The server (in the case of Client mount) may be specified in one of two ways : the first, in the pServer parameter, is by server name using named pipes. The second, in the pServerIP parameter, makes use of a regular TCP/IP address to define the server. If the pServerIP parameter is non-null, this takes precedence over the name parameter.

Under special conditions, a slave session to a given current session may be required in order to divide updates or queries, or to isolate certain processing segments for design reasons. For this purpose, the pMasterSession_id parameter may be used - if this parameter is non-null, it must be a valid current session to the required lava database. The pUser and pPassword parameters must be completed to authenticate the connection request, but all other parameters are left null (0). The connection is established to the same database as the master session, but forms an autonomous session with autonomous transaction frames if updates are performed. Slave sessions are closed as for normal sessions, using the CloseSession procedure.

See also

CloseSession, Mount, Dismount, CreateDatabase, DisableUser, EnableUser

CloseSession

The CloseSession procedure closes an existing session on a lava database.

```
PROCEDURE [PASCAL] CloseSession(  pSession_id :  LONGINT )  :  LONGINT;
```

Parameters

pSession_id	:	An existing, current session ID.

Return values
A standard rc (return code).

Remarks
The session is closed, and becomes invalid for future use.
Any open transaction frames are rolled back.

See also
OpenSession, Mount, Dismount, CreateDatabase

User Manipulation

The commands in this section are used to set up and adjust user accounts. In Exclusive mode, the user accounts on the exclusive database itself are adjusted, whereas in Client mode user accounts on the server database are adjusted.

CreateUser
DisableUser
DropUser
EnableUser

See also
API Categories

CreateUser

The CreateUser procedure allows the creation of new user account with basic attributes. Further attributes may be set using specific privilege instructions.

```
PROCEDURE [PASCAL] CreateUser(            pSession_id :  LONGINT;
                                          pUser       :  ARRAY OF CHAR;
                                          pPassword   :  ARRAY OF CHAR;
                                          pSchema     :  ARRAY OF CHAR
                                  )  :   LONGINT;
```

Parameters

pSession_id	:	A valid session ID, which identifies the database on which the user account is to be created.
pUser	:	The user name for the new user account.
pPassword	:	The initial password for the account.
pSchema	:	The default schema for the user account.

Return values
A standard rc (return code).

Remarks
The nominated user name must not be allocated to any existing account. Neither the user name nor the password may be a null (empty) string.

See also
DropUser, EnableUser, DisableUser, User Manipulation

DropUser

The DropUser procedure drops (deletes) a user account. Use of this feature is discouraged, as any audit references to the given user ID will become unresolvable. Instead of DropUser, the user account should be disabled using the DisableUser call.

```
PROCEDURE [PASCAL] DropUser (pUser :  ARRAY OF CHAR) : LONGINT;
```

Parameters

pUser : The user name of the account to be dropped.

Return values

A standard rc (return code).

Remarks

The nominated user account (for the specified user name) must exist and be current.

See also

CreateUser, EnableUser, DisableUser, User Manipulation

DisableUser

The DisableUser procedure flags the nominated user account as invalid - i.e. the nominated user will not be valid for opening a session through the OpenSession command.

```
PROCEDURE [PASCAL] DisableUser (    pSession_id :  LONGINT;
                                    pUser       :  ARRAY OF CHAR
                                ) : LONGINT;
```

Parameters

pSession_id	:	A valid session ID
pUser	:	The user name of the account to be disabled.

Return values
A standard rc (return code).

Remarks
The nominated user account (for the specified user name) must exist and be current.

After successful execution the nominated user account can no longer be used in an OpenSession command, until or unless the EnableUser procedure is executed with the specified user account.

See also
CreateUser, DropUser, EnableUser, OpenSession, User Manipulation

EnableUser

The EnableUser procedure flags the nominated user account as valid - i.e. the nominated user will be valid for opening a session through the OpenSession command, even if previously rendered invalid through the DisableUser command.

```
PROCEDURE [PASCAL] EnableUser (     pSession_id    :  LONGINT;
                                    pUser          :  ARRAY OF CHAR
                                ) : LONGINT;
```

Parameters

pSession_id	:	A valid session ID
pUser	:	The user name of the account to be enabled.

Return values
A standard rc (return code).

Remarks
The nominated user account (for the specified user name) must exist and be current.

After successful execution the nominated user account can be used in an OpenSession command.

See also
CreateUser, DropUser, DisableUser, OpenSession, User Manipulation

Lava Schema Manipulation

CreateSchema
DropSchema

See also
API Categories

CreateSchema

The CreateSchema procedure creates a new database schema in the database nominated by the session ID. See the section on Lava schemas for further information on the implementation and usage of schemas.

```
PROCEDURE [PASCAL] CreateSchema(      pSession_id       :  LONGINT;
                                      pSchema           :  ARRAY OF CHAR;
                                VAR   pSchema_id        :  LONGINT;
                                      pPrivilegeEnable  :  BOOLEAN;
                                      pOwnerSchema_id   :  LONGINT;
                                      pQuota            :  LONGINT;
                                      pDescription      :  ARRAY OF CHAR
                                 )    :  LONGINT;
```

Parameters

pSession_id	:	A valid session ID
pSchema	:	The name for the schema to be created
pSchema_id	:	A reference to the variable to receive the ID of the created schema.
pPrivilegeEnable	:	Boolean flag which indicates whether the new schema is to have access privileges enabled after creation
pOwnerSchema_id	:	(Not currently implemented - future provision) the ID of the owner schema for subschemas. Should be set to nil (0).
pQuota	:	(Not currently implemented - future provision) the limit, in Mb, to be set on data stored (in the form of data tables) in the new schema. Should be set to nil (0)
pDescription	:	A user-generated description of the schema

Return values
A standard rc (return code).

Remarks
The schema is initially empty after creation, with the exception of the default variable length tables for the schema.

See also
DropSchema, CreateUser, Lava Schema Manipulation

DropSchema

The DropSchema procedure deletes an existing schema including all tables belonging to the schema.

```
PROCEDURE [PASCAL] DropSchema  (     pSession_id       :  LONGINT;
                                     pSchema           :  ARRAY OF CHAR
                                )   :  LONGINT;
```

Parameters

pSession_id	:	A valid session ID
pSchema	:	The name of a valid schema

Return values
A standard rc (return code).

Remarks
All data contained in tables belonging to the schema will be lost on successful execution.

The following schemas are invalid for this command, and will return a non-zero return code, indicating failure : System, Parse, Design, Dictionary, Util, Event, Backup, Linker, Template, Virtual, Sheet, Scratch. These are system schemas and may not be dropped as this will impair database operations.

See also
CreateSchema, Lava Schema Manipulation

Lava Table Search Functions

The search functions presented in the Lava API are dedicated high-performance query facilities intended to search only one table at a time. Multi-table (join) searches are available in the SQL interface - see the Select Statement in the SQL documentation for further details.

Two separate search (query) facilities are presented - the SeekQueryResult family of queries provide multi-column search facilities, while the FirstColumnEntry family of queries provide a very high performance single column query.

CloseQuery	Terminates a multi-column query
FirstColumnEntry	Starts a single-column query
NextColumnEntry	Finds the next match for a single-column query
NextQueryResult	Finds the next match for a multi-column query
PreviousColumnEntry	Finds the previous match for a single-column query
SeekQueryResult	Starts a multi-column query
SetQueryParameter	Sets column conditional parameters for a multi-column query

See also
API Categories

SetQueryParameter

The SetQueryParameter procedure sets a single column condition for a multi-column query (search). If only one column is to be searched, use FirstColumnEntry instead.

```
PROCEDURE [PASCAL] SetQueryParameter (
                   pSession_id          :  LONGINT;
              VAR  pQuery               :  Sys_Query_Type;
                   pIndex               :  LONGINT;
                   pObject_id           :  LONGINT;
                   pColumn_sequence     :  LONGINT;
                   pCondition           :  LONGINT;
                   pColumn_type         :  LONGINT;
                   pColumn_Value_Long   :  LONGINT;
                   pColumn_Value_Float  :  LONGREAL;
             VAR   pColumn_Value_String :  ARRAY OF CHAR
                                        )  :   LONGINT;
```

Parameters

pSession_id	:	A valid session ID
pQuery	:	A reference structure variable of type Sys_Query_Type, which should be cleared before the first call to SetQueryParameter.
pIndex	:	The index (1-based) of the column condition to be specified.
pObject_id	:	The object ID of the table to be searched
pColumn_sequence	:	The column sequence for this column condition
pCondition	:	The condition to be applied - see Comparison Constants
pColumn_type	:	The data type for the column to be compared against - see Data Type Constants
pColumn_Value_Long	:	The value for comparison if the data type of the column is integer, else 0.
pColumn_Value_Float	:	The value for comparison if the data type of the column is float, else 0.
pColumn_Value_String	:	The string value for comparison if the data type of the column is string, else a blank string.

Return values
A standard rc (return code).

Remarks
The pQuery structure should be cleared to zero before calling SetQueryParameter for the first time, as the seek relies on the next column condition after the last valid condition to be blank as a termination.

The column conditions established using the SetQueryParameter procedure must commence with pIndex = 1, and be sequentially specified with no gaps between the values of pIndex.

The maximum number of column conditions that may be specified is 9. Any attempt to specify more than 9 column conditions will result in an error return.

On completion of the query process, the query should be terminated by calling CloseQuery to ensure that all allocated memory is released after completion.

See also
SeekQueryResult, NextQueryResult, CloseQuery, Lava Table Search Functions

CloseQuery

The CloseQuery procedure ensures that any memory allocation during the query process is released.

```
PROCEDURE [PASCAL] CloseQuery (pQuery  :  Sys_Query_Type) : LONGINT;
```

Parameters

pQuery	:	A reference structure variable of type Sys_Query_Type,

Return values
A standard rc (return code).

Remarks
If CloseQuery is not executed on conclusion of the query process, memory leakage can occur.

See also
SeekQueryResult, NextQueryResult, SetQueryParameter, Lava Table Search Functions

NextQueryResult

The NextQueryResult procedure allows retrieval of the next match for a multiple-column query.

```
PROCEDURE [PASCAL] NextQueryResult(          pSession_id :  LONGINT;
                                       VAR   pQuery      :  Sys_Query_Type;
                                       VAR   pRowID      :  LONGINT;
                                             pDelta      :  LONGINT;
                                             pCleanup    :  BOOLEAN
                                       )   :   BOOLEAN;
```

Parameters

pSession_id	:	A valid session ID
pQuery	:	A reference structure variable of type Sys_Query_Type, which has been set up using SetQueryParameter.
pDelta	:	The increment for the result - 1 if the next result is required, or -1 if the previous result is required.
pCleanup	:	A boolean indicator which, if TRUE, allows the procedure to perform query cleanup if no match is found.

Return values
TRUE (1) if the next match is valid, FALSE (0) otherwise.

Remarks
Prior to calling NextQueryResult, SetQueryParameter must be used to initialize the pQuery structure, and SeekQueryResult must be called to establish the result set and find the first match. If either of these steps has not been completed, the results of the NextQueryResult procedure may be inconsistent with requirements.

If it will be a requirement to retrace parts of the query by reversing direction, the pCleanup flag must be set to FALSE to ensure that the query does not clear the query information when the first mismatch is encountered. In this case, CloseQuery must be called explicitly on conclusion of the query.

See also
SetQueryParameter, SeekQueryResult, CloseQuery, Lava Table Search Functions

SeekQueryResult

The SeekQueryResult procedure must be used to establish the first match for a multi-column query, after the pQuery structure has been successfully initialized using SetQueryParameter for every search column.

```
PROCEDURE [PASCAL] SeekQueryResult(       pSession_id :  LONGINT;
                                          pObject_id  :  LONGINT;
                                    VAR   pQuery      :  Sys_Query_Type;
                                          pLowerBound :  BOOLEAN;
                                    VAR   pRowID      :  LONGINT;
                                          pCleanup    :  BOOLEAN
                                   )  :   BOOLEAN;
```

Parameters

pSession_id	:	A valid session ID
pObject_id	:	The object ID of the query table - used for validation purposes
pQuery	:	A reference structure variable of type Sys_Query_Type, which has been set up using SetQueryParameter.
pLowerBound	:	A boolean flag which indicates, for inequality range searches, that the match should commence at the lower bound of the match range
pRowID	:	A reference longint variable in which the row ID of the match is returned, if a valid match is found.
pCleanup	:	A boolean indicator which, if TRUE, allows the procedure to perform query cleanup if no match is found.

Return values
TRUE (1) if the next match is valid, FALSE (0) otherwise.

Remarks
Prior to calling SeekQueryResult, SetQueryParameter must be used to initialize the pQuery structure. If this step has not been completed, the results of the SeekQueryResult procedure will be inconsistent with requirements.

In general, the pCleanup flag should be set to TRUE to allow the query to clear the query information if no match is encountered. If not, CloseQuery must be called explicitly on conclusion of the query.

For inequality comparisons which yield a range of results, the query builds a result set comprising the matches to the query. In this case, the pLowerBound flag should be set appropriately to indicate whether the first match should be at the lower or upper bound of the result set.

See also
SetQueryParameter, NextQueryResult, CloseQuery, Lava Table Search Functions

FirstColumnEntry

The FirstColumnEntry procedure provides a search facility on a single column of a table. This procedure must be executed before a NextColumnEntry or PreviousColumnEntry may be executed.

```
PROCEDURE [PASCAL] FirstColumnEntry  (
                   pSession_id          :  LONGINT;
                   pObject_id           :  LONGINT;
                   pColumn_sequence     :  LONGINT;
              VAR  pValueAddress        :  LONGINT;
              VAR  pColumnScan          :  ColumnScan_Type;
                   pCondition           :  LONGINT
                                      ) : BOOLEAN; (* found *)
```

Parameters

pSession_id	:	A valid session ID.
pObject_id	:	The object ID for the search table.
pColumn_sequence	:	The column sequence for the search column.
pValueAddress	:	This parameter must be an independent doubleword (longint) variable initialized to the address of the search value.
pColumnScan	:	A reference variable of type ColumnScan, used to control the search
pCondition	:	The search condition - see Comparison Constants

Return values
The search returns TRUE (1) if a match is found for the search condition, FALSE (0) otherwise.

Remarks
The session, object (table) and column sequence must be valid.

The search column may not be of type boolean.

If the search condition is equal or greater, the match is the first row at the lower end of the valid search range (if more than one row matches the search condition). In this case, NextColumnEntry will successively find each match to the search condition. If the search condition is less, the match is the first row at the upper end of the valid search range. In this case, PreviousColumnEntry will successively find the remaining matches.

See also
NextColumnEntry, SeekQueryResult, SetQueryParameter, Lava Table Search Functions

NextColumnEntry

The NextColumnEntry procedure allows location of successive matches to a given search which was set up using FirstColumnEntry.

```
PROCEDURE [PASCAL] NextColumnEntry  (
                VAR   pColumnScan             :  ColumnScan_Type;
                      pUnique                 :  BOOLEAN;
                VAR   pValueAddress           :  LONGINT
                                        ) : BOOLEAN; (* found *)
```

Parameters

pColumnScan	:	A reference variable of type ColumnScan_Type, used to control the search
pUnique	:	A boolean which, if set to TRUE (1), indicates that the search should only return unique values. If FALSE (0), the search returns all matches to the search condition.
pValueAddress	:	The search value address reference variable.

Return values

The search returns TRUE (1) if a further match is found for the search condition, FALSE (0) otherwise. The search also returns FALSE if the end of the table is encountered.

Remarks

The initialization for the search, through FirstColumnEntry, must be successful for NextColumnEntry to have the possibility of finding further matches.

The pColumnScan reference structure must be as for the initial call to FirstColumnEntry, and may not be modified by the calling program. The fields in this structure may, however, be referred to for further detail on the match parameters.

See also

FirstColumnEntry, PreviousColumnEntry, Lava Table Search Functions

PreviousColumnEntry

The PreviousColumnEntry procedure allows location of successive (preceding) matches to a given search which was set up using FirstColumnEntry.

```
PROCEDURE [PASCAL] PreviousColumnEntry (
                VAR   pColumnScan             :  ColumnScan_Type;
                      pUnique                 :  BOOLEAN;
                VAR   pValue                  :  LONGINT
                                             ) : BOOLEAN; (* found *)
```

Parameters

pColumnScan	:	A reference variable of type ColumnScan_Type, used to control the search
pUnique	:	A boolean which, if set to TRUE (1), indicates that the search should only return unique values. If FALSE (0), the search returns all matches to the search condition.
pValueAddress	:	The search value address reference variable.

Return values
The search returns TRUE (1) if a further match is found for the search condition, FALSE (0) otherwise. The search also returns FALSE if the beginning of the search table is encountered.

Remarks
The initialization for the search, through FirstColumnEntry, must be successful for PreviousColumnEntry to have the possibility of finding further matches.

As PreviousColumnEntry searches for preceding matches to the current match entry, the search must either have condition less than, or NextColumnEntry must have been successfully executed at least once before the PreviousColumnEntry procedure can succeed.

The pColumnScan reference structure must be as for the initial call to FirstColumnEntry, and may not be modified by the calling program. The fields in this structure may, however, be referred to for further detail on the match parameters.

See also
FirstColumnEntry, NextColumnEntry, Lava Table Search Functions

Lava Entry ID Functions

The Entry ID functions provide for the retrieval of Lava ID values for named entries in certain tables. As many of the Lava API functions require ID values in certain parameters, the functions presented below are important translation facilities to render many of the more powerful API functions (such as CreateTable or TableRows usable.

FindSchema	
FindUser	
GetObject_id	

See also
API Categories

FindSchema

The GetSchema_id procedure executes a search for the nominated schema name, and returns the corresponding schema ID if found.

```
PROCEDURE [PASCAL] FindSchema (         pSchema     :  ARRAY OF CHAR;
                                    )   :   LONGINT;
```

Parameters

pSchema : The name of the schema (case insensitive) to be located.

Return values

The ID of the nominated schema, or 0 if not found.

Remarks

If the search is successful, the ID of the schema is returned. If unsuccessful, 0 is returned.

The nominated schema must exist on the local database - in other words, for Client mount (the default mode of operation for an application), the nominated schema must already have been distributed to the client database (see Distributed Client Operation for further information on distributing schemas) for this function to succeed; the schema search will not be performed on the server. In order to identify a schema on the server, a SQL select statement such as *select id from sys_schemas where schema_name = 'my schema'* executed through the LavaCommand procedure will provide the ID of a non-distributed schema from the server.

See also

GetObject_id, Lava Entry ID Functions

GetObject_id

The GetObject_id procedure attempts to find the nominated object, and if successful returns the object ID.

```
PROCEDURE [PASCAL] GetObject_id (          pSession_id    :  LONGINT;
                                    VAR    pObjectName    :  ARRAY OF CHAR;
                                           pSchema_id     :  LONGINT;
                                    VAR    pObject_id     :  LONGINT
                                    )   :   LONGINT;
```

Parameters

pSession_id	:	A valid session ID.
pObjectName	:	The name (case insensitive) of the required object.
pSchema_id	:	The schema ID for the schema to which the search is to be limited.
pObject_id	:	A reference longint variable in which the located object ID will be returned.

Return values
A standard rc (return code).

Remarks
If the search is successful, the ID of the object is returned in pObject_id and the return code is 0. If unsuccessful, 0 is returned in pObject_id and the non-zero return code indicates the failure reason.

See also
FindSchema, Lava Entry ID Functions

FindUser

The FindUser procedure returns the user ID (the row ID of the user account in the Sys_Users table) for a nominated user name.

```
PROCEDURE [PASCAL] FindUser (pUser :  ARRAY OF CHAR) : LONGINT;
```

Parameters

pUser : The user name for the required user.

Return values

The user ID for the requested user if found, 0 if no match is found.

Remarks

The user name search is case insensitive.

See also

Lava Entry ID Functions

Lava Table Manipulation

The table manipulation procedures provide facilities to create, manipulate, query and drop tables both on the client database and on the server.

AllocateColumnSpace	
AssertTablePointer	
ColumnSpec	
CreateTable	
CreateTableInstance	
DropTable	
FreeColumnSpace	
RenameTable	
RenameTableColumn	
TableColumns	
TableRows	
TableSize	
TruncateTable	

See also
API Categories

TableColumns

The TableColumns procedure returns a detailed column layout for the nominated table (object).

```
PROCEDURE [PASCAL] TableColumns(
                     pSession_id       :  LONGINT;
                     pObject_id        :  LONGINT;
                VAR  pTableFormat      :  TableColumnType;
                     pExpandDimensions :  BOOLEAN;
                VAR  pFlatFormat       :  TableColumnType
                                       );
```

Parameters

pSession_id	:	A valid session ID
pObject_id	:	The object ID representing the table for which column detail is required.
pTableFormat	:	A reference variable of type TableColumnType, which specifies the base structure for the column detail.
pExpandDimensions	:	A boolean parameter indicating whether the column information is to be expanded to eliminate dimensioned columns
pFlatFormat	:	A reference variable of type TableColumnType, which receives the expanded column information if pExpandDimensions is set to TRUE.

Return values
None. If the function fails, the format structure is returned with a column count of 0.

Remarks
The specified object ID must specify a valid object which is of type table.

The table format reference variable, pTableFormat, contains an initially null pointer to a column structure array. If the object is located successfully, the TableColumn procedure allocates sufficient heap memory to accommodate the columns of the table.

If pExpandDimensions is true, the TableColumns procedure will also allocate heap memory for the expanded column information in the pFlatFormat reference structure.

In order to avoid memory leakage, FreeColumnSpace should be called with the pTableFormat structure (and, where appropriate, the pFlatFormat structure) to allow the allocated memory to be freed.

See also
ColumnSpec, AllocateColumnSpace, FreeColumnSpace, CreateTable, Lava Table Manipulation

TableRows

The TableRows procedure returns the number of data rows in the nominated object (table).

```
PROCEDURE [PASCAL] TableRows (pObject_id : LONGINT) : LONGINT;
```

Parameters

pObject_id	:	The object ID for the object representing the table for which the rowcount is required.

Return values
The number of rows in the table represented by the nominated object ID

Remarks
The object ID must be valid and must refer to a valid table.

See also
TableSize, , Lava Table Manipulation

TableSize

The TableSize procedure returns the size in bytes of the nominated object (table).

```
PROCEDURE [PASCAL] TableSize (         pObject_id  :  LONGINT;
                                 VAR   pLength     :  QUADINTEGER
                               ) : LONGINT;
```

Parameters

pObject_id	:	The object ID referring to the table for which the size is required.
pLength	:	A reference variable of type Quadinteger (quadword) in which the size of the table, in bytes, is returned.

Return values
A standard rc (return code).

Remarks
The object ID must be valid and must refer to a valid table.

See also
TableRows, , Lava Table Manipulation

ColumnSpec

The ColumnSpec procedure allows specification of the detailed attributes for a single column in a table column format structure. This procedure is used in order to fill in the details of columns to be defined for a given table when creating a new table (see CreateTable). The procedure is called once for each column to be added.

