IBM® Netezza® Analytics
Release 3.3.x.0

Lua Developer's Guide

Revised: Oct. 05, 2017

Part Number 00J2390-06 Rev. 8

Note: Before using this information and the product that it supports, read the information in Notices and Trademarks on page 140.

Contents

Preface

1 Introduction and Installation

2 Using nzLua

3 The Lua Programming Language

4 Lua Functions and Libraries

6 nzLua Libraries

7 nzLua API

APPENDIX A

Examples

APPENDIX B

Notices and Trademarks

Index

List of Tables

Preface

Preface

Audience for This Guide

The IBM Netezza Lua Developer's Guide is written for programmers who intend to create nzLua UDXs for IBM Netezza Analytics

Purpose of This Guide

This guide describes the IBM Netezza nzLua API, which is a part of IBM Netezza Analytics. The IBM Netezza nzLua provides a mechanism for writing UDXs for Lua programmers.

Symbols and Conventions

Note on Terminology: The terms *User-Defined Analytic Process (UDAP)* and *Analytic Executable (AE)* are synonymous.

The following conventions apply:

- Italics for emphasis on terms and user-defined values, such as user input.
- Upper case for SQL commands, for example, INSERT or DELETE.
- Bold for command line input, for example, **`nzsystem stop`**.
- Bold to denote parameter names, argument names, or other named references.
- Angle brackets (< >) to indicate a placeholder (variable) that should be replaced with actual text, for example, `inza-<release_number>.zip`.
- A single backslash (“\”) at the end of a line of code to denote a line continuation. Omit the backslash when using the code at the command line, in a SQL command, or in a file.
- When referencing a sequence of menu and submenu selections, the “>” character denotes the different menu options, for example, **Menu Name > Submenu Name > Selection**.

If You Need Help

If you are having trouble using the IBM Netezza appliance, IBM Netezza Analytics or any of its components:

1. Retry the action, carefully following the instructions in the documentation.
2. Go to the IBM Support Portal at: http://www.ibm.com/support. Log in using your IBM ID and password. You can search the Support Portal for solutions. To submit a support request, click the '**Service Requests & PMRs**' tab.
3. If you have an active service contract maintenance agreement with IBM, you may contact customer support teams via telephone. For individual countries, please visit the Technical Support section of the IBM Directory of worldwide contacts (http://www14.software.ibm.com/webapp/set2/sas/f/handbook/contacts.html#phone)

Comments on the Documentation

We welcome any questions, comments, or suggestions that you have for the IBM Netezza documentation. Please send us an e-mail message at netezza-doc@wwpdl.vnet.ibm.com and include the following information:

- The name and version of the manual that you are using
- Any comments that you have about the manual
- Your name, address, and phone number

We appreciate your comments.

CHAPTER 1
Introduction and Installation

Lua is an extension programming language designed to support general procedural programming with data description facilities. It also offers good support for object-oriented programming, functional programming, and data-driven programming. Lua is intended to be used as a powerful, lightweight scripting language for any program that needs one.

Overview

Being an extension language, Lua has no notion of a "main" program; it works only when embedded in a host program, called the *embedding program* or simply the *host*. This host program can invoke functions to execute a piece of Lua code, can write and read Lua variables, and can register C functions to be called by Lua code. Through the use of C functions, Lua can be augmented to handle a wide range of different domains, creating customized programming languages that share a syntactical framework.

Using the Lua API, Netezza has created nzLua, which can be used to create user-defined functions, aggregates, and table functions on the Netezza appliance. Some of the features of Lua, such as the ability to access external files, ability to execute external programs, and debugging features have been disabled in nzLua. In addition to the features that have been removed, many functions have been added to nzLua to make the language easy to use for developers who are familiar with the standard functions available in SQL.

The version of Lua used to create nzLua (LuaJIT) uses just-in-time compilation techniques to compile frequently used code paths directly into 80x86 instructions. Although slower than a UDX built using C++, the JIT compiler does result in execution which is 5x to 50x faster than is typical for other interpreted languages such as Perl, Python, and JavaScript (including Javascript V8, which also uses a JIT compiler).

To make nzLua as easy to learn as possible, the nzLua development kit includes an examples directory containing sample nzLua user-defined functions, aggregates, and table functions. Many of the examples include extensive comments that explain how to use the features available in nzLua.

Installation

The nzLua package is distributed with the Netezza Analytics software and is normally installed as part of the express installation (see the *IBM Netezza Analytics Administrator's Guide*). The binaries for nzLua are installed in the directory /nz/extensions/nz/nzlua/bin and this directory should be added to a developer's PATH environment variable in the .bashrc file or appropriate rc file if a shell other than bash is being used.

```
export PATH="$PATH:/nz/extensions/nz/nzlua/bin"
```

It is also possible, using the cartridge manager, to install nzLua without installing the complete IBM Netezza Analytics software. First, follow the instructions for manually installing Netezza Analytics in the *IBM Netezza Analytics Administrator's Guide*, then instead of running the inzaCartridgeInstaller.sh script directly, invoke the cartridge manager to install nzLua, as shown below.

```
nzcm -i nzlua
nzcm -r nzlua -d inza
```

CHAPTER 2
Using nzLua

Compiling nzLua Programs

The nzl command in the /nz/extensions/nz/nzlua/bin directory is used to compile and install nzLua scripts. This section details the Unix and database permissions required to use the nzl program, as well as other options available with the nzl program.

It is not necessary to use the nz Unix login or the admin database login to compile and install nzLua programs. Any user with the appropriate Unix and database permissions, as outlined in the sections below, can compile and install an nzLua program. A Unix account on the Netezza host server, however, is required.

Unix Environment Variables and Permission Requirements

Using the nzl command to compile and install an nzLua UDX requires the user to set the $NZ_USER and $NZ_PASSWORD environment variables. If the $NZ_DATABASE environment variable has not been set, the -d option must be used to indicate the name of the database where the UDX is to be installed. The nzl command does not use passwords that have been cached using the nzpassword command. If schemas are enabled, and if you do not want to use the default schema for the specified database, you can use the $NZ_SCHEMA environment variable or the -s option to indicate the name of the schema where the UDX is to be installed.

A user must have execute permissions on the programs in the /nz/extensions/nz/nzlua/bin directory and read access to the files in the /nz/extensions/nz/nzlua//include, /nz/extensions/nz/nzlua/lib, and /nz/extensions/nz/nzlua/src directories. The default permissions for these directories should already be set correctly by the nzLua install process.

Database Permission Requirements

Users who need to compile and install nzLua code must have the permissions to create functions and

aggregates, execute permissions on the nzLua shared libraries, and the unfence privilege. Note that the libnzlua library name is based on the nzLua version. For example, Version 2.0.0 creates the library as libnzlua_2_0_0, whereas version 1.2.5 would create the library as libnzlua_1_2_5.

```
/* these permissions are granted by default */
grant execute on libnzlua_2_0_0 to public;
grant execute on libgmp to public;
```

The create_inza_db_developer.sh and create_inza_db_user.sh scripts can be used to grant the necessary permissions for a developer or a user. It is also possible to manually grant these permissions using the grant command. A developer would need these permissions:

```
grant create function to <user | group>;
grant create aggregate to <user | group>;
grant unfence to <user | group>;
grant execute on function to <user | group>;
grant execute on aggregate to <user | group>;
```

A normal user would need a more limited set of permissions.

```
grant execute on function to <user | group>;
grant execute on aggregate to <user | group>;
```

It is also possible to grant permissions for a normal user for a specific function or aggregate using the grant command or using the Netezza admin tool. When using the grant command in this manner it is necessary to specify both the function name as well as the arguments.

```
grant execute on testfunction(varchar(255),int,int) to <user | group>;
```

Compiling and Installing a Program

Compiling and installing an nzLua script is very simple. The /nz/extensions/nz/nzlua/examples directory has many sample nzLua programs that can be used as learning aids as well as verification that nzLua has been installed correctly. Many of the examples are well commented and serve as an excellent way to learn how to develop UDX programs in the Netezza database using nzLua.

Here is an example of compiling the nzlua_version.nzl script and then using the nzlua_version() UDF it creates. The installation directory always includes the version number of nzLua that has been installed. In this case the directory shown is for nzLua 2.0.0.

```
cd /nz/extensions/nz/nzlua/examples
nzl nzlua_version.nzl
nzsql -c "select nzlua_version()"
```

The output of installing the nzlua_version.nzl program should look like this:

```
Compiling: nzlua_version.nzl
################################################################
UdxName      =  nzlua_version
UdxType      =  UDF
Arguments    =
Result       =  VARCHAR(10)
```

```
Dependencies =  INZA..LIBNZLUA_2_0_0
NZUDXCOMPILE OPTIONS: (--unfenced --mem 2m )
CREATE FUNCTION
Created udf
Done
```

And the result of executing the command:

```
nzsql -c "select nzlua_version()"
 NZLUA_VERSION
---------------
 2.0.0
(1 row)
```

The version number in the output should match the version of nzLua that has been installed.

Uninstalling an nzLua UDX

The nzl command can be used to uninstall an nzLua UDX by specifying the -u option.

```
nzl -u nzlua_version.nzl
```

A UDX can also be dropped from the database using the SQL commands DROP FUNCTION (for normal functions) or DROP AGGREGATE (for aggregated functions).

```
nzsql -c "drop function nzlua_version()"
```

Note that with DROP FUNCTION and DROP AGGREGATE, the function signature must exactly match the argument types of the UDX to be dropped. This is necessary because Netezza allows function overloading where multiple functions have the same name but accept different arguments. For example, to drop a function that accepts two integer arguments and a varchar(255), the command would be:

```
drop function myfunction(integer,integer,varchar(255));
```

Development and Debugging

The nzlua Command Line Program

The nzLua distribution comes with a command line version of nzLua, which can be used for development and testing of code outside of the database. The nzlua command line program is located in the /nz/extensions/nz/nzlua/bin directory. All of the functions that are available inside of Netezza using nzLua are also usable from the command line version of nzlua.

```
nzlua - <<-END
      print("Hello world!")
      for i=1,10 do
            print( "i = " || i )
      end
END
```

In the example above, the nzlua command line program was invoked using the - argument (`nzlua - <<-END`). When passing a script to the nzlua command line program using a HERE document in a shell script, the - option must be used so that nzlua will abort and return an error code back to the shell script if an error occurs. If the - option is not used, nzlua will operate in interactive mode and therefore will not exit when an error occurs.

The nzlua Table Function

The nzlua table function is created during the install process in the INZA database. This function allows arbitrary nzLua code to be executed inside of the Netezza appliance by an external application directly from a SQL statement. This function is very useful for developing and testing code as well as for ETL scripting.

There are several examples of using the nzlua table function in the /nz/extensions/nz/nzlua/examples/MapReduce directory of the nzlua distribution. There is also the /nz/extensions/nz/nzlua/examples/elt.sql script, which uses the nzlua table function in combination with the Netezza SPUPad to perform some basic ELT processing.

Here is a simple example of using the nzlua table function in a SQL statement by passing the nzLua source code in as the first argument.

```
select *
from table(inza..nzlua('
function processRow(x)
    t={}
    for i=1,x do
        t[i] = { i, i*i }
    end
    return t
end

function getShape()
    columns={}
    columns[1] = { "x", integer }
    columns[2] = { "x_squared", integer }
    return columns
end',
11));
```

nzLua Code Libraries

nzLua provides a mechanism to create libraries of nzLua code that can then be reused by many programs. An nzLua code library is created by using a .nzll extension instead of .nzl for the nzLua program. The library can then be loaded by any other nzLua UDX by using the require statement. For example, if a file was named mylualib.nzll, another nzLua UDX could dynamically load the code in that library using:

```
require "mylualib"
```

See the appendix, APPENDIX A, for a sample code library, as well as the /nz/extensions/nz/nzlua/examples directory of the nzLua distribution, which has several sample nzLua code libraries.

The Netezza database system view _v_library will contain a list of all of the nzLua libraries that have been installed. The library names will always be prefixed with "NZLUALIB_". For example, if the original source file is named testlib.nzll, the library in the Netezza database will be named NZLUALIB_TESTLIB. The SQL below can be used to see all of the nzLua code libraries:

```
select library from _v_library where upper(library) like 'NZLUALIB%';
```

The nzLua require function replaces the require function in the Lua language.

Cross Database Code Libraries

For nzLua UDX procedures that you use for cross-database queries, you must specify the following paths to the library:

- For NPS V7.0.3 and later, you must provide the full path for the library as database.schema.objname.
- For NPS versions previous to V7.0.3, you must provide the full path for the library as databasename..libraryname.

If you do not use this format, you get the following error:

```
Add require("LIBRARYNAME"); to this program.
```

For example:

```
require "prod..mylibrary"
```

Upgrading to a New Version

The nzLua libraries are created in the database with a unique name for each version. For example, the version 1.2.8 library will be named LIBNZLUA_1_2_8 and the version 2.0.0 library will be named LIBNZLUA_2_0_0.

Note: You must recompile each nzLua UDX by using the nzl command from the new version of nzLua before the UDXs can take advantage of bug fixes or performance improvements that might be available in a newer version of nzLua.

System Views

There are several views in the database that are useful references when using nzLua. These views can be used to find all of the nzLua UDXs that have been created in each database and to identify the version of nzLua that has been installed.

_v_library

The _v_library view shows all of the shared libraries that have been installed in the Netezza database using the "CREATE LIBRARY" command. A developer using nzLua does not need to issue the create library command since it is automatically invoked during the install process as well as by the nzl command line program. For further information on the CREATE LIBRARY command, see the *Netezza User-Defined Analytic Process Developer's Guide*.

Column	Description
LIBRARY	The name of the shared library. The nzLua library is named LIBNZLUA_<version>. For example: LIBNZLUA_2_0_0. The nzLua code libraries (see nzLua Code Libraries) are named in the form NZLUALIB_<name>.
DEPENDENCIES	The libraries that a library depends on. The LIBNZLUA library depends on the LIBGMP library.
AUTOMATICLOAD	True/false indicator of whether the library should be automatically loaded by the Netezza database. The nzLua code libraries always report false for this column.

Example

```
select *
from _v_library;
```

_v_function

The _v_function view lists all of the functions (user-defined as well as built-in) that are available in the Netezza database. All of the functions that have been created with nzLua will have a dependency for the LIBNZLUA library.

Column	Description
FUNCTION	The name of the function.
DESCRIPTION	The description can be set using the Netezza COMMENT command or by defining the getComment() method. See getComment().
ARGUMENTS	The argument types accepted by the function.
DETERMINISTIC	True/false indicator of whether the function is deterministic. See getOptions().

Column	Description
VARARGS	True/false indicator of whether the function is a varargs function. See getArgs().
DEPENDENCIES	The libraries that a function depends on. All nzLua UDXs have a dependency on the LIBNZLUA library. Dependencies for other libraries may also exist based on usage of the require function.
RETURNS	The result type(s) of the function.
LOCATION	For table functions, indicates if the function runs in parallel or if only a single copy of the UDTF runs on the Netezza host system. See getOptions().

The query below can be used to display all of the nzLua user-defined functions and table functions that have been installed in the database. The version of nzLua used for each UDX can be determined based on the name of the LIBNZLUA library shown in the dependencies column.

Example

```
select function, dependencies, arguments, returns
from _v_function
where dependencies like '%NZLUA%'
order by 1;
```

_v_aggregate

The _v_aggregate view lists all of the aggregates (user-defined as well as built-in) that are available in the Netezza database. All of the aggregates that have been created with nzLua will have a dependency for the LIBNZLUA library.

Column	Description
AGGREGATE	The name of the aggregate.
ARGUMENTS	The argument types accepted by the aggregate.
STATE	Tthe types of the state variables for the aggregate. See UDA API Methods.
VARARGS	True/false indicator of whether the aggregate is a varargs aggregate. See getArgs().
RETURN_TYPE	The result type(s) of the aggregate.
DEPENDENCIES	The libraries which an aggregate depends on. All nzLua UDXs have a dependency on the LIBNZLUA library. Dependencies for other libraries may also exist, based on usage of the require function.

The query below can be used to display all of the nzLua user-defined functions and table functions that have been installed in the database. The version of nzLua used for each UDX can be determined based off the name of the LIBNZLUA library shown in the dependencies column.

Example

```
select aggregate, arguments, return_type, state, dependencies
from _v_aggregate
where dependencies like '%NZLUA%'
order by 1;
```

CHAPTER 3
The Lua Programming Language

This section of the documentation is a modified version of the Lua reference documentation, which is available at www.lua.org. This section describes the lexis, the syntax, and the semantics of nzLua. In other words, it describes which tokens are valid, how they can be combined, and what their combinations mean. Some features of Lua have been removed and nzLua also extends the Lua syntax and keywords to be more familiar to a SQL programmer.

The language constructs will be explained using the usual extended BNF notation, in which {*a*} means 0 or more *a*'s, and [*a*] means an optional *a*. Non-terminals are shown like non-terminal, keywords are shown like **kword**, and other terminal symbols are shown like `=´.

Lexical Conventions

Names (also called *identifiers*) in Lua can be any string of letters, digits, and underscores, not beginning with a digit. This coincides with the definition of names in most languages. (The definition of letter depends on the current locale: any character considered alphabetic by the current locale can be used in an identifier.) Identifiers are used to name variables and table fields.

The following *keywords* are reserved and cannot be used as names:

```
and       break     do        else      elseif
end       false     for       function  if
in        local     nil       not       null      or
repeat    return    then      true      until     while
```

Lua is a case-sensitive language: `and` is a reserved word, but `And` and `AND` are two different, valid names. As a convention, names starting with an underscore followed by uppercase letters (such as _VERSION) are reserved for internal global variables used by Lua.

The following strings denote other tokens:

```
+     -     *     /     %     ^     #
==    ~=    <=    >=    <     >     =     <>    !=
(     )     {     }     [     ]
;     :     ,     .     ->    ..    ||    …
```

Literal strings can be delimited by matching single or double quotes, and can contain the following C-like escape sequences: '`\a`' (bell), '`\b`' (backspace), '`\f`' (form feed), '`\n`' (newline), '`\r`' (carriage return), '`\t`' (horizontal tab), '`\v`' (vertical tab), '`\\`' (backslash), '`\"`' (quotation mark [double quote]), and '`\'`' (apostrophe [single quote]). Moreover, a backslash followed by a real newline results in a newline in the string. A character in a string can also be specified by its numerical value using the escape sequence `\`*ddd*, where *ddd* is a sequence of up to three decimal digits. (Note that if a numerical escape is to be followed by a digit, it must be expressed using exactly three digits.) Strings in Lua can contain any 8-bit value, including embedded zeros, which can be specified as '`\0`'.

Literal strings can also be defined using a long format enclosed by *long brackets*. We define an *opening long bracket of level n* as an opening square bracket followed by *n* equal signs followed by another opening square bracket. So, an opening long bracket of level 0 is written as `[[`, an opening long bracket of level 1 is written as `[=[`, and so on. A *closing long bracket* is defined similarly; for instance, a closing long bracket of level 4 is written as `]====]`. A long string starts with an opening long bracket of any level and ends at the first closing long bracket of the same level. Literals in this bracketed form can run for several lines, do not interpret any escape sequences, and ignore long brackets of any other level. They can contain anything except a closing bracket of the proper level.

```
str1 = "this is a string"
str2 = 'this is also a string'
str3 = [[this string uses level 0 long brackets]]
str4 = [=[this string uses level 1 long brackets]=]
str5 = [==[this string uses level 2 long brackets]==]
```

For convenience, when the opening long bracket is immediately followed by a newline, the newline is not included in the string. As an example, in a system using ASCII (in which '`a`' is coded as 97, newline is coded as 10, and '`1`' is coded as 49), the five literal strings below denote the same string:

```
a = 'alo\n123"'
a = "alo\n123\""
a = '\97lo\10\04923"'
a = [[alo
123"]]
a = [==[
alo
123"]==]
```

A *numerical constant* can be written with an optional decimal part and an optional decimal exponent. Lua also accepts integer hexadecimal constants, by prefixing them with `0x`. Examples of valid numerical constants are

```
3   3.0   3.1416   314.16e-2   0.31416E1   0xff   0x56
```

A *comment* starts with a double hyphen (`--`) anywhere outside a string. If the text immediately after `--` is not an opening long bracket, the comment is a *short comment*, which runs until the end of the line. Otherwise, it is a *long comment*, which runs until the corresponding closing long bracket. Long comments are frequently used to disable code temporarily. Long comments use the same form of

brackets as are used for long strings. A level 0 long comment starts with --[[and ends with]]. A level 1 long comment starts with --[=[and ends with]=], etc.

```
-- This is a short comment

--[[
This is a long comment
]]

--[=[
This long comment uses level 1 brackets
]=]
```

Values and Types

Lua is a *dynamically typed language*. This means that variables do not have types; only values do. There are no type definitions in the language. All values carry their own type.

All values in Lua are *first-class values*. This means that all values can be stored in variables, passed as arguments to other functions, and returned as results.

There are six basic types in nzLua: *null*, *boolean*, *number*, *string*, *function*, and *table*.

Null is the type of the value **null**, whose main property is to be different from any other value; it usually represents the absence of a useful value. The Lua parser has been extended for nzLua so that the keyword **null** is equivalent to the standard Lua keyword **nil**. Unlike a NULL value in SQL, **null** (and **nil**) does equal itself, therefore **null** == **null** returns **true**.

Boolean is the type of the values **false** and **true**. Both **null** and **false** make a condition false; any other value makes it true.

Number represents real (double-precision floating-point) numbers.

String represents arrays of characters. Lua is 8-bit clean: strings can contain any 8-bit character, including embedded zeros ('`\0`').

The type *table* implements associative arrays, that is, arrays that can be indexed not only with numbers, but with any value (except **null**). Tables can be *heterogeneous*; that is, they can contain values of all types (except **null**). Tables are the sole data structuring mechanism in Lua; they can be used to represent ordinary arrays, symbol tables, sets, records, graphs, trees, etc. To represent records, Lua uses the field name as an index. The language supports this representation by providing **`a.name`** as syntactic sugar for `a["name"]`. There are several convenient ways to create tables in Lua (see Table Constructors).

Like indices, the value of a table field can be of any type (except **null**). In particular, because functions are first-class values, table fields can contain functions. Thus tables can also carry *methods* (see Function Definitions).

Tables and functions values are *objects*: variables do not actually *contain* these values, only *references* to them. Assignment, parameter passing, and function returns always manipulate references to such values; these operations do not imply any kind of copy.

Coercion

Lua provides automatic conversion between string and number values at run time. Any arithmetic operation applied to a string tries to convert this string to a number, following the usual conversion rules. Conversely, whenever a number is used where a string is expected, the number is converted to a string, in a reasonable format. For complete control over how numbers are converted to strings, use the `format` function from the string library.

Note that coercion only applies to arithmetic operations and does not apply to comparison operations. Any comparison between two variables of different types will always return false (see Relational Operators).

Variables

Variables are places that store values. There are three kinds of variables in Lua: global variables, local variables, and table fields.

A single name can denote a global variable or a local variable (or a function's formal parameter, which is a particular kind of local variable):

```
var ::= Name
```

Name denotes identifiers, as defined in Lexical Conventions.

Any variable is assumed to be global unless explicitly declared as a local (see Local Declarations). Local variables are *lexically scoped*: local variables can be freely accessed by functions defined inside their scope (see Visibility Rules).

Before the first assignment to a variable, its value is **null**.

Square brackets are used to index a table:

```
var ::= prefixexp `[´ exp `]´
```

The syntax `var.Name` is just syntactic sugar for **`var["Name"]`**:

```
var ::= prefixexp `.´ Name
```

All global variables live as fields in ordinary Lua tables, called *environment tables* or simply *environments*. Each function has its own reference to an environment, so that all global variables in this function will refer to this environment table. When a function is created, it inherits the environment from the function that created it. To get the environment table of a Lua function, you call getfenv. To replace it, you call setfenv.

An access to a global variable `x` is equivalent to `_env.x`, where `_env` is the environment of the running function. (The `_env` variable is not accessible in Lua. We use them here only for explanatory purposes.)

Statements

Lua supports an almost conventional set of statements, similar to those in Pascal or C. This set includes assignments, control structures, function calls, and variable declarations.

Chunks

The unit of execution of Lua is called a *chunk*. A chunk is simply a sequence of statements, which are executed sequentially. Each statement can be optionally followed by a semicolon:

```
chunk ::= {statement [`;´]}
```

There are no empty statements and thus '`;;`' is not legal.

Lua handles a chunk as the body of an anonymous function with a variable number of arguments (see Function Calls as Statements). As such, chunks can define local variables, receive arguments, and return values.

A chunk can be stored in a file or in a string inside the host program. To execute a chunk, Lua first pre-compiles the chunk into instructions for a virtual machine, and then it executes the compiled code with an interpreter for the virtual machine. Frequently used code paths will be dynamically compiled into 80x86 machine instructions by the JIT compiler.

Blocks

A block is a list of statements; syntactically, a block is the same as a chunk:

```
block ::= chunk
```

A block can be explicitly delimited to produce a single statement:

```
statement ::= do block end
```

Explicit blocks are useful to control the scope of variable declarations. Explicit blocks are also sometimes used to add a **return** or **break** statement in the middle of another block (see Control Structures).

Assignment

Lua allows multiple assignments. Therefore, the syntax for assignment defines a list of variables on the left side and a list of expressions on the right side. The elements in both lists are separated by commas:

```
stat ::= varlist `=´ explist
varlist ::= var {`,´ var}
explist ::= exp {`,´ exp}
```

Expressions are discussed in Expressions.

Before the assignment, the list of values is *adjusted* to the length of the list of variables. If there are

more values than needed, the excess values are thrown away. If there are fewer values than needed, the list is extended with as many **null**s as needed. If the list of expressions ends with a function call, then all values returned by that call enter the list of values, before the adjustment (except when the call is enclosed in parentheses; see Expressions).

The assignment statement first evaluates all its expressions and only then are the assignments performed. Thus the code

```
i = 3
i, a[i] = i+1, 20
```

sets `a[3]` to 20, without affecting `a[4]` because the `i` in `a[i]` is evaluated (to 3) before it is assigned 4. Similarly, the line

```
x, y = y, x
```

exchanges the values of `x` and `y`, and

```
x, y, z = y, z, x
```

cyclically permutes the values of `x`, `y`, and `z`.

An assignment to a global variable `x = val` is equivalent to the assignment `_env.x = val`, where `_env` is the environment of the running function. (The `_env` variable is not defined in Lua. We use it here only for explanatory purposes.)

Control Structures

The control structures **if**, **while**, and **repeat** have the usual meaning and familiar syntax:

```
statement ::= while exp do block end
statement ::= repeat block until exp
statement ::= if exp then block
              {elseif exp then block}
              [else block]
              end
```

Lua also has a **for** statement, in two flavors (see Control Structures).

The condition expression of a control structure can return any value. Both **false** and **null** are considered false. All values different from **null** and **false** are considered true (in particular, the number 0 and the empty string are also true).

In the **repeat–until** loop, the inner block does not end at the **until** keyword, but only after the condition. So, the condition can refer to local variables declared inside the loop block.

The **return** statement is used to return values from a function or a chunk (which is just a function). Functions and chunks can return more than one value, and so the syntax for the **return** statement is

```
statement ::= return [explist]
```

The **break** statement is used to terminate the execution of a **while**, **repeat**, or **for** loop, skipping to the next statement after the loop:

```
statement ::= break
```

A **break** ends the innermost enclosing loop.

The **return** and **break** statements can only be written as the *last* statement of a block. If it is really necessary to **return** or **break** in the middle of a block, then an explicit inner block can be used, as in the idioms `do return end` and `do break end`, because now **return** and **break** are the last statements in their (inner) blocks.

For Statement

The **for** statement has two forms: one numeric and one generic.

The numeric **for** loop repeats a block of code while a control variable runs through an arithmetic progression. It has the following syntax:

```
statement ::= for name `=´ exp `,´ exp [`,´ exp] do block end

for i=1,1000 do
    t[i] = t[i] + 1
end
```

The *block* is repeated for *name* starting at the value of the first *exp*, until it passes the second *exp* by steps of the third *exp*. More precisely, a **for** statement like

```
for v = e1, e2, e3 do block end
```

is equivalent to the code:

```
do
  local var, limit, step = tonumber(e1), tonumber(e2), tonumber(e3)
  if not (var and limit and step) then error() end
  while (step > 0 and var <= limit) or (step <= 0 and var >= limit)
do
    local v = var
    block
    var = var + step
  end
end
```

Note the following:

- All three control expressions are evaluated only once, before the loop starts. They must all result in numbers.
- *var*, *limit*, and *step* are invisible variables. The names shown here are for explanatory purposes only.
- If the third expression (the step) is absent, then a step of 1 is used.

You can use **break** to exit a **for** loop.

The loop variable `v` is local to the loop; you cannot use its value after the **for** ends or is broken. If you need this value, assign it to another variable before breaking or exiting the loop.

The generic **for** statement works over functions, called *iterators*. On each iteration, the iterator

function is called to produce a new value, stopping when this new value is **null**. The generic **for** loop has the following syntax:

```
stat ::= for namelist in explist do block end

namelist ::= Name {`,´ Name}
```

The pairs function is a build in iterator function which returns all of the key/value pairs in a Lua table.

```
for key,value in pairs(t) do
   block
end
```

A **for** statement like

```
for var_1, ···, var_n in explist do block end
```

is equivalent to the code:

```
do
    local f, s, var = explist
    while true do
         local var_1, ···, var_n = f(s, var)
         var = var_1
         if var == null then break end
         block
    end
end
```

Note the following:

- *explist* is evaluated only once. Its results are an *iterator* function, a *state*, and an initial value for the first *iterator variable*.
- *f*, *s*, and *var* are invisible variables. The names are here for explanatory purposes only.
- You can use **break** to exit a **for** loop.
- The loop variables *var_i* are local to the loop; you cannot use their values after the **for** ends. If you need these values, then assign them to other variables before breaking or exiting the loop.

Function Calls as Statements

To allow possible side-effects, function calls can be executed as statements:

```
statement ::= functioncall
```

In this case, all returned values are thrown away. Function calls are explained in Function Calls.

Local Declarations

Local variables can be declared anywhere inside a block to limit the variable's scope. In addition to limiting the scope, accessing a local variable is faster than accessing a global variable. For both of these reasons, local variables should be used whenever possible. The declaration can include an initial assignment:

```
      statement ::= local namelist [`=´ explist]
local x
local y = 12345
```

If present, an initial assignment has the same semantics of a multiple assignment (see Assignment). Otherwise, all variables are initialized with **null**.

```
local a,b = 'abc', 'xyz'
```

A chunk is also a block (see Chunks), and so local variables can be declared in a chunk outside any explicit block. The scope of such local variables extends until the end of the chunk.

The visibility rules for local variables are explained in Visibility Rules.

Expressions

The basic expressions in Lua are the following:

```
exp ::= prefixexp
exp ::= nil | null | false | true
exp ::= Number
exp ::= String
exp ::= function
exp ::= tableconstructor
exp ::= `...´
exp ::= exp binop exp
exp ::= unop exp
prefixexp ::= var | functioncall | `(´ exp `)´
```

Numbers and literal strings are explained in Lexical Conventions; variables are explained in Variables; function definitions are explained in Function Definitions; function calls are explained in Function Calls; table constructors are explained in Table Constructors. Vararg expressions, denoted by three dots ('. . .'), can only be used when directly inside a vararg function; they are explained in Function Definitions.

Binary operators comprise arithmetic operators (see Arithmetic Operators), relational operators (see Relational Operators), logical operators (see Logical Operators), and the concatenation operator (see Concatenation). Unary operators comprise the unary minus (see Arithmetic Operators), the unary **not** (see Logical Operators), and the unary *length operator* (see The Length Operator).

Both function calls and vararg expressions can result in multiple values. If an expression is used as a statement, which is only possible for function calls (see Precedence), then its return list is adjusted to zero elements, thus discarding all returned values. If an expression is used as the last (or the only) element of a list of expressions, then no adjustment is made (unless the call is enclosed in parentheses). In all other contexts, Lua adjusts the result list to one element, discarding all values except the first one.

Here are some examples:

```
f()                 -- adjusted to 0 results
g(f(), x)           -- f() is adjusted to 1 result
g(x, f())           -- g gets x plus all results from f()
```

```
a,b,c = f(), x     -- f() is adjusted to 1 result (c gets null)
a,b = ...          -- a gets the first vararg parameter, b gets
                   -- the second (both a and b can get null if there
                   -- is no corresponding vararg parameter)
a,b,c = x, f()     -- f() is adjusted to 2 results
a,b,c = f()        -- f() is adjusted to 3 results
return f()         -- returns all results from f()
return ...         -- returns all received vararg parameters
return x,y,f()     -- returns x, y, and all results from f()
{f()}              -- creates a list with all results from f()
{...}              -- creates a list with all vararg parameters
{f(), null}         -- f() is adjusted to 1 result
```

Any expression enclosed in parentheses always results in only one value. Thus, `(f(x,y,z))` is always a single value, even if `f` returns several values. (The value of `(f(x,y,z))` is the first value returned by `f` or **null** if `f` does not return any values.)

Arithmetic Operators

Lua supports the usual arithmetic operators: the binary + (addition), – (subtraction), * (multiplication), / (division), % (modulo), and ^ (exponentiation); and unary – (negation). If the operands are numbers, or strings that can be converted to numbers (see Coercion), then all operations have the usual meaning. Exponentiation works for any exponent. For instance, `x^(-0.5)` computes the inverse of the square root of `x`. Modulo is defined as

```
a % b == a - math.floor(a/b)*b
```

That is, it is the remainder of a division that rounds the quotient towards minus infinity.

Relational Operators

The relational operators in nzLua are

```
==     ~=     <     >     <=     >=     !=     <>
```

These relational operators always result in **false** or **true**.

Equality (==) first compares the type of its operands. If the types are different, then the result is **false**. Otherwise, the values of the operands are compared. Numbers and strings are compared in the usual way. Objects (tables, userdata, and functions) are compared by *reference*: two objects are considered equal only if they are the *same* object. Every time you create a new object (a table, userdata, thread, or function), this new object is different from any previously existing object.

The conversion rules of Coercion *do not* apply to equality comparisons. Thus, `"0"==0` evaluates to **false**, and `t[0]` and `t["0"]` denote different entries in a table.

The operators ~=, <>, and != are exactly the negation of equality (==).

The order in which relational operators work is as follows:

- If both arguments are numbers, they are compared as such.
- If both arguments are strings, their values are compared according to the current locale.

Logical Operators

The logical operators in Lua are **and**, **or**, and **not**. Like the control structures (see Control Structures), all logical operators consider both **false** and **null** as false and anything else as true.

The negation operator **not** always returns **false** or **true**. The conjunction operator **and** returns its first argument if this value is **false** or **null**; otherwise, **and** returns its second argument. The disjunction operator **or** returns its first argument if this value is different from **null** and **false**; otherwise, **or** returns its second argument. Both **and** and **or** use short-cut evaluation; that is, the second operand is evaluated only if necessary. Here are some examples:

```
10 or 20            --> 10
10 or error()       --> 10
null or "a"         --> "a"
null and 10         --> null
false and error()   --> false
false and           --> false
false or null       --> null
10 and 20           --> 20
```

(In this guide, `-->` indicates the result of the preceding expression.)

Concatenation

The string concatenation operator in Lua is denoted by two dots ('`..`') and nzLua also supports using the SQL style concatenate operation using the two pipe characters ('`||`'). If both operands are strings or numbers, then they are converted to strings according to the rules mentioned in Coercion.

```
str1 = "foo" .. "bar"
str2 = str1 || "baz"
```

The Length Operator

The length operator is denoted by the unary operator `#`. The length of a string is its number of bytes (that is, the usual meaning of string length when each character is one byte).

The length of a table `t` is defined to be any integer index `n` such that `t[n]` is not **null** and `t[n+1]` is **null**; moreover, if `t[1]` is **null**, `n` can be zero. For a regular array, with non-null values from 1 to a given `n`, its length is exactly that `n`, the index of its last value. If the array has "holes" (that is, **null** values between other non-null values), then `#t` can be any of the indices that directly precedes a **null** value (that is, it may consider any such **null** value as the end of the array).

Precedence

Operator precedence in Lua follows the table below, from lower to higher priority:

```
or
and
<     >     <=    >=    ~=    ==    <>    !=
..    ||
+     -
```

```
*      /      %
not    #      - (unary)
^
```

As usual, you can use parentheses to change the precedence of an expression. The concatenation ('. .' and '||') and exponentiation ('^') operators are right associative. All other binary operators are left associative.

Table Constructors

Table constructors are expressions that create tables. Every time a constructor is evaluated, a new table is created. A constructor can be used to create an empty table or to create a table and initialize some of its fields. The general syntax for constructors is

```
tableconstructor ::= `{´ [fieldlist] `}´
fieldlist ::= field {fieldsep field} [fieldsep]
field ::= `[´ exp `]´ `=´ exp | Name `=´ exp | exp
fieldsep ::= `,´ | `;´
```

Create an empty table.

```
T = {}
```

Create a table as an array where t[1] = 12, t[2] = 34, and t[3] = 56.

```
t = { 12, 34, 56 }
```

Each field of the form `[exp1] = exp2` adds an entry to the table with key `exp1` and value `exp2`. The code below creates a table which is identical to t = {12,34,56}.

```
t = { [1] = 12, [2] = 34, [3] = 56 }
```

A field of the form `name = exp` is equivalent to `["name"] = exp`. The code below creates a table where t["a"] = 12, t["b"] = 23, and t["c"] = 56. The two tables created below are equivalent.

```
t1 = { a = 12, b = 23, c = 56 }
t2 = { ["a"] = 12, ["b"] = 23, ["c"] = 56 }
```

Fields of the form `exp` are equivalent to `[i] = exp`, where `i` are consecutive numerical integers, starting with 1. Fields in the other formats do not affect this counting. The two tables created below are equivalent.

```
t1 = { 12, 34, 56 }
t2 = { [3] = 56, 12, 34 }
```

If the last field in the list has the form exp and the expression is a function call or a vararg expression, then all values returned by this expression enter the list consecutively (see Function Calls). To avoid this, enclose the function call or the vararg expression in parentheses (see Expressions).

```
function foo() return 34, 56 end
t = { 12, foo() }
```

Vararg function which creates a table that contains all of the arguments passed to the function. In this case the resulting table is the same as t = { 12, 34, 56 }

```
function maketable(...) return { ... } end
t = maketable(12,34,56)
```

The field list can have an optional trailing separator, as a convenience for machine-generated code. The table below is a valid table initialization because the , after the third value is ignored.

```
t = { 12, 34, 56, }
```

Function Calls

A function call in Lua has the following syntax:

```
functioncall ::= prefixexp args
```

In a function call, first prefixexp and args are evaluated. If the value of prefixexp has type *function*, then this function is called with the given arguments. Otherwise, the prefixexp "call" metamethod is called, having as first parameter the value of prefixexp, followed by the original call arguments.

The form

```
functioncall ::= prefixexp `:´ Name args
```

can be used to call "methods". A call `v:name(`*`args`*`)` is syntactic sugar for `v.name(v,`*`args`*`)`, except that `v` is evaluated only once.

Arguments have the following syntax:

```
args ::= `(´ [explist] `)´
args ::= tableconstructor
args ::= String
```

All argument expressions are evaluated before the call. A call of the form `f{`*`fields`*`}` is syntactic sugar for `f({`*`fields`*`})`; that is, the argument list is a single new table. A call of the form `f'`*`string`*`'` (or `f"`*`string`*`"` or `f[[`*`string`*`]]`) is syntactic sugar for `f('`*`string`*`')`; that is, the argument list is a single literal string.