Before this procedure may be called, the pTableFormat structure must first be initialized by calling AllocateColumnSpace in order to allocate sufficient memory for the column information.

```
PROCEDURE [PASCAL] ColumnSpec (
                 pIndex          : LONGINT;
         VAR     pTableFormat    : LavaDB.TableColumnType;
                 pName           : ARRAY OF CHAR;
                 pLength         : INTEGER;
                 pType           : INTEGER;
                 pDigits         : INTEGER;
                 pFraction       : INTEGER;
                 pNullable       : BOOLEAN;
                 pSystem         : BOOLEAN;
                 pPrimaryVDT     : BOOLEAN;
                 pCached         : BOOLEAN;
                 pDimension      : INTEGER;
                 pFormat         : LONGINT
                                 );
```

Parameters

pIndex	:	The index of the column to be specified. The first column has index 1.
pTableFormat	:	A reference structure of type TableColumnType into which the column array is specifed.
pName	:	The column name
pLength	:	The length of the column - used only for non-fixed type columns, such as strings.
pType	:	The type of the column.
pDigits	:	(not in use - future provision) The number of integer digits for columns of type decimal.
pFraction	:	(not in use - future provision) The number of fraction digits for columns of type decimal.
pNullable	:	(not in use - future provision) Boolean parameter indicating whether the column is nullable.
pSystem	:	(Only for system use) Indicates whether the column is restricted to system use
pPrimaryVDT	:	A boolean parameter indicating whether the column contains the primary VDT for the table.
pCached	:	A boolean parameter indicating whether the column should be buffered at mount time.
pDimension	:	The dimension for array columns. The dimension should be specified as 0 for non-array columns.
pFormat	:	(not in use - future provision) The format ID for a column format entry

Return values
None.

Remarks
The ColumnSpec procedure is used to define the column attributes for all columns in a table format prior to creating a new table using the CreateTable procedure.

The required memory for all columns that are to be defined using ColumnSpec must be allocated in advance using the AllocateColumnSpace procedure.

Once the information in the table format structure is no longer required, the FreeColumnSpace procedure must be called with the table format structure to free the allocated column space.

ColumnSpec is called once for each column to be added to the table format. The pIndex parameter indicates which column is to be defined, and this 1-based index must be incremented strictly sequentially for each column defined - calling ColumnSpec with out of sequence column information will result in an unexpected table format.

See also
AllocateColumnSpace, FreeColumnSpace, CreateTable, Lava Table Manipulation

CreateTableInstance

The CreateTableInstance procedure defines an instance table modelled on an existing object (table). This function allows for the definition of specific tables which can be used to store instance data for a particular control during its execution. This allows data required for management of the control to be stored, retrieved and manipulated without having to filter a consolidated table reaching across several current instances of the control for the data pertaining to a given instance.

```
PROCEDURE [PASCAL] CreateTableInstance  (
                   pSession_id       :  LONGINT;
                   pTableName        :  ARRAY OF CHAR;
                   pSourceObject_id  :  LONGINT;
                   pObjectType       :  LONGINT;
             VAR   pObject_id        :  LONGINT;
                   pRows             :  LONGINT;
             VAR   pTablePointer     :  SYSTEM.PTR
                                           ) : LONGINT;
```

Parameters

pSession_id	:	A valid session ID
pTableName	:	The name of the instance table
pSourceObject_id	:	The object ID representing the source (model) table
pObjectType	:	The object type of the instance table (see Object Types)
pObject_id	:	A reference longint which receives the object ID of the new instance table if the creation succeeds
pRows	:	The required number of initial rows to be allocated if the instance table is a virtual table
pTablePointer	:	The address of a longint variable to serve as the reference pointer for the instance table

Return values
A standard rc (return code).

Remarks
CreateTableInstance is used to create a temporary instance table. Although this can be used for many potential purposes, the original intent of the function is to allow high-speed processing of instance data with ease of programming, as fewer search criteria are needed to locate the information for the instance.

In the most general case, CreateTableInstance would be the first step in instantiating the data for a particular control. This would be followed by an initialization procedure which extracts appropriate data from the permanent data tables, and inserts this data into the instance table. This step can, in most cases, be accomplished by 1 or more SQL “insert as select” statements. The data for the control is then maintained in the instance table during the lifetime of the control. On expiry of the control, typically when the user executes a close on the encapsulating window, the data from the instance table is copied back to the permanent data table, and the instance table is dropped.

The pTablePointer parameter is provided in order to allow the instance table to be addressed directly through an array. See the section Array Access to Virtual Tables for further details on this mechanism.

See also
CreateTable, Lava Table Manipulation

CreateTable

The CreateTable procedure allows creation of a new data table. In addition to the specification of column details for the table, several attributes may be set to modify the nature and purpose of the table to be created.

Prior to calling the CreateTable command, the pTableFormat parameter needs to be correctly initialized through use of the ColumnSpec procedure. This defines detailed column attributes required for the table creation process.

```
PROCEDURE   [PASCAL]  CreateTable (
                  pSession_id       :  LONGINT;
                  pTableName        :  ARRAY OF CHAR;
            VAR   pObject_id        :  LONGINT;
                  pObjectType       :  LONGINT;
            VAR   pTableFormat      :  TableColumnType;
                  pSchema_id        :  LONGINT;
                  pInitialSize      :  LONGINT;
                  pTimeDomain       :  BOOLEAN;
                  pUpdateFileQueue  :  BOOLEAN;
                  pUpdateSysTables  :  BOOLEAN;
                  pAccessType       :  LONGINT;
                  pReplicateTable   :  BOOLEAN;
                  pReclaim          :  BOOLEAN;
                  pReserveRows      :  LONGINT;
                  pLocation         :  LONGINT
                                    )  :   LONGINT;
```

Parameters

pSession_id	:	A valid session ID.
pTableName	:	The name of the table to be created. This name must be unique within the schema selected for the table. Table names may be non-unique across database schemas.
pObject_id	:	The object ID of the created table (if successful) is returned in this reference longint variable.
pObjectType	:	The pObjectType specifies, in a bitfielded longint (dword) parameter, the type options for the table. (see Object Types)
pTableFormat	:	A reference TableColumnType structure, which has already been constructed using the ColumnSpec procedure
pSchema_id	:	The schema ID for the schema within which the table is to be created - this may be obtained from FindSchema if only a textual schema name is available
pInitialSize	:	For virtual and replicator tables, the initial size in bytes of the memory allocation for the table
pTimeDomain	:	(future provision - not in use)
pUpdateFileQueue	:	For user table declarations, this boolean flag should always be TRUE
pUpdateSysTables	:	For user table declarations, this boolean flag should always be TRUE
pAccessType	:	(future provision - not in use)
pReplicateTable	:	A boolean flag which indicates, for physical tables, whether the table should be replicated by default on mount
pReclaim	:	A boolean flag which indicates whether deleted rows are re-used for added rows. If FALSE, rows are added only to the end of the table.

pReserveRows	:	A numeric value indicating the number of rows which should be reserved for addition by default on distribution of the table to a client
pLocation	:	A constant specifying the location (server or client) and default distribution status of the created table - see Table Location

Return values
A standard rc (return code).

Remarks
CreateTable is used both to create conventional tables (virtual and physical) as well as stack tables. No special attributes are required for stack tables, but a table which is created for stack purposes (and has had stack operations such as Push or Pop executed on it) should be reserved for stack purposes only, as using procedures such as DeleteRow will result in unexpected table layout and stack results.

The table name, specified in the pTableName parameter, must be unique within the schema (there is a special exception for tables of type RESULTSET - see Object Types - but these tables are normally created by the SQL command execution and not by the user)

The pObjectType parameter is compiled through addition of the required object constants - see Object Types - an example being .SQL_OBJECT_TABLE + SQL_OBJECT_FRAMED, which would create a physical table which supports transaction frames.. See also the example code provided.

Note that in the case of virtual tables, by default the virtual table will exist only for the current database session. In order to create a virtual table which persists across database sessions (mounts), the table must be created on the server (see the pLocation parameter) and must have the SQL_OBJECT_PERSISTENT attribute within the specified pObjectType.

The pTableFormat structure must be fully specified before the call to CreateTable. This involves multiple calls to the ColumnSpec procedure to define the columns for the table. See the example code for further details.

The pSchema_id parameter must specify an existing schema, although - in the case of a server table creation - the schema may not exist locally. In all cases, the user must have adequate access privilege to create a table within the nominated schema. Conventionally, the schema should be existent on the local database.

The pInitialSize parameter only applies to virtual tables and replicator tables, where a memory image of the table will be allocated on creation. See Virtual Tables for further information.

The pReplicateTable parameter applies only to physical tables (it should be specified as FALSE for virtual tables), and will result in the table being replicated into memory - see Replicator Tables for further information on table replication.

The pReclaim flag specifies whether the table permits deleted rows to be re-used - if FALSE, rows added to the table (through a SQL insert or use of the AddRow procedure) will always add to the end of the table. The pReclaim flag may only be TRUE if the table contains a Row_status column

The pReserveRows parameter only applies to physical tables created on the server. The number of rows specified here is the default number of rows reserved to a client for row addition when the table is distributed. See Distributed Client Operation for further details.

The pLocation parameter specifies in which database (client or server) the table is to be created, and in the case of server creation, whether the table is to be immediately distributed to the client. See Table Location for the applicable constants - as an example, the combination
SQL_OBJECT_SERVER + SQL_OBJECT_DISTRIBUTED
would create the table on the server and distribute the created table to the client.

See also

ColumnSpec, CreateTableInstance, Lava Table Manipulation

Example code
Table Creation

AssertTablePointer

The AssertTablePointer procedure may be used to notify the Lava database kernel of a user-defined table pointer to contain the address of a virtual table. The pointer is maintained by the Lava kernel and always points at the correct address for the virtual table.

```
PROCEDURE [PASCAL] AssertTablePointer(
                       pSession_id       :  LONGINT;
                       pObject_id        :  LONGINT;
                 VAR   pTablePointer     :  SYSTEM.PTR
                                         ) : LONGINT;
```

Parameters

pSession_id	:	A valid session ID
pObject_id	:	The object ID for the virtual table for which the pointer is required.
pTablePointer	:	A reference variable, which should be a 4-byte integer to receive the address of the virtual table.

Return values
A standard rc (return code).

Remarks
The object ID should refer to a virtual table belonging to (defined by) the caller.

The integer (dword) variable referenced by pTablePointer should be defined as a static variable, since the Lava Database kernel will continue to modify the content of the address provided for as long as the nominated object (table) exists. If the variable is defined in transient memory, such as the stack (i.e. a local procedure variable) these updates will result in memory corruption.

The database kernel will update the nominated variable every time a re-allocation of memory results in the memory space of the nominated table moving. The nominated pointer is therefore always valid.

The pointer nominated for update should be defined locally as a pointer to an array of structures which coincide exactly with the column layout of the nominated table. This will result in the array so defined overlaying the content of the virtual table exactly, row for row. In this way, the virtual table may be addressed (within certain constraints) directly as if it were an array.

See the description for Array Access to Virtual Tables for further details.

See also
Lava Table Manipulation

DropTable

The DropTable procedure drops the nominated table permanently. No recovery options exist, and all data contained in the nominated table is lost.

```
PROCEDURE [PASCAL] DropTable (   pSession_id :  LONGINT;
                                 pObject_id  :  LONGINT
                               )  :   LONGINT;
```

Parameters

pSession_id	:	A valid session ID
pObject_id	:	The object ID representing the table to be dropped

Return values
A standard rc (return code).

Remarks
The nominated session ID must have appropriate access rights to the table.

Any relations defined to the dropped table will be dropped automatically, and will not be automatically reinstated even if a table by the same name is created subsequently.

See also
CreateTable, Lava Table Manipulation

TruncateTable

The TruncateTable procedure truncates the nominated table. All data previously contained in the table is lost, and no recovery options exist.

```
PROCEDURE [PASCAL] TruncateTable(      pSession_id :  LONGINT;
                                       pObject_id  :  LONGINT
                                   )   :  LONGINT;
```

Parameters

pSession_id	:	A valid session ID
pObject_id	:	The object ID representing the table to be truncated

Return values
A standard rc (return code).

Remarks
The nominated session must have the required access permission on the nominated table to execute a truncate.

After the truncate all data in the table is irretrievably lost. The only way to restore the data is by restoring a backup made before the truncation.

See also
DropTable, CreateTable, Lava Table Manipulation

RenameTableColumn

The RenameTableColumn procedure renames a column in the nominated table to a given string.

```
PROCEDURE [PASCAL] RenameTableColumn( pSession_id       :  LONGINT;
                                      pObject_id        :  LONGINT;
                                      pColumnSequence   :  LONGINT;
                                      pColumnName       :  ARRAY OF CHAR
                                    ) : LONGINT;
```

Parameters

pSession_id	:	A valid session ID
pObject_id	:	The object ID representing the table containing the column
pColumnSequence	:	The 1-based sequence (index) of the column to be renamed
pColumnName	:	A string parameter specifying the new name for the column

Return values
A standard rc (return code).

Remarks
The nominated session must have adequate access permissions to execute the rename.

Although the specified column name can be any text string, and the table will still be accessible through the Lava API (since columns are addressed through the column sequence), if the nominated column name is not compliant with SQL requirements for a column name (must commence with an alpha character, comprise only conventional ASCII, and may not contain spaces) the column will not be accessible through SQL.

See also
CreateTable, Lava Table Manipulation

RenameTable

The RenameTable procedure renames the nominated table to a newly specified name.

```
PROCEDURE [PASCAL] RenameTable  (      pSession_id    :  LONGINT;
                                       pObject_id     :  LONGINT;
                                       pName          :  ARRAY OF CHAR
                                ) : LONGINT;
```

Parameters

pSession_id	:	A valid session ID
pObject_id	:	The object ID representing the table to be renamed
pName	:	A string parameter specifying the new name for the table

Return values
A standard rc (return code).

Remarks
The nominated session must have adequate access permissions to execute the rename.

Although the specified table name can be any text string, and the table will still be accessible through the Lava API (since tables are addressed through the object ID representing the table), if the nominated name is not compliant with SQL requirements for a table name (must commence with an alpha character, comprise only conventional ASCII, and may not contain spaces) the table will not be accessible through SQL.

See also
CreateTable, Lava Table Manipulation

AllocateColumnSpace

The AllocateColumnSpace procedure ensures that the pTableFormat structure has adequate memory allocated to accommodate all the required columns for the table being defined.

```
PROCEDURE [PASCAL] AllocateColumnSpace (
                       VAR   pTableFormat    :  TableColumnType;
                             pColumnCount    :  LONGINT
                                             ) : LONGINT;
```

Parameters

pTableFormat	:	A reference `TableColumnType` structure for which the column array allocation is to be performed
pColumnCount	:	The number of columns to be defined for the table

Return values
A standard rc (return code).

Remarks
It is essential that the number of columns to be defined is specified correctly on the first invocation of AllocateColumnSpace. A second invocation of this procedure is not permitted.

After completion of the table creation (i.e. after CreateTable has been called with the completed table format) the caller should invoke FreeColumnSpace to free the allocated memory so as to avoid memory leakage.

See also
CreateTable, FreeColumnSpace, Lava Table Manipulation

FreeColumnSpace

The FreeColumnSpace procedure releases any memory allocated as a result of invocation of the AllocateColumnSpace procedure.

```
PROCEDURE [PASCAL] FreeColumnSpace (
                        VAR   pTableFormat    :  TableColumnType
                                              ) : LONGINT;
```

Parameters

pTableFormat	:	A reference `TableColumnType` structure for which the column array allocation is to be released

Return values

A standard rc (return code).

Remarks

After FreeColumnSpace has been invoked on the table format structure, the column array (defined as a pointer to allocated memory in this structure) is no longer accessible.

See also

CreateTable, AllocateColumnSpace, Lava Table Manipulation

Transaction Frames

The Lava Transaction Frame mechanism presents the ability to perform updates (including adds, updates and deletes) on table data with the option to commit or rollback the update.

In addition, the mechanism presents the ability to specify transaction checkpoints (through the Set_Transaction procedure) which creates a nested transaction which may be committed or rolled back independently of the master transaction.

Transaction nesting is supported to arbitrary depth.

LocksExist	
Set_Transaction	
Commit	
Rollback	
TransactionExists	

See also
API Categories

LocksExist

The LocksExist procedure returns a boolean value indicating whether locks (resulting from row-level updates, either SQL or row API) exist on the nominated table.

```
PROCEDURE [PASCAL] LocksExist(pObject_id : LONGINT) : BOOLEAN;
```

Parameters

pObject_id	:	The object ID representing the table to be checked for locks

Return values

The function returns TRUE (1) if there are currently active locks on the nominated object, and FALSE (0) if none are present.

Remarks

If the LocksExist procedure returns TRUE, the implication is that at least one current (uncommitted) transaction contains a pending update on the nominated table.

See also

Set_Transaction, Commit, Rollback, Transaction Frames

Set_Transaction

The Set_Transaction procedure provides a means of creating a named transaction checkpoint, which may be used to perform a partial commit or rollback without terminating the master transaction

```
PROCEDURE [PASCAL] Set_Transaction(    pSession_ID       :  LONGINT;
                                       pTransactionName  :  ARRAY OF CHAR;
                                       pAutoCommit       :  LONGINT;
                                       pSystem           :  LONGINT
                                   )  :   LONGINT;
```

Parameters

pSession_id	:	A valid session ID.
pTransactionName	:	The required name for the checkpoint
pAutoCommit	:	(reserved for future use - not implemented)
pSystem	:	(reserved for future use - not implemented)

Return values
A standard rc (return code).

Remarks
The nominated session ID must be the same session as the one used to perform the currently pending updates in a particular transaction frame for that transaction frame to receive the nested checkpoint. It is possible to create any number of sessions to a given Lava Server, and if the wrong session ID is specified the resulting checkpoint will be unrelated to the current transaction frame.

It is permissible to call Set_Transaction when no current transaction frame exists. This will force a transaction frame, and will give the transaction frame the name specified. A commit or rollback to this transaction name (in this case) will be equivalent to a complete commit or rollback.

The specified transaction (checkpoint) name may be any valid ASCII string.

See also
LocksExist, Commit, Rollback, Transaction Frames

Commit

The Commit procedure performs a commit for the current transaction, either complete or nested depending on specified parameters.

```
PROCEDURE [PASCAL] Commit(      pSession_id    :  LONGINT;
                                pTransaction   :  ARRAY OF CHAR
                              )  :   LONGINT;
```

Parameters

pSession_id	:	A valid session ID with a pending transaction frame.
pTransactionName	:	The required transaction name for a partial commit

Return values
A standard rc (return code).

Remarks
If no transaction frame exists for the nominated session, the call has no effect.

If the pTransaction parameter is specified as an empty string, the commit is a complete commit, performing a final update to the data tables on which pending transactions (updates) exist within the transaction frame for the nominated session. The transaction frame is dropped on successful commit

If the pTransaction parameter is a non-empty string, the nominated transaction (checkpoint) name must exist within the current major transaction frame for the nominated session. If not, the commit will fail and will return an error - no action will be performed, and the transaction frame will remain uncommitted.

Provided the nominated transaction (checkpoint) is found, a partial commit will be executed. The nominated checkpoint may be at any level in the current transaction frame - it is not necessary to commit nested transaction frames in exact reverse order of creation.

See also
LocksExist, Set_Transaction, Rollback, Transaction Frames

Rollback

The Rollback procedure performs a rollback for the current transaction, either complete or nested depending on specified parameters.

```
PROCEDURE [PASCAL] Rollback(     pSession_id    :  LONGINT;
                                 pTransaction   :  ARRAY OF CHAR
                               )  :   LONGINT;
```

Parameters

pSession_id	:	A valid session ID with a pending transaction frame.
pTransactionName	:	The required transaction name for a partial rollback

Return values
A standard rc (return code).

Remarks
If no transaction frame exists for the nominated session, the call has no effect.

If the pTransaction parameter is specified as an empty string, the rollback is a complete rollback, discarding all pending transactions (updates) within the transaction frame for the nominated session. The transaction frame is dropped on successful rollback.

If the pTransaction parameter is a non-empty string, the nominated transaction (checkpoint) name must exist within the current major transaction frame for the nominated session. If not, the rollback will fail and will return an error - no action will be performed, and the transaction frame will remain as before the call to Rollback.

Provided the nominated transaction (checkpoint) is found, a partial rollback will be executed. The nominated checkpoint may be at any level in the current transaction frame - it is not necessary to rollback nested transaction frames in exact reverse order of creation.

See also
LocksExist, Set_Transaction, Commit, Transaction Frames

TransactionExists

The TransactionExsists function returns a boolean value which indicates whether actual pending updates are present for the nominated session.

```
PROCEDURE [PASCAL] TransactionExists(pSession_id    :  LONGINT) : BOOLEAN;
```

Parameters

pSession_id	:	A valid session ID

Return values
A boolean indicating presence (TRUE) or absence (FALSE) of an actual transaction pending for the nominated session

Remarks
If no transaction frame exists for the nominated session, the return value is FALSE.

Even if a transaction frame exists for the nominated session, if no update entries (of any form) are found in the transaction frame, the return value is FALSE.

See also
LocksExist, Set_Transaction, Commit, Rollback, Transaction Frames

Lava Private Memory Management

The Lava kernel supports a set of memory management calls which allow all essential memory management functions. The major benefits from this interface are automatic management of the memory extent, and the ability to query memory management tables using SQL.

CreatePrivateMemory	
DropPrivateMemory	
GetPrivateMemoryAddress	
ExtendPrivateMemory	
WritePrivateMemory	
ClearPrivateMemory	
ReadPrivateMemory	

See also
API Categories

CreatePrivateMemory

The CreatePrivateMemory procedure is the Lava equivalent of the memalloc or heapalloc functions presented by the Windows API. Similarly to these functions, CreatePrivateMemory allocates the memory required and returns the Lava equivalent of a "handle" (a Lava Virtual ID) to the allocated memory. In addition, the Lava procedure activates several facilities which provide for automatic extension of the allocated memory, and validation of access to memory as well as checks on limits when reading and writing into the allocated memory - provided that the Lava procedures are used to access the allocated segment.

```
PROCEDURE [PASCAL] CreatePrivateMemory (  pSession_id       :  LONGINT;
                                          pInitialSize      :  LONGINT;
                                          pIncrementPercent :  LONGINT
                                       ) : LONGINT;
```

Parameters

pSession_id	:	A valid session ID
pInitialSize	:	The number of bytes to be allocated at this time
pIncrementPercent	:	The percentage of the initial allocation by which to extend the allocation if a write into the allocation area using WritePrivateMemory exceeds the current allocation

Return values

If the function succeeds, the return value is the Virtual ID for the entry. If the function fails, the return value is nil (0).

Remarks

In order to obtain a conventional memory address for the allocated memory, the GetPrivateMemoryAddress procedure may be used, but using a memory address to access the memory will eliminate the advantage of validation and automatic extension which results from using the provided memory access procedures.

See also

DropPrivateMemory, GetPrivateMemoryAddress, ExtendPrivateMemory, WritePrivateMemory, ClearPrivateMemory, ReadPrivateMemory, Lava Private Memory Management

DropPrivateMemory

The DropPrivateMemory procedure is the equivalent of the heapfree or virtualfree functions presented by the Windows API to free previously allocated memory. Similarly, DropPrivateMemory frees the memory and deletes the management entries for a previously allocated private memory entry.

```
PROCEDURE   [PASCAL]  DropPrivateMemory  (      pSession_id    :  LONGINT;
                                               pVirtual_id    :  LONGINT
                                         ) : LONGINT;
```

Parameters

pSession_id	:	A valid session ID
pVirtual_id	:	A valid and current virtual ID, previously returned by the CreatePrivateMemory procedure

Return values
A standard rc (return code).

Remarks
After execution of the drop, the virtual ID is no longer valid for private memory operations and will result in an error code if used.