As an exception to the free-format syntax of Lua, you cannot put a line break before the '`(`' in a function call. This restriction avoids some ambiguities in the language. If you write

```
a = f
(g).x(a)
```

Lua would see that as a single statement, `a = f(g).x(a)`. So, if you want two statements, you must add a semi-colon between them. If you actually want to call `f`, you must remove the line break before `(g)`.

A call of the form `return` *functioncall* is called a *tail call*. Lua implements *proper tail calls* (or *proper tail recursion*): in a tail call, the called function reuses the stack entry of the calling function. Therefore, there is no limit on the number of nested tail calls that a program can execute. However, a tail call erases any debug information about the calling function. Note that a tail call only happens with a particular syntax, where the **return** has one single function call as argument; this syntax makes the calling function return exactly the returns of the called function. So, none of the following examples are tail calls:

```
return (f(x))        -- results adjusted to 1
```

```
return 2 * f(x)
return x, f(x)       -- additional results
f(x); return         -- results discarded
return x or f(x)     -- results adjusted to 1
```

Function Definitions

The syntax for function definition is

```
function ::= function funcbody
funcbody ::= `(´ [parlist] `)´ block end
```

The following syntactic sugar simplifies function definitions:

```
statement ::= function funcname funcbody
statement ::= local function Name funcbody
funcname ::= Name {`.´ Name} [`:´ Name]
```

The statement

```
function f () body end
```

translates to

```
f = function () body end
```

The statement

```
function t.a.b.c.f () body end
```

translates to

```
t.a.b.c.f = function () body end
```

The statement

```
local function f () body end
```

translates to

```
local f; f = function () body end
```

not to

```
local f = function () body end
```

(This only makes a difference when the body of the function contains references to `f`.)

A function definition is an executable expression, whose value has type *function*. When Lua pre-compiles a chunk, all its function bodies are pre-compiled too. Then, whenever Lua executes the function definition, the function is *instantiated* (or *closed*). This function instance (or *closure*) is the final value of the expression. Different instances of the same function can refer to different external local variables and can have different environment tables.

Parameters act as local variables that are initialized with the argument values:

```
parlist ::= namelist [`,´ `...´] | `...´
```

When a function is called, the list of arguments is adjusted to the length of the list of parameters, unless the function is a variadic or *vararg function*, which is indicated by three dots ('. . .') at the end

of its parameter list. A vararg function does not adjust its argument list; instead, it collects all extra arguments and supplies them to the function through a *vararg expression*, which is also written as three dots. The value of this expression is a list of all actual extra arguments, similar to a function with multiple results. If a vararg expression is used inside another expression or in the middle of a list of expressions, then its return list is adjusted to one element. If the expression is used as the last element of a list of expressions, then no adjustment is made (unless that last expression is enclosed in parentheses).

As an example, consider the following definitions:

```
function f(a, b) end
function g(a, b, ...) end
function r() return 1,2,3 end
```

Then, we have the following mapping from arguments to parameters and to the vararg expression:

```
CALL            PARAMETERS
f(3)             a=3, b=null
f(3, 4)          a=3, b=4
f(3, 4, 5)       a=3, b=4
f(r(), 10)       a=1, b=10
f(r())           a=1, b=2

g(3)             a=3, b=null, ... -->  (nothing)
g(3, 4)          a=3, b=4,    ... -->  (nothing)
g(3, 4, 5, 8)    a=3, b=4,    ... -->  5  8
g(5, r())        a=5, b=1,    ... -->  2  3
```

Results are returned using the **return** statement (see Control Structures). If control reaches the end of a function without encountering a **return** statement, then the function returns with no results.

Visibility Rules

Lua is a lexically scoped language. The scope of variables begins at the first statement *after* their declaration and lasts until the end of the innermost block that includes the declaration. Consider the following example:

```
x = 10                -- global variable
do                    -- new block
  local x = x         -- new 'x', with value 10
  print(x)            --> 10
  x = x+1
  do                  -- another block
    local x = x+1     -- another 'x'
    print(x)          --> 12
  end
  print(x)            --> 11
end
print(x)              --> 10  (the global one)
```

Notice that, in a declaration like `local x = x`, the new `x` being declared is not in scope yet, and so the second `x` refers to the outside variable.

Because of the lexical scoping rules, local variables can be freely accessed by functions defined inside their scope. A local variable used by an inner function is called an *upvalue*, or *external local variable*, inside the inner function.

Notice that each execution of a **local** statement defines new local variables. Consider the following example:

```
a = {}
local x = 20
for i=1,10 do
  local y = 0
  a[i] = function () y=y+1; return x+y end
end
```

The loop creates ten closures (that is, ten instances of the anonymous function). Each of these closures uses a different `y` variable, while all of them share the same `x`.

Error Handling

Because nzLua is embedded inside of the Netezza database, all nzLua actions start with the database calling a function in the nzLua code. Whenever an error occurs, control returns to the Netezza database and the SQL statement will be aborted with the appropriate error message.

Lua code can explicitly generate an error by calling the error function. If you need to trap errors in Lua, you can use the pcall or xpcall functions.

```
ok,result1,result2 = pcall(myfunction, arg1, arg2, ...);
if not ok then error("myfunction failed!") end
```

Garbage Collection

Lua performs automatic memory management. This means that you have to worry neither about allocating memory for new objects nor about freeing it when the objects are no longer needed. Lua manages memory automatically by running a *garbage collector* from time to time to collect all *dead objects* (that is, objects that are no longer accessible from Lua). All memory used by Lua is subject to automatic management: tables, userdata, functions, strings, etc.

When working with very large data elements it may be necessary to force an immediate garbage collection cycle after dereferencing a variable to prevent nzLua from running out of memory.

```
bigtable = null
collectgarbage()
```

Closures

Lua is a functional programming language with full support for closures. When a new function is created, any local variables become bound to that instance of the function. This can be used to create iterator functions as well as to create private variables for object oriented programming.

One of the simplest examples of using a closure is to create an iterator function which can be used in

a for loop.

```
function counter(i)
    local x=0
    return function()
        x=x+1
        if x > i then return null end
        return x
    end
end
```

In the example above, each time the counter function is called, it creates a new iterator function. The values of the local variables x and i are bound to that instance of the iterator function and are not accessible from any other function. Function parameters are always considered to be local variables as are any variables which are declared using the local keyword. The counter function can be used in combination with a for loop as shown below.

```
sum=0
for i in counter(5) do
    for j in counter(5) do
        sum = sum + j
    end
end
```

The closure concept can also be used to support data privacy for object oriented programming as is shown in this next example.

```
function newAccount(balance)
    local t={}

    t.deposit = function(amount)
        balance = balance + amount
        return balance
    end

    t.withdraw = function(amount)
        balance = balance - amount
        return balance
    end

    t.getBalance = function()
        return balance
    end

    return t
end

account = newAccount(1000)
account.deposit(100)
account.withdraw(500)
balance = account.getBalance()
```

It is more common to use metatables (see Metatables) to create objects than it is to use closures, but metatables do not provide a good way to implement data privacy as can be done using closures.

Metatables

Metatables are the Lua method for creating objects and altering the behavior of operators such as +, -, [], etc.

Every value in Lua can have a *metatable*. This *metatable* is an ordinary Lua table that defines the behavior of the original value under certain special operations. You can change several aspects of the behavior of operations over a value by setting specific fields in its metatable. For instance, when a non-numeric value is the operand of an addition, Lua checks for a function in the field `"__add"` in its metatable. If it finds one, Lua calls this function to perform the addition.

We call the keys in a metatable *events* and the values *metamethods*. In the previous example, the event is `"add"` and the metamethod is the function that performs the addition.

Tables and full userdata have individual metatables (although multiple tables and userdata can share their metatables). Values of all other types share one single metatable per type; that is, there is one single metatable for all numbers, one for all strings, etc.

A metatable controls how an object behaves in arithmetic operations, order comparisons, concatenation, length operation, and indexing. For each of these operations Lua associates a specific key called an *event*. When Lua performs one of these operations over a value, it checks whether this value has a metatable with the corresponding event. If so, the value associated with that key (the metamethod) controls how Lua will perform the operation.

Metatables control the operations listed next. Each operation is identified by its corresponding name. The key for each operation is a string with its name prefixed by two underscores, '__'; for instance, the key for operation "add" is the string `"__add"`. The semantics of these operations is better explained by a Lua function describing how the interpreter executes the operation.

Arithmetic metamethods

The arithmetic metamethods define how an object will behave when used within an arithmetic operation. The arithmetic metamethods are listed here:

- __add is called for the + operator
- __sub is called for the - operator
- __mul is called for the * operator
- __div is called for the / operator
- __pow is called for the ^ operator
- __mod is called for the % operator
- __unm is called for negation (for example y = -x)

Example

```
mt={}
mt["__add"] = function(a,b)
    if type(b) == "string" then
        a[#a+1] = b
    elseif getmetatable(a) == getmetatable(b) then
```

```
        for k,v in pairs(b) do
            a[#a+1] = v
        end
    else
        error("Invalid datatype for + operator!",0)
    end
    return a
end

t={}
setmetatable(t,mt);

-- Now use + to call the __add metamethod of the table t
t1 = t + "foo"                  --> t[1] = "foo"
t1 = t + "bar"                  --> t[2] = "bar"
```

Relational metamethods

When a table with a metamethod is used with a comparison operator the __eq, __lt, and __le metamethods are called. Lua only calls the relational metamethods when both tables have the same metatable. Lua always returns false when any relational operator is used to compare two objects which do not have the same metatables.

- __eq is called for the equality operators ==, !=, <>, and ~=
- __lt is called for the less than operator < and >=
- __le is called for the less than or equal to operator <= and >

Note that a>=b is the same as not (a<b), a!=b is not (a==b), and a>b is not(a<=b). The Lua interpreter automatically translates the relational operators so that the developer needs only implement the three metamethods shown above.

Example

```
mt={}
mt["__eq"] = function(a,b)
    for k,v in pairs(a) do
        if b[k] != v then return false end
    end
    for k,v in pairs(b) do
        if a[k] != v then return false end
    end
    return true
end

t1={1,2,3}
t2={1,2,3}
setmetatable(t1,mt)
setmetatable(t2,mt)
if t1 != t2 then error("Tables are not equal!",0) end
```

Table Access metamethods

- __index is called for the [] operator
- __newindex is called for [] used as an assignment (a[3] = value)

The __index and __newindex metamethods are only called when the table does not have a value defined for a given index. For example, given t = { ["a"] = 123, ["b"] = 456}, if t has a metatable with the __index method defined, the __index metamethod would not be called for t["a"] but it would be called for t["foo"].

Example

```
mt={}
mt["__newindex"] = function(t,index,value)
    if type(index) != "number" then
        error("Can only use numbers for table keys!",0)
    end
    rawset(t,index,value) -- use rawset to ignore metamethods
end
t={["foo"]=33}
setmetatable(t,mt)

t["foo"] = 99        --> ok (t["foo"] already defined)
t["bar"] = 11        --> error
t[123] = 456         --> ok
```

Other metamethods

- __metatable is used to protect an object's metatable
- __tostring is called to convert the object into a string

If the __metatable index is defined in a metatable, it is no longer possible to use the setmetatable function to change an object's metatable. Calling getmetatable(object) will return the value stored in the __metatable index.

The __tostring metamethod is called by the tostring() function or in any other context where the object would normally be converted into a string such as when the .. operator is used to concatenate two strings together.

Example

```
mt={}
mt.__metatable = "locked"
mt.__tostring = function(t)
    error("Cannot convert to a string!",0)
end

t={}
setmetatable(t,mt)

setmetatable(t,null)        --> error
str = tostring(t)           --> error
```

The : operator

The : operator is used to call methods on an object. Using the : operator requires that the __index metamethod is defined as the metatable itself. If the __index metamethod is set to the metatable itself, the result is to translate t:test(123) into mt["test"](t,123), where mt is the metatable which has been defined using setmetatable for the table t.

Example

```
mt={}
mt.__index = mt
mt.test = function(self,value)
    if self.total == null then
        self.total = 0
    end
    self.total = self.total + value
    return self.total
end

t={}
setmetatable(t,mt)
x = t:test(123)            --> x = 123
y = t:test(1)              --> y = 124
z = t.total                --> z = 124
```

CHAPTER 4
Lua Functions and Libraries

The standard Lua libraries provide a set of useful functions that operate on the basic Lua data types. Many of the standard Lua libraries have been disabled in nzLua since they allow access to the filesystem or other operating system functions. The standard functions available in nzLua are divided into the following groups:

- Basic Functions
- String Manipulation
- Table Manipulation
- Math Functions

Basic Functions

The basic library provides some core functions to Lua. If you do not include this library in your application, you should check carefully whether you need to provide implementations for some of its facilities.

assert(v [, message])

Issues an error when the value of its argument `v` is false (i.e., **null** or **false**); otherwise, returns all its arguments. `message` is an error message; when absent, it defaults to "assertion failed!"

collectgarbage()

Run a full garbage collection phase to free memory for all objects which are no longer referenced by any variable. Unlike the Lua collectgarbage function, the nzLua collectgarbage function does not accept any arguments and can only be used to run a full garbage collection cycle.

error(message [, level])

Terminates the last protected function called and returns `message` as the error message. Function `error` never returns.

Usually, `error` adds some information about the error position at the beginning of the message. The `level` argument specifies how to get the error position. With level 1 (the default), the error position is where the `error` function was called. Level 2 points the error to where the function that called `error` was called; and so on. Passing the second argument as level 0 avoids the addition of error position information to the message.

Example

```
error( "Something did not work correctly!", 0 )
```

getfenv([f])

Returns the current environment in use by the function. `f` can be a Lua function or a number that specifies the function at that stack level: Level 1 is the function calling `getfenv`. If the given function is not a Lua function, or if `f` is 0, `getfenv` returns the global environment. The default for `f` is 1.

getmetatable(t)

If `object` does not have a metatable, returns **null**. Otherwise, if the object's metatable has a `"__metatable"` field, returns the associated value. Otherwise, returns the metatable of the given object. The getmetatable function in nzLua cannot be used to get the metatable from a userdata value.

ipairs(t)

Returns three values: an iterator function, the table `t`, and 0, so that the construction

```
for i,v in ipairs(t) do body end
```

will iterate over the pairs (`1,t[1]`), (`2,t[2]`), ···, up to the first integer key absent from the table.

loadstring(string [, chunkname])

Compile a string and return the result. To load and run a given string, use the idiom

```
assert(loadstring(s))()
```

When absent, `chunkname` defaults to the given string.

next(table [, index])

Allows a program to traverse all fields of a table. Its first argument is a table and its second argument is an index in this table. `next` returns the next index of the table and its associated value. When called with **null** as its second argument, `next` returns an initial index and its associated value. When

called with the last index, or with **null** in an empty table, `next` returns **null**. If the second argument is absent, then it is interpreted as **null**. In particular, you can use `next(t)` to check whether a table is empty.

The order in which the indices are enumerated is not specified, *even for numeric indices*. (To traverse a table in numeric order, use a numerical **for** or the ipairs function.)

The behavior of `next` is *undefined* if, during the traversal, you assign any value to a non-existent field in the table. You may however modify existing fields. In particular, you may clear existing fields.

pairs(t)

Returns three values: the next function, the table `t`, and **null**, so that the construction

```
for k,v in pairs(t) do body end
```

will iterate over all key–value pairs of table `t`.

See function next for the caveats of modifying the table during its traversal.

Example

```
sum = 0
t = { 1, 1, 2, 3, 5, 8, 13, 21, 34, 55 }
for key, value in pairs(t) do
    if key % 2 == 0 then
        sum = sum + value
    end
end
```

pcall(f, arg1, ···)

Calls function `f` with the given arguments in *protected mode*. This means that any error inside `f` is not propagated; instead, `pcall` catches the error and returns a status code. Its first result is the status code (a Boolean), which is true if the call succeeds without errors. In such case, `pcall` also returns all results from the call, after this first result. In case of any error, `pcall` returns FALSE plus the error message.

Example

```
ok, result = pcall( myfunction, arg1, arg2, arg3 )
if not ok then error( "myfunction() failed!", 0 ) end
```

rawequal(v1, v2)

Checks whether `v1` is equal to `v2`, without invoking any metamethod. Returns a Boolean.

rawget(table, index)

Gets the real value of `table[index]`, without invoking any metamethod. `table` must be a table; `index` may be any value.

rawset(table, index, value)

Sets the real value of `table[index]` to `value`, without invoking any metamethod. `table` must be a table, `index` any value different from NULL, and `value` any Lua value.

This function returns `table`.

select(index, ···)

If `index` is a number, returns all arguments after argument number `index`. Otherwise, `index` must be the string `"#"`, and `select` returns the total number of extra arguments it received.

Example

```
function sumargs(...)
    local sum = 0
    for i=1,select('#', ...) do
        sum = sum + select(i, ...)
    end
    return sum
end
```

setfenv(f, table)

Sets the environment to be used by the given function. `f` can be a Lua function or a number that specifies the function at that stack level: Level 1 is the function calling `setfenv`. `setfenv` returns the given function.

As a special case, when `f` is 0 `setfenv` changes the environment of the running thread. In this case, `setfenv` returns no values.

setmetatable(table, metatable)

Sets the metatable for the given table. (You cannot change the metatable of a userdata type from Lua, only from C.) If `metatable` is **null**, removes the metatable of the given table. If the original metatable has a `"__metatable"` field, raises an error.

This function returns `table`.

tonumber(e [, base])

Tries to convert its argument to a number. If the argument is already a number or a string convertible to a number, then `tonumber` returns this number; otherwise, it returns **null**.

An optional argument specifies the base to interpret the numeral. The base may be any integer between 2 and 36, inclusive. In bases above 10, the letter '`A`' (in either upper or lower case) represents 10, '`B`' represents 11, and so forth, with '`Z`' representing 35. In base 10 (the default), the number can have a decimal part, as well as an optional exponent part (see Values and Types). In other bases, only unsigned integers are accepted.

tostring (e)

Receives an argument of any type and converts it to a string in a reasonable format. For complete control of how numbers are converted, use string.format().

If the metatable of `e` has a "`__tostring`" field, then `tostring` calls the corresponding value with `e` as argument, and uses the result of the call as its result.

type (v)

Returns the type of its only argument, coded as a string. The possible results of this function are "`null`" (a string, not the value **null**), "`number`", "`string`", "`boolean`", "`table`", "`function`", "`thread`", and "`userdata`".

unpack (list [, i [, j]])

Returns the elements from the given table. This function is equivalent to

```
return list[i], list[i+1], ···, list[j]
```

except that the above code can be written only for a fixed number of elements. By default, `i` is 1 and `j` is the length of the list, as defined by the length operator (see The Length Operator).

xpcall (f, err)

This function is similar to pcall, except that you can set a new error handler.

`xpcall` calls function `f` in protected mode, using `err` as the error handler. Any error inside `f` is not propagated; instead, `xpcall` catches the error, calls the `err` function with the original error object, and returns a status code. Its first result is the status code (a boolean), which is true if the call succeeds without errors. In this case, `xpcall` also returns all results from the call, after this first result. In case of any error, `xpcall` returns **false** plus the result from `err`.

String Manipulation

This library provides generic functions for string manipulation, such as finding and extracting substrings, and pattern matching. When indexing a string in Lua, the first character is at position 1 (not at 0, as in C). Indices are allowed to be negative and are interpreted as indexing backwards, from the end of the string. Thus, the last character is at position -1, and so on.

The string library provides all its functions inside the table `string`. It also sets a metatable for strings where the `__index` field points to the `string` table. Therefore, you can use the string functions in object-oriented style. For instance, `string.byte(s, i)` can be written as `s:byte(i)`.

The string library assumes one-byte character encodings.

string.byte (s [, i [, j]])

Returns the internal numerical codes of the characters `s[i]`, `s[i+1]`, ···, `s[j]`. The default value for `i` is 1; the default value for `j` is `i`.

Note that numerical codes are not necessarily portable across platforms.

string.char (···)

Receives zero or more integers. Returns a string with length equal to the number of arguments, in which each character has the internal numerical code equal to its corresponding argument.

Note that numerical codes are not necessarily portable across platforms.

string.find (s, pattern [, init [, plain]])

Looks for the first match of `pattern` in the string `s`. If it finds a match, then `find` returns the indices of `s` where this occurrence starts and ends; otherwise, it returns **null**. A third, optional numerical argument `init` specifies where to start the search; its default value is 1 and can be negative. A value of **true** as a fourth, optional argument `plain` turns off the pattern matching facilities, so the function does a plain "find substring" operation, with no characters in `pattern` being considered "magic". Note that if `plain` is given, then `init` must be given as well.

If the pattern has captures, then in a successful match the captured values are also returned, after the two indices.

string.format (formatstring, ···)

Returns a formatted version of its variable number of arguments following the description given in its first argument (which must be a string). The format string follows the same rules as the `printf` family of standard C functions. The only differences are that the options/modifiers `*`, `l`, `L`, `n`, `p`, and `h` are not supported and that there is an extra option, `q`. The `q` option formats a string in a form suitable to be safely read back by the Lua interpreter: the string is written between double quotes, and all double quotes, newlines, embedded zeros, and backslashes in the string are correctly escaped when written. For instance, the call

```
string.format('%q', 'a string with "quotes" and \n new line')
```

will produce the string:

```
"a string with \"quotes\" and \
 new line"
```

The options `c`, `d`, `E`, `e`, `f`, `g`, `G`, `i`, `o`, `u`, `X`, and `x` all expect a number as argument, whereas `q` and `s` expect a string.

This function does not accept string values containing embedded zeros, except as arguments to the `q` option.

string.gmatch (s, pattern)

Returns an iterator function that, each time it is called, returns the next captures from `pattern` over string `s`. If `pattern` specifies no captures, then the whole match is produced in each call.

As an example, the following loop

```
s = "hello world from Lua"
for w in string.gmatch(s, "%a+") do
  print(w)
end
```

will iterate over all the words from string `s`, printing one per line. The next example collects all pairs `key=value` from the given string into a table:

```
t = {}
s = "from=world, to=Lua"
for k, v in string.gmatch(s, "(%w+)=(%w+)") do
  t[k] = v
end
```

For this function, a '^' at the start of a pattern does not work as an anchor, as this would prevent the iteration.

string.gsub (s, pattern, repl [, n])

Returns a copy of `s` in which all (or the first `n`, if given) occurrences of the `pattern` have been replaced by a replacement string specified by `repl`, which can be a string, a table, or a function. `gsub` also returns, as its second value, the total number of matches that occurred.

If `repl` is a string, then its value is used for replacement. The character `%` works as an escape character: any sequence in `repl` of the form `%`*n*, with *n* between 1 and 9, stands for the value of the *n*-th captured substring (see below). The sequence `%0` stands for the whole match. The sequence `%%` stands for a single `%`.

If `repl` is a table, then the table is queried for every match, using the first capture as the key; if the pattern specifies no captures, then the whole match is used as the key.

If `repl` is a function, then this function is called every time a match occurs, with all captured substrings passed as arguments, in order; if the pattern specifies no captures, then the whole match is passed as a sole argument.

If the value returned by the table query or by the function call is a string or a number, then it is used as the replacement string; otherwise, if it is **false** or **null**, then there is no replacement (that is, the original match is kept in the string).

Here are some examples:

```
x = string.gsub("hello world", "(%w+)", "%1 %1")
--> x="hello hello world world"

x = string.gsub("hello world", "%w+", "%0 %0", 1)
--> x="hello hello world"
```

```
x = string.gsub("hello world from Lua", "(%w+)%s*(%w+)", "%2 %1")
--> x="world hello Lua from"

x = string.gsub("home = $HOME, user = $USER", "%$(%w+)", os.getenv)
--> x="home = /home/roberto, user = roberto"

x = string.gsub("4+5 = $return 4+5$", "%$(.-)%$", function (s)
      return loadstring(s)()
    end)
--> x="4+5 = 9"

local t = {name="lua", version="5.1"}
x = string.gsub("$name-$version.tar.gz", "%$(%w+)", t)
--> x="lua-5.1.tar.gz"
```

string.len (s)

Receives a string and returns its length. The empty string `""` has length 0. Embedded zeros are counted, so `"a\000bc\000"` has length 5.

string.lower (s)

Receives a string and returns a copy of this string with all uppercase letters changed to lowercase. All other characters are left unchanged. The definition of what an uppercase letter is depends on the current locale.

string.match (s, pattern [, init])

Looks for the first *match* of `pattern` in the string `s`. If it finds one, then `match` returns the captures from the pattern; otherwise it returns **null**. If `pattern` specifies no captures, then the whole match is returned. A third, optional numerical argument `init` specifies where to start the search; its default value is 1 and can be negative.

string.rep (s, n)

Returns a string that is the concatenation of `n` copies of the string `s`.

string.reverse (s)

Returns a string that is the string `s` reversed.

string.sub (s, i [, j])

Returns the substring of `s` that starts at `i` and continues until `j`; `i` and `j` can be negative. If `j` is absent, then it is assumed to be equal to -1 (which is the same as the string length). In particular, the call `string.sub(s,1,j)` returns a prefix of `s` with length `j`, and `string.sub(s, -i)` returns a suffix of `s` with length `i`.

string.upper (s)

Receives a string and returns a copy of this string with all lowercase letters changed to uppercase. All other characters are left unchanged. The definition of what a lowercase letter is depends on the current locale.

Patterns

Character Class

A *character class* is used to represent a set of characters. The following combinations are allowed in describing a character class:

***x*:** (where *x* is not one of the *magic characters* ^$()%.[]*+-?) represents the character *x* itself.
.: (a dot) represents all characters.
%a: represents all letters.
%c: represents all control characters.
%d: represents all digits.
%l: represents all lowercase letters.
%p: represents all punctuation characters.
%s: represents all space characters.
%u: represents all uppercase letters.
%w: represents all alphanumeric characters.
%x: represents all hexadecimal digits.
%z: represents the character with representation 0.
%*x*: (where *x* is any non-alphanumeric character) represents the character *x*. This is the standard way to escape the magic characters. Any punctuation character (even the non magic) can be preceded by a '%' when used to represent itself in a pattern.

[*set*]: represents the class which is the union of all characters in *set*. A range of characters can be specified by separating the end characters of the range with a '-'. All classes %*x* described above can also be used as components in *set*. All other characters in *set* represent themselves. For example, [%w_] (or [_%w]) represents all alphanumeric characters plus the underscore, [0-7] represents the octal digits, and [0-7%l%-] represents the octal digits plus the lowercase letters plus the '-' character.

The interaction between ranges and classes is not defined. Therefore, patterns like [%a-z] or [a-%%] have no meaning.

[^*set*]: represents the complement of *set*, where *set* is interpreted as above.

For all classes represented by single letters (%a, %c, etc.), the corresponding uppercase letter represents the complement of the class. For instance, %S represents all non-space characters.

The definitions of letter, space, and other character groups depend on the current locale. In particular, the class [a-z] may not be equivalent to %l.

Pattern Item

A pattern item can be:

- a single character class, which matches any single character in the class;

- a single character class followed by '`*`', which matches 0 or more repetitions of characters in the class. These repetition items will always match the longest possible sequence;
- a single character class followed by '`+`', which matches 1 or more repetitions of characters in the class. These repetition items will always match the longest possible sequence;
- a single character class followed by '`-`', which also matches 0 or more repetitions of characters in the class. Unlike '`*`', these repetition items will always match the *shortest* possible sequence;
- a single character class followed by '`?`', which matches 0 or 1 occurrence of a character in the class;
- `%`*n*, for *n* between 1 and 9; such item matches a substring equal to the *n*-th captured string (see below);
- `%b`*xy*, where *x* and *y* are two distinct characters; such item matches strings that start with *x*, end with *y*, and where the *x* and *y* are *balanced*. This means that, if one reads the string from left to right, counting *+1* for an *x* and *-1* for a *y*, the ending *y* is the first *y* where the count reaches 0. For instance, the item `%b()` matches expressions with balanced parentheses.

Pattern

A *pattern* is a sequence of pattern items. A '`^`' at the beginning of a pattern anchors the match at the beginning of the subject string. A '`$`' at the end of a pattern anchors the match at the end of the subject string. At other positions, '`^`' and '`$`' have no special meaning and represent themselves.

Captures

A pattern can contain sub-patterns enclosed in parentheses; they describe *captures*. When a match succeeds, the substrings of the subject string that match captures are stored (*captured*) for future use. Captures are numbered according to their left parentheses. For instance, in the pattern "`(a*(.)%w(%s*))`", the part of the string matching "`a*(.)%w(%s*)`" is stored as the first capture (and therefore has number 1); the character matching "`.`" is captured with number 2, and the part matching "`%s*`" has number 3.

As a special case, the empty capture `()` captures the current string position (a number). For instance, if we apply the pattern "`()aa()`" on the string "`flaaap`", there will be two captures: 3 and 5.

A pattern cannot contain embedded zeros. Use `%z` instead.

Table Manipulation

This library provides generic functions for table manipulation. It provides all its functions inside the table `table`.

Most functions in the table library assume that the table represents an array or a list. For these functions, when we talk about the "length" of a table we mean the result of the length operator.

table.concat (table [, sep [, i [, j]]])

Given an array where all elements are strings or numbers, returns `table[i]..sep..table[i+1]`

`··· sep..table[j]`. The default value for `sep` is the empty string, the default for `i` is 1, and the default for `j` is the length of the table. If `i` is greater than `j`, returns the empty string.

table.insert (table, [pos,] value)

Inserts element `value` at position `pos` in `table`, shifting up other elements to open space, if necessary. The default value for `pos` is `n+1`, where `n` is the length of the table (see The Length Operator), so that a call `table.insert(t,x)` inserts `x` at the end of table `t`.

table.maxn (table)

Returns the largest positive numerical index of the given table, or zero if the table has no positive numerical indices. (To do its job this function does a linear traversal of the whole table.)

table.remove (table [, pos])

Removes from `table` the element at position `pos`, shifting down other elements to close the space, if necessary. Returns the value of the removed element. The default value for `pos` is `n`, where `n` is the length of the table, so that a call `table.remove(t)` removes the last element of table `t`.

table.sort (table [, comp])

Sorts table elements in a given order, *in-place*, from `table[1]` to `table[n]`, where `n` is the length of the table. If `comp` is given, then it must be a function that receives two table elements, and returns true when the first is less than the second (so that `not comp(a[i+1],a[i])` will be true after the sort). If `comp` is not given, then the standard Lua operator `<` is used instead.

The sort algorithm is not stable; that is, elements considered equal by the given order may have their relative positions changed by the sort.

Mathematical Functions

This library is an interface to the standard C math library. It provides all its functions inside the table `math`.

math.acos (x)

Returns the arc cosine of `x` (in radians).

math.asin (x)

Returns the arc sine of `x` (in radians).

math.atan (x)

Returns the arc tangent of `x` (in radians).

math.atan2 (y, x)

Returns the arc tangent of `y/x` (in radians), but uses the signs of both parameters to find the quadrant of the result. (It also handles correctly the case of `x` being zero.)

math.ceil (x)

Returns the smallest integer larger than or equal to `x`.

math.cos (x)

Returns the cosine of `x` (assumed to be in radians).

math.cosh (x)

Returns the hyperbolic cosine of `x`.

math.deg (x)

Returns the angle `x` (given in radians) in degrees.

math.exp (x)

Returns the value e^x.

math.floor (x)

Returns the largest integer smaller than or equal to `x`.

math.fmod (x, y)

Returns the remainder of the division of `x` by `y` that rounds the quotient towards zero.

math.frexp (x)

Returns `m` and `e` such that $x = m2^e$, `e` is an integer and the absolute value of `m` is in the range *[0.5, 1)* (or zero when `x` is zero).

math.huge

The value `HUGE_VAL`, a value larger than or equal to any other numerical value.

math.ldexp (m, e)

Returns $m2^e$, where `e` should be an integer.

math.log (x)

Returns the natural logarithm of `x`.

math.log10 (x)

Returns the base-10 logarithm of `x`.

math.max (x, ...)

Returns the maximum value among its arguments.

math.min (x, ...)

Returns the minimum value among its arguments.

math.modf (x)

Returns two numbers, the integral part of `x` and the fractional part of `x`.

math.pi

The value of *pi*.

math.pow (x, y)

Returns x^y . The expression `x^y` can also be used to compute this value.

math.rad (x)

Returns the angle `x` (given in degrees) in radians.

math.random ([m [, n]])

This function is an interface to the simple pseudo-random generator function `rand` provided by ANSI C. (No guarantees can be given for its statistical properties.)

When called without arguments, returns a uniform pseudo-random real number in the range *[0,1)*. When called with an integer number `m`, `math.random` returns a uniform pseudo-random integer in the range *[1, m]*. When called with two integer numbers `m` and `n`, `math.random` returns a uniform pseudo-random integer in the range *[m, n]*.

math.randomseed (x)

Sets `x` as the "seed" for the pseudo-random generator: equal seeds produce equal sequences of numbers.

math.sin (x)

Returns the sine of `x` (assumed to be in radians).

math.sinh (x)

Returns the hyperbolic sine of `x`.

math.sqrt (x)

Returns $\sqrt{x}$. The expression `x^0.5` can also be used to compute this value.

math.tan (x)

Returns the tangent of `x` (assumed to be in radians).

math.tanh (x)

Returns the hyperbolic tangent of `x`.

CHAPTER 5
nzLua Functions

In addition to the standard functions offered by Lua, nzLua provides many functions which extend the capabilities of Lua or are designed to be more familiar to developers who know SQL. These functions are divided into the following groups:

- Date/Time Functions
- Encryption and Hashing Functions
- Math Functions
- Netezza Database Functions
- Regular Expression Functions
- String Manipulation
- Other Functions
- Array Module
- Bignum Module
- JSON Module
- StringBuffer Module

Date and Time Functions

The date and time functions operate on nzLua timestamp values. In nzLua, all timestamps are encoded as numbers using the standard POSIX time format (also known as Unix Time), where the value is stored as the number of seconds since January 1, 1970.

add_months(timestamp, months)

Add a number of months to a date. The months value can be positive or negative. If the resulting date is invalid, for example, if you add one month to January 30, 2012, the date is set to the last day of the month.

Example

```
ts = add_months(to_date("2012-01-30", "YYYY-MM-DD"), 1)
```

date_part(units, timestamp)

Extract the given units out of a timestamp. The units value is a string and can be any of the following values:

"day"	"minute"	"second"	"year"
"hour"	"month"	"week"	"dow"
"doy"			

where:

- "dow" returns the correct day of the week, for example, 1 for Sunday as the first day of the week, and 7 for Saturday as the last day of the week.
- "doy" returns the Julian day of the year, for example, 1 for the first day of the year, 365 or 366 for the last day of the year.

Example

```
year = date_part("year", ts )
day = date_part("dow", ts)
```

date_trunc(units, timestamp)

Truncate a date to the given precision. The precision value can be any of the following values:

"day"	"minute"	"second"	"year"
"hour"	"month"	"week"	

Example

```
ts = date_trunc("day", ts )
```

days(number_of_days)

Convert number_of_days to an interval.

Example

```
interval = days(5)
```

hours(number_of_hours)

Convert number_of_hours to an interval.

Example

```
interval = hours(23)
```

interval_decode(interval)

Break an interval down into days, hours, minutes, and seconds. The interval_decode function returns 4 values.

Example

```
days, hours, minutes, seconds = interval_decode(12345678)
```

interval_encode(days [,hours[, minutes[, seconds]]])

Convert interval components into an interval value. In nzLua, an interval is represented as a number of seconds.

Example

```
seconds = interval_encode( 11, 22, 33, 44 )
```

The *tformat* specification

The tformat specification is used by the to_char and to_date functions. The values supported by nzLua are a subset of those allowed by the SQL to_char and to_date functions. With the exceptions of DAY, DY, MON, and MONTH the format characters are not case sensitive (YYYY, yyyy, and YyyY all represent the same thing - a 4 digit year)

AM / A.M.	Meridian indicator
CC	Century
DAY	Full day name
Day	Full day name with first character capitalized
DD	Day in 01-31 format
DDD	Day of the year
DY	Three letter abbreviated day name
Dy	Three letter abbreviated day name with first character capitalized
MI	Minutes in 00-59 format
MM	Month in 01-12 format
MON	Three letter abbreviated month name in upper case
Mon	Three letter abbreviated month name with first character capitalized
MONTH	Full month name in upper case
Month	Full month name with first character capitalized
HH	Hours in 00-12 format
HH12	Hours in 00-12 format
HH24	Hours in 00-23 format
PM / P.M.	Meridian indicator
SS	Seconds in 00-59 format
YY	Two digit year

YYYY	Four digit year

time_decode(timestamp)

Break a timestamp down into year, month, day, hours, minutes, seconds, milliseconds. The milliseconds value is accurate for timestamps in the years 1900 through 2100, but can be off by 1 millisecond for timestamps outside of that range due to rounding issues.

Example

```
year, month, day, hour, min, sec, ms = time_decode(ts)
```

time_encode(year,month,day[,hours[,minutes[,seconds[,milliseconds]]]])

Create a timestamp from the individual time components. Any value which is not specified will default to 0.

Example

```
ts = time_encode( 2010, 1, 1 )
```

to_char(timestamp, *tformat*)

Convert a timestamp into a string using the *tformat* value. The *tformat* specification is similar to the format strings used by the SQL to_char function and is documented in the *tformat* section of this chapter.

Example

```
str = to_char( ts, "YYYY-MM-DD HH24:MI:SS" )
```

to_date(string, *tformat*)

Convert a string into a timestamp using the *tformat* value. The *tformat* specification is similar to the format strings used by the SQL to_date function and is documented in the *tformat* section of this chapter.

Example

```
ts = to_date( "July 05, 2005", "MONTH DD, YYYY")
```

Encryption and Hashing Functions

The encryption functions use a password to transform data in a way that can be reversed using a decryption function along with the password. The output of an encryption function is always as large or larger than the original input. A hashing function transforms the data in a way which is not reversible and is generally much smaller than the original input. Many different inputs will map to the same output for a hashing function, whereas all inputs to a encryption function will generate a unique output for a given password.

crc32(string)

Generate a 32-bit integer value from an input string. The crc32 algorithm is normally used to detect changes or errors in an arbitrarily large chunk of data.

Example

```
checksum = crc32( "the quick brown fox jumps over the lazy dog" )
```

hex(string)

Convert a string into a hexadecimal representation of the string. The input string can be binary data or a normal string.

Example

```
str = hex("Hello world!")
```

md5(string)

Calculate the MD5 hash value for a string. The md5 algorithm generates a 128-bit binary string result.

Example

```
value = md5( "the quick brown fox jumps over the lazy dog")
```

hex(string)

Encode a string by converting each byte into a 2-byte hexadecimal representation.

Example

```
hexed = hex("Hello world!")
```

sha1(string)

Calculate the SHA1 hash value for a string. The sha1 algorithm generates a 160-bit binary string result.

Example

```
value = sha1("the quick brown fox jumps over the lazy dog")
```

unhex(string)

Convert a hex encoded string back to a binary representation.

Example

```
str = unhex("48656C6C6F20776F726C6421")
```

Math Functions

abs(value)

Return the absolute value of a number. The abs function works for normal Lua numbers as well as bignum values.

Examples

```
value = abs(-2392349.82394)
value = abs(bignum.new("-982480924092049823.09283048098234"))
```

nrandom(mean, stddev)

Generate a normally distributed random number given a mean and standard deviation value.

Example

```
num = nrandom( 1000, 100 )
```

random([x [,y]])

Generate a random number. With no arguments, the random() function returns a double value between 0 and 1. With one argument, random() generates an integer value between 1 and x (inclusive). With two arguments, random() generates an integer value between x and y (inclusive).

The random number generator uses the Mersenne Twister algorithm.

Examples

```
if random() < .25 then ... end
value = random(100)
year = random(1990, 2010)
```

round(value [,digits])

Round a value to a specified number of digits. If no digit value is specified, it defaults to 0 and rounds to the nearest integer value.