See also
CreatePrivateMemory, GetPrivateMemoryAddress, ExtendPrivateMemory, WritePrivateMemory, ClearPrivateMemory, ReadPrivateMemory, Lava Private Memory Management

GetPrivateMemoryAddress

The `GetPrivateMemoryAddress` procedure is presented for the sake of completeness, and to allow advanced users to access the memory allocated by the private memory functions directly. However, use of this facility is discouraged as it is unprotected and could cause memory violations or memory corruption. Most especially, it is not good practice to write into the memory using this address - use the WritePrivateMemory procedure provided instead, which will validate memory addressed and extend the memory allocated if necessary.

```
PROCEDURE   [PASCAL]  GetPrivateMemoryAddress  (
                         pSession_id    :  LONGINT;
                         pVirtual_id    :  LONGINT;
                   VAR   pBufferAddress :  LONGINT
                                              ) : LONGINT;
```

Parameters

pSession_id	:	A valid session ID
pVirtual_id	:	A valid and current virtual ID, previously returned by the CreatePrivateMemory procedure
pBufferAddress	:	A reference longint variable which receives the address of the allocated memory segment

Return values
A standard rc (return code).

Remarks
Where the address of the memory segment is to be used to overlay a structure, previously correctly sized for allocation through the CreatePrivateMemory procedure, this facility is justified. It is important, however, not to succumb to the temptation to write into the memory outside of the overlayed structure, but to use the WritePrivateMemory or ExtendPrivateMemory facilities to ensure that the validation provided by the private memory system is maintained.

See also
CreatePrivateMemory, DropPrivateMemory, ExtendPrivateMemory, WritePrivateMemory, ClearPrivateMemory, ReadPrivateMemory, Lava Private Memory Management

ExtendPrivateMemory

The ExtendPrivateMemory procedure extends a previously allocated memory segment by the number of bytes specified.

```
PROCEDURE   [PASCAL]  ExtendPrivateMemory  (    pSession_id    :  LONGINT;
                                                 pVirtual_id    :  LONGINT;
                                                 pExtendSize    :  LONGINT
                                                 ) : LONGINT;
```

Parameters

pSession_id	:	A valid session ID
pVirtual_id	:	A valid and current virtual ID, previously returned by the CreatePrivateMemory procedure
pExtendSize	:	The number of bytes by which the memory allocation is to be extended.

Return values
A standard rc (return code).

Remarks
Using the ExtendPrivateMemory procedure is only necessary if, for specific reasons, the memory will not be accessed using the WritePrivateMemory procedure, which would automatically extend memory whenever required.

See also
CreatePrivateMemory, DropPrivateMemory, GetPrivateMemoryAddress, WritePrivateMemory, ClearPrivateMemory, ReadPrivateMemory, Lava Private Memory Management

WritePrivateMemory

The WritePrivateMemory procedure allows the caller to write data into a memory segment previously allocated using the CreatePrivateMemory procedure.

```
PROCEDURE   [PASCAL]  WritePrivateMemory  (     pSession_id    :  LONGINT;
                                                pVirtual_id    :  LONGINT;
                                                pBufferAddress :  LONGINT;
                                                pOffset        :  LONGINT;
                                                pSize          :  LONGINT
                                              ) : LONGINT;
```

Parameters

pSession_id	:	A valid session ID
pVirtual_id	:	A valid and current virtual ID, previously returned by the CreatePrivateMemory procedure
pBufferAddress	:	The address of the source data to be written into the private memory segment
pOffset	:	The offset into the private memory segment at which point the data pointed to by pBufferAddress is to be written
pSize	:	The number of byte to be written into the private memory segment from the source data

Return values
A standard rc (return code).

Remarks
If the memory addressed from pOffset to pOffset + pSize reaches beyond the current extent of the allocated private memory, the memory allocation will be automatically extended to encompass the requested memory write. This also holds even if the start point specified by pOffset is beyond the current allocation.

If, for some reason (such as an out of memory condition), the WritePrivateMemory procedure is required to extend the memory allocation and is unable to do so, an error code is returned. Similarly, if pOffset is negative, an error code is returned.

See also
CreatePrivateMemory, DropPrivateMemory, GetPrivateMemoryAddress, ExtendPrivateMemory, ClearPrivateMemory, ReadPrivateMemory, Lava Private Memory Management

ClearPrivateMemory

The ClearPrivateMemory procedure clears a segment of allocated memory to zero.

```
PROCEDURE   [PASCAL]  ClearPrivateMemory (      pSession_id    :  LONGINT;
                                                pVirtual_id    :  LONGINT;
                                                pOffset        :  LONGINT;
                                                pSize          :  LONGINT
                                            ) : LONGINT;
```

Parameters

pSession_id	:	A valid session ID
pVirtual_id	:	A valid and current virtual ID, previously returned by the CreatePrivateMemory procedure
pOffset	:	The offset into the private memory segment at which point the cleared segment commences
pSize	:	The number of byte to be cleared in the private memory segment

Return values
A standard rc (return code).

Remarks
If the memory addressed from pOffset to pOffset + pSize reaches beyond the current extent of the allocated private memory, the memory allocation will be automatically extended to encompass the requested memory clear. This also holds even if the start point specified by pOffset is beyond the current allocation.

See also
CreatePrivateMemory, DropPrivateMemory, GetPrivateMemoryAddress, ExtendPrivateMemory, WritePrivateMemory, ReadPrivateMemory, Lava Private Memory Management

ReadPrivateMemory

The ReadPrivateMemory procedure reads (copies) a segment of memory from a private memory allocation

```
PROCEDURE [PASCAL]  ReadPrivateMemory  (   pSession_id       :  LONGINT;
                                           pVirtual_id       :  LONGINT;
                                           pBufferAddress    :  LONGINT;
                                           pOffset           :  LONGINT;
                                           pSize             :  LONGINT
                                        ) : LONGINT;
```

Parameters

pSession_id	:	A valid session ID
pVirtual_id	:	A valid and current virtual ID, previously returned by the CreatePrivateMemory procedure
pBufferAddress	:	The address into which the data from the virtual memory is to be copied
pOffset	:	The offset into the private memory segment at which point the data to be copied commences
pSize	:	The number of byte to be copied (read) from the private memory segment into the user buffer

Return values
A standard rc (return code).

Remarks
The user buffer must be large enough (pSize bytes) to receive the copied data.

If the memory addressed from pOffset to pOffset + pSize reaches beyond the current extent of the allocated private memory, the memory allocation will be automatically extended to encompass the requested memory clear. This also holds even if the start point specified by pOffset is beyond the current allocation.

See also
CreatePrivateMemory, DropPrivateMemory, GetPrivateMemoryAddress, ExtendPrivateMemory, WritePrivateMemory, ClearPrivateMemory, Lava Private Memory Management

Lava Replicator Table Functions

The procedures presented in this section are for advanced use only - programmers who attempt to use these calls should be sure to have a sound grasp of the principles of Lava Virtual Tables, and be experienced in memory manipulation.

ReplicatorToDisk	
ExtendReplicatorTable	
Virtual_Realloc	

See also
API Categories

ReplicatorToDisk

The ReplicatorToDisk procedure writes the entire content of the replicator (a Virtual table which is the Replicator memory copy of a disk table) to disk.

Under normal circumstances, this is never required as the Lava database kernel ensures that the disk and memory copies are consistent. The ReplicatorToDisk procedure is only required if the replicator is independently modified through array access, leaving the disk copy out of date.

```
PROCEDURE   [PASCAL]  ReplicatorToDisk  (    pSession_id    :  LONGINT;
                                             pObject_id     :  LONGINT
                                         ) : LONGINT;
```

Parameters

pSession_id	:	A valid session ID
pObject_id	:	The object ID of the table in question

Return values
A standard rc (return code).

Remarks
ReplicatorToDisk can only be used in Exclusive mount mode - in Client mode, the procedure is non-functional as Client databases consist solely of Virtual tables.

This procedure should only be used when severe performance constraints are encountered. Even then, other techniques for improving performance should be explored before resorting to ReplicatorToDisk.

See also
Virtual_Realloc, Lava Replicator Table Functions

ExtendReplicatorTable

The ExtendReplictorTable extends the virtual memory of a replicator (a Virtual table which is the Replicator memory copy of a disk table) to accommodate the specified number of rows.

```
PROCEDURE [PASCAL] ExtendReplicatorTable  ( pObject_id   : LONGINT;
                                            pMinimunRows : LONGINT
                                          ) : LONGINT;
```

Parameters

pObject_id	:	The object ID of the replicator table
pMinimumRows	:	The number of rows to be provided for

Return values
A standard rc (return code).

Remarks
ExtendReplicatorTable can only be used in Exclusive mount mode - in Client mode, the procedure is non-functional as Client databases consist solely of Virtual tables.

Generally, it is not necessary to call this procedure as the Lava Database kernel will always ensure that sufficient memory is allocated for current row requirements.

See also
Virtual_Realloc, Lava Replicator Table Functions

Virtual_Realloc

The Virtual_Realloc procedure ensures that the virtual table nominated has sufficient memory allocated to accommodate the number of rows specified.

```
PROCEDURE [PASCAL] Virtual_Realloc   (    pObject_id     :  LONGINT;
                                          pMinimumRows   :  LONGINT
                                        )  :   LONGINT;
```

Parameters

pObject_id	:	The object ID of the virtual table
pMinimumRows	:	The number of rows to be provided for

Return values
A standard rc (return code).

Remarks
The Virtual_Realloc procedure is normally only used if performance constraints require fewer memory allocations while adding rows to a virtual table.

In special cases, and for advanced users, the Virtual_Realloc procedure can be used to ensure that a virtual table has a minimum number of addressable rows if the table has been mapped onto a memory array into which a given set of rows is to be written without using the AddRow procedure. This technique should be used with care, as memory violations can result if memory bounds are overstepped.

See also
ReplicatorToDisk, Lava Replicator Table Functions

Lava Row-level Table Interface

In order to allow comprehensive access to data in any Lava table, a complete row-level API is defined to allow addition, modification and deletion of any row. The interface in this category applies only to non-raw tables, i.e. tables with a specified RowStatus column. For raw tables, see the following category, Lava Raw Table Interface.

GetColumn	
GetRow	
PutColumn	
PutRow	
AddRow	
DeleteRow	

See also
API Categories

GetColumn

The GetColumn procedure allows retrieval of a single column (of a single row) from a nominated table. This interface allows the retrieval of both fixed-length and variable-length columns.

```
PROCEDURE [PASCAL] GetColumn(          pSession_id       :  LONGINT;
                                       pObject_id        :  LONGINT;
                                       pBufferAddress    :  LONGINT;
                                VAR    pBytesRead        :  LONGINT;
                                       pRowID            :  LONGINT;
                                       pColumnSequence   :  LONGINT
                                )  :    LONGINT;
```

Parameters

pSession_id	:	A valid session ID
pObject_id	:	The object ID for the required table
pBufferAddress	:	The address of a memory buffer (declared by the caller) to receive the column data
pBytesRead	:	A reference longint variable that receives the length in bytes of the column retrieved
pRowID	:	The row ID of the table row containing the column to be retrieved
pColumnSequence	:	The 1-based column sequence of the column to be retrieved

Return values
A standard rc (return code).

Remarks

The column to be retrieved may be of any type.

This procedure is the only method provided to retrieve the content of variable-length columns.

The buffer to receive the column data must be declared by the caller, and must be long enough to receive all column data. The pBytesRead value must be initialized to the length of the buffer before calling the procedure to allow validation.

The pBytesRead parameter will be updated by the GetColumn procedure to reflect the actual number of bytes retrieved.

See also
GetRow, PutColumn, PutRow, AddRow, DeleteRow, Lava Row-level Table Interface

GetRow

The GetRow procedure allows retrieval of a single table row. Variable-length columns are not retrieved in entirety; only the fixed length (base length) portion is retrieved.

```
PROCEDURE [PASCAL] GetRow(          pSession_id    :  LONGINT;
                                    pObject_id     :  LONGINT;
                                    pBufferAddress :  LONGINT;
                           VAR      pBytesRead     :  LONGINT;
                                    pRowID         :  LONGINT;
                                    pPassByAddress :  BOOLEAN;
                           VAR      pBufferPointer :  LONGINT
                           )  :      LONGINT;
```

Parameters

pSession_id	:	A valid session ID
pObject_id	:	The object ID for the required table
pBufferAddress	:	The address of a memory buffer (declared by the caller) to receive the row data
pBytesRead	:	A reference longint variable that receives the length in bytes of the row retrieved
pRowID	:	The row ID of the table row to be retrieved
pPassByAddress	:	System option only - not allowed for non-system callers
pBufferPointer	:	System option only - not allowed for non-system callers

Return values
A standard rc (return code).

Remarks

The pBytesRead parameter must be initialized to the correct row length - a value smaller than the row length will cause processing to fail and an error to be returned.

See also
GetColumn, PutColumn, PutRow, AddRow, DeleteRow, Lava Row-level Table Interface

PutColumn

The PutColumn procedure allows a single column (of a single row) to be updated in the nominated table. This permits both fixed-length and variable-length columns to be updated.

```
PROCEDURE [PASCAL] PutColumn(          pSession_id     :  LONGINT;
                                       pObject_id      :  LONGINT;
                                       pBufferAddress  :  LONGINT;
                                VAR    pBytes          :  LONGINT;
                                       pRowID          :  LONGINT;
                                       pColumnSequence :  LONGINT
                                )  :    LONGINT;
```

Parameters

pSession_id	:	A valid session ID
pObject_id	:	The object ID for the required table
pBufferAddress	:	The address of a memory buffer (declared by the caller) which contains the column data
pBytes	:	A reference longint variable that receives the length in bytes of the column processed
pRowID	:	The row ID of the table row containing the column to be updated
pColumnSequence	:	The 1-based column sequence of the column to be updated

Return values
A standard rc (return code).

Remarks

The column to be updated may be of any type.

This procedure is the only method provided to update the content of variable-length columns.

The buffer containing the column data must be at least as long as the column to be updated, even in the case of string columns.

The pBytes parameter will be updated by the PutColumn procedure to reflect the actual number of bytes updated.

See also
GetColumn, GetRow, PutRow, AddRow, DeleteRow, Lava Row-level Table Interface

PutRow

The PutRow procedure allows the update of a single table row. Variable-length columns are not updated in entirety; only the fixed length (base length) portion is updated.

```
PROCEDURE [PASCAL] PutRow(          pSession_id    :  LONGINT;
                                    pObject_id     :  LONGINT;
                                    pBufferAddress :  LONGINT;
                                    pBytes         :  LONGINT;
                                    pRowID         :  LONGINT
                                ) :   LONGINT;
```

Parameters

pSession_id	:	A valid session ID
pObject_id	:	The object ID for the required table
pBufferAddress	:	The address of a memory buffer containing the updated row data
pBytes	:	The length of the data buffer in bytes
pRowID	:	The row ID of the table row to be updated

Return values
A standard rc (return code).

Remarks

The pBytes parameter must be specified as the correct row length - a value unequal to the row length will cause processing to fail and an error to be returned.

The row to be updated should exist, but is permitted to have a deleted Row status.

See also
GetColumn, GetRow, PutColumn, AddRow, DeleteRow, Lava Row-level Table Interface

AddRow

The AddRow procedure permits the addition of new rows to a table.

```
PROCEDURE [PASCAL] AddRow(          pSession_id    :  LONGINT;
                                    pObject_id     :  LONGINT;
                                    pBufferAddress :  LONGINT;
                                    pBytes         :  LONGINT;
                               VAR  pRowID         :  LONGINT
                               )  :   LONGINT;
```

Parameters

pSession_id	:	A valid session ID
pObject_id	:	The object ID for the required table
pBufferAddress	:	The address of a memory buffer containing the new row data
pBytes	:	The length of the data buffer in bytes
pRowID	:	A reference longint variable that will be set to the row ID of the table row added

Return values
A standard rc (return code).

Remarks

The pBytes parameter must be specified as the correct row length - a value unequal to the row length will cause processing to fail and an error to be returned.

Depending on the table attributes, the new data row will not necessarily be added at the end of the table. If the table has the Reclaim attribute set (see CreateTable) the AddRow procedure will attempt to find an unused (deleted) slot in which to place the new row.

See also
GetColumn, GetRow, PutColumn, PutRow, DeleteRow, Lava Row-level Table Interface

DeleteRow

The DeleteRow procedure provides a mechanism for deleting data rows from non-Raw tables.

```
PROCEDURE [PASCAL] DeleteRow(    pSession_id :  LONGINT;
                                 pObject_id  :  LONGINT;
                                 pRowID      :  LONGINT
                               )  :   LONGINT;
```

Parameters

pSession_id	:	A valid session ID
pObject_id	:	The object ID for the required table
pRowID	:	The row ID for the row to be deleted.

Return values
A standard rc (return code).

Remarks

The nominated row is flagged as deleted, and is no longer a valid entry for retrieval with functions such as GetRow or GetColumn, and will not be selected in any SQL select statement.

DeleteRow may not be used on a Raw table (see Lava Raw Tables).

After deletion, if the table is flagged as Reclaim (see CreateTable), the slot may be used for newly added rows.

See also
GetColumn, GetRow, PutColumn, PutRow, AddRow, Lava Row-level Table Interface

Lava Raw Table Interface

This category deals with interface to tables which do not have a RowStatus column - i.e. rows in the table cannot be flagged as deleted, in order to remove a row from the table it must be physically eliminated, and all subsequent rows in the table moved up. Raw tables are also known as packed tables, as there are no blank or unused entries in a Raw table.

The usage of Raw tables is an advanced concept, and should only be used by programmers who fully understand the concept of packed tables and the limitations and performance constraints imposed by this mechanism.

InsertRow_VirtualRaw	
DeleteRow_VirtualRaw	

See also
API Categories

InsertRow_VirtualRaw

The InsertRow_VirtualRaw procedure allows a data row to be added to a Raw table. The row may be added at any point in the table, allowing sorted or ordered inserts. There is a performance penalty when inserting an early row ID into a large Raw table, as all memory subsequent to the insertion point must be moved up to make space for the new entry.

```
PROCEDURE [PASCAL] InsertRow_VirtualRaw( pObject_id           :  LONGINT;
                                         pUsedRows            :  LONGINT;
                                         pRowID               :  LONGINT;
                                         pBufferAddress       :  LONGINT;
                                         pUpdateControlFile   :  BOOLEAN
                                       )  :   LONGINT;
```

Parameters

pObject_id	:	The object ID representing the table to be updated
pUsedRows	:	The total number of rows in the Raw table prior to the insertion
pRowID	:	The row ID before which the row is to be inserted
pBufferAddress	:	The address of the memory buffer containing the new row data
pUpdateControlFile	:	A boolean flag which, for non-system users, should always be set to TRUE.

Return values
A standard rc (return code).

Remarks
Due to the nature of the Raw table interface, the procedures are “unsafe” and can cause the entire application to fail if all parameters are not exactly correct on each invocation. For this reason, most programmers are advised not to use this procedure; the normal row-level interface to standard tables should be used instead.

The buffer presented must be of the same length as the row length of the Raw table to be updated.

In order to add rows to the end of a Raw table, the AddRow procedure should be used - InsertRow_VirtualRaw is intended strictly to insert rows before existing rows in the table.

See also
DeleteRow_VirtualRaw, PutRow, AddRow, Lava Row-level Table Interface, Lava Raw Table Interface

DeleteRow_VirtualRaw

The DeleteRow_VirtualRaw procedure allows deletion of rows from a Raw table. After deletion, all rows subsequent to the deleted row are moved up over the deleted row, as Raw tables never have any unused or free rows. There is a performance penalty when deleting an early row ID into a large Raw table, as all memory subsequent to the insertion point must be moved up to eliminate the deleted row.

```
PROCEDURE [PASCAL] DeleteRow_VirtualRaw ( pObject_id           :  LONGINT;
                                          pRowID               :  LONGINT;
                                          pUpdateControlFile   :  BOOLEAN
                                        )  :   LONGINT;
```

Parameters

pObject_id	:	The object ID representing the table to be updated
pRowID	:	The row ID of the row to be deleted
pUpdateControlFile	:	A boolean flag which, for non-system users, should always be set to TRUE.

Return values
A standard rc (return code).

Remarks
Due to the nature of the Raw table interface, the procedures are “unsafe” and can cause the entire application to fail if all parameters are not exactly correct on each invocation. For this reason, most programmers are advised not to use this procedure; the normal row-level interface to standard tables should be used instead.

See also
InsertRow_VirtualRaw, PutRow, AddRow, Lava Row-level Table Interface, Lava Raw Table Interface

Distributed Client Operation

The concept of Distributed Client is one of the most import new concepts in the Lava database. In brief, Distributed Client allows the client application to operate at all times as if the server database is mounted locally in exclusive mode. All distributed schemas can be accessed with the speed and ease of use of a private, single user database, while the Distributed Client mechanism takes care of communication with the server.

For a more detailed description of Distributed Client, see the related description under Key Concepts.

If the database has been mounted in client mode, and is successfully connected to a Lava server, schemas (with the exception of the System schema) may be distributed to the client - this renders access to the tables in the schema very fast and easy, and automatically enables peer-to-peer data sharing, or workgroup operation, on all tables within the schema.

RequestUpdateEvent	
DistributeSchema	

See also
API Categories

RequestUpdateEvent

The RequestUpdateEvent procedure provides a means for the client application to acquire notification of changes in the database with respect to any tables distributed to the client.

Directly after client initialization (through the CreateDatabase procedure) the client requests distribution of any schemas the client will be monitoring, displaying or modifying. This is achieved through the DistributeSchema procedure. This results in all tables belonging to the nominated schemas being distributed to the client database.

At this point, the RequestUpdateEvent procedure is called, once for each schema to be monitored. This results in a Windows message being posted to the event handler of the client application on every occasion where data modification occurs in any of the distributed tables belonging to the nominated schema.

```
PROCEDURE [PASCAL] RequestUpdateEvent   (    pHandle        :  LONGINT;
                                             pSchema_id     :  LONGINT;
                                             pEvent         :  LONGINT;
                                             pbIndividual   :  BOOLEAN
                                          ) : LONGINT;
```

Parameters

pHandle	:	The handle of the window to which update events are to be posted
pSchema_id	:	The ID of the schema for which table modification is to be notified
pEvent	:	The event code to be posted to the client window
pbIndividual	:	A boolean flag indicating whether table-individual notification is required - see in Remarks below.

Return values
A standard rc (return code).

Remarks
On modification of any of the tables in the nominated schema, the provided window handle will receive a message (nominated in pEvent) to its event handler (or Window Procedure - see WindowProc in the Windows API documentation).

If pbIndividual is specified as FALSE, a single message is posted for each batch of updates from the server (which may update multiple tables) with wParam and lParam both null (see WindowProc in the Windows API documentation).

If pbIndividual is specified as TRUE, one message is posted for each table modified, with wParam equal to the object ID of the table being updated as a result of a server-distributed update. The lParam parameter is null.

See also
DistributeSchema, Distributed Client Operation

DistributeSchema

The DistributeSchema procedure requests distribution of all the tables in the nominated schema from the server. On successful completion, and after the propagation delay incurred as a result of line bandwidth, the client has the complete data of the entire schema present on the workstation.

In addition, from the time of distribution onward, the server updates any data modified as a result of updates or additions performed by other clients with this schema distributed.

```
PROCEDURE [PASCAL] DistributeSchema(   pSession_id    :  LONGINT;
                                       pSchema        :  ARRAY OF CHAR;
                                       pbInitial      :  BOOLEAN;
                                       pbFinal        :  BOOLEAN
                                    ) : LONGINT;
```

Parameters

pSession_id	:	A valid session ID, which links to the correct server
pSchema	:	The name of the schema to be distributed
pbInitial	:	A boolean flag indicating whether this is the first schema in a batch of distribution requests
pbFinal	:	A boolean flag indicating whether this is the last schema in a batch of distribution requests

Return values
A standard rc (return code).

Remarks

The schema is specified in name form (text) instead of the conventional ID format, as the nominated schema may not exist on the freshly mounted client database, and hence it is not possible to locally determine the schema ID - hence the name of the schema is more practical. The name is case insensitive.