Example

```
round(1234.123, 2)
```

srandom([value])

Seed the random number generator. The random number generator is automatically seeded using the Linux /dev/urandom file so it is not necessary to use the srandom() function except in situations where an identical series of numbers needs to be generated.

Example

```
srandom(12345)
srandom(1234 + getDatasliceId())
```

trunc(value)

Drop the non-integer portion of a number. No rounding is performed.

Example

```
trunc(1.99999999)
```

Netezza Database Functions

These functions give information about the Netezza environment where a function is executing or produce a result that relies on the SPU or dataslice where the nzLua code is executing.

getCurrentUsername()

Returns the name of the user who invoked the UDX.

getDatasliceId ()

Returns the dataslice identifier where a UDX is currently executing or 0 if the UDX is running on the host system.

getDatasliceCount()

Returns the number of dataslices in the Netezza database.

getLocus()

Returns a value that indicates where the UDX is currently executing. The values returned by getLocus() are one of the following values.

DBOS	The UDX is running on the host system.
SPU	The UDX is running in parallel on the SPUs.
POSTGRES	The UDX is in the compile step (not yet executing).
NZLUA	The code is being executed outside of the Netezza database using the nzlua command line.

getMaxMemory()

Returns the maximum amount of memory in megabytes that can currently be used by nzLua. The max memory usage of an nzLua UDX is capped at 32MB by default. The setMaxMemory function can be used to modify the default memory usage cap.

getMemoryUsage()

Returns the amount of memory currently being consumed by nzLua.

getNextId()

Returns an unique integer value each time the function is called. The first call will return the number getDatasliceId()+1 and each subsequent call will add getDatasliceCount() to the value. This guarantees that when running in parallel, each dataslice will see a different sequence of values.

getSpuCount()

Returns the number of SPUs in the Netezza system. Each SPU contains one or more dataslices.

isFenced()

Returns true if nzLua is running in fenced mode.

isUserQuery()

Returns false if the nzLua UDX is being called while the Netezza database is gathering JIT stats or some other non user generated query, otherwise returns true.

require(library)

The Lua require function has been replaced in nzLua with a custom require function. The nzLua version of require is used to load an nzLua shared code library.

setMaxMemory(mb)

Sets the maximum memory usage cap (in megabytes) for the nzLua UDX. The value can be between 1MB and 512MB; the default memory cap is 32MB. The setMaxMemory function should be called from the initialize() function.

Excessive memory usage by a UDX can severely disrupt performance of the Netezza system during high concurrency workloads. Using more than 128MB may require careful job scheduling in some environments to avoid thrashing due to insufficient RAM.

Example

```
function initialize()
    setMaxMemory(128)
end
```

Regular Expression Functions

The regular expression functions use the standard POSIX extended regular expression syntax. The POSIX syntax is not documented in the nzLua documentation since it is readily available in many forms on the internet, books, and manuals of many other programming languages.

regexp_capture(string, pattern [,start_position])

Return each individual string captured using the regular expression. A capture is any part of the regular expression that has been surrounded with parenthesis. If the optional third argument is given, regexp_capture will skip to specified start_position before attempting to match the pattern.

A capture can contain subcaptures (nested parenthesis), resulting in a portion of the matched string being returned multiple times. In the third example below, the c1 variable will be assigned the value "123def" and the c1sub variable will be assigned the value "123".

If regexp_capture is not able to match the entire pattern, it returns the value null for all captures.

When a capture such as (.*) matches nothing, it is returned as the zero length string ''.

When a capture matches 0 times, such as ([a-z]){0,4} or (ABC)*, the capture is returned as the boolean value false instead of the zero length string '' to indicate that no value was matched.

If the regular expression pattern does not match the input string, the value **null** is returned.

Examples

```
foo, bar = regexp_capture( "foobar", "(f..)(b..)" )
c1, c2, c3 = regexp_capture( "foo;bar;baz", "^(.*);(.*);(.*)$")
c1, c1sub = regexp_capture("abc123def", "(([0-9]+).*)" )
```

regexp_count(string, pattern)

Count the number of matches that are found in the string for the regular expression pattern.

Example

```
vowels = regexp_count("the quick brown fox", "[aeiou]")
```

regexp_extract(string, pattern [,start [,result]])

Return the value that matched a regular expression. When the result parameter is passed, instead of returning the first match, the regexp_extract function will return the *N*th match, where N = result.

Example

```
str = regexp_extract("How much food does a barbarian eat?", "foo|bar")
str = regexp_extract("How much food does a barbarian eat?", "foo|bar", 1,
3)
```

regexp_extract_all(string, pattern)

Return all matches found for the regular expression as a Lua table.

Example

```
t = regexp_extract_all("How much food does a barbarian eat?", "foo|bar" )
```

regexp_find(string, pattern [,start])

Find the start and end position of a pattern within a string. If no match is found, regexp_find returns 0.

Examples

```
start, stop = regexp_find("the quick brown fox", "b[a-z]*")
```

regexp_gmatch(string, pattern)

Return an iterator function that can be used in a for loop to process each substring matched by the regular expression pattern.

Example

```
t={}
for word in regexp_gmatch("the quick brown fox", "[a-zA-Z]+") do
    t[#t+1] = word
end
```

regexp_gsplit(string, pattern)

Return an iterator function which can be used in a for loop to process each substring that does not match the regular expression pattern.

Example

```
t={}
for word in regexp_gsplit("the quick brown fox", "[ ]+") do
    t[#t+1] = word
end
```

regexp_like(string, pattern [,start])

Return true if a string matches a regular expression, otherwise return false. Note that regexp_like does not require the entire string to match, only that some portion of the string matches the regular expression. To check if the entire string matches a pattern, the ^ and $ characters must be used to match the start and end of the string.

Examples

```
if regexp_like("foo,12345,bar", "[0-9]+") then ... end
if regexp_like("9872398479", "^[0-9]+$") then ... end
```

regexp_replace(string, pattern, value)

Replace all occurrences of a regular expression pattern with the specified value. The value string can contain a back reference to a value captured with the regular expression pattern. For example %1 is the first value capture, %2 is the second, etc. Two consecutive % characters must be used in the value string to represent the % character.

Examples

```
str = regexp_replace("The quick brown", "[aeiou]", "" )
str = regexp_replace("foobar", "(...)(...)", "%2%1d" )
```

regexp_split(string, pattern)

Returns a table containing each value found by breaking up a string into each part of the string that does not match the regular expression.

Examples

```
words = regexp_split( "attack at dawn", "[\t\n ]+")
```

String Manipulation

basename(string)

Extract the file name out of a string that contains the directory plus the filename. The directory separator can be either of the Unix style "/" or the Windows style "\" characters.

Example

```
file = basename("/usr/local/nz/bin/nzsql")
```

chr(byteint {, byteint})

Converts one or more byte values into string that has a length equal to the number of arguments.

Examples

```
tab = chr(9)
crlf = chr(13,10)
hello = chr(104,101,108,108,111)
```

dirname(string)

Extract the directory out of a string that contains the directory plus the filename. The directory separator can be either of the Unix style "/" or the Windows style "\" characters.

Example

```
dir = dirname("/usr/local/nz/bin/nzsql")
```

join(table, delimiter)

Append all values in a table together into a string result, each value separated by the delimiter string. The delimiter can be an empty string (or null).

Example

```
t = { "t", "e", "s", "t" }
test = join(t, "" )
```

length(string)

Return the length of a string.

replace(string, search_string, replace_string)

Replace all occurrences of *search_string* in a string with *replace_string*. The replacement string can be an empty string. The replace function is faster than regexp_replace for simple cases where a regular expression is not needed.

Examples

```
str = replace("tree", "ee", "eat")
nospaces = replace("the quick brown fox", " ", "")
```

rpad(string, width [, character])

Make a string *width* characters long. If the string is currently less than *width* characters, make the string *width* characters long by right padding the string with the space character by default or with the specified padding character. If the string is currently more than *width* characters wide, truncate the string to *width* characters by removing characters from the right side of the string.

Examples

```
str = rpad("test", 8)
str = rpad("testing",4)
str = rpad("testing", 20, "#")
```

rtrim(string [,string])

Remove all whitespace characters from the end of the string. Whitespace is defined as the space, tab, newline, formfeed, and vertical tab. A list of characters can be specified using the second argument to make rtrim remove all characters in that string rather than the default whitespace characters.

Examples

```
str = rtrim( "test       ")
str = rtrim( "test#######", "#")
str = rtrim( str, chr(9,10,13,32))
```

split(string, string [, result])

With two parameters, the split function breaks up a string based on a delimiter defined by the second string and returns a table containing all of the resulting tokens. When the three parameter form is used the split function returns a scalar value which is the *N*th value found in the string by tokenizing the string.

Examples

```
t = split( "one;two;three;four", ";" )
three = split("one;two;three;four", ";", 3 )
```

strlen(string)

Return the length of a string.

strpos(string, string)

Return the start and end positions of the first occurrence of a *search_string* within a string.

Example

```
start, stop = strpos( "the quick brown fox", "brown" )
```

trim(string [, string])

Trim whitespace from the start and end of a string. Whitespace is defined as the space, tab, vertical tab, newline, and form feed characters. Optionally, a list of characters can be specified that will be trimmed from the start and end of the string instead of the default whitespace characters.

Examples

```
str = trim( "    test    " )
str = trim( "<test>", "<>" )
```

upper(string)

Convert all characters in a string to upper case. Does not support UTF8 characters.

substr(string, start [, length])

Extract a substring from a string. If no length is specified, substr extracts all values up to the end of the string. If the start position is negative, it is used as an offset from the end of the string.

Examples

```
foo = substr( "foobar", 1, 3 )
bar = substr( "foobar", -3 );
```

urldecode(string)

Decode a string that has been encoded using the RFC3896 standard where unsafe characters have been replaced with their %XX hexadecimal equivalent.

Example

```
str = urldecode("Why+is+the+sky+blue%3F")
```

urlencode(string)

Encode a string using the RFC3896 URL encoding method where unsafe characters are replaced with their equivalent lowercase %xx hexadecimal value and spaces are replaced with the + character. The only characters not encoded are defined by this set [a-zA-Z0-9] and the four characters '_', '.', '-', and '~'.

Example

```
str = urlencode("what do you mean?")
```

urlparsequery(string)

Decode a URL query string into name/value pairs and return the result as a table. The query string is assumed to be encoded using RFC3896 format. If the string contains a '?' character, the

urlparsequery function will ignore all characters prior to the first '?' character.

Example

```
parms = urlparsequery('foo=this+is+foo&bar=23&sparta=This+is+Sparta!')
parms = urlparsequery('http://www.ibm.com/index.html?foo=bar')
```

Other Functions

decode(string, decode_format [, start])

The decode function takes a binary string that has been created using the encode function and converts it back into values based on the format specification. The encode/decode functions are similar to the Perl pack and unpack functions and are very useful for storing multiple values in a VARCHAR field during analytical or ETL processing.

Example

```
str = encode( 'Zi2A10', 'foo', 123456789, 987654321, 'bar' )
str1,int1,int2,str2 = decode(str,'Zi2A10')
```

decode_format specification

The format string consists of a set of characters which indicate the type of argument for each relative position. The format characters for the encode function are:

```
        a     fixed width string, right padded with NULLs
        A     fixed width string, right padded with spaces
        B     bignum (encoded as a null terminated string)
        d     double (8 bytes)
        h     short (2 bytes, -32768 through 32767))
        H     unsigned short (2 bytes, 0 through 65535)
        i     integer (4 bytes, -2147483648 through 2147483647)
        I     unsigned integer (4 bytes, 0 through 4294967295)
        l     long long (8 bytes)
        L     unsigned long long (8 bytes)
        N     fixed width number (ascii encoded)
        p     position within the string (only valid for decode)
        v     variable length binary string up to 65535 bytes
        V     variable length binary string up to 4294967295 bytes
        y     byte (-128 through 127)
        Y     unsigned byte (0 through 255)
        Z     null terminated string

    Modifiers
        >     encode number using big endian format
        <     encode number using little endian format
```

For all format types other than A and N, a number can be specified after the format character to repeat that type some number of times.

The A and N options behaves differently from the other options. Both will always only generate or consume a single fixed-width input value. For the fixed length formats A and N, the encode function

will truncate the value if is larger than the specified width of the field.

Format 'i10' means 10 integers whereas format 'A10' means one fixed-width string that is 10 characters wide. 'N5' means a fixed-width table.maxndecnumber that is 5 characters wide.

The l and L (long long) formats only support 53 bits of information due to Lua's number format being a double which uses 53 bits to store the number and 11 bits to store the exponent.

The < and > operators apply to a single format character. The format string 'i>ii<' encodes the first integer using big endian, the second using native encoding (the default), and the third number using little endian format. The format string 'i10>' indicates to encode 10 integers using big endian format.

encode(format, value1, ...)

The encode function takes a list of arguments and stores them into an efficient binary string representation based on the format specification. The encode/decode functions are similar to the pack/unpack functions in Perl. See the decode_format specification section of this chapter for information on the format characters.

Example

```
str = encode("iiiZ", 111, 222, 333, "this is a string")
```

foreach(table, function)

Pass each value in a table to the specified function.

Example

```
sum=0
function add(value) sum = sum + value end
foreach({1,2,3,4,5}, add)
```

map(table, function)

Pass each key/value pair in a table to a function. The function should return a new key/value pair or null. The map function will return a table containing all of the results from the function.

Example

```
function swap(key,value) return value, key end
t = map({5,4,3,2,1}, swap)
```

nullif(value1, value2)

Return null if value1 == value2, otherwise return value1.

Example

```
value = nullif(100,100)
```

nvl(value1, value2)

If value1 is not null then return value1, otherwise return value2.

Example

```
value = nvl(null,999)
```

pop(table)

Remove the last value from a table and return it.

Example

```
t={}
push(t,1)
push(t,2)
value = pop(t)
```

push(table,value)

Add a new value to the end of a table.

Example

```
t={}
push(t,1)
push(t,2)
value = pop(t)
```

switch(table, value [, ...])

Lua does not provide a switch or case statement, therefore nzLua provides an alternative using a switch function. The first argument to switch is a lookup table that contains a set of functions. The second argument selects which function to call. All additional arguments will be passed to the function that matches the lookup value.

The first argument passed to the functions in the lookup table will always be the value that was used to look up the function. The rest of the arguments will be passed as they were provided to the switch statement. If the lookup value is not found in the table, the switch function calls the default function.

Example

```
ops={}
ops['+'] = function(op,a,b) return a+b end
ops['-'] = function(op,a,b) return a-b end
ops['*'] = function(op,a,b) return a*b end
ops.default = function(op,a,b)
    error( 'Invalid operation: ' || op, 0 )
end

result = switch(ops,'*',6,7)
```

CHAPTER 6
nzLua Libraries

Array Module

The array module allows for the creation of memory efficient arrays that can easily be serialized to a string. A Lua table uses far more memory per element than an array, especially when the array is used to store 8, 16, or 32-bit values. All numerical values in a Lua table are stored as 64-bit double precision numbers and a table also requires additional memory overhead to store the value.

Once an array has been created, the values can be accessed just as they can be with a Lua table. All values in the array are initialized to 0 when the array is created..

Example

```
arr = Array.new(1000, "int32")
for i=1,1000 do arr[i] = i end
```

Array.new(size[, arraytype])

Create a new array that contains size elements. If the arraytype is not specified it defaults to an array of doubles.

Example

```
arr = Array.new(1000)
arr = Array.new(1000, "int32")
arr = Array.new(5000, "uint16")
```

Array.bytes(array)

Return the amount of memory consumed by the array.

Array.size(array)

Return the number of elements which can be stored in the array.

Array.serialize(array)

Encode the array as a string that can be stored in the SPUPad or passed to the Netezza database as a varchar value and later deserialized back into an array.

Example

```
saveString("myarray", Array.serialize(arr))
```

Array.deserialize(string [, arraytype])

Create an array using a string which was generated using the Array.serialize function. If no arraytype is specified, double is assumed. The size of the array created is determined by the length of the string and the arraytype.

Arraytype

Table 1: arraytype Valid Types

Type	Description
int8	8-bit integer with a value between -128 and 127
int16	16-bit integer with a value between -32768 and 32767
int32	32-bit integer with a value between -2147483648 and 2147483647
uint8	8-bit integer with a value between 0 and 255
uint16	16-bit integer with a value between 0 and 65535
uint32	32-bit integer with a value between 0 and 4294966295
double	64-bit IEEE 754 double precision value

BigNum Module

The BigNum module supports mathematical calculations on numbers that are too large to fit into a standard double precision value. A double is limited to approximately 15 digits of precision, whereas a Netezza database numeric value can have up to 38 digits of precision. nzLua encodes numbers that do not fit into a double precision value as BigNum values. The OPT_FORCE_BIGNUM option (see UDX Options) can be used to modify when integer and numeric values will be passed to nzLua as BigNum values instead of doubles.

Although performance when using a BigNum number in a calculation is good, it is still far slower than using a double and should be avoided where possible.

BigNum + x

The + operator creates a new BigNum instance and assigns the result of adding x to it. The x value can be a number, a string, or another BigNum.

Example

```
x = BigNum.new() + "898923424898234234.8982394"
y = x + x
```

Note: BigNum:add(x) is more efficient than BigNum + x because it does not create a new instance of the BigNum.

BigNum - x

The - operator creates a new BigNum instance and assigns to it the value of the BigNum minus x. The x value can be a number, a string, or a BigNum. Use bn:minus(value) instead of bn = bn - value where possible due to higher performance.

Example

```
bn = BigNum.new(1)
result = bn + 1
```

BigNum * x

The * operator creates a new BigNum instance and assigns to it the value of the BigNum multiplied by x. The x value can be a number, a string, or a BigNum. Use bn:mul(value) instead of the * operator where possible for better performance.

Example

```
result = BigNum.new("9898239498234898") * 3.14159
```

BigNum / x

The / operator creates a new BigNum instance and assigns the result of dividing the BigNum by x. The x value can be a number, a string, or a BigNum.

BigNum.abs(BigNum)

Return the absolute value of a BigNum. The standard abs function can also be applied directly to a BigNum value. Using this form of the abs function creates a new BigNum variable, leaving the prior variable unchanged.

Example

```
x = BigNum.abs(x)
```

BigNum:abs()

Directly update a BigNum value to its absolute value. Since this method does not create a new

variable, it operates much faster than the BigNum.abs(value) method.

Example

```
x = BigNum.new("-92834982394898234.2348982349")
x:abs()
```

BigNum:add(value)

Directly add a value to a BigNum variable. This method does not create a new BigNum variable and thus gives higher performance than using the + operator.

Example

```
x:add(y)
x:add(12345)
x:add("9828932322234233423489.2382923423483")
```

BigNum.compare(a,b)

Compare two values, one of which must be a BigNum value. Returns 0 if a equals b, a negative number of a is less than b, or a positive number of a is greater than b.

Example

```
if BigNum.compare(x,y) < 0 then
    error( "X must be >= Y", 0)
end
```

BigNum:compare(x)

Compare a BigNum value to x. Returns 0 if the BigNum equals x, a negative number if the BigNum is less than x, or a positive number if the BigNum is greater than x.

Example

```
if x:compare(y) < 0 then
    error( "X must be >= Y", 0 )
end
```

BigNum:div(x)

Update a BigNum value by dividing it by x, where x can be a string, a Lua number, or another BigNum value. Using BigNum:div(x) is approximately 50% faster than using "x = x / y" because it does not create a new BigNum value, instead it updates the already existing BigNum.

Example

```
x = BigNum.new("98298398234.989823")
x:div(1000)
```

BigNum:eq(x)

Return true if the BigNum value equals x, false otherwise. Lua does not support operator overloading for comparison between two different types (such as comparing a BigNum to a normal number),

therefore the BigNum:compare() or other functions must be used to compare BigNum values to other non BigNum values.

Example

```
x = BigNum.new("12345")
if not x:eq(12345) then
    error( "Something is wrong!", 0 )
end
```

BigNum.forceArg(x, [true|false])

Force argument x to be passed into nzLua as a BigNum value, even if the value would otherwise be passed as a normal nzLua number. This function should be called in the initialize method of the UDX.