The pbInitial flag must be TRUE (1) for the first call to DistributeSchema after mount, FALSE (0) otherwise.
The pbFinal flag must be TRUE (1) for the last call to DistributeSchema, FALSE (0) otherwise.
If only one schema is to be distributed, both pbInitial and pbFinal should be flagged as TRUE.

See also
RequestUpdateEvent, Distributed Client Operation

Lava Thread Support

Where the application program requires independent threads and the programmer would like to see the thread in the lava thread table in order to be able to track threads within the application, the Lava threading interface may be used to start any required threads. (For a more comprehensive treatment of threads in the Windows environment, see the category “Processes and Threads” in the Microsoft MSDN).

StartThread	
CloseThread	

See also
API Categories

StartThread

The StartThread procedure starts a new thread with the nominated parameters. In addition, the thread is placed in a thread wrapper which provides default exception handling which will avoid errors such as divide by zero and invalid float load causing the thread (or application) to abort.

```
PROCEDURE [PASCAL] StartThread (
                  pSession_id          :  LONGINT;
                  pThreadType          :  LONGINT;
                  pTitle               :  ARRAY OF CHAR;
                  pThreadProc          :  WinBase.LPTHREAD_START_ROUTINE;
                  pParameterAddress    :  WinDef.LPVOID;
                  pIdent               :  LONGINT
                                       ) : LONGINT;
```

Parameters

pSession_id	:	A valid session ID
pThreadType	:	A caller-specified thread type identifier, which must be greater than 1000
pTitle	:	The caller title (text) of the thread
pThreadProc	:	The address of the thread to be started
pParameterAddress	:	The address of the parameters to the thread
pIdent	:	A caller-specified thread identifier, which must be greater than 1000

Return values
A standard rc (return code).

Remarks
The thread procedure must comply with the Windows ThreadProc prototype (see ThreadProc in the Windows API) which allows a 4-byte parameter (dword) and should return a dword on exit.

The pParameterAddres should specify the exact value of the dword parameter to be passed to the thread procedure on creation (typically a pointer to a structure).

On exit, and before returning, the thread procedure should call CloseThread to allow removal of the thread from the Lava thread management tables.

See also
CloseThread, Lava Thread Support

CloseThread

The CloseThread procedure notifies the Lava database kernel that a thread previously started using StartThread is about to be terminated. It should be the last procedure called before the thread exits.

```
PROCEDURE [PASCAL] CloseThread ();
```

Parameters
None.

Return values
A standard rc (return code).

Remarks
CloseThread must be called immediately prior to the thread returning (exiting) in order for the Lava kernel to track the status of the thread.

See also
StartThread, Lava Thread Support

Lava Stack Tables

In order to support programming algorithms which require an extensible stack for data storage and manipulation, a special kind of table which supports stack operations in its native mode is provided in the Lava kernel.

Stack tables are defined in memory, and are extremely fast. As with all Lava tables, memory management is automatically applied, which means that stack tables are effectively as large as workstation memory.

Aside from characteristics of operation, stack tables are defined and maintained as for any other Lava virtual table - see Virtual Tables for principles of Lava memory tables, and Lava Table Manipulation for creating and manipulating stack tables.

Aside from being specified as virtual tables, the characteristics of stack tables result from the stack operations themselves - a stack table is merely a virtual table on which stack operations are performed. It is permissible to perform a PutRow operation to a stack table, for example, provided that you fully understand the principles behind row IDs and virtual table structures - such an update will merely change the value of that item on the stack. It is not permissible, however, to perform a DeleteRow on a stack table - this violates the stack sequence and will cause undesirable behaviour from the stack API.

Push	
Pop	
GetStackTop	
ClearStack	

See also
API Categories

Push

The Push procedure pushes a new stack item onto the top of a stack table. The stack depth grows by 1.

```
PROCEDURE [PASCAL] Push(          pSession_id    :  LONGINT;
                                  pObject_id     :  LONGINT;
                                  pBufferAddress :  LONGINT;
                                  pBytes         :  LONGINT
                              )   :   LONGINT;
```

Parameters

pSession_id	:	A valid session ID
pObject_id	:	The object ID representing the stack table
pBufferAddress	:	The address of the data buffer for the stack item (data row)
pBytes	:	The number of bytes in the data buffer

Return values
A standard rc (return code).

Remarks
The object ID should represent a valid stack table. Performing stack operations on a conventional (non-stack) table may have unexpected results.

The number of bytes specified in pBytes should coincide exactly with the size in bytes of the data row as defined at table creation (see CreateTable). If not, an error will result and the data will not be pushed on the stack.

See also
Push, Pop, GetStackTop, ClearStack, Lava Stack Tables

Pop

The Pop procedure pops the top item (row) off a stack table, returns this item to the caller, and diminishes the stack depth by 1.

```
PROCEDURE [PASCAL] Pop   (          pSession_id    :  LONGINT;
                                    pObject_id     :  LONGINT;
                                    pBufferAddress :  LONGINT;
                               VAR  pBytesRead     :  LONGINT
                                    )  :   LONGINT;
```

Parameters

pSession_id	:	A valid session ID
pObject_id	:	The object ID representing the stack table
pBufferAddress	:	The address of the data buffer to receive the stack item (data row)
pBytesRead	:	A reference longint variable which must be initialized by the caler to the size of the row data for the stack table. The Pop procedure returns the number of bytes read into the data buffer in this variable.

Return values
A standard rc (return code).

Remarks
The object ID should represent a valid stack table. Performing stack operations on a conventional (non-stack) table may have unexpected results.

The number of bytes specified in pBytesRead should coincide exactly with the size in bytes of the data row as defined at table creation (see CreateTable). If not, an error will result and the data will not be popped off the stack.

See also
Push, Pop, GetStackTop, ClearStack, Lava Stack Tables

GetStackTop

The GetStackTop procedure reads the top item (data row) off the stack table, but does not diminish the stack depth. See Pop for a conventional stack pop operation.

```
PROCEDURE [PASCAL] GetStackTop   (        pSession_id    :  LONGINT;
                                          pObject_id     :  LONGINT;
                                          pBufferAddress :  LONGINT;
                                    VAR   pBytesRead     :  LONGINT
                                    )  :   LONGINT;
```

Parameters

pSession_id	:	A valid session ID
pObject_id	:	The object ID representing the stack table
pBufferAddress	:	The address of the data buffer to receive the stack item (data row)
pBytesRead	:	A reference longint variable which must be initialized by the caler to the size of the row data for the stack table. The Pop procedure returns the number of bytes read into the data buffer in this variable.

Return values
A standard rc (return code).

Remarks
The object ID should represent a valid stack table. Performing stack operations on a conventional (non-stack) table may have unexpected results.

The number of bytes specified in pBytesRead should coincide exactly with the size in bytes of the data row as defined at table creation (see CreateTable). If not, an error will result and the data will not be popped off the stack.

See also
Push, Pop, GetStackTop, ClearStack, Lava Stack Tables

ClearStack

The ClearStack procedure clears a stack table, which resets the stack depth to 0.

```
PROCEDURE [PASCAL] ClearStack  (       pSession_id    :  LONGINT;
                                       pObject_id     :  LONGINT
                                  ) : LONGINT;
```

Parameters

pSession_id	:	A valid session ID
pObject_id	:	The object ID representing the stack table

Return values
A standard rc (return code).

Remarks
The object ID should represent a valid stack table. Performing stack operations on a conventional (non-stack) table may have unexpected results.

See also
Push, Pop, GetStackTop, ClearStack, Lava Stack Tables

SQL Interface

The SQL interface to Lava is operated through a very simple, single procedure call. The results of the call (if data is to be returned, for example with *select* statements) are returned in the form of a result table, which is a conventional Lava Virtual Table with additional attributes.

The location of execution (Client or Server) is determined in terms of the location (distribution status) of the tables involved. If all tables nominated within the SQL command are distributed, the command is executed on the Client. If any table nominated in the SQL command is not distributed, the entire command is executed remotely, on the server. See SQL Command Execution for further information.

See also
API Categories

LavaCommand

The LavaCommand procedure executes a SQL command on a Lava database. The location of execution (Server or Client) will depend on the mount mode and table distribution details - See SQL Command Execution for further details.

```
PROCEDURE  [PASCAL] LavaCommand(          pSession_id :  LONGINT;
                                  VAR     pSQL        :  ARRAY OF CHAR;
                                  VAR     pResult_id  :  LONGINT
                                ) : LONGINT;
```

Parameters

pSession_id	:	A valid session ID
pSQL	:	A string reference variable containing the SQL command to be executed
pResult_id	:	If the command yields results the result object ID is returned in this reference longint variable.

Return values
A standard rc (return code).

Remarks
The specified SQL command is executed, and if results are yielded as a result of execution, these are placed in a Lava result table (in essence a Lava Virtual Table), for which the object ID is returned in pResult_id. If an error is encountered or no results are yielded by the SQL executed (for example an ALTER TABLE command would yield no results) the pResult_id is zero (0).

The result data can be extracted either by using the GetRow procedure to extract one row of data at a time, or by using Array Access to Virtual Tables to map a pointer to an array of row structures over the result set.

Example code
SQL Execution and Data Extraction

Miscellaneous Interfaces

The interface and utility procedures presented in this section are miscellaneous procedures none of which fit into any of the conventional categories.

BlockCRC	
DayOfWeek	
EndActivity	
ExtractFileName	
Extract_VDT_Time	
Format_VDT	
FormatNumber	
GetCommandParm	
GetDate	
GetServerDateTime	
GetTime	
GregorianDate	
HeapSort	
HPtimestamp	
JulianDate	
JulianTime	
LogEvent	
MessageBox	
MonthDays	
ParseCommandLine	
Random	
ServerDate	
ShowActivity	
SplitFullName	
StartActivity	
StringCRC	

See also
API Categories

LogEvent

The LogEvent procedure logs an error or event to the Lava event log (see Sys_Event_Log for more information)

```
PROCEDURE [PASCAL] LogEvent (
  pFunction_id, pEvent_id, pObject_id, pSession_id, pLog_id : LONGINT
                                ) : LONGINT;
```

Parameters

pFunction_id	:	An identifier which logs the function (procedure) - user function numbers should be greater than 1000.
pEvent_id	:	The identifier for the event - this should coincide with an event type ID, or be recognizable to the caller.
pObject_id	:	The object ID of a table involved in the event or error
pSession_id	:	The caller's session ID
pLog_id	:	The previous event ID if a chain of events is being established.

Return values
The Event ID for the logged event entry.

Remarks
Logged events can be queried through regular table row interfaces to trace event chains.

See also
Miscellaneous Interfaces

GetServerDateTime

The GetServerDateTime procedure returns the DateTime as defined by the Lava Server, in VDT format.

```
PROCEDURE [PASCAL] GetServerDateTime() : LONGREAL;
```

Parameters
None

Return values
The server DateTime in VDT format.

Remarks
The time returned is that of the server on which the Lava Server is running to which the client is connected, or, if the mount type is exclusive, it is the local time of the workstation / server on which the database is running.

See also
Miscellaneous Interfaces

FormatNumber

The FormatNumber procedure creates a formatted string representing a given number in the requested format.

```
PROCEDURE [PASCAL] FormatNumber(
                     pNumberAddress       :  LONGINT;
                VAR  pOutput              :  ARRAY OF CHAR;
                     pPrimary             :  LONGINT;
                     pSecondary           :  LONGINT;
                     pDecimals            :  LONGINT;
                     pb1000Separator      :  BOOLEAN;
                     pCurrency            :  ARRAY OF CHAR;
                     pbTrailingCurrency   :  BOOLEAN;
                     pbUnicode            :  BOOLEAN
                               ) : LONGINT;
```

Parameters

pNumberAddress	:	The address of the source number to be formatted.
pOutput	:	A reference string to receive the formatted output number
pPrimary	:	The primary format code - see Primary Format Codes and the Remarks section for composition of this parameter
pSecondary	:	The secondary format code - see Secondary Format Codes and the Remarks section
pDecimals	:	The number of decimals to be depicted
pb1000Separator	:	A boolean flag indicating whether a 1000s separator (comma) should be inserted
pCurrency	:	A string providing the currency symbol / string if the primary format code specifies a currency format
pbTrailingCurrency	:	A boolean flag indicating whether the currency should be placed trailing (TRUE) or leading (FALSE)
pbUnicode	:	A boolean flag indicating whether the output string should be in Unicode (TRUE) or ASCII (FALSE)

Return values
A standard rc (return code).

Remarks
The string variable specified for the output should be long enough to accept the fully formatted string as specified.

The primary format code (pPrimary) must be compiled from a primary output format (see Primary Format Codes, FORMAT_P_GENERAL through FORMAT_P_BOOLEAN) and an input number type (see Primary Format Codes, FORMAT_P_LONGREAL through FORMAT_P_QUADINTEGER). A valid example of a primary format code would be FORMAT_P_DATE + FORMAT_P_LONGINT, which specifies that the input number is a longint (4-byte signed integer), and the primary output format is date format.

The secondary format code specifies details of formatting where the primary output format is ambiguous - for example, in the case of the FORMAT_P_DATE format selected for the primary format above, the secondary format may be FORMAT_S_DATE_2, specifying the type of date format. If the primary format is unambiguous, the secondary format should be specified as FORMAT_S_NULL.

See also
Miscellaneous Interfaces

Format_VDT

The Format_VDT procedure formats a standard VDT variable, input in the form of a longreal (8-byte float), into standard VDT string output form.

```
PROCEDURE [PASCAL] Format_VDT (         pVDT        :  LONGREAL;
                                 VAR  VDT_String  :  ARRAY OF CHAR
                                );
```

Parameters

pVDT	:	The VDT to be formatted
VDT_String	:	A reference string variable to receive the formatted output string

Return values
None

Remarks
The reference string variable must be 20 characters in length to be able to receive the formatted VDT output string.

The VDT is output in the form YYYY.MM.DD HH:MM:SS.hhh

See also
Miscellaneous Interfaces

GetDate

The GetDate procedure returns the current date in individual fields

```
PROCEDURE [PASCAL] GetDate(VAR day,month,year,dayOfWeek:INTEGER);
```

Parameters

```
      day          :     Reference integer variable. The day of the month
      month        :     Reference integer variable. The current month
      year         :     Reference integer variable. The current year
      dayOfWeek    :     Reference integer variable. Day of the week; Sunday
                         is represented as 1
```

Return values
None

Remarks
The date returned is as per the workstation date. Use GetServerDateTime to obtain the server date and time.

See also
GetTime, Miscellaneous Interfaces

GetTime

The GetTime procedure returns the current time in individual fields

```
PROCEDURE [PASCAL] GetTime(VAR sec,min,hour:INTEGER);
```

Parameters

```
      sec   :     Reference integer variable. Current second
      min   :     Reference integer variable. Current minute
      hour  :     Reference integer variable. Current hour (24 hour format)
```

Return values
None

Remarks
The time returned is as per the workstation date. Use GetServerDateTime to obtain the server date and time.

See also
GetDate, Miscellaneous Interfaces

HPtimestamp

The HPtimestamp procedure returns a high-performance timestamp for timing or performance measurement purposes.

```
PROCEDURE [PASCAL] HPtimestamp (VAR time: QUADINTEGER);
```

Parameters

time : A reference Quadinteger (8-byte signed integer) which is set to the timestamp value.

Return values

None

Remarks

The timestamp is merely a fast, synchronous counter, which is hardware dependent. The exact per-unit meaning of the count depends on the workstation configuration and manufacturer, and has to be determined on a per-machine basis, but is usually one count per microsecond in most cases.

See also

Miscellaneous Interfaces

JulianDate

The JulianDate procedure converts a standard year, month, day specification into a Lava Julian date, consistent with all Lava date fields and VDT date indicators.

```
PROCEDURE [PASCAL] JulianDate (pYear, pMonth, pDay : INTEGER) : LONGINT;
```

Parameters

pYear	:	The year (CE, Current Era)
pMonth	:	Month of the year
pDay	:	Day of the month

Return values
The Lava Julian date for the given standard date.

Remarks
A Julian date is a numerically coded date, from a (normally arbitrary) starting date, which varies from application to application. In the case of the Lava Julian date, day 1 is January 1st, 300 CE.

Standard arithmetic (subtraction, addition of deltas) are valid on Julian dates, as they are strictly sequential, taking into account variants such as leap years and days of the month. The result can then be converted to a Gregorian date using the GregorianDate procedure.

See also
GetServerDateTime, VDT, GregorianDate, Miscellaneous Interfaces

JulianTime

The JulianTime procedure allows conversion of a conventional (hour, minute, second) time to a time format compatible with a Lava VDT.

```
PROCEDURE [PASCAL] JulianTime(
       pHour, pMinute, pSecond, pMillisecond : INTEGER
                               ) : LONGREAL;
```

Parameters

pHour	:	Required hour in 24-hour fomrat
pMinute	:	Required minute
pSecond	:	Required second
pMillisecond	:	Required milliseconds

Return values
A VDT-format time (presented in a longreal), with the date portion (integer portion) zero.

Remarks
The time value returned is represented as a fraction, coded into the fractional part of the returned longreal. The integer portion of the return is always 0.

Standard arithmetic can be applied (in general) to time in the returned format, provided that the fraction does not become less than 0 or greater than the number of milliseconds in a day.

Reverse conversion may be accomplished by Extract_VDT_Time.

See also
Extract_VDT_Time, Miscellaneous Interfaces

ServerDate

The ServerDate procedure retrieves the current date as defined by the Lava Server, coded in Julian date format.

```
PROCEDURE [PASCAL] ServerDate ()  :   LONGINT;
```

Parameters
None.

Return values
The current Server date in Julian date format.

Remarks
If the mount mode is Exclusive, the date on the workstation / server on which the database is mounted will be returned.

See also
GetServerDateTime, VDT, Miscellaneous Interfaces

Extract_VDT_Time

The procedure Extract_VDT_Time allows the extraction of the (Julian format) time portion of a VDT into conventional hour-minute-second format.

```
PROCEDURE [_APICALL] Extract_VDT_Time (
                        pVDT        :  LONGREAL;
                    VAR pTime       :  TimeClass
                                       );
```

Parameters

pVDT	:	A longreal (8 byte float) containing a standard Lava VDT
pTime	:	A reference TimeClass structure into which the decoded time values are placed

Return values
None.

Remarks
The integer portion (date portion) of the VDT is ignored.

The time coded into the TimeClass structure is in 24-hour format.

See also
JulianDate, JulianTime, VDT, GetServerDateTime, Miscellaneous Interfaces

GregorianDate

The GregorianDate procedure extracts a standard date (year, month, day) from a given Julian date.

```
PROCEDURE [PASCAL] GregorianDate (        pJulianDate : LONGINT;
                                  VAR     pDate       : DateClass);
```

Parameters

pJulianDate	:	The Julian date to be decoded
pDate	:	A reference DateClass structure which receives the decoded date.

Return values
None.

Remarks
The given Julian date must be according to the Lava Julian date standard (see VDT).

The decoded date will be in the range 300.01.01 through 2999.12.31.

See also
JulianDate, JulianTime, VDT, GetServerDateTime, Miscellaneous Interfaces

MonthDays

The MonthDays procedure returns the correct number of days in any specified month since the year 300, taking into account all leap-years.

```
PROCEDURE [PASCAL] MonthDays (pYear, pMonth : INTEGER) : LONGINT;
```

Parameters

pYear	:	The required year
pMonth	:	The required month

Return values
The number of days in the given month.

Remarks
Provided that the input year is 300 or greater, this procedure will always return the correct number of days, considering all standard or special leap-years.

See also
Miscellaneous Interfaces

DayOfWeek

The DayOfWeek procedure returns the 1-based numeric representation of the day of the week given a Lava Julian date. Sunday is coded as day 1.

```
PROCEDURE [PASCAL] DayOfWeek   (  pDate  :  LONGINT) : INTEGER;
```

Parameters

Return values
The 1-based day of the week for the nominated Julian date, with 1 representing Sunday.

Remarks
The given Julian date must be a Lava-standard Julian date.

See also
GetServerDateTime, VDT, Miscellaneous Interfaces

Random

The Random procedure generates a random number between the limits provided.

```
PROCEDURE [PASCAL] Random (      pLow     : LONGINT;
                                 pHigh    : LONGINT
                           ) : LONGINT;
```

Parameters

pLow	:	The lower limit of the returned random number
pHigh	:	The upper limit of the returned random number

Return values

A random number between the nominated limits.

Remarks

The Random procedure is a pseudo-random generator with very good scatter and a very low rate of repitition.

See also

Miscellaneous Interfaces

HeapSort

The HeapSort procedure is a very high speed sort (order n log n) which sorts an arbitrary input into a sorted form. The sort is in place in terms of the input data, but it does occupy stack memory for the heap management of the sort.

```
PROCEDURE [PASCAL] HeapSort (  N     : LONGINT;
                               Less  : CompareProc;
                               Swap  : SwapProc);
```

Parameters

N	:	The number of items to be sorted
Less	:	A procedure (written by the caller) which returns TRUE if the first item referenced is smaller than or precedes the second item in terms of the required sort order
Swap	:	A procedure (written by the caller) which swaps the first and second referenced items in the input data

Return values
None

Remarks
The user must specify how many items are in the input data, as this is crucial to the management of the heapsort.

The two procedures, Less and Swap, are caller-written and must comply with the interface defined in HeapSort Procedure Types. The first, the CompareProc type, accepts a low and high index into the input data, and is required to compare the two items and return TRUE (1) if the first item should precede the second item in terms of the required sort order. The second, the SwapProc type, again accepts a low and high index, and swaps the two items in the input data.

See also
Miscellaneous Interfaces

ParseCommandLine

The ParseCommandLine procedure parses a conventional Windows command line into a searchable command parameter array

```
PROCEDURE  [PASCAL] ParseCommandLine (
                      pCmdAddress    :  LONGINT;
                VAR   pCmdParam      :  CommandLineType
                                     );
```

Parameters

pCmdAddress	:	The address of the command line as returned by the Windows API GetCommandLineA() function
pCmdParam	:	A reference structure of type CommandLineType which receives the parsed command line

Return values
None

Remarks
The command line address must be a reference to a valid command line.

The syntax provided for in the command line parameters includes :

- Parameter name prefixed by either '-' (dash) or '/' (slash)
- String values framed by double-or single quotes
- String values unframed by quotes, de facto delimited by the commencement of the next parameter
- Numeric values (integer and real)
- Switches (no value to the parameter, the switch is the parameter presence or absence in the command line

The general form of a parameter in the command line is **/ParmName = ParmValue**

See also
GetCommandParm, Miscellaneous Interfaces

GetCommandParm

The GetCommandParm procedure searches a previously parsed command line (see ParseCommandLine) for a nominated parameter name, and returns the value of the parameter.

```
PROCEDURE [_APICALL] GetCommandParm       (
                       VAR   pCmdParam -          :  CommandLineType;
                             pIdent               :  ARRAY OF CHAR;
                       VAR   pValue               :  ARRAY OF CHAR;
                             pNumericValue        :  POINTER TO LONGREAL;
                             pbCaseInsensitive    :  BOOLEAN
                                                  ) : BOOLEAN;
```

Parameters

pCmdParam	:	A reference variable of CommandLineType which contains the results of the ParseCommandLine procedure
pIdent	:	The parameter required
pValue	:	A reference string into which the value of the parameter (if found) will be placed
pNumericValue	:	The address of a longreal (8 byte float) variable into which the decoded numeric value of the parameter will be placed - nil if not required
pbCaseInsensitive	:	A boolean flag set to TRUE if the parameter name search is to be case insensitive

Return values
TRUE (1) if the command parameter is found, or FALSE (0) if not.