The forceArg function is supported for a UDF or UDTF but not for a UDA.

Example

```
function initialize()
    BigNum.forceArg(1,true)
    BigNum.forceArg(3,true)
end
```

BigNum:format(type,precision)

Return a string representation of the BigNum value. The type argument can be one of 'e', 'g', or 'x'

```
e     Use scientific format, example: 1.2345678901234e18
g     Use fixed format, example: 123456789.012345
x     Use hexadecimal format, example: 0x1.2d687e3df217cec28a18p+20
```

The precision argument determines the maximum number of characters which will be used to output the number.

BigNum:ge(value)

Return true if the BigNum is greater than or equal to the argument. The argument can be a number, a string, or another BigNum.

BigNum:gt(value)

Return true if the BigNum is greater than the argument. The argument can be a number, a string, or another BigNum.

BigNum.isbignum(value)

Return true if value is a BigNum or false otherwise.

Example

```
if bignum.isbignum(x) then
    x:add(1)
```

```
else
    x = x + 1
end
```

BigNum:le(value)

Return true if the BigNum is less than or equal to the argument. The argument can be a number, a string, or another BigNum.

BigNum:lt(value)

Return true if the BigNum is less than the argument. The argument can be a number, a string, or another BigNum.

BigNum:mul(value)

Update the BigNum value by multiplying it by the argument. The performance of using BigNum:mul() is much faster than using the * operator, since it does not result in creating a new BigNum variable.

BigNum.neg(BigNum)

Return the negative of a BigNum. Creates a new instance of the BigNum.

BigNum:neg()

Update the value of the BigNum to be its negative, does not create a new instance.

BigNum.new([value [,digits]])

Create a new BigNum variable. If no value is provided, the BigNum is initialized to 0 and 38 digits of precision (128 bits). The value can be a number value, a string that can be converted into a number, or another BigNum value. The acceptable values for precision are between 38 and 305 digits (128 to 1024 bits).

Values larger than 38 digits cannot be returned as a numeric value to the Netezza database because the Netezza numeric datatype is limited to 38 digits of precision. The value could, however, be returned as a varchar.

Example

```
a = BigNum.new()
a = BigNum.new(12345)
a = BigNum.new("87293849232423432423482398 98.982983",100)
x = BigNum.new(a)
```

BigNum:set(x)

Update a BigNum value to be x. The x argument can be a number, a string, or another BigNum value.

Example

```
bn = BigNum.new()
bn:set("9828939823498")
```

BigNum:sub(x)

Update the value of a BigNum to be the result of subtracting x. This is more efficient than using the - operator since it does not create a new instance of a BigNum.

Example

```
bn = BigNum.new(12341234)
bn:sub(1234)
```

BigNum.todouble(BigNum)

Return the BigNum value as a double. This does not alter the value of the BigNum.

Example

```
x = BigNum.todouble(bn)
```

BigNum:todouble()

Return the BigNum value as a double. Does not modify the value of the BigNum.

Example

```
x = bn:todouble()
```

BigNum.tostring(BigNum)

Return the BigNum value as a string. Does not modify the value of the BigNum.

Example

```
str = BigNum.tostring(x)
```

BigNum:tostring()

Return the BigNum value as a string. Does not modify the value of the BigNum.

Example

```
str = bn:tostring()
```

BigNum.trunc(BigNum)

Create a new instance of a BigNum value and set it to the result of truncating the BigNum value that was passed as an argument.

Example

```
x = BigNum.trunc(bn)
```

BigNum:trunc()

Update the BigNum value by truncating it. This is more efficient than using BigNum.trunc() becauese it does not create a new instance of a BigNum.

Example

```
bn:trunc()
```

Bit Module

The bit module performs bitwise operators on Lua numbers. The input value is transformed into a 32-bit signed integer value and all of the bit functions return a 32-bit signed integer value.

bit.tohex(value [,size])

Convert the 32-bit integer portion of a number to hex format. The optional size parameter limits the result to *size* hex characters. A negative size results in upper case hex characters and a positive size results in lower case hex characters.

bit.bnot(value)

Return the bitwise NOT value of the argument.

bit.bor(value1, ...)

Return the bitwise OR value of all of the arguments.

bit.band (value1, ...)

Return the bitwise AND value of all of the arguments.

bit.bxor(value1, ...)

Bitwise XOR of all of the arguments.

bit.lshift(value, n)

Left shift the value by *n* bits.

bit.rshift(value, n)

Right shift the value by *n* bits.

bit.rol(value, n)

Rotate the bits in the value left by *n* bits.

bit.ror(value,n)

Rotate the bits in the value right by *n* bits.

bit.bswap(value)

Swap the upper 4 bits with the lower 4 bits.

JSON Module

The JSON module allows nzLua to parse JSON formatted strings into a Lua table or to convert a Lua table into a JSON formatted string.

json.decode(string)

Parse a JSON encoded string and return the result as a table. When decoding a JSON encoded string, the JSON null value will be written to the output table as the special value json.NULL since it is not possible to store the **null** value in a Lua table.

Example

```
t = json.decode('{"a":123,"b":456}')
```

json.encode(table [,compatible])

Convert a Lua table into a JSON encoded string. The special value json.NULL can be used to store a null value in a Lua table since it is not possible to store a Lua **null** value in a table.

JSON supports an array with integer indexes or an object that has string indexes whereas Lua allows a table to use integer and string values for keys in the same table. To allow any Lua table to be serialized to a string, nzLua by default uses an encoding format that may result in a serialized table that is not compatible with standard JSON. For example, the table below cannot be encoded in the standard JSON format since it has both integer indexes and a string index.

```
t = {111,222,333,foo="abc"}
```

Using json.encode(t) on this table will yield the string '{1:111,2:222,3:333,"foo":"abc"}' which is not a legal encoding for JSON, whereas json.encode(t,true) will yield the string '{"1":111,"2":222,"3":333,"foo":"abc"}'. In the compatible format, all of the integer keys of the table are converted to string keys.

The json.encode function makes no attempt to detect recursive tables, therefore the code shown below will result in a stack overflow error.

```
t = {1,2,3}
t[4] = t
str = json.encode(t)
```

Example

```
t = {a=123,b=987,c=333,d=json.NULL}
str = json.encode(t)
```

SQLite Module

A SQLite database has been embedded inside of nzLua to provide a way to persist data outside of a Netezza transaction. The primary use case for this feature is for debug logging, but it can also be used to store lookup tables or additional data that cannot be directly returned to the Netezza database from a UDX (for example, when a UDTF needs to return multiple distinct result sets instead of a single result set). The SPUPad feature (see 7.5) will generally give better performance for this purpose, but all data in the SPUPad disappears at the end of each transaction whereas the data in the SQLite database will remain until either the database is restarted or a s-blade fails.

The maximum amount of data that can be stored in a SQLite database is capped at 1GB. A large SQLite database could substantially impact overall performance of the Netezza database due to random I/O requests for index lookups and index maintenance during inserts. The SQLite database will be cached in memory where possible to avoid excessive random I/O. However, as the database grows in size, memory may not be available. This results in many small I/O requests, which then interferes with sequential reads for the Netezza database.

A unique SQLite database exists for each dataslice. Two concurrently running UDXs on the same dataslice will access the same SQLite database. It is not possible for a UDX to share data across dataslices using SQLite. The SQLite database does not support row level locking. Any insert, update, or delete statement locks the entire SQLite database for the duration of a transaction. By default, SQLite operates in autocommit mode where each DML statement is committed immediately.

Detailed documentation on SQLite, including the SQL syntax supported by SQLite, can be found at http://www.sqlite.org. Some examples of using SQLite from within an nzLua UDX can be found in the directory /nz/extensions/nz/nzlua/examples/SQLite.

db:close()

Close a database connection, freeing all memory.

Example

```
db=sqlite.open('testdb')
db:close()
```

db:cols(sql)

Execute a SQL statement and return an iterator function that can be used in a for loop. The values will be returned using the Lua multiple return value feature.

Example

```
sum=0
db=sqlite.open('testdb')
for x in db:cols[[select x from foo]] do
    sum=sum+x
end
```

db:exec(commands)

Execute one or more SQL commands in a SQLite database.

Example

```
db=sqlite.open('testdb')
db:exec[[
    create table foo(x integer);
    insert into foo values (1);
    insert into foo values (2);
]]
```

db:first_cols(sql)

Execute a SQL statement and return the first row of data. The values are returned directly using the standard Lua feature of returning multiple values from a function.

Example

```
db=sqlite.open('testdb')
min_x, max_x = db:first_cols[[select min(x), max(x) from foo]]
```

db:first_irow(sql)

Execute a SQL statement and return the first row of data.

Example

```
db=sqlite.open('testdb')
row=db:first_irow[[select max(x) from foo]]
max_x = row[1]
```

db:first_row(sql)

Execute a SQL statement and return the first row of data. The columns are returned as a table using the column names from the SQL statement instead of as an integer indexed array.

Example

```
db=sqlite.open('testdb')
row=db:first_row[[select max(x) as biggest from foo]]
max_x = row.biggest
```

db:irows(sql)

Execute a SQL statement and return an iteration function that can be used in a for loop.

Example

```
sum=0
db=sqlite.open('testdb')
for row in db:irows[[select x from foo]] do
    sum=sum+row[1]
end
```

db:prepare(sql)

Create a prepared statement that can be repeatedly used with different bind variables. The bind parameters are represented by the ? character in the SQL statement.

Example

```
db=sqlite.open('testdb')
stmt,err=db:prepare('select count(*) from foo where x > ?')
```

db:rows(sql)

Execute a SQL statement and return an iteration function that can be used in a for loop. The values will be returned as a table that uses the column names as the index.

Example

```
sum=0
db=sqlite.open('testdb')
for row in db:rows[[select x from foo]] do
    sum=sum+row.x
end
```

sqlite.begin(dbname)

Start a transaction in the given database. It is more efficient to insert many rows in the same transaction instead of relying on autocommit after each row.

sqlite.commit(dbname)

Commit a transaction in the given database. The finalize method (see finalize()) can be used to commit all rows prior at the end of UDX execution.

Example

```
function finalize()
    sqlite.commit('testdb')
end
```

sqlite.drop_if_exists(dbname,tablename)

Drop a SQLite table if it exists, returns true for success.

Example

```
sqlite.drop_if_exists('testdb', 'foo')
```

sqlite.dropdb(dbname)

Drop a SQLite database and all associated storage.

Example

```
sqlite.dropdb('testdb')
```

sqlite.execute(dbname, sql, parameters)

Execute a SQL statement and return the number of rows modified. The execute function should be used for insert, update, delete, and DDL statements.

Example

```
sqlite.execute('testdb', 'create table foo(x integer, y varchar(255))')
sqlite.execute('testdb', 'insert into foo values(?,?)', 123, 'abc')
sqlite.execute('testdb', 'drop table foo')
```

sqlite.listdb()

Return a list of all SQLite databases and the sizes of each database. The list is returned as a Lua table where the first index is the name of the database and the second index is the size (in bytes) of the database.

Example

```
dblist = sqlite.listdb()
for i=1,#dblist do
    total_size = total_size + dblist[i][2]
end
```

sqlite.open(dbname)

Open a connection to a SQLite database and return a database object.

Example

```
db,err = sqlite.open('mydb')
if not db then error(err,0) end
```

sqlite.rollback(dbname)

Rollback a transaction in the specified database.

Example

```
sqlite.rollback('testdb')
```

sqlite.select(dbname,sql,parameters)

Execute a select statement, returns an iterator function that can be used in conjunction with a for

loop.

Example

```
for row in sqlite.select('testdb', 'select x from foo where x > ?', 0) do
    sum = sum + row[1]
end
```

sqlite.selectOne(dbname,sql,parameters)

Execute a select statement and return the first row as a Lua table.

Example

```
row = sqlite.selectOne('testdb', 'select max(x) from foo')
max_x = row[1]
```

sqlite.table_exists(dbname,tablename)

Check if a table exists.

Example

```
if not sqlite.table_exists('testdb', 'foo') then
    error('Table foo does not exist!')
end
```

sqlite.tables(dbname)

Return a Lua table that contains a list of all the SQLite tables defined in the specified database.

Example

```
t = sqlite.tables('testdb')
```

stmt:bind(valuelist)

Bind values to the bind parameters of a prepared statement.

Example

```
db=sqlite.open('testdb')
stmt,err=db:prepare('select count(*) from foo where x between ? and ?')
stmt:bind(100,200)
```

stmt:close()

Close a prepared statement, freeing any memory associated with the statement.

Example

```
db=sqlite.open('testdb')
stmt,err=db:prepare('select count(*) from foo where x between ? and ?')
stmt:close()
```

stmt:cols()

Execute a prepared statement using the currently assigned bind variables and return an iterator function. The column values will be returned as multiple return values.

Example

```
db=sqlite.open('testdb')
stmt,err=db:prepare[[select x from foo where x between ? and ?]]
stmt:bind(1,100)
for x in stmt:irows() do
    sum=sum + x
end
```

stmt:exec()

Execute a prepared statement using the currently assigned bind variables.

Example

```
db=sqlite.open('testdb')
stmt,err=db:prepare[[create table foo (x integer)]]
stmt:exec()
```

stmt:first_cols()

Execute a prepared statement using the currently assigned bind variables and return the first row of data as multiple return values.

Example

```
db=sqlite.open('testdb')
stmt,err=db:prepare[[select max(x), min(x) from foo where x between ? and
?]]
stmt:bind(1,100)
max,min=stmt:first_cols()
```

stmt:first_irow()

Execute a prepared statement using the currently assigned bind variables and return the first row of data as an integer indexed table.

Example

```
db=sqlite.open('testdb')
stmt,err=db:prepare[[select max(x), min(x) from foo where x between ? and
?]]
stmt:bind(1,100)
row=stmt:first_cols()
max,min=row[1],row[2]
```

stmt:first_row()

Execute a prepared statement using the currently assigned bind variables and return the first row of

data as a table using the column names as the index.

Example

```
db=sqlite.open('testdb')
stmt,err=db:prepare[[
    select max(x) max_x, min(x) min_x
    from foo
    where x between ? and ?
]]
stmt:bind(1,100)
row = stmt:first_row()
max, min = row.max_x, row.min_x
```

stmt:irows()

Execute a prepared statement using the currently assigned bind variables and return an iterator function. The column values will be returned as an integer indexed array.

Example

```
db=sqlite.open('testdb')
stmt,err=db:prepare[[select x from foo where x between ? and ?]]
stmt:bind(1,100)
for row in stmt:irows() do
    sum=sum + row[1]
end
```

stmt:rows()

Execute a prepared statement using the currently assigned bind variables and return an iterator function. The column values will be returned as a table using the column names as an index.

Example

```
db=sqlite.open('testdb')
stmt,err=db:prepare[[select x from foo where x between ? and ?]]
stmt:bind(1,100)
for row in stmt:rows() do
    sum=sum + row.x
end
```

StringBuffer Module

The StringBuffer module allows a string to be built up by appending to the end of the StringBuffer. Just as in Java, C#, and other garbage collected languages, strings are immutable objects in Lua. The StringBuffer module should be used for building large strings by appending to the end of the StringBuffer instead of using string concatenation. Once the string has been built using the StringBuffer, it can be converted into a normal Lua string using the toString method of the StringBuffer object or using the standard Lua tostring function.

A StringBuffer can be used in place of a Lua string to return results back to the Netezza database. Returning a StringBuffer as a function result is more efficient than converting the StringBuffer into a string using tostring() or the :toString() method and then returning the resulting string.

StringBuffer.new()

Create a new StringBuffer.

Example

```
sb = StringBuffer.new()
```

StringBuffer:append(string)

Append a string to the end of a StringBuffer. The StringBuffer will automatically be resized as necessary.

Example

```
sb = StringBuffer.new()
sb:append('string1')
sb:append(';')
sb:append('string2')
```

StringBuffer:clear()

Reset the StringBuffer to zero length. This does not free any memory, instead only truncating whatever is currently in the StringBuffer to zero length. It is much more efficient to reuse a StringBuffer by using StringBuffer:clear() than it is to create a new StringBuffer.

Example

```
sb = StringBuffer.new()
sb:append('foo');
sb:clear()
```

StringBuffer:delete(start, length)

Delete length bytes out of a StringBuffer starting at the given start position. If start is negative it is used as an offset from the end of the StringBuffer instead of the beginning.

Example

```
sb = StringBuffer.new()
sb:append('foobar')
sb:delete(4,6)
```

StringBuffer:insert(position, str)

Insert a string into the middle of a StringBuffer starting at the specified position.

Example

```
sb = StringBuffer.new()
sb:append('foobar')
sb:insert(4,'d')
```

StringBuffer:length()

Return the current length of the string stored in the StringBuffer.

Example

```
length = sb:length()
```

StringBuffer:replace(start,stop,str)

Replace a portion of the StringBuffer between the start and stop positions with another string.

Example

```
sb = StringBuffer.new()
sb:append('testing')
sb:replace(3,4, 'x')
```

StringBuffer:setLength(length)

Decrease the length of the StringBuffer to length bytes. If the length specified is greater than the current StringBuffer length, the new length is ignored.

Example

```
sb = StringBuffer.new()
sb:append('foobar')
sb:setLength(3)
```

StringBuffer:size()

Return the size of the StringBuffer. This is the amount of memory currently consumed by the StringBuffer. The size is automatically increased as necessary when strings are appended to the end of the StringBuffer. Note that StringBuffer:clear() does not decrease the size of the StringBuffer, it only sets the length to 0.

Example

```
bytes = sb:size()
```

StringBuffer:substr(start [,length])

Extract a substring out of a StringBuffer. If length is not specified, extract all characters until the end of the string.

Example

```
sb = StringBuffer.new()
sb:append('foobar')
bar = sb:substr(4,3)
```

StringBuffer:toString()

Convert a StringBuffer into a Lua string. The Lua tostring function can also be used to convert a StringBuffer into a Lua string. It is more efficient to return a StringBuffer as a result back to the Netezza database rather than using toString to convert a StringBuffer to a string and then returning the string.

Example

```
sb = StringBuffer.new()
sb:append('the quick brown fox')
str1 = sb:toString()
str2 = tostring(sb)
```

XML Module

The nzLua XML module is a implemented as a document object model (DOM) XML parser. An XML object keeps track of the current position within the document. The position can be moved to other nodes in the document by using the various methods available in the XML object. Other methods can be used to get information about the current node.

```
str = [[
<config version="1.0">
        <option type="type1">value1</option>
        <option type="type2">value2</option>
</config>
]]

doc = xml.parse(str)
value1 = doc:getText('/config/option[2]')      ==> 'value2'
doc:goPath('/config/option')
value2 = doc:text()                            ==> 'value1'
```

Many of the XML methods accept a document path as an argument. The path in nzLua is a limited subset of the standard XPath syntax. The nzLua path syntax does not support pattern matching, expressions, or filter conditions. The options available for an XML path within nzLua are listed in Table 2.

Table 2: XML Path Options

Expression	Description
/	The root node.
.	The current node.
..	Parent of the current node.
/foo	The node named foo, which is a child of the Root node.
/foo/bar	The node bar, which is a child of the node foo, which is a child of the root node.
foo	The first node named foo, which is a child of the current node.
foo[3]	The third node named foo, which is a child of the current node.
/foo/bar[2]	The second element named bar, which is a child of the root node foo.
/foo/bar[last()]	The last element named bar, which is a child of the root node foo.
./bar	The node bar, which is a child of the current node.

XML:append(name [, text [, attributes [, go]]])

Add a new element to the document as a sibling immediately after the current element.

If text is specified, the element will contain a text node that is set to the value of the text string.

The optional attributes argument is a table that should contain a set of name/value pairs. The table will be used to set the attributes of the new XML element.

The fourth argument is a boolean value. If it has the value true the document position will be set to be the newly created element. Otherwise the document position will not be changed.