Remarks
The pIdent string must be long enough to receive the value of the parameter

The pNumericValue address may be passed as nil (0) if a decode to numeric is not required.

See also
ParseCommandLine, Miscellaneous Interfaces

BlockCRC

The BlockCRC procedure calculates a 32-bit CRC, or cyclic redundancy check (which may be incremental) for the given memory block.

```
PROCEDURE [PASCAL] BlockCRC(           pCount            :  LONGINT;
                                       pSourceCRC        :  LONGINT;
                                       pBufferAddress    :  LONGINT
                                ) : LONGINT;
```

Parameters

pCount	:	The number of bytes to be calculated
pSourceCRC	:	The starting value for the CRC - zero (0) if this is the first block to be calculated
pBufferAddress	:	The address of the buffer for which the CRC is to be calculated

Return values
The CRC for the nominated block of memory

Remarks
If a CRC is to be calculated incrementally, i.e. across a number of blocks of memory which will make up the total block for which the CRC is required, the first call to BlockCRC should specify a pSourceCRC of zero - subsequent calls to BlockCRC should specify the last value returned by BlockCRC as the starting CRC value. The final value retuned for the last memory block is the CRC for the entire set of memory blocks.

See also
StringCRC, Miscellaneous Interfaces

StringCRC

The StringCRC procedure calculates a CRC (cyclic redundancy check) for an ASCII string of characters.

```
PROCEDURE [PASCAL] StringCRC(VAR pString - : ARRAY OF CHAR) : LONGINT;
```

Parameters

pString	:	A reference string variable containing the character string for which the CRC is required - the string must be null-terminated

Return values
The CRC for the nominated string.

Remarks
The given string must be null terminated, as the string processing algorithm terminates only when the first null character is found.

By implication, this procedure will not work for Unicode strings, as there are often null values in Unicode strings prior to the end of the string.

See also
BlockCRC, Miscellaneous Interfaces

EndActivity

The EndActivity procedure terminates an activity window.

```
PROCEDURE [PASCAL]  EndActivity  ()   :  LONGINT;
```

Parameters
None.

Return values
A standard rc (return code).

Remarks
EndActivity acts on an Activity window which must have been previously declared using the StartActivity procedure.

See also
StartActivity, ShowActivity, Miscellaneous Interfaces

ShowActivity

The Showactivity procedure refreshes an activity window, previously opened using the StartActivity procedure.

```
PROCEDURE [PASCAL]  ShowActivity  (    pCaption    :  ARRAY OF CHAR;
                                       pText       :  ARRAY OF CHAR);
```

Parameters

pCaption	:	An ASCII string defining the window caption for the activity window.
pText	:	An ASCII string to be displayed within the activity window

Return values
None

Remarks
An activity window must currently be open for this procedure to have any effect.

The pCaption string may be left empty (null in the first character) if the previous caption is to be left unchanged.

The text provided in the pText parameter will be displayed, centred, in the activity window.

The activity window is closed by using the EndActivity procedure.

See also
StartActivity, EndActivity, Miscellaneous Interfaces

StartActivity

The StartActivity procedure opens an activity window, used to indicate process activity to the user.

```
PROCEDURE  [PASCAL]    StartActivity (
               pXpos, pYpos, pWidth, pHeight, Alpha : LONGINT
                                     )  :  LONGINT;
```

Parameters

pXpos	:	Window X position (left) in pixels
pYpos	:	Window Y position (top) in pixels
pWidth	:	Window width, in pixels
pHeight	:	Window height, in pixels
Alpha	:	Alphablend ratio (transparency) - 0 is totally opaque, and 100 is totally transparent

Return values
A standard rc (return code).

Remarks
Only one activity window can be opened by any one thread at any one time. The window is refreshed using the ShowActivity procedure, and terminated (closed) with the EndActivity procedure.

See also
ShowActivity, EndActivity, Miscellaneous Interfaces

MessageBox

The MessageBox procedure displays an advanced, formatted message box with up to 4 custom label, automatically sized buttons, and returns the user's button selection to the caller.

```
PROCEDURE  [PASCAL] MessageBox (    pApplication      :  ARRAY OF CHAR;
                                    pCaption          :  ARRAY OF CHAR;
                                    pMsgText          :  ARRAY OF CHAR;
                                    pOwnerHandle      :  LONGINT;
                                    pXPos, pYPos      :  LONGINT;
                                    pWidth, pHeight   :  LONGINT;
                                    pButton1          :  LONGINT;
                                    pButton1Text      :  ARRAY OF CHAR;
                                    pButton2          :  LONGINT;
                                    pButton2Text      :  ARRAY OF CHAR;
                                    pButton3          :  LONGINT;
                                    pButton3Text      :  ARRAY OF CHAR;
                                    pButton4          :  LONGINT;
                                    pButton4Text      :  ARRAY OF CHAR;
                                    pImageIndex       :  LONGINT
                                )   :  WinDef.HWND;
```

Parameters

pApplication	:	The name of the application invoking the message box (used for tracking purposes)
pCaption	:	The requested caption of the message box
pMsgText	:	The text to be displayed in the message box
pOwnerHandle	:	The handle of the owner window
pXPos, pYPos	:	The X and Y (pixel) position for the message box
pWidth, pHeight	:	The requested width and height of the message window
pButton1	:	A numeric constant to be returned if this button is pressed
pButton1Text	:	The required text for the button
pButton2	:	A numeric constant to be returned if this button is pressed
pButton2Text	:	The required text for the button
pButton3	:	A numeric constant to be returned if this button is pressed
pButton3Text	:	The required text for the button
pButton4	:	A numeric constant to be returned if this button is pressed
pButton4Text	:	The required text for the button
pImageIndex	:	Not implemented at this time - reserved for future use. Should be set to zero (0).

Return values
The handle of the message box window

Remarks
The buttons to be used should be assigned from button 1 sequentially. For buttons not required, the button constant should be set to 0 and the text to a null string (‘’).

See also
Miscellaneous Interfaces

ExtractFileName

The ExtractFileName procedure returns the filename from a full filepath string

```
PROCEDURE [PASCAL] ExtractFileName (         pFilePath   :  ARRAY OF CHAR;
                                      VAR    pFileName   :  ARRAY OF CHAR
                                    );
```

Parameters

pFilePath	:	The full filepath containing the required filename
pFileName	:	A reference string to receive the extracted filename

Return values
None

Remarks
The filename and extension are extracted and copied to the pFileName reference string.

Ensure that the reference string is long enough to receive the filename.

See also
Miscellaneous Interfaces

SplitFullName

The SplitFullName procedure divides a complete filepath string into the separate portions : drive, path, filename, extension.

```
PROCEDURE [PASCAL] SplitFullName (        pFilePath   :  ARRAY OF CHAR;
                                    VAR   pDrive      :  CHAR;
                                    VAR   pPath       :  ARRAY OF CHAR;
                                    VAR   pFileName   :  ARRAY OF CHAR;
                                    VAR   pExtension  :  ARRAY OF CHAR;
                                    VAR   pUNC        :  ARRAY OF CHAR
                                   );
```

Parameters

pFilePath	:	The full filepath to be split
pDrive	:	A reference character variable to receive the drive character
pPath	:	A reference string to receive the folder path, excluding the drive character
pFileName	:	A reference string to receive the filename, excluding extension
pExtension	:	A reference string to receive the file extension
pUNC	:	A reference string to receive the UNC computer specification

Return values

None

Remarks

If the path is a local path, the drive character is returned in the pDrive character, and the pUNC reference string is set to a null string.

If the path is a UNC path, the UNC computer location is returned in the pUNC string, and the pDrive character is set to null (0).

See also

Miscellaneous Interfaces

Lava Backup System

The Lava Backup System is presented through a default interface which permits backup and restore of individual schemas. The Backup API is presented to permit advanced programmers to write more specific backup and restore routines where there is such a requirement.

CreateBackupSet	
BackupObjectData	
FinaliseBackup	
OpenBackupSet	
RestoreObjectData	
CloseBackupSet	
SetBackupFolder	

See also
API Categories

CreateBackupSet

The CreateBackupSet procedure is the primary or first call in a sequence of procedure calls required to define a Lava backup. CreateBackupSet initializes the new backup and names the set.

```
PROCEDURE [PASCAL] CreateBackupSet(
                        pSession_id       :  LONGINT;
                   VAR  pDatasetHandle    :  LONGINT;
                        pEncryptionKey    :  ARRAY OF CHAR;
                        pDataSet          :  ARRAY OF CHAR;
                        pSchema           :  ARRAY OF CHAR;
                        pbAppendTimestamp :  BOOLEAN
                                  )  :   LONGINT;
```

Parameters

pSession_id	:	A valid session ID
pDatasetHandle	:	A reference longint (dword) variable which receives the new data set handle
pEncryptionKey	:	A string specification for the encryption key to be applied to the backup
pDataSet	:	A string identifier for the backup set
pSchema	:	The schema to which the tables to be backed up belong
pbAppendTimestamp	:	A boolean flag indicating whether the current Date Time should be appended to the dataset name

Return values
A standard rc (return code).

Remarks
The nominated session must have adequate access privilege to be allowed to read all tables to be nominated for the backup set.

Only one backup set can be created by a given client application at any one time - i.e. CreateBackupSet must be followed by 1 or more calls to BackupObjectData, followed by a call to FinaliseBackup. Different clients may execute backups simultaneously to one another.

If no encryption is required, the encryption key may be set to a null string.

The encryption key will be stored in an encrypted form within the backup header, and the backup set will only be restorable if the correct encryption key is specified. Care should be taken to ensure that the intended encryption key is not accidentally mis-typed by the user, as such a backup set cannot be restored.

The backup set is created in the folder nominated as the backup folder for the active client - see SetBackupFolder. If no default backup folder has been set, the procedure aborts and returns an error code.

The dataset name specified in pDataSet must be unique across all currently existing backups in the current backup folder.

The pSchema parameter should specify the schema to which the backup tables belong. Although it is possible to create a backup set spanning schemas, as a rule this is discouraged as it becomes more difficult to select a valid and contained backup set if the tables in the backup are not clearly delineated and belong to a single functional group, which is typically limited by schema boundaries.

If pbAppendTimestamp is TRUE, the current server date time (see GetServerDateTime) is appended to the nominated data set name. This will, under all circumstances, ensure that the dataset name is unique.
Any occurrences of the characters '/' (forward slash) or ':' (colon) in the specified dataset name will be replaced

by '_' (underscore) as these characters are illegal within Windows filenames.

The backup file created will automatically acquire the extension **.LBS**, for Lava Backup Set.

See also
BackupObjectData, FinaliseBackup, OpenBackupSet, RestoreObjectData, CloseBackupSet, SetBackupFolder, Lava Backup System

Example code
Backup Set Creation

BackupObjectData

The BackupObjectData procedure includes a given object (table) into a backup set currently being created.

```
PROCEDURE [PASCAL] BackupObjectData    (     pSession_id       :  LONGINT;
                                             pObject_id        :  LONGINT;
                                             pSystem           :  BOOLEAN;
                                             pForce_id         :  BOOLEAN;
                                             pEncrypted        :  BOOLEAN
                                          )  :   LONGINT;
```

Parameters

pSession_id	:	A valid session ID
pObject_id	:	The object ID representing the table to be included in the backup set
pSystem	:	(not for application programmer use - should be set to FALSE)
pForce_id	:	(not for application programmer use - should be set to FALSE)
pEncrypted	:	A boolean indicator which enables or disables data encryption for this table

Return values
A standard rc (return code).

Remarks
BackupObjectData may only be invoked after CreateBackupSet has been called and has returned a zero return code. If this is not the case, the results from this call will be unpredictable and probably unusable by subsequent backup procedures.. See the example code for a typical backup sequence.

A maximum of 100 tables may be included in a single backup - an attempt to include more than 100 tables will result in an error return, and no action will be taken.

For Client mount (the default mount method for a client application) the pObject_id must refer to a table which has been distributed from the server (see Distributed Client Operation). If not, the backup image for this table will be empty and probably invalid.

If the pEncrypted flag is set to FALSE (0), the image for this table within the backup will be unencrypted regardless of whether an encryption key was asserted in the call to CreateBackupSet.

See also
CreateBackupSet, FinaliseBackup, OpenBackupSet, RestoreObjectData, CloseBackupSet, SetBackupFolder, Lava Backup System

Example code
Backup Set Creation

FinaliseBackup

The FinaliseBackup procedure actually performs the requested backup based on information supplied through CreateBackupSet and BackupObjectData.

```
PROCEDURE [PASCAL] FinaliseBackup (  pDatasetHandle    :  LONGINT
                                  )  :   LONGINT;
```

Parameters

pDatasetHandle : The handle to the current backup dataset as returned by CreateBackupSet

Return values

A standard rc (return code).

Remarks

The backup set must be valid and have been set up correctly - see the example code for a valid setup sequence.

The FinaliseBackup procedure will extract the data from the nominated tables included in the backup, and create the backup set in the nominated Windows file.

The backup set is compressed using the Lava compression engine, and individual table images are encrypted if this was specified in earlier calls.

See also

CreateBackupSet, BackupObjectData, OpenBackupSet, RestoreObjectData, CloseBackupSet, SetBackupFolder, Lava Backup System

Example code

Backup Set Creation

OpenBackupSet

The OpenBackupSet procedure opens an existing backup set in order to execute a restore of the tables included in the backup.

```
PROCEDURE [PASCAL] OpenBackupSet(
                     pDataSet             : ARRAY OF CHAR;
                VAR  pDatasetHandle       : LONGINT;
                VAR  pBackupHeader        : BackupSetType;
                VAR  pBackupObjectArray   : ObjectArrayType;
                     pEncryptionKey       : ARRAY OF CHAR
                                  )  :   LONGINT;
```

Parameters

pFileName	:	The Windows filename for the required backup set.
pDatasetHandle	:	A reference longint (dword) variable which receives the handle of the backup set
pBackupHeader	:	A reference structure which receives the backup header for the nominated backup
pBackupObjectArray	:	A reference array of structures which receives the backup object array
pEncryptionKey	:	A string specification for the encryption key to be applied to the backup

Return values
A standard rc (return code).

Remarks
The backup folder must be correctly set to the folder containing the required backup set, using the SetBackupFolder procedure.

If the nominated backup set as specified in the pDataSet parameter is not found in the current backup folder, an error code is returned and the open fails.

The encryption key specified in the pEncryptionKey parameter must concur exactly (including character case) with the encryption key specified when the backup set was created using the CreateBackupSet procedure.

The object array returned in the pBackupObjectArray parameter specifies the objects included in the backup.

See also
CreateBackupSet, BackupObjectData, FinaliseBackup, RestoreObjectData, CloseBackupSet, SetBackupFolder, Lava Backup System

Example code
Backup Set Restore

RestoreObjectData

The RestoreObjectData procedure restores a single table from the backup set currently open after calling the OpenBackupSet procedure.

```
PROCEDURE [PASCAL] RestoreObjectData (
                        pSession_id          :  LONGINT;
                        pDatasetHandle       :  LONGINT;
                  VAR   pBackupHeader        :  BackupSetType;
                  VAR   pBackupObjectArray   :  ObjectArrayType;
                        pObjectIndex         :  LONGINT;
                        pTruncate            :  BOOLEAN
                                          )  :   LONGINT;
```

Parameters

pSession_id	:	A valid session ID
pDatasetHandle	:	The handle of an open backup set
pBackupHeader	:	The backup header for the open backup set
pBackupObjectArray	:	The object array for the open backup set
pObjectIndex	:	The 1-based index of the object to be restored from the backup
pTruncate	:	A boolean flag indicating whether the target table is to be truncated (alternatively dropped)

Return values
A standard rc (return code).

Remarks
The backup set must be currently open for the restore to operate.

The nominated object index must be a valid index for an object contained in the backup.

If the pTruncate option is TRUE, the target table is merely truncated and the backup data restored to the existing table. For this option to work, it is essential that the table layout (column layout) be identical to that at the time the backup was made. If this is not the case, the pTruncate option must be set to FALSE, to allow the backup to re-create the table in order for the column layout to match the backup data.

If the table is re-created during the restore process, the object ID for the table is likely to change as a result of the restore.

Once all the required tables have been restored, the backup set should be closed using the CloseBackupSet procedure.

See also
CreateBackupSet, BackupObjectData, FinaliseBackup, OpenBackupSet, CloseBackupSet, SetBackupFolder, Lava Backup System

Example code
Backup Set Restore

CloseBackupSet

The CloseBackupSet procedure closes a backup previously opened for restore purposes.

```
PROCEDURE [PASCAL] CloseBackupSet(pDatasetHandle    :  LONGINT);
```

Parameters

pDatasetHandle	:	The handle of an open backup set

Return values
A standard rc (return code).

Remarks
If the backup set is open, the procedure closes the backup set and returns 0. If the provided handle is not valid, the procedure fails and returns an error code.

See also
CreateBackupSet, BackupObjectData, FinaliseBackup, OpenBackupSet, RestoreObjectData, SetBackupFolder, Lava Backup System

Example code
Backup Set Creation

SetBackupFolder

The SetBackupFolder procedure sets the current (default) backup folder, in which any backup set initialized by CreateBackupSet or opened by OpenBackupSet will be located.

```
PROCEDURE [PASCAL] SetBackupFolder(pFolder  : ARRAY OF CHAR);
```

Parameters

pFolder	:	A complete Windows file path

Return values
A standard rc (return code).

Remarks
The Lava backup folder is unconditionally set to the folder path specified - the onus is on the caller to ensure that the path is valid prior to creating or opening a backup set.

See also
CreateBackupSet, BackupObjectData, FinaliseBackup, OpenBackupSet, RestoreObjectData, CloseBackupSet, Lava Backup System

Example code
Backup Set Creation, Backup Set Restore

Lava DataGrid Control

See also
API Categories

CreateGrid

```
PROCEDURE [PASCAL] CreateGrid (         pParent              :  LONGINT;
                                        pGridHandle          :  LONGINT;
                                        pXPos                :  LONGINT;
                                        pYPos                :  LONGINT;
                                        pWidth               :  LONGINT;
                                        pHeight              :  LONGINT;
                                        pUserSession_id      :  LONGINT;
                                        pObjectSession_id    :  LONGINT;
                                        pObject_id           :  LONGINT;
                                        pGridIdent           :  LONGINT;
                                        pColumnHeader        :  BOOLEAN;
                                        pRowIndicator        :  BOOLEAN;
                                        pGridLines           :  BOOLEAN
                                    )  :  LONGINT (* Grid handle *);
```

Parameters

Return values
The handle of the Grid window

Remarks

RefreshGrid

```
PROCEDURE [PASCAL] RefreshGrid(     pGridHandle       :  LONGINT;
                                    pObject_id        :  LONGINT;
                                    pSession_id       :  LONGINT;
                                    pInstanceIdent    :  LONGINT
                                ) : LONGINT;
```

Parameters

Return values
A standard rc (return code).

Remarks

SetColumnWidth

```
PROCEDURE [PASCAL] SetColumnWidth(     pGridHandle       :  LONGINT;
                                       pColumnSequence   :  LONGINT;
                                       pWidth            :  LONGINT);
```

Parameters

Return values
None

Remarks

SetColumnTitle

```
PROCEDURE [PASCAL] SetColumnTitle(     pGridHandle       :  LONGINT;
                                       pColumnSequence   :  LONGINT;
                                       pTitle            :  ARRAY OF CHAR);
```

Parameters

Return values
None

Remarks

SetGridRow

```
PROCEDURE [PASCAL] SetGridRow(   pGridHandle :  LONGINT;
                                 pRow        :  LONGINT;
                                 pHighlight  :  BOOLEAN;
                                 pNotify     :  BOOLEAN;
                                 pWrap       :  BOOLEAN
                                );
```

Parameters

Return values
None

Remarks

GetGridRow

```
PROCEDURE [PASCAL] GetGridRow(         pGridHandle :  LONGINT;
                                VAR    pGridRow    :  LONGINT;
                                VAR    pTableRow   :  LONGINT
                               ) : LONGINT;
```

Parameters

Return values
A standard rc (return code).

Remarks

SetColumnVisible

```
PROCEDURE [PASCAL] SetColumnVisible (     pGridHandle       :  LONGINT;
                                          pColumnSequence   :  LONGINT;
                                          pbVisible         :  BOOLEAN
                                    );
```

Parameters

Return values
None

Remarks

Lava Compression

The Lava Compression library is presented to allow the user to perform fast, high-ratio compression on arbitrary memory segments. Compression ratios are comparable with those achieved in archiving packages like WinZip, although typically about 5% to 10% lower. However, the speed of compression is very high - on modern workstations, the compression algorithm will compress and decompress 1 Mb of data in approximately 0.3 seconds.

See also
API Categories

Compress

The Compress procedure accepts the address and size of a memory segment, and compresses (and optionally encrypts) the data into a target memory segment which is allocated on the heap.

```
PROCEDURE [PASCAL] Compress (
                        pSourceAddress       :  LONGINT;
                        pSourceSize          :  LONGINT;
                  VAR   pTargetAddress       :  LONGINT;
                  VAR   pTargetSize          :  LONGINT;
                        pHeap                :  LONGINT;
                        pKey                 :  LavaRunX.QUADINTEGER
                             ) : LONGINT;
```

Parameters

pSourceAddress	:	The base address of the memory segment to be compressed
pSourceSize	:	The size (in bytes) of the source memory segment
pTargetAddress	:	A reference longint which receives the address of the compressed segment
pTargetSize	:	The size (in bytes) of the compressed data
pHeap	:	A valid handle to a Windows heap which is large enough to contain the compressed data
pKey	:	The encryption key (an 8-byte integer) - optional

Return values
A standard rc (return code).

Remarks

The source segment must be in memory at the time of compression.

The target data will be allocated on the heap provided - therefore, this specified heap handle must have sufficient free space to accept the compressed data.

If encryption of the target data is not required, the encryption key (pKey) should be specified as 0.

Decompress

The Decompress procedure is provided to allow decompression of data compressed using the Compress algorithm.

```
PROCEDURE [PASCAL] Decompress  (
                       pSourceAddress       :  LONGINT;
                       pSourceSize          :  LONGINT;
                 VAR   pTargetAddress       :  LONGINT;
                       pTargetSize          :  LONGINT;
                       pHeap                :  LONGINT;
                       pKey                 :  LavaRunX.QUADINTEGER
                                  ) : LONGINT;
```

Parameters

pSourceAddress	:	The base address of the compressed data segment
pSourceSize	:	The size (in bytes) of the compressed data
pTargetAddress	:	A reference longint which receives the address of the decompressed data
pTargetSize	:	The size (in bytes) of the decompressed data, provided by the caller
pHeap	:	A valid handle to a Windows heap which is large enough to contain the decompressed data
pKey	:	The encryption key (an 8-byte integer)

Return values
A standard rc (return code).

Remarks
The caller must provide the target size of the decompressed data. This avoid the decompression algorithm having to re-allocate memory or allocate more memory than required, thereby improving performance and memory utilization.

The heap handle provided must have enough free space for the uncompressed data - since compression ratios can be very high in some cases, it is important to ensure that this is true.

The encryption key provided must be the same as when compression was performed, or the decompression will fail (or, in the worst case, cause memory violations due to an invalid compressed image after incorrect decryption)

Lava Editor Control

The Lava Editor Control is a self-contained control presenting an advanced text editor within a specified window. The control has the ability to load and save Windows text files, and has all the features that would be required from a programming editor.

AppendText	
ClearContent	
ClearSelection	
CloseEditWindow	
Copy	
Cut	
GetSelectedText	
GotoOffset	
GotoPos	
LoadFile	
NewEditWindow	
Paste	
Replace	
ResetAllBookmarks	
ResetBookmark	
ResizeWindow	
SaveFile	
Search	
SearchFiles	
SelectSegment	
SetBookmark	
SetGutter	
SetScrollBar	
TextExtract	
TextModified	

See also
API Categories

TextExtract

The TextExtract procedure extracts the entire textual content of a nominated edit window to an allocated memory space.

```
PROCEDURE [PASCAL] TextExtract (          hEdit              :  WinDef.HWND;
                                          pHeapHandle        :  LONGINT;
                                    VAR   pVirtualAddress    :  LONGINT;
                                    VAR   pBytes             :  LONGINT
                                    ) : LONGINT;
```

Parameters

hEdit	:	The handle of an existing edit window
pHeapHandle	:	A valid Windows heap handle, which will be used for allocating the memory to receive the extracted text
pVirtualAddress	:	A reference longint (dword) variable which receives the address of the allocated memory
pBytes	:	The total size of the extracted text in bytes

Return values
A standard rc (return code).

Remarks
The edit handle supplied must be current.

The heap handle supplied must have sufficient free space to accommodate the full text to be extracted. If no other heap handle is available, the caller may use the Windows API function GetProcessHeap() which returns the current process heap handle.

The onus is on the caller to free the allocated memory once it is no longer required - this must be done using the Windows API function HeapFree, and the same heap handle must be used as was passed to TextExtract

See also
Lava Editor Control

AppendText

The AppendText procedure appends nominated text to the specified edit window at the end of any existing text.

```
PROCEDURE [PASCAL] AppendText (           hEdit    :  WinDef.HWND;
                                    VAR   pText  - :  ARRAY OF CHAR
                                    )  : LONGINT;
```

Parameters

hEdit	:	The handle of an existing edit window
pText	:	A reference string variable containing null-terminated text

Return values
A standard rc (return code).

Remarks
The nominated text may contain linefeed characters (10H) to delimit text lines.

See also
Lava Editor Control

ClearContent

The ClearContent procedure clears an edit window.

```
PROCEDURE [PASCAL] ClearContent   (hEdit : WinDef.HWND) : LONGINT;
```

Parameters

hEdit : The handle of an existing edit window

Return values

A standard rc (return code).

Remarks

The content of the edit window is irretrievably lost

See also

Lava Editor Control

SearchFiles

The SearhFiles procedure provides a mechanism for searching an arbitrary number of files in a folder or a folder tree, with wildcard features provided.

```
PROCEDURE [PASCAL] SearchFiles(  pSession_id          :  LONGINT;
                                 pSearchFolder        :  ARRAY OF CHAR;
                                 bSearchSubfolders    :  BOOLEAN;
                                 pFileSpec            :  ARRAY OF CHAR;
                                 pSearchString        :  ARRAY OF CHAR;
                                 pMatchCount          :  LONGINT;
                                 bMatchCase           :  BOOLEAN;
                                 bNewSearch           :  BOOLEAN
                               ) :  BOOLEAN;
```

Parameters

pSession_id	:	A valid session ID
pSearchFolder	:	The folder within which the file(s) to search are located
pSearchSubfolders	:	A boolean flag indicating whether subfolders to the nominated folder in pSearchFolder should be searched
pFileSpec	:	A (potentially wildcarded) file specification for the file(s) to be searched
pSearchString	:	The string to be searched for, with provision for wildcards
pMatchCount	:	The maximum number of matches (hits) to be permitted in each file searched
bMatchCase	:	A boolean flag indicating whether the search is case sensitive
bNewSearch	:	A boolean flag indicating whether the match table is to be truncated prior to commencing the search

Return values
TRUE (1) if any hits are found for the nominated search in any of the nominated files.

Remarks
The search string may contain wildcards. Supported wildcards are * (star) which represents any number of any characters, and ? (question mark) representing any one character.

The matches (if any are found) are added to the SearchMatch table in the Util schema (see Util.Searchmatch for details of the match table layout).

If bNewSearch is TRUE, the Util.Searchmatch table is truncated before the search is commenced.

See also
Lava Editor Control

Search

The Search procedure searches the text in the nominated edit window from the cursor position to the first match.

```
PROCEDURE [PASCAL] Search  (     hEdit                :  WinDef.HWND;
                                 pSearch              :  ARRAY OF CHAR;
                                 pMatchcase           :  BOOLEAN;
                                 pDown                :  BOOLEAN;
                                 pWholeWords          :  BOOLEAN
                               ) : BOOLEAN;
```

Parameters

hEdit	:	The handle of an existing edit window
pSearch	:	The string to be searched for, with provision for wildcards
pMatchCase	:	A boolean flag indicating whether the search is case sensitive
pDown	:	A boolean flag indicating the search direction
pWholeWords	:	A boolean flag indicating whether only whole words are a valid match

Return values
TRUE (1) if any hits are found for the nominated search.

Remarks
The search string may contain wildcards. Supported wildcards are * (star) which represents any number of any characters, and ? (question mark) representing any one character.

If the pDown parameter is TRUE, the search is performed downward in the text.

If the pWholeWords parameter is TRUE, only complete words are considered a match.

See also
Lava Editor Control

Replace

The Replace procedure performs a search-and-replace on the nominated edit window

```
PROCEDURE [PASCAL] Replace (     hEdit          :  WinDef.HWND;
                                 pSearch        :  ARRAY OF CHAR;
                                 pReplace       :  ARRAY OF CHAR;
                                 pMatchcase     :  BOOLEAN;
                                 pDown          :  BOOLEAN;
                                 pWords         :  BOOLEAN;
                                 pReplaceAll    :  BOOLEAN;
                                 pbConfirm      :  BOOLEAN
                               ) : LONGINT;
```

Parameters

hEdit	:	The handle of an existing edit window
pSearch	:	The string to be searched for, with provision for wildcards
pReplace	:	The replacement string
pMatchCase	:	A boolean flag indicating whether the search is case sensitive
pDown	:	A boolean flag indicating the search direction
pWords	:	A boolean flag indicating whether only whole words are a valid match
pReplaceAll	:	A boolean flag indicating whether the replacement is only for the first or for all matched occurrences
pbConfirm	:	A boolean flag indicating whether a confirmation dialog must be invoked to request confirmation of each potential replacement

Return values
The number of replacements executed

Remarks
The search string may contain wildcards. Supported wildcards are * (star) which represents any number of any characters, and ? (question mark) representing any one character.

If the pDown parameter is TRUE, the search is performed downward in the text.

If the pWords parameter is TRUE, only complete words are considered a match.

If pReplaceAll is TRUE, the replacement algorithm continues to the end of the text; if FALSE only the first match occurrence is replaced

If pbConfirm is TRUE the Replace procedure invokes a confirmation dialog for each match (potential replacement) for user confirmation.

See also
Lava Editor Control

GotoPos

The GotoPos procedure places the cursor at the required row and column in the text.

```
PROCEDURE [PASCAL] GotoPos (        hEdit          :  WinDef.HWND;
                                    pRow           :  LONGINT;
                                    pColumn        :  LONGINT
                                ) : INTEGER;
```

Parameters

hEdit	:	The handle of an existing edit window
pRow	:	The required row
pColumn	:	The required column

Return values
A standard rc (return code).

Remarks
If either the row or column is invalid (less than zero or greater than the highest row or column) the cursor is placed at the end of the text.

See also
Lava Editor Control

GotoOffset

The GotoOffset procedure places the cursor at the given byte offset into the text (from the first character)

```
PROCEDURE [PASCAL] GotoOffset (     hEdit           :  WinDef.HWND;
                                    pOffset         :  LONGINT
                                ) : LONGINT;
```
(* The edit caret is placed at the byte offset pOffset from row 1, column 1. *)

Parameters

hEdit	:	The handle of an existing edit window
pOffset	:	The required byte offset

Return values
A standard rc (return code).

Remarks
The byte offset is the offset from the first byte (row 1 column 1) of the text. The offset includes linefeed characters at the end of each line.

See also
Lava Editor Control

Copy

The Copy procedure copies the selected region in the text into the Windows clipboard.

```
PROCEDURE [PASCAL] Copy (hEdit : WinDef.HWND) : LONGINT;
```

Parameters

hEdit	:	The handle of an existing edit window

Return values
A standard rc (return code).

Remarks
If no text is currently selected in the nominated edit window, the procedure has no effect and the clipboard is left unchanged.

See also
Lava Editor Control

GetSelectedText

The GetSelectedText procedure copies the currently selected text in the nominated edit window into the provided text buffer.

```
PROCEDURE [PASCAL] GetSelectedText(          hEdit       :  WinDef.HWND;
                                      VAR    pBuffer     :  ARRAY OF CHAR
                                   ) : LONGINT;
```

Parameters

hEdit	:	The handle of an existing edit window
pBuffer	:	A reference text buffer to receive the copied text

Return values
A standard rc (return code).

Remarks
The provided buffer must be long enough to receive the selected text.

If no text is currently selected in the nominated edit window, the provided text buffer is left unchanged.

See also
Lava Editor Control

SelectSegment

The SelectSegment procedure selects the text as delimited in the nominated edit window.

```
PROCEDURE [PASCAL] SelectSegment(
              hEdit        :  LONGINT;
              pStartRow, pStartColumn, pEndRow, pEndColumn : LONGINT);
```

Parameters

hEdit	:	The handle of an existing edit window
pStartRow	:	The starting row for the selection region
pStartColumn	:	The starting column for the selection region
pEndRow	:	The end row for the selection region
pEndColumn	:	The end column for the selection region

Return values
None

Remarks
If any of the specified row or column boundaries are invalid, the selection request is ignored.

See also
Lava Editor Control

Paste

The Paste procedure pastes the content of the clipboard into the text of the nominated edit window at the current cursor position.

```
PROCEDURE [PASCAL] Paste (hEdit : WinDef.HWND) : LONGINT;
```

Parameters

hEdit	:	The handle of an existing edit window

Return values
A standard rc (return code).

Remarks
If the current clipboard content is not of type text, the procedure has no effect.

See also
Lava Editor Control

Cut

The Cut procedure cuts the currently selected text in the nominated edit window to the clipboard.

```
PROCEDURE [PASCAL] Cut (hEdit : WinDef.HWND) : INTEGER;
```

Parameters

hEdit	:	The handle of an existing edit window

Return values
A standard rc (return code).

Remarks
If no text is currently selected in the nominated edit window, the procedure leaves the clipboard unaffected.

See also
Lava Editor Control

ClearSelection

The ClearSelection procedure deletes the current selection region in the nominated edit window.

```
PROCEDURE [PASCAL] ClearSelection (hEdit : WinDef.HWND) : INTEGER;
```

Parameters

hEdit	:	The handle of an existing edit window

Return values
A standard rc (return code).

Remarks
If no text is currently selected in the nominated edit window, the procedure exits without any changes to the text.

See also
Lava Editor Control

NewEditWindow

The NewEditWindow procedure creates a new edit window with the window limits specified. The window is empty after creation.

```
PROCEDURE [PASCAL] NewEditWindow  (    Parent          :  WinDef.HWND;
                                       bReadOnly       :  BOOLEAN;
                                       pFont           :  WinDef.HWND
                                  ) : WinDef.HWND;
```

(* Creates a new editor window. The document may be flagged as bReadOnly. If the pFont is non-zero
the nominated font is used as the default font for the document *)

Parameters

Parent	:	The handle of the parent window to the created edit window.
bReadOnly	:	A boolean flag indicating whether the edit window is to be editable or read-only
pFont	:	If non-zero, the font for the edit window uses the nominated font handle

Return values
The handle of the new edit window.

Remarks
If the bReadOnly flag is TRUE, no text modification will be supported in the created edit window.

If the pFont parameter is zero, the default edit font is used.

See also
Lava Editor Control

ResizeWindow

The ResizeWindow procedure resizes the nominated edit window to new limits.

```
PROCEDURE [PASCAL] ResizeWindow (      hEdit    :  WinDef.HWND;
                                       pLeft    :  INTEGER;
                                       pTop     :  INTEGER;
                                       pWidth   :  INTEGER;
                                       pHeight  :  INTEGER);
```

Parameters

hEdit	:	The handle of an existing edit window
pLeft	:	The required left (pixel) position
pTop	:	The required top (pixel) position
pWidth	:	The required pixel width
pHeight	:	The required pixel height

Return values
None

Remarks
If any of the specified limits are invalid, the edit window is left unchanged.

See also
Lava Editor Control

SetGutter

The SetGutter procedure specifies whether the nominated edit window should be configured with a gutter for bookmark indicators.

```
PROCEDURE [PASCAL] SetGutter(    hEdit    :  WinDef.HWND;
                                 bGutter  :  BOOLEAN);
```

Parameters

hEdit	:	The handle of an existing edit window
bGutter	:	A boolean flag indicating the requirement for a gutter.

Return values
None

Remarks
If the bGutter value is FALSE, any bookmarks set in the text within the edit window will not be visibly indicated.

See also
Lava Editor Control

SetScrollBar

The SetScrollBar procedure sets the horizontal and vertical scroll bars on or off.

```
PROCEDURE [PASCAL] SetScrollBar( hEdit                       : WinDef.HWND;
                                 pbVertical, pbHorizontal    : BOOLEAN);
```

Parameters

hEdit	:	The handle of an existing edit window
pbVertical	:	A boolean flag indicating the requirement for a vertical scrollbar
pbHorizontal	:	A boolean flag indicating the requirement for a horizontal scrollbar

Return values
None

Remarks
The scroll bars in a new edit window are enabled by default, but can be disabled by setting the respective parameter to FALSE.

See also
Lava Editor Control

CloseEditWindow

The CloseEditWindow procedure closes the nominated edit window.

```
PROCEDURE [PASCAL] CloseEditWindow (hEdit  :  WinDef.HWND);
```

Parameters

hEdit	:	The handle of an existing edit window

Return values
None

Remarks
The nominated edit window is closed regardless of content and modification status - if the content of the window must be saved before closing, the SaveFile must be called explicitly before calling CloseEditWindow.

See also
Lava Editor Control

TextModified

The TextModified procedure returns the current modification status for the nominated edit window.

```
PROCEDURE [PASCAL] TextModified (hEdit :  WinDef.HWND) : BOOLEAN;
```

Parameters

hEdit	:	The handle of an existing edit window

Return values
TRUE (1) if the content of the edit window has changed since last load or save, FALSE (0) otherwise.

Remarks
Modification is considered to be TRUE if any action has taken place which has left the content of the edit window different from when the window was opened using NewEditWindow, or the content was saved using SaveFile.

See also
Lava Editor Control

LoadFile

The LoadFile procedure loads the contents of a specified Windows file into the nominated edit window.

```
PROCEDURE [PASCAL] LoadFile (    hEdit       :  WinDef.HWND;
                                 pFileName   :  WinDef.LPSTR
                               ) : INTEGER;
```

Parameters

hEdit	:	The handle of an existing edit window
pFileName	:	The filename (and path) of the file to be loaded

Return values
A standard rc (return code).

Remarks
The file content is loaded into the edit window regardless of current content - any content prior to the load request is discarded.

See also
Lava Editor Control

SaveFile

The SaveFile procedure saves the content of the nominated edit window to the nominated Windows file.

```
PROCEDURE [PASCAL] SaveFile (    hEdit       :  WinDef.HWND;
                                 pFileName   :  WinDef.LPSTR
                               ) : INTEGER;
```

Parameters

hEdit	:	The handle of an existing edit window
pFileName	:	The filename (and path) of the file to be loaded

Return values
A standard rc (return code).

Remarks
If the specified file does not exist, it is created. If it exists, the content is replaced by the content of the edit window.

After the save request has been successfully executed, the modification status of the edit window is set to FALSE.

See also
Lava Editor Control

SetBookmark

The SetBookmark procedure sets a bookmark in the nominated edit window.

```
PROCEDURE [PASCAL] SetBookmark (          hEdit              :  LONGINT;
                                          pRow               :  LONGINT;
                                          pType              :  LONGINT
                                       ) : LONGINT;
```

Parameters

hEdit	:	The handle of an existing edit window
pRow	:	The row at which a bookmark is to be set
pType	:	The type of bookmark to be set (see Editor Bookmark Types)

Return values
A standard rc (return code).

Remarks
If the specified row is invalid, no action is taken and an error code is returned.

See also
Lava Editor Control

NextBookmark

The NextBookmark procedure moves the cursor (insertion point) in the nominated edit window to the next bookmark downward in the text from the current cursor position.

```
PROCEDURE [PASCAL] NextBookmark (  hEdit             :  LONGINT
                                ) : LONGINT;
```

Parameters

hEdit	:	The handle of an existing edit window

Return values
A standard rc (return code).

Remarks
If no bookmark is found downward in the document, the cursor remains at its current position, and the procedure returns an error code.

See also
Lava Editor Control

PreviousBookmark

The PreviousBookmark procedure moves the cursor (insertion point) in the nominated edit window to the previous bookmark upwards in the text from the current cursor position.

```
PROCEDURE [PASCAL] PreviousBookmark (  hEdit             :  LONGINT
                                    ) : LONGINT;
```

Parameters

hEdit	:	The handle of an existing edit window

Return values
A standard rc (return code).

Remarks
If no bookmark is found upwards in the document, the cursor remains at its current position, and the procedure returns an error code.

See also
Lava Editor Control

ResetBookmark

The ResetBookmark procedure resets a bookmark at the specified row in the nominated edit window.

```
PROCEDURE [PASCAL] ResetBookmark (        hEdit             :  LONGINT;
                                          pRow              :  LONGINT
                                    ) : LONGINT;
```

Parameters

hEdit	:	The handle of an existing edit window
pRow	:	The row at which a bookmark is to be reset

Return values
A standard rc (return code).

Remarks
If no bookmark is found at the specified row, an error code is returned.

See also
Lava Editor Control

ResetAllBookmarks

The ResetAllBookmarks procedure resets any bookmarks currently set in the nominated edit window.

```
PROCEDURE [PASCAL] ResetAllBookmarks (     hEdit       :  LONGINT
                                       ) : LONGINT;
```

Parameters

hEdit	:	The handle of an existing edit window

Return values

A standard rc (return code).

See also

Lava Editor Control

The Lava System Schemas

This chapter describes pertinent details and relations in the Lava system schemas, which may be used by administrators and advanced programmers to obtain information on various aspects of the Lava system.

The Lava database kernel is entirely self-driven, and all information used and managed by the kernel is stored in conventional Lava data tables. There are no custom lists or queues of any sort in the kernel, and information about any aspect of the entire database may be obtained entirely from the system tables, provided that the table layout and necessary relations are well understood.

The Lava kernel itself is divided into several schemas, of which the Backup, Event, Parse and System schemas are described below. The Linker schema, used exclusively by the Lava Linker, is documented in a separate reference (to be released shortly) dealing only with this topic. The remainder of the default kernel schemas are not described in this reference, as they are not utilized by the Lava database kernel itself, and therefore neither influence the operation of the kernel nor have any value in determining system operational attributes.

In all cases below, only tables which have information of value to administrators, programmers or advanced users are described. In each of these schemas, other tables are present which are used by the Lava database kernel, but which do not have information which can easily be interpreted at user level.

Schematic Conventions

In the entity-relation diagrams in this chapter, the following conventions apply :

- The “many” end of a many-to-one relationship is indicated by a ball on the relation line
- Tables pertinent to interpreting the diagram but not central to the theme are presented unexpanded; i.e. only the table name is shown without column details.
- In certain cases a pseudo-column, *Version*, is listed where the detailed column list in the textual documentation will list the individual columns *Row_status*, *ID*, *VDT*, *User_id* and *Cache_id*. The *Version* pseudo-column is a macro in the data dictionary utility which produces the standard identification columns.

The following schema views are provided :

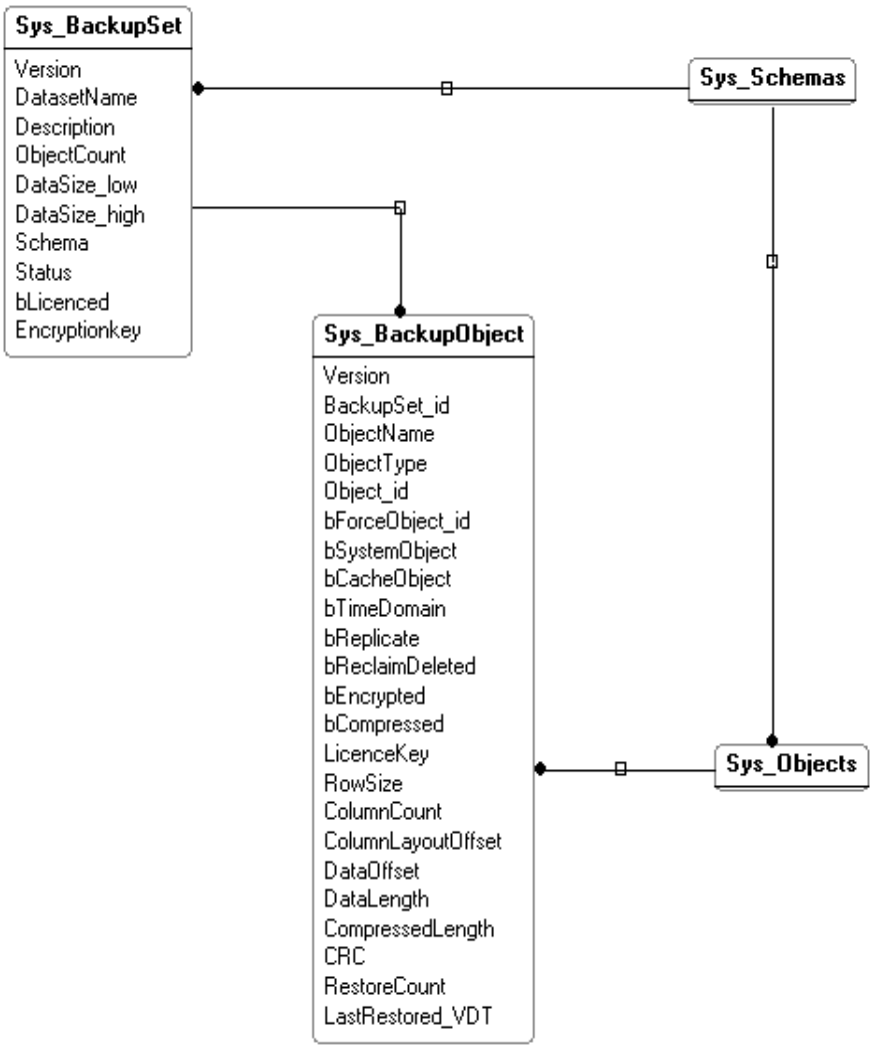
Backup Schema	
Event Schema	
Parse Schema	
System Schema : Relational Integrity	
System Schema : User / Session Tables	
System Schema : Objects / Tables	

The tables described in this section are listed alphabetically below.