Example

```
attrs={}
attrs.hostname='127.0.0.1'
attrs.port='5480'
attrs.database='system'

doc = xml.parse('<options><date>2011-07-01</date></options>')
doc:goPath('/options/date')
doc:append('connection', null, attrs, true)
```

XML:appendChild(name [, text [, attributes [, go]]])

Create a new child element node and attach it as the last child of the current element.

If text is specified, the element will contain a text node that is set to the value of the text string. The text value can be null.

The optional attributes argument can be a table that contains a set of name/value pairs. The table will be used to set the attributes of the new XML element. The attributes value can be null.

The fourth argument is a boolean value. If it is set to true the document position will be set to be the newly created element. Otherwise the document position will not be changed.

Example

```
doc = xml.new('options')
doc:appendChild('host', '127.0.0.1')
doc:appendChild('port', '5480')
doc:appendChild('database', 'system')
str = tostring(doc)
```

XML:clear([path])

Delete all attributes and child elements from an element. If no path is specified, clear will affect the current element. If a valid path is specified, clear will affect the specified element and will also set the current position to be the element specified by the path. Returns true on success or false on failure.

Example

```
doc = xml.parse([[<foo><bar>bartext</bar><baz>baztext</baz></foo>]])
doc:goPath('/foo')
doc:clear()
```

XML:delete([path])

Delete the current element and all child elements from the document. If the path is specified, the element identified by the path will be deleted. Returns true if an element was deleted or false otherwise. The current position is set to be the parent of the deleted node. The root element of the document cannot be deleted using the delete method.

Example

```
doc:goPath('/foo/bar[3]')
doc:delete()
if not doc:delete('/foo/bar') then
    error('Failed to delete element /foo/bar!')
end
```

XML:deleteAttribute(name)

XML:deleteAttribute(path,name)

Delete an attribute from an element. If no path is given, the attribute will be deleted from the

current element node. If a path is given, the attribute will be deleted from the node identified by the path. Returns false if the path is supplied and the path does not identify an element in the document, otherwise returns true even if the element does not have an attribute with the given name.

Example

```
doc:deleteAttribute('date')
doc:deleteAttribute('/options/database', 'name')
```

XML:free()

Release all memory allocated to the parsed XML document.

Example

```
doc = XML.parse(str)
doc:free()
```

XML:getAttribute(attribute)

XML:getAttribute(path,attribute)

Return the value of the attribute. Returns null if the attribute does not exist.

The single argument form returns the value of an attribute of the current element.

The two argument form returns the value of an attribute of the element identified by the path. The first argument is the path and the second argument is the attribute.

Example

```
doc = XML.parse([[<cfg><opt type="test"/></cfg>]])
str = doc:getAttribute('/cfg/opt', 'type')
```

XML:getAttributes([path])

Return the value of all attributes of an element as a table. If no path is specified getAttributes returns the attributes of the current element.

Example

```
doc = XML.parse([[<config><options type="test"/></config>]])
attributes = doc:getAttributes('/config/options')
if attributes.type != 'test' then
    error("Invalid result!",0)
end
```

XML:getCount([path])

Count the number of elements that are children of an element. If no arguments are given, the count reflects the number of children of the current element. The one argument form returns the number of elements which are a child of the element at the specified path.

Example

```
count=doc:getCount()
```

```
count=doc:getCount('/foo/bar')
```

XML:getName()

Return the name of the current element.

Example

```
name = doc:getName()
```

XML:getNameCount([path,] name)

Count the number of child elements that have a given name. When only one argument is specified, this method will return a count of the number of child elements of the current node that have the given name. When the two argument form is used, the first argument must be a path and the second argument is the name of the elements to count.

Example

```
doc=xml.parse(str)
count=doc:getNameCount('bar')
count=doc:getNameCount('/foo/bar', 'baz')
```

XML:getPosition()

Return the a string that is the path to the current element.

Example

```
path = doc:getPosition()
```

XML:getText([path])

Return the value of the first text node that is attached to an element or null if there is no text node attached to the element.

The zero argument form returns the the value of the text node attached to the current element.

The one argument form returns the value of the text node attached to the element identified by the path string.

Example

```
doc = XML.parse([[<cfg><opt>foo</opt></cfg>]])
foo = doc:getText('/cfg/opt')

doc:goPath('/cfg')
foo = doc:getText()
```

XML:goPath(path)

Move to the element specified by the XML path string. If the path is valid, return true; otherwise return false.

Example

```
doc = XML.parse(str)
if not doc:goPath('/foo/bar') then
    error("Invalid XML element /foo/bar",0)
end
```

XML:goChild()

Move to the first child element of the current element. Returns true on success, or false if there is no child element. If there is not a child element, the current position is not changed.

Example

```
doc = XML.parse(str)
doc:goPath('/foo/bar')
if not doc:goChild() then
    error("Invalid document!",0)
end
```

XML:goNext([name])

Move to the next sibling element. Returns true on success or false otherwise. Does not move if there is no element after the current position. If the optional name is specified, move to the next sibling element that has the given name, skipping over any elements that do not match the given name.

Example

```
doc = XML.parse(str)
doc:goPath('/foo/bar')
if not doc:goNext() then
    return null
end
```

XML:goParent()

Move the current position to be the parent of the current node.

Example

```
doc:parse(str)
doc:goPath('/foo/bar/baz')
doc:goParent()
if doc:getPath() != '/foo/bar' then
    error('Invalid!',0)
end
```

XML:goPrev([name])

Move to the previous sibling element. Returns true on success or false otherwise. Does not move if there is no previous sibling element to move to. If the optional name is specified, move to the previous sibling element that has the given name, skipping over any elements that do not match the given name.

Example

```
doc = XML.parse(str)
doc:goPath('/foo/bar')
if doc:goPrev() then
    return doc:value()
end
```

XML:goRoot()

Move to the root element of the document.

Example

```
doc:goRoot()
```

XML:insert(name [, text [, attributes [, move]]])

Add a new element to the document as a sibling immediately before the current element.

If text is specified, the element will contain a text node that is set to the value of the text string. The text value can be null.

The optional attributes argument is a table that contains a set of name/value pairs. The table will be used to set the attributes of the new XML element. The attributes value can be null.

The fourth argument is a boolean value. If it is set to true the document position will be set to be the newly created element. Otherwise the document position will not be changed.

Example

```
doc = xml.parse('<options><host>127.0.0.1</host></options>')
doc:goPath('/options/host')
doc:insert('port','5480')
```

XML.parse(string)

Parse a string and return an XML object. The current location is set to the root element of the document. The parse function will throw an error if the string is not valid XML (errors can be caught using the pcall function).

Example

```
doc = XML.parse([[
<config version="1.0">
        <option type="type1">value1</option>
        <option type="type2">value2</option>
</config>]])
```

XML:setAttribute(attribute,value)

XML:setAttribute(path,attribute,value)

Set the value of a single attribute.

When the two argument form is used, the first argument is the attribute name and the second string is the attribute value. The two argument form modifies the current element.

When the three argument form is used, the first argument is the path to an element, the second argument is the attribute name, and the third argument is the attribute value.

Example

```
doc = xml.parse('<config><connection host='127.0.0.1'/></config>')
doc:setAttribute('/config/connection', 'host', '208.94.146.70')

doc:goPath('/config/connection')
doc:setAttribute('host','127.0.0.1')
```

XML:setAttributes(table)

XML:setAttributes(path, table)

Set the value of all attributes of an element. Any previous attributes are removed prior to setting the new attributes.

The single argument form modifies the attributes of the current element. The first argument must be a table that contains a set of name/value pairs. The table will be used to set the values of all the attributes of the current element.

When the two argument form is used, the first argument is a path to an element and the second argument is a table that contains the name/value pairs.

Example

```
config={host='208.94.146.70',port=5480,database='system'}
doc = xml.parse([[<config><connection host='127.0.0.1'/></config>]])
doc:setAttributes('/config/connection', config)

doc:goPath('/config/connection')
doc:setAttributes(config)
```

XML:setCDATA(cdata)

XML:setCDATA(path, cdata)

Set the value of the CDATA text node to the string argument. If the single argument form is used, setCDATA will set the value of the CDATA text node that is attached to the current node position. When the two argument form is used, the first argument is used as a path and the second argument is used as the value of the CDATA text node.

Example

```
doc = xml.parse('<test><typing>the quick brown fox</typing></test>')
doc:setCDATA('/test/typing', 'The quick brown fox jumps over the lazy
dog.')
```

XML:setText(text)

XML:setText(path,text)

Set the value of the text node that is attached to the current element to the string argument. If the single argument form is used, setText will set the value of the text node that is attached to the current node position. When two string arguments are used, the first string is used as a path and the second argument is used as the value of the text node.

Example

```
doc = xml.parse('<test><typing>the quick brown fox</typing></test>')
doc:setText('/test/typing', 'The quick brown fox jumps over the lazy
dog.')
```

XML:setName(name)

XML:setName(path,name)

Change the name of an element to the specified name. If only one argument is specified, the name of the current element will be changed. If two arguments are specified, the first argument is used as a path to an element and the second argument is used as the element's name.

Example

```
doc = xml.parse([[<options><host>127.0.0.1</host></options>]])
doc:setName('/options', 'configuration')
```

XML:useCDATA(path [,false])

Set the first text node that is a child of the element at the given path to be a CDATA text node instead of a regular text node. If the second argument is false, the text node will be a regular text node instead of a CDATA text node.

Example

```
doc = xml.new('test')
doc:setText([[the quick brown fox jumps over the lazy dog]])
doc:useCDATA('/test',true)
```

CHAPTER 7
nzLua API

The nzLua API defines how nzLua interacts with the Netezza database. nzLua can be used to create functions, aggregates, and table functions. Each type of UDX has a specific set of API calls associated with it. In addition, all types of nzLua UDX have a common set of API calls. The documentation for the API calls is therefore broken down into the following sections.

- API calls common to all nzLua programs
- API calls used by a UDF (user-defined function)
- API calls used by a UDA (user-defined aggregate)
- API calls used by a UDTF (user-defined table function)
- SPUPad API (save results in memory, retrieve with another UDX)
- Global variables
- Constants

The abbreviation UDX is generally used as a shorthand method to represent the term "user defined X" where X can be one of [function | aggregate | table function].

UDX API Methods

The UDX API methods can be used with all types of nzLua programs. Every nzLua UDX must have a getName(), getType(), and getArgs() function. The getOptions(), initialize(), and finalize() methods are optional.

getName()

Every nzLua UDX must have a getName() method defined. The getName() method determines the name of the function that will be created in the database when the nzLua code is compiled and installed. The getName() method must return a single string. The string returned by getName() is not case sensitive and can contain only normal characters that can be used in an unquoted identifier in the Netezza database. This limits a Netezza UDX name to the standard ASCII alphabetic characters

and UTF8 alphabetic characters and the underscore symbol.

When the UDX is compiled, the name will be converted to the default system catalog case that is normally upper case.

Example

```
function getName()
    return "testfunction"
end
```

getType()

The getType() method is mandatory and is used to indicate the type of program that is represented by the nzLua code. The getType() method must return a string that is one of "udf", "uda", or "udtf". During the nzLua compile phase, the compiler calls the getType() method so that it knows how to compile and install the program. See the other sections in this chapter for the API calls specific to each type of UDX, as well as the Appendix that contains several example nzLua programs of each type.

Examples

```
function getType()
    return "udf"
end

function getType()
    return "udtf"
end
```

getArgs()

The getArgs() method returns a table that contains one row for each argument that will be accepted by the UDX. In the Netezza database, arguments are always positional, therefore the first argument in the list represents the first argument to the UDX, the second row in the table represents the second argument, etc.

The data types that are accepted by an nzLua UDX are listed in Table 3.

Table 3: Accepted nzLua UDX data types

Datatype	Description
date	A date value, encoded as Unix time (seconds since January 1, 1970)
bigint	A 64-bit integer value. Depending on the OPT_FORCE_BIGNUM option, a bigint may be stored as a double or as a bignum value. See getOptions().
boolean	True or false encoded as a Lua boolean value.
byteint	A 1-byte integer (-128 through 127), encoded as a double in nzLua.fixed-width

Datatype	Description
date	A date value, encoded as Unix time (seconds since January 1, 1970)
char(size)	A fixed-width character string, encoded as a Lua string.
double	IEEE double precision floating point number.
float	IEEE single precision floating point number, encoded as a double.
integer	A 32-bit signed integer encoded as a double.
nchar(size)	A fixed-width UTF8 string value encoded as a Lua string.
nvarchar(size)	A variable-width UTF8 string value encoded as a Lua string.
numeric(any)	A numeric value that can have any scale and precision. Depending on the value and the OPT_FORCE_BIGNUM setting, a numeric(any) may be passed into nzLua as a double or a BigNum value.
numeric(scale,precision)	A numeric value with a predefined scale and precision. Depending on the value and the OPT_FORCE_BIGNUM setting, a numeric(any) may be passed into nzLua as a double or a BigNum value. Large numeric values (beyond 14 digits) will always be passed into nzLua as a BigNum and not as a normal Lua number. See BigNum Module for details on using a BigNum value.
smallint	A 16-bit integer value encoded as a double.
timestamp	A timestamp value with millisecond precision encoded as Unix time (seconds since January 1, 1970).
varchar(any)	A string of any length encoded as a Lua string value.
varchar(size)	A variable-width string of predefined length encoded as Lua string.
varargs	If the varargs datatype is used, it can be the only argument accepted by the nzLua UDX. A UDX defined to accept varargs can be called with any combination of datatypes. The UDX is responsible for checking the datatypes and behaving appropriately.

The getArgs() method returns a nested table in the form below. The argument name is not currently used by nzLua or the Netezza database, but has been provided for documentation purposes as well as providing compatibility should the Netezza database eventually add support for named arguments.

```
{{ARGUMENT_1_NAME, ARGUMENT_1_TYPE},
 {ARGUMENT_2_NAME, ARGUMENT_2_TYPE},
 ...,
 {ARGUMENT_X_NAME, ARGUMENT_X_TYPE}}
```

Example

```
function getArgs()
    local args={}
    args[1] = { "x", integer }
    args[2] = { "y", integer }
    return args
end

function getArgs()
    return {{"str", varchar(255)}}
end

function getArgs()
    return varargs
end
```

getComment()

The getComment() method is an optional method that can be used to set the description field in the _v_function or _v_aggregate views. During compilation of the nzLua UDX, if the getComment() method returns a string, the result of the getComment() function will be used to execute a COMMENT command in SQL to modify the UDX description.

Example

```
function getComment()
    return "This is a comment"
end
```

getOptions()

The getOptions() method is used to alter the default behavior of a UDX, for example if the UDX is called for NULL values. Some options are only applicable to specific types of UDX. The list of options and their descriptions is available in the Constants section of the UDX API documentation (see UDX Options).

Examples

```
function getOptions()
    local options={}
    options[OPT_NULL_INPUT] = true
    options[OPT_DETERMINISTIC] = false
    return options
end
```

initialize()

The initialize method is called before the first row of data is passed to a UDX. The most common operations performed by initialize() are to validate data types for a VARARGS UDX, initialize variables,

or to restore data from the SPUPad (see SPUPad API).

For a UDA, the initialize method will be called prior to the first row of data being passed to the accumulate method but will not be called prior to the first row of data being passed to the merge or finalResult methods.

Example

```
function initialize()
    counter = 0

    -- restore data from the SPUPad
    t = restoreTable("mytable")
end

function initialize()
    for i,type in pairs(ARGTYPE) do
        if type != TYPE_STRING and type != TYPE_UTF8STRING then
            error("This function only accepts STRING arguments!",0)
        end
    end
end
```

finalize()

The finalize method is called after all rows have been processed by the UDX. The finalize method can be used to store data in the SPUPad (see SPUPad API) or utilize some other feature that operates outside of the normal data flow of the SQL statement. The finalize method does not accept any arguments or return any values.

Any errors generated during the finalize method call are trapped and ignored, therefore the finalize method should only be used to execute statements that are certain to not fail or in cases where failure of the functions being called is not fatal.

Example

```
function finalize()
    saveTable("mytable", myresults)
end
```

skipArgs()

The skipArgs() method is used to avoid passing the first *N* arguments in for every row of data. Generally this is used when the first argument is a large constant string, such as configuration data or nzLua source code. The first arguments will be passed into the initialize method (and the calculateShape() method of a UDTF), but will not be passed to the accumulate() (UDA), evaluate() (UDF), or processRow() (UDTF) methods. By default all arguments will always be passed to these functions unless the skipArgs method is defined by the UDX.

Example

```
-- avoid passing in the first two argument for each row
function skipArgs()
    return 2
```

```
end
```

UDF API Methods

A user defined function is the most commonly used type of UDX. A UDF is generally used in the SELECT and WHERE clauses of a SQL statement. The Appendix contains several UDF examples. In the SQL statement shown below, the UDF named "testudf" is shown in the SELECT and WHERE clauses of the statement.

```
select x, y, testudf(x,y)
from foo
where testudf(x,y) > 100
```

The methods that must be implemented to create a UDF are getName(), getType(), getArgs(), evaluate(), and getResult(). The optional methods that can be used in a UDF are calculateSize(), getOptions(), initialize(), and finalize().

calculateSize()

When the return type of a user-defined function is defined as VARCHAR(ANY), NUMERIC(ANY), or NVARCHAR(ANY) the calculateSize method will be called at runtime to dynamically determine the size of the result column.

nzLua passes a table as the single argument to the calculateSize method. The table contains one record for each argument as is shown in the table below.

Table 4: calculateSize Argument Table

Name	Description
args.count	The number of arguments the UDF was invoked with.
args[i].type	The argument type. For more information, see the “Argument Types“ section.
args[i].length	The length of a char, varchar, nchar, or nvarchar argument.
args[i].precision	The precision of a numeric argument.
args[i].scale	The scale of a numeric argument.

Example

```
function calculateSize( args )
    return args[1].length + args[2].length
end

function getResult()
    return varchar(any)
end
```

evaluate()

The evaluate() method is called once for each row of data in the SQL statement and must return a single value back to the Netezza database. The arguments passed into the evaluate method will match the format specified by the getArgs() method. An evaluate() method that returns the result of adding two numbers is shown here.

```
function evaluate(x, y)
    return x + y
end
```

getResult()

The getResult() method is called only during the compile step of the UDF and it is used to indicate the data type that will be returned by the evaluate() method. The getResult() method must return one of the standard data types that are documented in Table 3: Accepted nzLua UDX data types.

When the getResult method returns varchar(any) or numeric(any), the calculateSize method will be invoked at runtime to dynamically determine the size of the result.

Here are a few examples of the getResult() method for a UDF.

```
function getResult()
    return integer
end

function getResult()
    return varchar(255)
end

function getResult()
    return numeric(18,4)
end
```

UDA API Methods

User defined aggregates are used in conjunction with a GROUP BY clause or the OVER clause. Implementing a UDA requires deeper understanding of how the Netezza database works internally when performing aggregations. For a standard UDA used with a GROUP BY statement, the Netezza database processes the data by sequentially following the steps outlined here:

4. For each row of data being processed, check to see if a row for the group the row is in has already been found. If it has, pass the row to the accumulate() method along with the current value of the state variables for that group. If the row belongs to a new group, first call the initState() method and then call the accumulate method with the newly initialized state values plus the row of data. The accumulate method should update the current state variables based on the row of data and return the updated state. No data is moved across the network until Step 2.
5. Once all rows of data have been processed on each dataslice, redistribute the final values of the state variables for each group of data so that all of the state variables for each group of

data are moved to the same dataslices (for example, if the GROUP BY was on customer_id, all of the states from each dataslice for customer_id=1 would be moved to the same dataslice).

6. Merge all of the state variables together by calling the merge() method. On each call to merge, two sets of state variables will be passed in. The merge() method merges the two states together and returns the result. The merge() method is called until there is only one state for each group based on the group by statement.
7. For each of the remaining states, call the finalResult() method, passing in the final values for each state. The finalResult() method then calculates a result based on the state variables and returns the result to the Netezza database for each group.

getState()

The getState method is used to define the types of the state variables used by the UDA. The state variables will be passed in to each call of initState, accumulate, merge, and finalResult. The format of the table returned by the getState method is identical to the getArgs and getShape methods.