Table	Schema	
SQL_ColumnNode	Parse	
SQL_FilterNode	Parse	

SQL_ObjectNode	Parse	
SQL_ParseRoot	Parse	
SQL_PlanList	Parse	
SQL_ValueList	Parse	
Sys_Alias	System	
Sys_BackupObject	Backup	
Sys_BackupSet	Backup	
Sys_Cache	System	
Sys_ColumnBufferPool	System	
Sys_Event_Log	Event	
Sys_Event_Type	Event	
Sys_Locks	System	
Sys_Objects	System	
Sys_ObjectPrivilege	System	
Sys_RelationColumns	System	
Sys_Relations	System	
Sys_Reserve	System	
Sys_Schemas	System	
Sys_Sessions	System	
Sys_Table_Columns	System	
Sys_Tables	System	
Sys_Threads	System	
Sys_Transactions	System	
Sys_UserObjectAccess	System	
Sys_Users	System	

Backup Schema



Sys_BackupObject

Row_status	
ID	
VDT	
User_id	
Cache_id	
BackupSet_id	

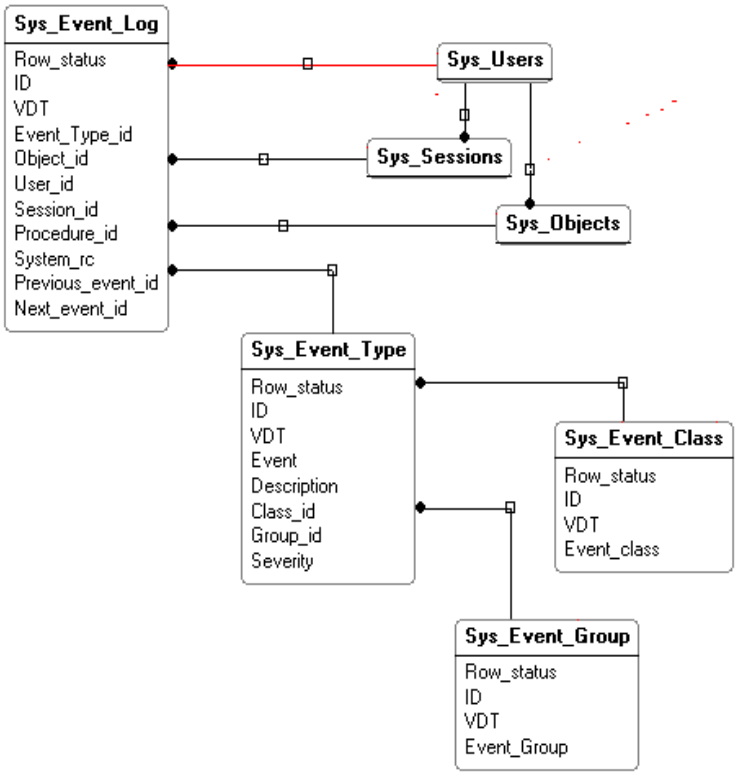
Backup Schema ERD

ObjectName	
ObjectType	
Object_id	
bForceObject_id	
bSystemObject	
bCacheObject	
bTimeDomain	
bReplicate	
bReclaimDeleted	
bEncrypted	
bCompressed	
LicenceKey	
RowSize	
ColumnCount	
ColumnLayoutOffset	
DataOffset	
DataLength	
CompressedLength	
CRC	
RestoreCount	
LastRestored_VDT	

Sys_BackupSet

Row_status	
ID	
VDT	
User_id	
Cache_id	
DatasetName	
Description	

ObjectCount	
DataSize_low	
DataSize_high	
Schema	
Status	
bLicenced	
Encryptionkey	



Event Schema

Sys_Event_Log

Row_status	
ID	
VDT	
Event_Type_id	
Object_id	
User_id	
Session_id	
Procedure_id	
System_rc	

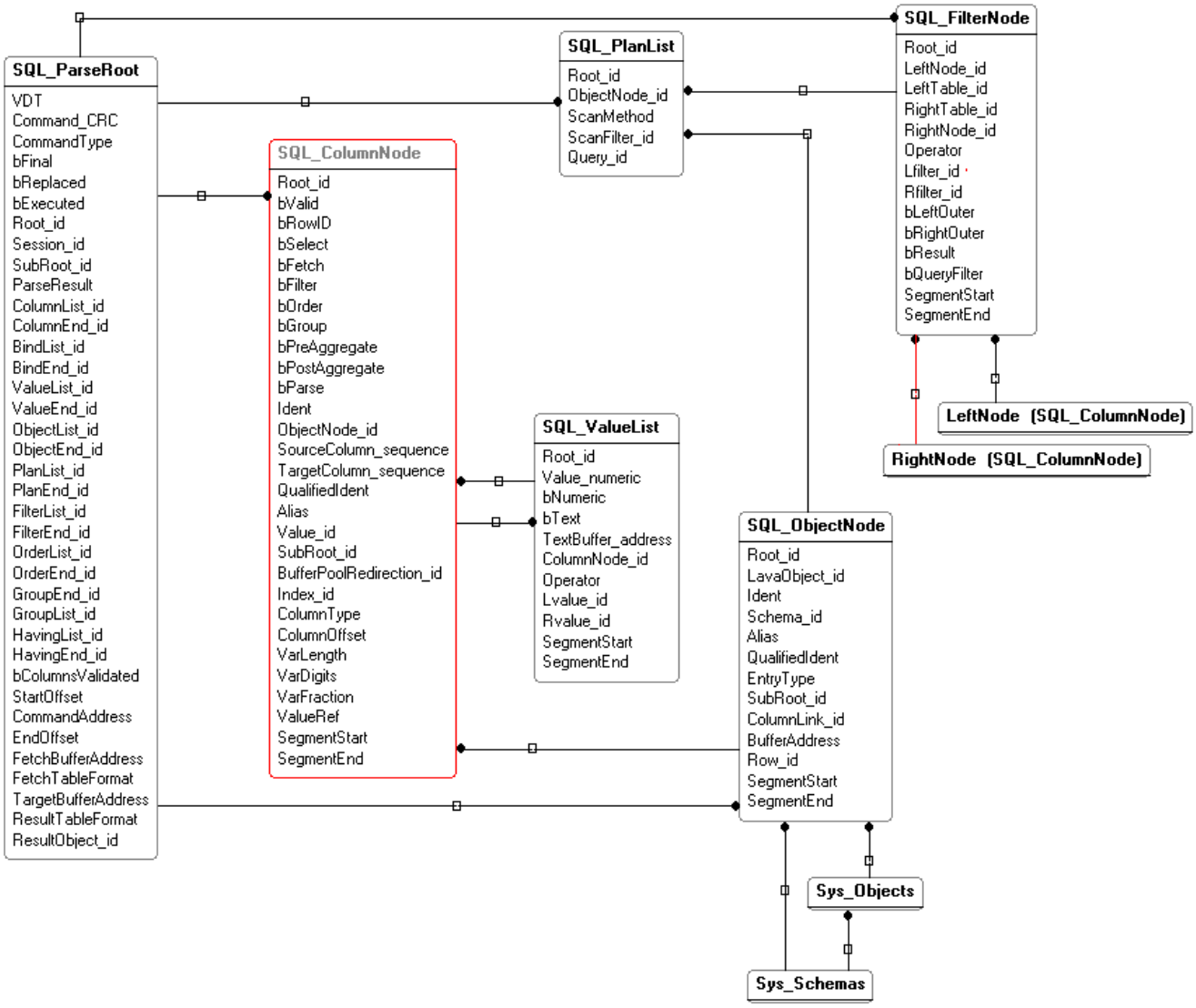
Event Schema ERD

Previous_event_id	
Next_event_id	

Sys_Event_Type

Row_status	
ID	
VDT	
Event	
Description	
Class_id	
Group_id	
Severity	

Parse Schema



SQL_ColumnNode

Root_id	
bValid	
bRowID	
bSelect	
bFetch	

Parse Schema ERD

bFilter	
bOrder	
bGroup	
bPreAggregate	
bPostAggregate	
bParse	
Ident	
ObjectNode_id	
SourceColumn_sequence	
TargetColumn_sequence	
QualifiedIdent	
Alias	
Value_id	
SubRoot_id	
BufferPoolRedirection_id	
Index_id	
ColumnType	
ColumnOffset	
VarLength	
VarDigits	
VarFraction	
ValueRef	
SegmentStart	
SegmentEnd	

SQL_FilterNode

Root_id	
LeftNode_id	
LeftTable_id	
RightTable_id	

RightNode_id	
Operator	
Lfilter_id	
Rfilter_id	
bLeftOuter	
bRightOuter	
bResult	
bQueryFilter	
SegmentStart	
SegmentEnd	

SQL_ObjectNode

Root_id	
LavaObject_id	
Ident	
Schema_id	
Alias	
QualifiedIdent	
EntryType	
SubRoot_id	
ColumnLink_id	
BufferAddress	
Row_id	
SegmentStart	
SegmentEnd	

SQL_ParseRoot

VDT	
Command_CRC	

CommandType	
bFinal	
bReplaced	
bExecuted	
Root_id	
Session_id	
SubRoot_id	
ParseResult	
ColumnList_id	
ColumnEnd_id	
BindList_id	
BindEnd_id	
ValueList_id	
ValueEnd_id	
ObjectList_id	
ObjectEnd_id	
PlanList_id	
PlanEnd_id	
FilterList_id	
FilterEnd_id	
OrderList_id	
OrderEnd_id	
GroupEnd_id	
GroupList_id	
HavingList_id	
HavingEnd_id	
bColumnsValidated	
StartOffset	
CommandAddress	
EndOffset	
FetchBufferAddress	
FetchTableFormat	

TargetBufferAddress	
ResultTableFormat	
ResultObject_id	

SQL_PlanList

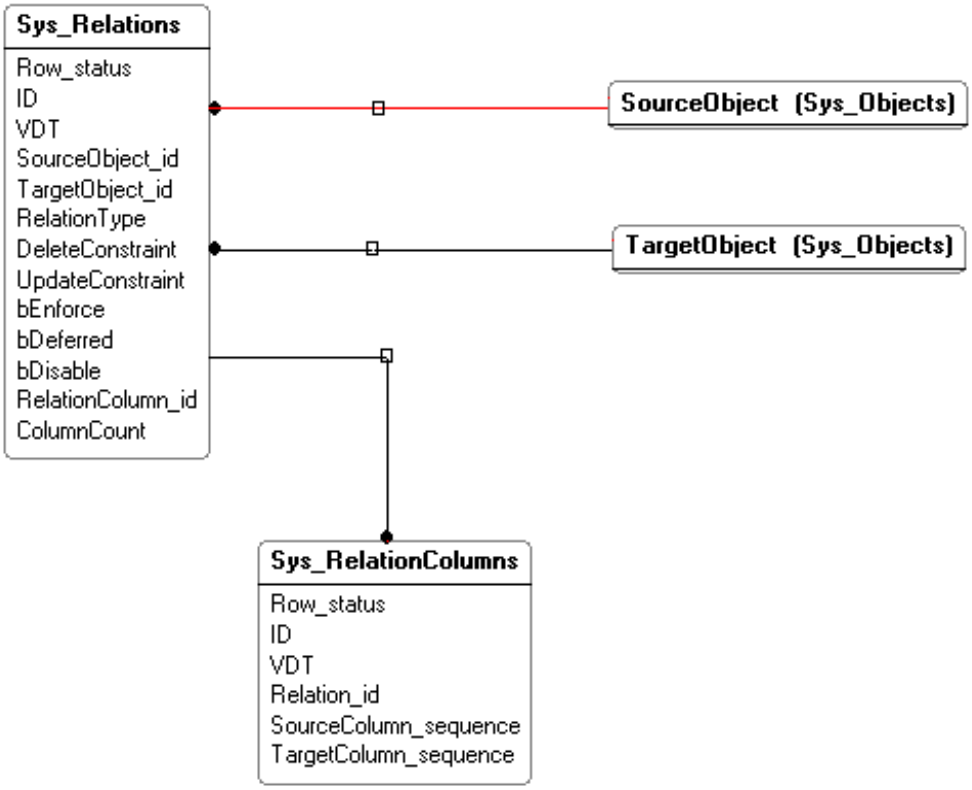
Root_id	
ObjectNode_id	
ScanMethod	
ScanFilter_id	
Query_id	

SQL_ValueList

Root_id	
Value_numeric	
bNumeric	
bText	
TextBuffer_address	
ColumnNode_id	
Operator	
Lvalue_id	
Rvalue_id	
SegmentStart	
SegmentEnd	

System Schema

Relational Integrity



Sys_RelationColumns

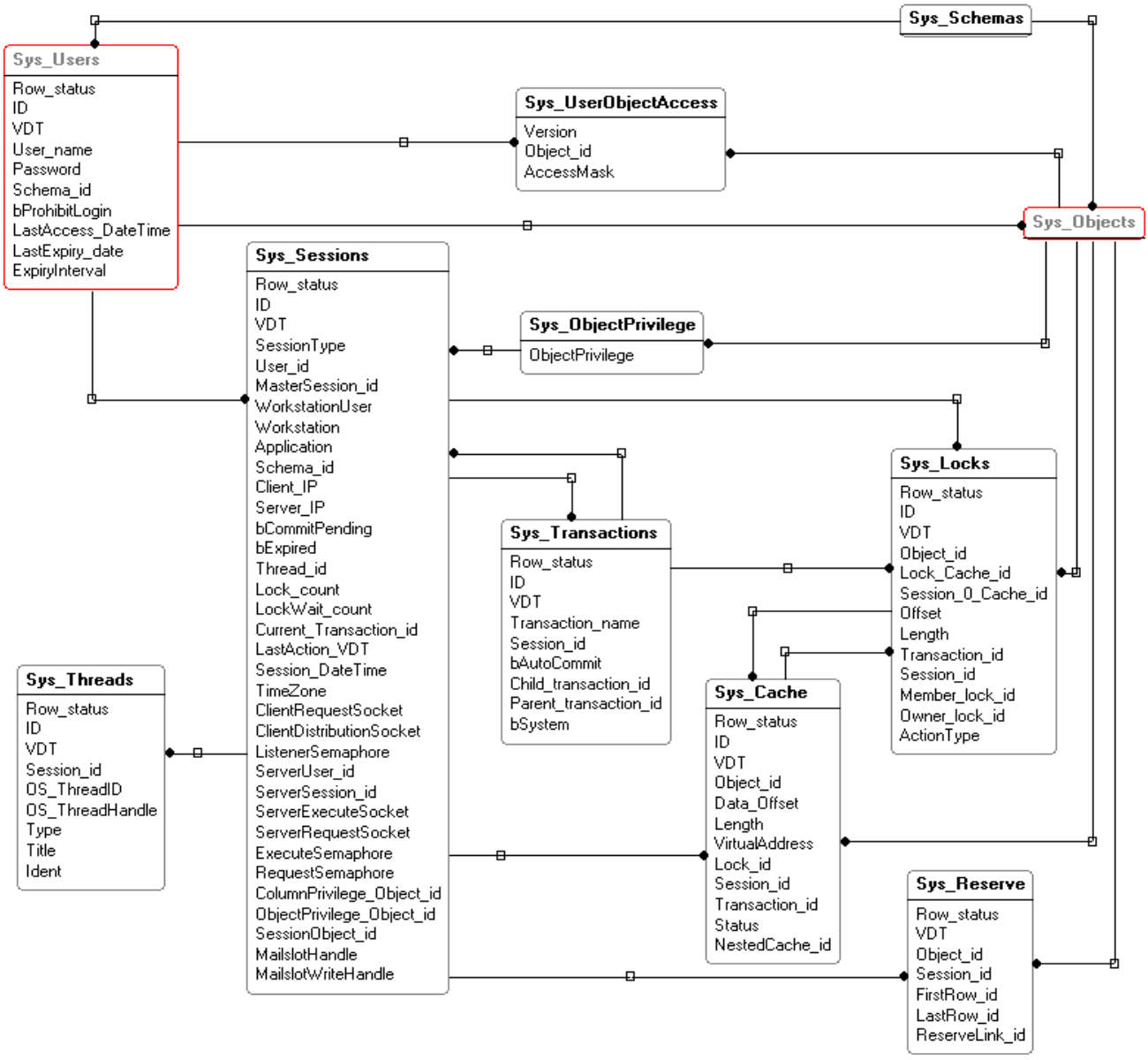
Row_status	
ID	
VDT	
Relation_id	
SourceColumn_sequence	
TargetColumn_sequence	

Sys_Relations

Relational Integrity ERD

Row_status	
ID	
VDT	
SourceObject_id	
TargetObject_id	
RelationType	
DeleteConstraint	
UpdateConstraint	
bEnforce	
bDeferred	
bDisable	
RelationColumn_id	
ColumnCount	

User / Session Tables



Sys_Cache

Row_status	
ID	
VDT	

User / Session ERD

Object_id	
Data_Offset	
Length	
VirtualAddress	
Lock_id	
Session_id	
Transaction_id	
Status	
NestedCache_id	

Sys_Locks

Row_status	
ID	
VDT	
Object_id	
Lock_Cache_id	
Session_0_Cache_id	
Offset	
Length	
Transaction_id	
Session_id	
Member_lock_id	
Owner_lock_id	
ActionType	

Sys_ObjectPrivilege

ObjectPrivilege	

Sys_UserObjectAccess

Row_status	
ID	
VDT	
User_id	
Cache_id	
Object_id	
AccessMask	

Sys_Reserve

Row_status	
VDT	
Object_id	
Session_id	
FirstRow_id	
LastRow_id	
ReserveLink_id	

Sys_Sessions

Row_status	
ID	
VDT	
SessionType	
User_id	
MasterSession_id	
WorkstationUser	
Workstation	
Application	

Schema_id	
Client_IP	
Server_IP	
bCommitPending	
bExpired	
Thread_id	
Lock_count	
LockWait_count	
Current_Transaction_id	
LastAction_VDT	
Session_DateTime	
TimeZone	
ClientRequestSocket	
ClientDistributionSocket	
ListenerSemaphore	
ServerUser_id	
ServerSession_id	
ServerExecuteSocket	
ServerRequestSocket	
ExecuteSemaphore	
RequestSemaphore	
ColumnPrivilege_Object_id	
ObjectPrivilege_Object_id	
SessionObject_id	
MailslotHandle	
MailslotWriteHandle	

Sys_Threads

Row_status	
ID	

VDT	
Session_id	
OS_ThreadID	
OS_ThreadHandle	
Type	
Title	
Ident	

Sys_Transactions

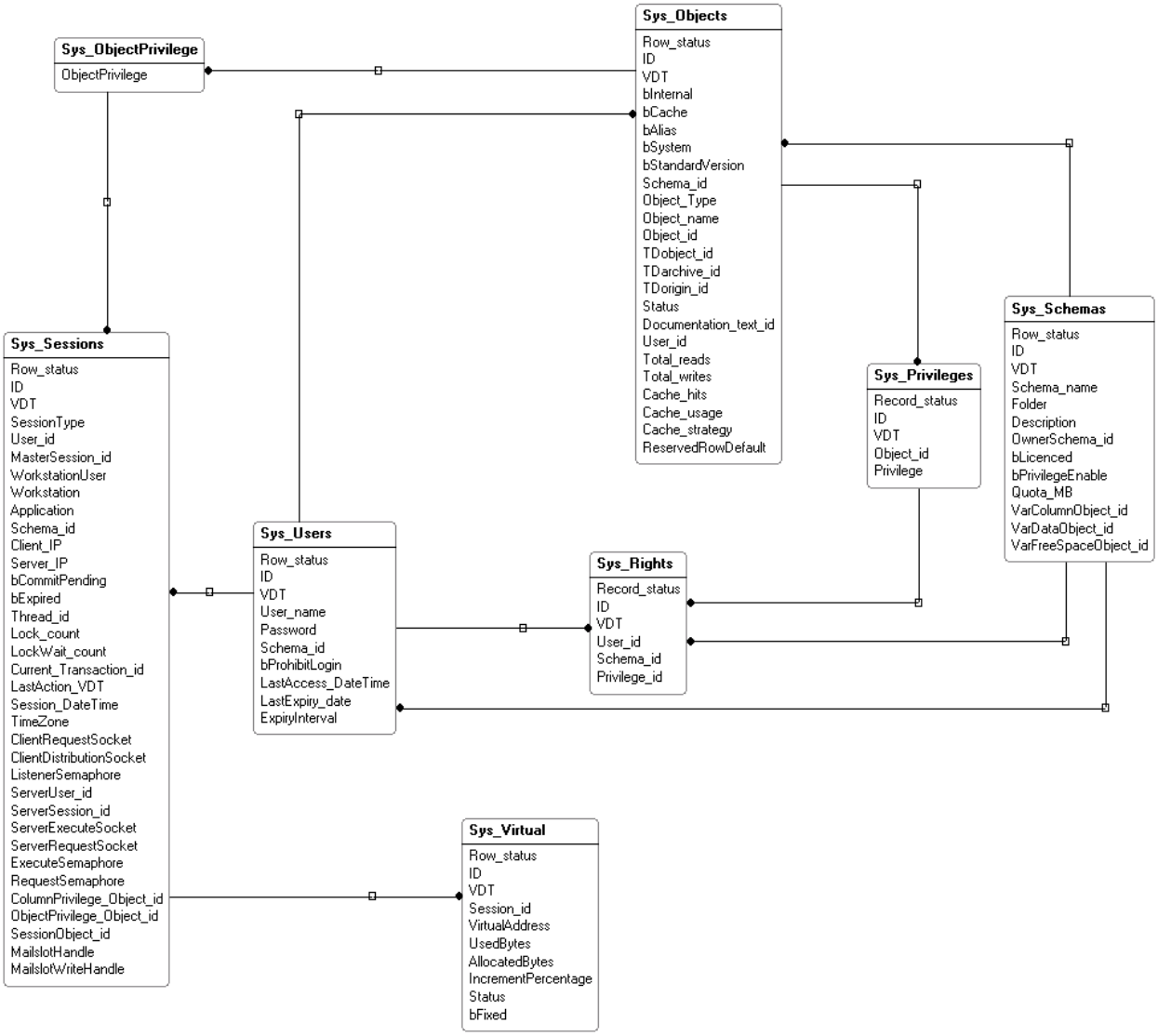
Row_status	
ID	
VDT	
Transaction_name	
Session_id	
bAutoCommit	
Child_transaction_id	
Parent_transaction_id	
bSystem	

Sys_Users

Row_status	
ID	
VDT	
User_name	
Password	
Schema_id	
bProhibitLogin	

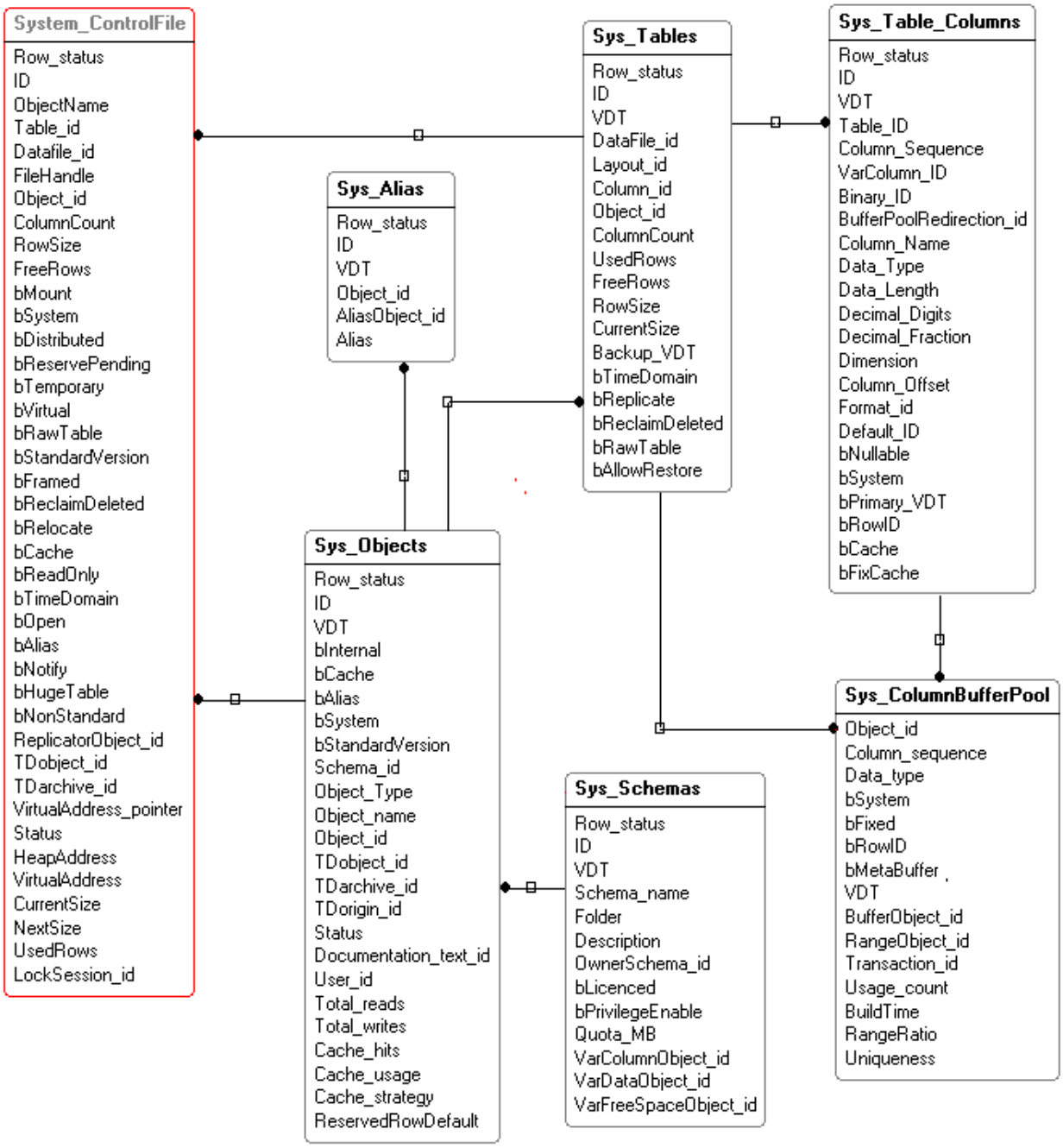
LastAccess_DateTime	
LastExpiry_date	
ExpiryInterval	

Users / Sessions



User / Session / Privilege ERD

Objects / Tables



System_ControlFile

Row_status	

Object / Table ERD

ID	
ObjectName	
Table_id	
Datafile_id	
FileHandle	
Object_id	
ColumnCount	
RowSize	
FreeRows	
bMount	
bSystem	
bDistributed	
bReservePending	
bTemporary	
bVirtual	
bRawTable	
bStandardVersion	
bFramed	
bReclaimDeleted	
bRelocate	
bCache	
bReadOnly	
bTimeDomain	
bOpen	
bAlias	
bNotify	
bHugeTable	
bNonStandard	
ReplicatorObject_id	
TDobject_id	
TDarchive_id	

VirtualAddress_pointer	
Status	
HeapAddress	
VirtualAddress	
CurrentSize	
NextSize	
UsedRows	
LockSession_id	

Sys_Alias

Row_status	
ID	
VDT	
Object_id	
AliasObject_id	
Alias	

Sys_ColumnBufferPool

Object_id	
Column_sequence	
Data_type	
bSystem	
bFixed	
bRowID	
bMetaBuffer	
VDT	
BufferObject_id	
RangeObject_id	
Transaction_id	

Usage_count	
BuildTime	
RangeRatio	
Uniqueness	

Sys_Objects

Row_status	
ID	
VDT	
bInternal	
bCache	
bAlias	
bSystem	
bStandardVersion	
Schema_id	
Object_Type	
Object_name	
Object_id	
TDobject_id	
TDarchive_id	
TDorigin_id	
Status	
Documentation_text_id	
User_id	
Total_reads	
Total_writes	
Cache_hits	
Cache_usage	
Cache_strategy	

ReservedRowDefault	

Sys_Schemas

Row_status	
ID	
VDT	
Schema_name	
Folder	
Description	
OwnerSchema_id	
bLicenced	
bPrivilegeEnable	
Quota_MB	
VarColumnObject_id	
VarDataObject_id	
VarFreeSpaceObject_id	

Sys_Tables

Row_status	
ID	
VDT	
DataFile_id	
Layout_id	
Column_id	
Object_id	
ColumnCount	
UsedRows	
FreeRows	

RowSize	
CurrentSize	
Backup_VDT	
bTimeDomain	
bReplicate	
bReclaimDeleted	
bRawTable	
bAllowRestore	

Sys_Table_Columns

Row_status	
ID	
VDT	
Table_ID	
Column_Sequence	
VarColumn_ID	
Binary_ID	
BufferPoolRedirection_id	
Column_Name	
Data_Type	
Data_Length	
Decimal_Digits	
Decimal_Fraction	
Dimension	
Column_Offset	
Format_id	
Default_ID	
bNullable	
bSystem	

bPrimary_VDT	
bRowID	
bCache	
bFixCache	

API Structures and Constants

This section of the reference specifies structures (record formats) and constants necessary for interfacing to the Lava API. The structures are listed in alphabetical order.

Structures and constants defined in this reference are specified in Pascal / Oberon format. Note that in this syntax, the variable type follows the variable name, the opposite to C syntax - for C declarations of these entities consult the appropriate addendum and header files.

Lava Structures

The following structures are required for various parameters in the Lava API. The listing is alphabetical, with the addition of hyperlinks to structures and types which expand to further definitions. For reference, the Oberon base types are listed with C equivalent.

Base Types

Oberon Type	C Equivalent	Description
ARRAY [x] OF		Array of x elements of whichever type is nominated
ARRAY [x] OF CHAR		Zero terminated ASCII character string of maximum x characters
BOOLEAN	No direct equivalent - closest is byte	1-byte boolean value - TRUE(1) or FALSE (0)
BYTE		1-byte value
CHAR		1-byte ASCII character
INTEGER		2-byte signed integer
LONGINT		4-byte signed integer
LONGREAL		8-byte float
POINTER TO		Pointer to whichever type is nominated
PTR		Anonymous pointer (pointer to untyped memory).
SHORTINT		1-byte signed integer
SHORTREAL		4-byte float

BackupSetType

```
   BackupSetType                =     RECORD
      VDT                       :     LONGREAL;
      ObjectCount               :     INTEGER;
      Schema                    :     Label;
      EncryptionKey             :     Label;
```

```
                                        END;
```

ColumnArray_Type

```
ColumnArray_Type            =  ARRAY 10000 OF Sys_Table_Columns_Type;
```

ColumnScan_Type

```
   ColumnScan_Type       =
      RECORD
         Session_id                        :        LONGINT;
         Object_id                         :        LONGINT;
         Column_Sequence                   :        LONGINT;
         BufferRedirection_id              :        LONGINT;
         Buffer_VDT                        :        LONGREAL;
         LastBufferIndex                   :        LONGINT;
         LastRowID                         :        LONGINT;
         DataType                          :        LONGINT;
         Condition                         :        LONGINT;
         ScanValueAddress                  :        LONGINT;
         ColumnValue                       :        LONGREAL;
         ColumnString                      :        ARRAY 12 OF CHAR;
         ColumnQuad                        :        QUADINTEGER;
      END;
```

CommandParamType

```
   CommandParamType         =  RECORD
      Ident                 :  ARRAY 60 OF CHAR;
      Value                 :  ARRAY 260 OF CHAR;
      FloatValue            :  LONGREAL;
   END;
```

CommandLineType

```
   CommandLineType          =  RECORD
      Count                 :  LONGINT;
      Param                 :  ARRAY 25 OF CommandParamType;
   END;
```

DateClass

```
   DateClass       = RECORD
      Year         :  INTEGER;
      Month        :  INTEGER;
      Day          :  INTEGER;
      Julian       :  LONGINT;
                   END;
```

Heap Sort Procedure Types

```
   CompareProc = PROCEDURE(pLowIdx, pHighIdx : LONGINT) : BOOLEAN;
   SwapProc    = PROCEDURE(pLowIdx, pHighIdx : LONGINT);
```

Label

```
   Label                    = ARRAY LABEL_LENGTH OF CHAR;
```

ObjectArrayType

```
   ObjectArrayType =  ARRAY 100 OF ObjectBackupType;
```

ObjectBackupType

```
   ObjectBackupType             =     RECORD
      ObjectName                :     Label;
      ObjectType                :     LONGINT;
      Object_id                 :     LONGINT;
      bForceObject_id           :     BOOLEAN;
      bSystemTable              :     BOOLEAN;
      bCacheTable               :     BOOLEAN;
      bTimeDomain               :     BOOLEAN;
      bReplicate                :     BOOLEAN;
      bReclaimDeleted           :     BOOLEAN;
      RecordLength              :     LONGINT;
      ColumnCount               :     INTEGER;
      ColumnLayoutOffset        :     LONGINT;
      DataOffset                :     QUADINTEGER;
      DataLength                :     QUADINTEGER;
      bEncrypted                :     BOOLEAN;
                                       END;
```

QUADINTEGER

```
   QUADINTEGER    =  RECORD
      Low         :  LONGINT;
      High        :  LONGINT;
   END;
```

Sys_Query_Type

```
   Sys_Query_Type       =
      RECORD
         Row_status                        :        SHORTINT;
         Column_id                         :        ARRAY 10 OF LONGINT;
         Value_type                        :        ARRAY 10 OF LONGINT;
         Value_length                      :        ARRAY 10 OF LONGINT;
         Value_float                       :        ARRAY 10 OF LONGREAL;
         Value_long                        :        ARRAY 10 OF LONGINT;
         Value_string_address              :        ARRAY 10 OF LONGINT;
         Condition                         :        ARRAY 10 OF LONGINT;
         ScanGroup                         :        ColumnScan_Type;
         MatchTable_id                     :        LONGINT;
         BufferAddress                     :        LONGINT;
         bPersistent                       :        BOOLEAN;
         bCaseInsensitive                  :        BOOLEAN;
         bRangeInequality                  :        BOOLEAN;
         InequalityIndex                   :        LONGINT;
      END;
```

Sys_Table_Columns_Type

```
Sys_Table_Columns_Type       =
   RECORD
      Row_status                          :        SHORTINT;
      ID                                  :        LONGINT;
      VDT                                 :        LONGREAL;
      Table_ID                            :        LONGINT;
      Column_Sequence                     :        LONGINT;
      VarColumn_ID                        :        LONGINT;
      Binary_ID                           :        LONGINT;
      BufferPoolRedirection_id            :        LONGINT;
      Column_Name                         :        ARRAY 50 OF CHAR;
      Data_Type                           :        LONGINT;
      Data_Length                         :        LONGINT;
      Decimal_Digits                      :        LONGINT;
      Decimal_Fraction                    :        LONGINT;
      Dimension                           :        LONGINT;
      Column_Offset                       :        LONGINT;
      Format_id                           :        LONGINT;
      Default_ID                          :        LONGINT;
      bNullable                           :        BOOLEAN;
      bSystem                             :        BOOLEAN;
      bPrimary_VDT                        :        BOOLEAN;
      bRowID                              :        BOOLEAN;
      bCache                              :        BOOLEAN;
      bFixCache                           :        BOOLEAN;
   END;
```

TableColumnPointer

```
TableColumnPointer          =  POINTER TO ColumnArray_Type;
```

TableColumnType

```
TableColumnType             =  RECORD
   RowSize                     :     LONGINT;
   ColumnCount                 :     INTEGER;
   bHeapAllocated              :     BOOLEAN;
   ColumnArray                 :     TableColumnPointer;
                               END;
```

TableFormatPointer

```
TableFormatPointer          =  POINTER TO TableColumnType;
```

TimeClass

```
TimeClass          = RECORD
   Hour            :  INTEGER;
   Minute          :  INTEGER;
   Second          :  INTEGER;
   MilliSecond     :  INTEGER;
                   END;
```

Util_SearchMatch_Type

```
   Util_SearchMatch_Type       =
      RECORD
         Row_status           :        SHORTINT;
         SourceName           :        ARRAY 100 OF CHAR;
         Path_id              :        LONGINT;
         Node_id              :        LONGINT;
         MatchType_id         :        LONGINT; (* see Search Match Types
*)
         EditHandle           :        LONGINT;
         Offset               :        LONGINT;
         Length               :        LONGINT;
         FoundString          :        ARRAY 100 OF CHAR;
      END;
```

Lava API Constants

Comparison Constants

```
   SQL_TOKEN_EQ                  = 9;
   SQL_TOKEN_NEQ                 = 10;
   SQL_TOKEN_LT                  = 11;
   SQL_TOKEN_LEQ                 = 12;
   SQL_TOKEN_GT                  = 13;
   SQL_TOKEN_GEQ                 = 14;
```

Data Type Constants

```
   BYTE_TYPE            = 1H;
   SHORTINT_TYPE        = 2H;       (* 2-byte integer, signed, max 65535 *)
   INTEGER_TYPE         = 3H;       (* 4-byte integer, signed, max 2.147 E
                                    09 *)
   QUADINTEGER_TYPE     = 4H;       (* 8-byte integer, signed, max 9.223 E
                                    18 *)
   FLOAT_TYPE           = 7H;       (* 8-byte real, 10-bit exponent *)
   BOOLEAN_TYPE         = 8H;       (* 1-byte boolean value, 0=FALSE,
                                    1=TRUE *)
   CHAR_TYPE            = 9H;       (* 1-byte character *)
   STRING_TYPE          = 0AH;      (* fixed length ascii string *)
   UNICODE_TYPE         = 2AH;      (* fixed length unicode string *)
   VARSTRING_TYPE       = 0BH;      (* variable length column ascii string
                                    with fixed length maximum in-column
                                    string storage *)
   VARUNICODE_TYPE      = 2BH;      (* variable length unicode string *)
   IP_TYPE              = 0FH;      (* IP address *)
   ROWSTATUS_TYPE       = 21H;      (* 1-byte unsigned row status *)
   ROWID_TYPE           = 23H;      (* 4-byte integer, unsigned, of type
                                    Row ID *)
   DATE_TYPE            = 33H;      (* 4-byte integer, unsigned, Julian
                                    date *)
   VDT_TYPE             = 37H;      (* 8-byte real, integer part = Julian
                                    date, fractional part = milliseconds
                                    after midnight *)
   TIME_TYPE            = 43H;      (* 4-byte integer, unsigned,
                                    milliseconds after midnight *)
```

Editor Bookmark Types

```
   BOOKMARK_BREAKPOINT              =  1;
   BOOKMARK_DISABLEDBREAKPOINT      =  2;
   BOOKMARK_USER                    =  3;
```

Backup Set Constants

```
   LABEL_LENGTH            = 100;
```

Shutdown Modes

```
   SHUTDOWN_NORMAL             -    = 1051;
```

```
   SHUTDOWN_IMMEDIATE          -    = 1052;
   SHUTDOWN_ABORT              -    = 1053;
```

Startup Modes

```
   OPEN_SERVER                 -    = 1002;
   OPEN_EXCLUSIVE              -    = 1003;
   OPEN_CLIENT                 -    = 1004;
   OPEN_STANDBYSERVER          -    = 1005;
```

Object Types

```
   SQL_OBJECT_TABLE                 = 1H;
   SQL_OBJECT_RESERVED              = 2H;
   SQL_OBJECT_VIEW                  = 3H;
   SQL_OBJECT_PROCEDURE             = 4H;
   SQL_OBJECT_TRIGGER               = 5H;
   SQL_OBJECT_FUNCTION              = 6H;
   SQL_OBJECT_PACKAGE               = 7H;
   SQL_OBJECT_PSEUDOTABLE           = 8H;

   SQL_OBJECT_MASK                  = 0FFFH;

   (*..Object type attributes *)

   SQL_OBJECT_ALLOWRESTORE          = 00001000H;      (* Table may be
                                                      restored   not valid
                                                      for most system
                                                      tables *)

   SQL_OBJECT_VIRTUAL               = 00010000H;
   SQL_OBJECT_RAW                   = 00020000H;
   SQL_OBJECT_PERSISTENT            = 00040000H;      (* Flags virtual
                                                      table as persistent
                                                      *)
   SQL_OBJECT_NONSTANDARD           = 00080000H;      (* Flags table as
                                                      nonstandard format
                                                      get / put cannot be
                                                      performed *)

   SQL_OBJECT_FRAMED                = 00100000H;
   SQL_OBJECT_RELOCATE              = 00200000H;
   SQL_OBJECT_INTERNAL              = 00400000H;

   SQL_OBJECT_RESULTSET             = 00800000H;      (* Flags table as a
                                                      sql result set -
                                                      allows creation of
                                                      duplicate table names
                                                      *)

   SQL_OBJECT_TIMEDOMAIN            = 01000000H;      (* Table is a time-
                                                      domain replicator *)
   SQL_OBJECT_TDARCHIVE             = 02000000H;      (* Table is a time-
                                                      domain archive
*)

   SQL_OBJECT_COLUMNBUFFER          = 04000000H;      (* Table is a column
```

```
                                                    buffer *)
   SQL_OBJECT_INSTANCECOPY          = 08000000H;    (* Table is an
                                                    instance copy of a
                                                    table prototype *)

   SQL_OBJECT_ATTRIBUTE_MASK        = 0FF0000H;
   SQL_OBJECT_VIRTUAL_MASK          = 00F0000H;
   SQL_OBJECT_INDEX_MASK            = 0F00000H;

(*.Composite types *)

   SQL_OBJECT_PERSISTENT_VIRTUAL_TABLE =  SQL_OBJECT_TABLE +
                                          SQL_OBJECT_VIRTUAL +
                                          SQL_OBJECT_PERSISTENT;
```

Primary Format Codes

```
   FORMAT_P_GENERAL           =  000H;
   FORMAT_P_NUMBER            =  001H;
   FORMAT_P_CURRENCY          =  002H;
   FORMAT_P_ACCOUNTING        =  003H;
   FORMAT_P_DATE              =  004H;
   FORMAT_P_TIME              =  005H;
   FORMAT_P_PERCENTAGE        =  006H;
   FORMAT_P_SCIENTIFIC        =  007H;

   FORMAT_P_HEXQUAD           =  008H;
   FORMAT_P_VDT               =  009H;
   FORMAT_P_IP                =  00AH;

   FORMAT_P_BOOLEAN           =  00BH;

   FORMAT_P_STRING            =  00CH;

   FORMAT_P_NULL              =  00DH;    (* Formatting routine outputs a
                                          blank string *)

   FORMAT_P_LONGREAL          =  000H;
   FORMAT_P_BYTE              =  100H;
   FORMAT_P_INTEGER           =  200H;
   FORMAT_P_LONGINT           =  300H;
   FORMAT_P_QUADINTEGER       =  400H;
   FORMAT_P_CHAR              =  500H;
   FORMAT_P_VARSTRING         =  600H;
   FORMAT_P_UNICODE           =  700H;
   FORMAT_P_UNDEFINED         =  800H;

   FORMAT_P_FORMAT_MASK       =  00FFH;
   FORMAT_P_TYPE_MASK         =  0F00H;
```

Search Match Types

```
   MATCH_NODETITLE                =  1;
   MATCH_NODETEXT                 =  2;
   MATCH_KEYWORD                  =  3;
   MATCH_DOCUMENTNAME             =  4;
   MATCH_DOCUMENTCONTENT          =  5;
```

```
   MATCH_FILECONTENT                =  6;
```

Secondary Format Codes

```
   FORMAT_S_NULL                =  0;

   FORMAT_S_BASE_BINARY         =  1;
   FORMAT_S_BASE_HEX            =  2;

   FORMAT_S_NEGATIVE_1          =  0;    (* -1,234.10      *)
   FORMAT_S_NEGATIVE_2          =  1;    (*  1,234.10  RED *)
   FORMAT_S_NEGATIVE_3          =  2;    (* (1,234.10)     *)
   FORMAT_S_NEGATIVE_4          =  3;    (* (1,234.10) RED *)

   FORMAT_S_DATE_1              =  1;    (* dd/mm/yy       *)
   FORMAT_S_DATE_2              =  2;    (* dd/mm/yyyy     *)
   FORMAT_S_DATE_3              =  3;    (* mm/dd/yy       *)
   FORMAT_S_DATE_4              =  4;    (* mm/dd/yyyy     *)
   FORMAT_S_DATE_5              =  5;    (* mmm dd, yyyy   *)
   FORMAT_S_DATE_6              =  6;    (* mmmmmmmmm dd, yyyy *)
   FORMAT_S_DATE_7              =  7;    (* dd mmm yy      *)
   FORMAT_S_DATE_8              =  8;    (* dd mmm yyyy    *)
   FORMAT_S_DATE_9              =  9;    (* yyyy/mm/dd     *)

   FORMAT_S_TIME_1              =  1;    (* hh:mm       *)
   FORMAT_S_TIME_2              =  2;    (* hh:mm:ss    *)
   FORMAT_S_TIME_3              =  3;    (* hh:mmXM     *)
   FORMAT_S_TIME_4              =  4;    (* hh:mm:ssXM  *)
   FORMAT_S_TIME_5              =  5;    (* hhmm        *)
   FORMAT_S_TIME_6              =  6;    (* hhmmss      *)
```

Table Location

```
   SQL_OBJECT_LOCAL              -    =  0001H;
   SQL_OBJECT_SERVER             -    =  0002H;
   SQL_OBJECT_DISTRIBUTED        -    =  0010H;

   SQL_OBJECT_LOCATION_MASK      -    =  000FH;
   SQL_OBJECT_OPTION_MASK        -    =  00F0H;
```

Appendix I : Lava Error Codes

Appendix II : Source Code Examples

The source code examples provided below are divided into four major categories - Oberon, Pascal (Delphi), Clarion and C. Although virtually any language from assembly through Basic can be used to interface to the Lava database, the examples provided should cover a wide enough scope that most programmers working in most languages should be able to find suitable source to illustrate a given principle.

Oberon Examples

Backup Set Creation

Backup Set Restore

Instance tables

SQL Execution and Data Extraction

See also
Virtual table pointers

Virtual table pointers

Table Creation

Appendix III : SQL Examples

Grouping aggregates, subqueries	An example including subqueries in the column

Simple examples

Advanced examples

Grouping aggregates, subqueries

The following example demonstrates the use of a subquery in the column list, as well as the use of a Group By clause to group aggregate results. In addition, the use of column and table aliases is demonstrated.

```
	select
		(select
			schema_name
		from
			sys_objects o, sys_schemas s
		where
			o.id = scf.id and
			s.id = o.schema_id) SchemaName,
		sum(usedrows * rowsize) ts
	from
		system_controlfile scf
	group by
		schemaname
```

The result set for the above query executed on a freshly created Lava Database is as follows :

	SchemaName	total size (bytes)
1	Backup	0.00
2	Design	487.00
3	Dictionary	0.00
4	Event	2,127,318.00
5	Linker	0.00
6	Parse	128,226.00
7	Scratch	0.00
8	Sheet	71,964.00
9	SYSTEM	5,075,458.00
10	Template	0.00
11	Util	1,212.00
12	Virtual	66.00

Group By example 1

Appendix IV : ODBC Interface

This appendix documents the Lava ODBC interface, which is provided for legacy SQL support. Although this is a functional driver, and it is possible to extract data from and update data in the Lava Database using the facilities provided in this driver, its use is **strongly discouraged**.

The techniques used to interface to server databases in the ODBC methodology are completely outdated, and should be avoided if at all possible. The Lava Database presents techniques for interfacing to the database which are vastly improved, many times faster and much more flexible than the ODBC techniques.
For information on the Please consult the sections Lava SQL Reference and The Lava API for information on the native Lava interfaces, and in particular the section Key Concepts in the Lava Database and the reference for the command LavaCommand which provides direct SQL access into the Lava Database should be consulted to acquire information on the totally revised techniques used in the native interface to the Lava Database. The coded example SQL Execution and Data Extraction may also be consulted for insight into the native mechanisms provided.