Example

```
function getState()
    statevars={}
    statevars[1] = { "count", integer }
    statevars[2] = { "sum",   double  }
    return statevars
end
```

initState()

The initState method is used to initialize the UDA's state variables to a known state. Generally this would be values such as **null** or 0. A table containing the state variables is passed into the initState method and after modifying the values the initState method, should return the updated values. The initState method is not mandatory, when not present the state variables will automatically be initialized to **null**.

Example

```
function initState(state)
    state[1] = 0
    state[2] = 0
    return state
end
```

accumulate()

The accumulate method is called for each row of data in the query. The accumulate method updates the state variables with the new row of data and returns the updated state. If the accumulate method returns null, the current values stored in the UDA state variables will not be updated.

Example

```
function accumulate(state,value)
    state[1] = state[1] + value
```

```
    state[2] = state[2] + 1
    return state
end
```

merge()

After the accumulate phase completes, the merge method is then used to merge all of the states generated by the accumulate method together into a final state for each group of data. If the merge method returns null, the current values stored in the UDA state variables will not be updated.

An analytic UDA (example: select myuda(product_id) over (partition by customer) from sales]) will not call the merge method. Merge is only called for a UDA when the UDA is used in the context of a GROUP BY statement.

Example

```
function merge(state1,state2)
    state1[1] = state1[1] + state2[1]
    state1[2] = state1[2] + state2[2]
    return state1
end
```

finalResult()

After the merge phase completes there will be a single set of state variables for each group of data. That set of state variables will be passed into the finalResult method, which transforms the state variables into the final result.

Example

```
function finalResult(state)
    return state[1] / state[2]
end
```

UDTF API Methods

A user defined table function is used in the FROM clause of a SQL statement. This enables a UDTF to return multiple columns and/or multiple rows for each input row. The Appendix contains several example UDTFs. The SQL below invokes a table function named "tftest", passing the values foo.x and foo.y as the first and second arguments. The tftest table function in this example returns three columns-- "a", "b", and "c".

```
select foo.x, foo.y, tf.a, tf.b, tf.c
from foo, table(testtf(foo.x, foo.y)) tf
where foo.x > 1000;
```

processRow()

The processRow() method is called once for each row of data and can return 0 or more rows of data back to the Netezza database. To return 0 rows the value **null** should be returned.

```
function processRow(x)
    return null
```

```
end
```

To return a single row, the processRow method should return a table that contains one value for each column being returned-- t[1] = first column, t[2] = second column, etc. This example returns one row that has three columns.

```
function processRow(x)
    return { x, x*x, x*x*x }
end
```

To return multiple rows, the processRow() method should return a nested table of the form:

```
{{row1_column1, row1_column2, ..., row1_columnX},
 {row2_column1, row2_column2, ..., row2_columnX},
 ...,
 {rowY_column1, rowY_column2, ..., rowY_columnX}}
```

For example, this processRow method returns 3 rows that have 2 columns.

```
function processRow(x)
    local rows={}
    rows[1] = { 1, random(x) }
    rows[2] = { 2, random(x) }
    rows[3] = { 3, random(x) }
    return rows
end
```

outputRow()

After each call to the processRow() method, if the outputRow() method is defined, it is called until it returns null. This allows a UDTF to output any number of rows efficiently. The outputRow() method is always called with one parameter, which is the number of times outputRow() has been called since the processRow() method was called.

Here is an example of using outputRow() in combination with processRow(). The processRow() method is called first and stores the value of x in the rows_to_output variable. The Netezza database then calls the outputRow() method until it returns null. On the first call, the rownum argument will have the value 1, the second call it will have the value 2, etc.

```
function processRow(x)
    rows_to_output = x
    return null
end

function outputRow(rownum)
    if rownum > rows_to_output then
        return null
    end
    return { rownum }
end
```

Once the outputRow() method has returned null, the Netezza database will then call the processRow() method again with the next row of data to be processed. The outputRow() method must return data in exactly the same format as is allowed for the processRow() method.

Even if the processRow() method returns a result, the outputRow() method will still be called after each call to processRow(). For example:

```
function processRow(x)
    rows_to_output = x
    return { 0 }
end

function outputRow(rownum)
    if rownum > rows_to_output then
        return null
    end
    return { rownum }
end
```

getShape()

The getShape() method is called by the nzl program during the nzLua UDX compile step to determine the columns and data types that will be returned by the table function. Unlike a UDF or UDA, a UDTF can return multiple columns. Here is a simple example:

```
function getShape()
    local shape = {}
    shape[1] = { "x", integer }
    shape[2] = { "y", integer }
    shape[3] = { "z", varchar(255) }
    return shape
end
```

In this example, the UDTF is defined to return three columns of data. Based on the getShape() method show above, the processRow() method should return three columns of data that match the format defined by the getShape() method. For example, the processRow() method definition could could look like this:

```
function processRow(a,b,c)
    return { a*10, b-10, "foobar" }
end
```

A special case for the getShape() method is to allow the UDTF to determine the output shape at runtime. When the getShape method returns the value VARSHAPE the calculateShape() method (see calculateShape()) will be called at runtime to determine the output columns based on the constant arguments that are used to invoke the UDTF.

```
function getShape()
    return VARSHAPE
end
```

calculateShape()

The calculateShape() method allows a UDTF to dynamically determine its output shape (the column names and data types returned by the UDTF). This makes it possible to build very versatile functions that can alter their behavior based on the input values. The standard nzlua table function is a good

example of what can be done using the calculateShape() method since it allows a user to submit nzLua source code that is then used to process the data passed into the nzlua table function from the query. The code for the nzlua table function is included in the examples directory with the nzLua distribution.

A single argument is passed to the calculateShape() method that contains a Lua table. The contents of the Lua table are shown here.

```
args.count          The number of arguments
args[i].name        Name of the argument (not currently implemented)
args[i].type        nzLua datatype for the argument
args[i].length      The length of a char, varchar, nchar, or nvarchar
args[i].precision   The precision for a numeric
args[i].scale       The scale for a numeric
args[i].isconst     Boolean indicating if the argument is a constant
args[i].value       Value of the argument if it is a constant
```

The value of every constant argument is passed into the calculateShape() method. For non-constant arguments, only the data type and size will be present. The calculateShape() method can then use the values of the constant arguments to determine the output columns and data types that the UDTF will return.

Example

```
function calculateShape(args)
    if args[1].value < 1 or args[1].value > 1024 then
        error("Invalid number of output columns!", 0)
    end
    local shape={}
    for i=1,args[1].value do
        shape[i] = { "c" || i, varchar(100) }
    end
    return shape
end
```

outputFinalRow()

The outputFinalRow() method is called only when the table function is invoked using the "table with final" syntax.

```
select * from table with final(tablefunction(arg1,...))
```

The purpose of the outputFinalRow() method is for a UDTF to output rows after all of the input to the table function has been processed. This processRow() method can be used to store all of the data in a Lua table. The outputFinalRow() method can then be used to perform a calculation on the data and output the final result for the table function.

The outputFinalRow() method behaves very similarly to the outputRow() method. Just as with outputRow(), outputFinalRow() is passed a single argument on each call that is the number of times the outputFinalRow() function has been called. This makes it easier for the developer to know what data should be returned on each call.

```
function outputFinalRow(rownum)
    if rownum > 1 then return null end
    return { result1, result2, result3 }
```

```
end
```

SPUPad API

The SPUPad provides a way for a UDX to store data that remains in memory throughout the life of a transaction. All memory allocated to the SPUPad is automatically released at the end of a transaction. The finalize() method (see finalize()) is often used to call saveString() or saveTable() to write the final results of a calculation to the SPUPad, since it is called after the final row of data has been processed by the UDX.

To use the SPUPad to pass data between multiple SQL statements, it is necessary to surrounded all of the SQL statements in a single begin/commit block due to the SPUPad being completely cleared at the end of each transaction. The primary use cases for storing data in the SPUPad are listed below:

- Multiple output streams during ELT processing
- An nzLua UDTF can be used to parse a set of input records, outputting the good records and storing the bad records in the SPUPad. A second SQL statement would then pull the bad records back out of the SPUPad and store them into an error table.
- Lookup tables
- In some cases it may be useful to store a lookup table in the SPUPad rather than joining the lookup table using SQL. This can be very useful in cases where, rather than finding an exact match, the UDX needs to find the "best" match based on the lookup data or a set of patterns. This type of "join" can be extremely difficult and inefficient to expression with SQL.
- Configuration data
- The SPUPad can be used to hold configuration data or source code, which will affect how a UDX operates. For example, nzLua source code could be stored in a table, loaded into the SPUPad via one SQL statement, and then used to process a set of data using a second SQL statement.

See SPUPad Example for an example of using the SPUPad. The nzLua examples provided as part of the nzLua distribution also have the save_string.nzl and restore_string.nzl scripts, which demonstrate usage of the SPUPad.

deleteString(name)

Delete a value that has been stored in the SPUPad. Although all data in the SPUPad is automatically cleared at the end of the transaction, freeing memory earlier using deleteString() can help to reduce memory usage by the UDX.

Example

```
deleteString("myarray")
```

deleteTable(name)

The deleteTable() function is exactly the same as the deleteString function, but is provided as a convenience function to be used with saveTable. The functions deleteString() and deleteTable() can be used interchangeably to delete strings or tables out of the SPUPad.

Example

```
deleteTable("mytable")
```

saveString(name, value)

Save a string value to the SPUPad as *name*. The saveString() function can be used to save the results of serializing an nzLua object to a string, such as an array object or json object. The encode/decode functions can also be used to transform data into a binary string that can be stored in the SPUPad.

The number of different strings stored in the SPUPad should be kept to a relatively low number. Rather than storing many strings, the saveTable() function or json.encode() should be used to save an entire Lua table under a single name.

Example

```
saveString("words", "the quick brown fox jumps over the lazy dog.")
saveString("array", array.serialize(myarray))
saveString("json", json.encode(mytable))
```

saveTable(name, table)

Save a table to the SPUPad as *name*. The saveTable() function requires that the table being saved must contain only string, number, or table data. The saveTable() and restoreTable() functions encode the table data using JSON format but are more memory efficient than using json.encode() + saveString() since the result of json.encode() does not have to be returned to nzLua.

When saving or restoring large tables it may be necessary to increase the memory allowed for the nzLua UDX by using the setMaxMemory() function (see Netezza Database Functions).

Example

```
t = {a","b","c", 1, 2, 3, {"x", "y", "z"}}
saveTable("mytable", t)
```

restoreString(name)

Restore a string that has been saved in the SPUPad. Returns null if the SPUPad does not contain a string by the given name. Since the SPUPad is cleared at the end of each transaction, the SPUPad can only be used when all SQL statements are inside of the same begin/commit block.

Example

```
str = restoreString("words")
myarray = array.deserialize(restoreString("myarray"))
mytable = json.decode(restoreString("json"))
```

restoreTable(name)

Restores a table that has been saved to the SPUPad using the saveTable function. The saveTable / restoreTable functions encode the table information using native Lua format and thus offer the best performance option for nzLua when using the SPUPad.

Example

```
t = restoreTable("mytable")
```

Global Variables

ARGCOUNT

ARGCOUNT is an integer value that indicates the number of arguments that were used to call the UDX. The ARGCOUNT variable is generally used with a VARARGS UDX since the number of arguments passed to the UDX is not known until runtime.

ARGTYPE

The ARGTYPE array contains the list of argument types for the UDX. For a standard UDX, the argument types will match the values returned by the getArgs() method and for a VARARGS UDX the argument types will be based on how the UDX is invoked. The values contained in the ARGTYPE array do not match Netezza datatypes, instead they match the nzLua datatypes shown in the Constants section.

Example

```
if ARGTYPE[1] != TYPE_NUMBER then
    error("Expected NUMBER for argument #1.",0)
end
```

ARGSIZE1

The ARGSIZE1 global variable stores the first size for arguments that have a size. For example, if the second argument's data type was VARCHAR(255), then ARGSIZE1[2] would equal 255.

Arguments that do not have sizes, such as an integer value, will have an ARGSIZE1 value of negative one.

ARGSIZE2

The ARGSIZE2 global variable stores the second size for arguments that have a size. For example, if the fourth argument's data type was NUMERIC(18,4), then ARGSIZE1[4] would be equal to 18 and ARGSIZE2[4] would be equal to 4.

Arguments that do not have a second size value such as a varchar, date, or an integer will have an ARGSIZE2 value of negative one.

ARGCONST

The ARGCONST global variable indicates which arguments are constants instead of being variables that come from a calculation or a table. For example, in this SQL statement:

```
select substr(str, 1, 4) from strings
```

The first argument is from a table, therefore ARGCONST[1] would be false. The second and third arguments for the substr function are constants and both ARGCONST[2] and ARGCONST[3] would be true.

Constants

The Lua language does not support true constant values, therefore it is possible for a developer to modify the values of any of these "constants".

Argument Types

When arguments are passed into an nzLua UDX they are translated into nzLua datatypes. The table below contains the list of the standard nzLua datatypes that will be in the ARGTYPE array (see ARGCOUNT).

Table 5: Datatype Translation

nzLua Datatype	Netezza Datatype
TYPE_BIGNUM	BIGINT, NUMERIC larger than 15 digits. The getOptions() method can be used to change the behavior for when values are passed into nzLua as bignum vs. numbers. See OPT_FORCE_BIGNUM in the “UDX Options“ section.
TYPE_BOOL	Boolean
TYPE_DATE	Date, Timestamp (nzLua translates both DATE and TIMESTAMP into unixtime)
TYPE_INTERVAL	Interval
TYPE_NUMBER	BYTEINT, SMALLINT, INTEGER, some BIGINT, some NUMERIC. For BIGINT/NUMERIC values, large numbers that cannot fit into a DOUBLE value will be passed into nzLua as TYPE_BIGNUM
TYPE_STRING	CHAR and VARCHAR
TYPE_TIMESTAMP	Date, Timestamp (nzLua translates both DATE and TIMESTAMP into Unixtime)
TYPE_UTF8STRING	NCHAR and NVARCHAR

UDX Options

These constants are used by the getOptions method (see getOptions()) to modify the behavior of the nzLua UDX.

Table 6: getOptions Method Constants

Option	Description
OPT_AGGTYPE	Only valid for a UDA. Default = any. This option determines if a UDA can be used in combination with the analytic OVER clause or the GROUP BY statement. Values indicate: ► any --the UDA can be used with OVER or GROUP BY ► analytic--UDA can only be used with the OVER clause ► grouped--UDA can only be used with the GROUP BY statement
OPT_DETERMINISTIC	A deterministic UDX always returns the same value for a given input. This allows the Netezza database to avoid calling the UDX multiple times if the arguments are the same. This option can be set to true or false.

OPT_FORCE_BIGNUM	This option determines how integer and numeric values are passed to an nzLua UDX. Normally a value that can fit into a double is always passed as a double. Valid values for OPT_FORCE_BIGNUM are: ► 0 - the default... only use bignum when necessary ► 1 - use bignum for all bigint, numeric64, and numeric128 values ► 2 - use bignum for all bigint, numeric, and double values ► 3 - use bignum for all integer, numeric, and floating point values
OPT_MEMORY	This setting does not directly affect memory for the UDX, instead it is used to tell the Netezza database how much memory the UDX uses (in megabytes). The OPT_MEMORY setting allows values between 1 and 128. The Netezza snippet scheduler uses this setting to make sure the system will not run out of memory while executing a query.
OPT_NULL_INPUT	When set to false, a UDF will return NULL if any of the arguments are NULL (the evaluate() method will not be called). The default value for OPT_NULL_INPUT is true so that the evaluate() method will be called when any of the UDF arguments are NULL.
OPT_PARALLEL	For a table function, setting OPT_PARALLEL=false forces the table function to run on the host as a single process instead of running in parallel on the SPUs.
OPT_REPLICATE_BY_SQL	When true, a UDX is registered with the DDL clause REPLICATE BY SQL, which triggers replication by SQL instead of the default replicate by value (REPLICATE BY VALUE DDL clause). This option is used only for replication. By default, this option is false.
OPT_REQUIRE_FINAL	For a table function, force the "WITH FINAL" syntax to be included to use the table function. See UDTF API Methods for more information.
OPT_UDA_MERGE_NULL	The default behavior (false) of a nzLua UDA is to not call the merge() method when all of the state variables are null for one of the states. If this value is set to true the merge() method will always be called even when all of the state variables are null.

OPT_VARARGS_AS_TABLE	Normally a VARARGS UDX receives the arguments using the Lua varargs syntax (...). Setting this option to true causes the VARARGS arguments to be passed to nzLua using a table instead of as a normal argument list.

APPENDIX A
Examples

The examples provided in this document are a subset of the examples that exist under the /nz/extensions/nz/nzlua/examples directory of the nzLua installation. The examples provided in this guide and in the /nz/extensions/nz/nzlua/examples directory are the fastest way to learn how to create user defined functions, aggregates, and table functions.

UDF Examples

UDF Example #1

This is a simple example UDF that adds two numbers together and returns the result.

```
function evaluate(a,b)
    return a + b
end

function getName()
    return "adder"
end

function getType()
    return "udf"
end

function getArgs()
    args={}
    args[1] = { "a", double }
    args[2] = { "b", double }
    return args
end

function getResult()
```

```
    return double
end
```

UDF Example #2

This UDF takes a string and counts the number of unique characters the string contains, returning the result.

```
function evaluate( str )
    count = 0
    chars = {}
    for ch in string.gmatch(str, "." ) do
        if chars[ch] == null then
            chars[ch] = 1
            count = count + 1
        end
    end

    return count
end

function getType()
    return "UDF"
end

function getName()
    return "unique_chars"
end

function getArgs()
    args={}
    args[1] = { "str", varchar(any) }
    return args
end

function getResult()
    return integer
end
UDF Example #3 (calculateSize)
```

In some situations it can be very useful to support a dynamic output size for a VARCHAR or NUMERIC result. This example shows using the calculateSize method to dynamically set the size of a VARCHAR result column as the sum of the length of the two VARCHAR arguments.

```
function evaluate(s1, s2 )
    return s1 || s2
end

function calculateSize( args )
    return args[1].length + args[2].length
end

function getType()
    return "UDF"
end

function getName()
```

```
    return "concat"
end

function getArgs()
    return {{ "", varchar("any") },
            { "", varchar("any") }}
end

function getResult()
    return varchar(any)
end
```

UDA Examples

Example #1

Calculate an average for a double precision value. This UDA performs the same function as the SQL AVG aggregate, but only for the DOUBLE data type.

```
SUM=1
COUNT=2

function initState(state)
    state[SUM] = 0
    state[COUNT] = 0
    return state
end

function accumulate(state, value)
    state[SUM]   = state[SUM] + value
    state[COUNT] = state[COUNT] + 1
    return state
end

function merge(state1, state2)
    state1[SUM]   = state1[SUM]   + state2[SUM]
    state1[COUNT] = state1[COUNT] + state2[COUNT]
    return state1
end

function finalResult(state)
    return state[SUM] / state[COUNT]
end

function getState()
    state = {}
    state[1] = { "sum", double }
    state[2] = { "count", double }
    return state
end

function getType()
    return "uda"
end
```

```
function getName()
    return "lua_avg"
end

function getArgs()
    args = {}
    args[1] = { "value", double }
    return args
end

function getResult()
    return double
end
```

Example #2

The second UDA example is an analytic UDA (a UDA that is invoked with an OVER clause). This UDA can be used to create unique session identifiers for weblog data based on a session timeout value.

```
--[[-----------------------------------------------------------
Example usage in SQL:

select
       customer,
       click_time,
       sessionize(click_time, 900) over (partition by customer
                                         order by click_time)
from
       weblog_data

--]]-----------------------------------------------------------

function initState(state)
    state[1] = 0
    state[2] = 0
    return state
end

function accumulate(state, ts, seconds)
    if ts - state[1] > seconds then
        state[2] = getNextId()
    end
    state[1] = ts
    return state
end

function finalResult(state)
    return state[2]
end

function getName()
    return "sessionize"
end

function getType()
    return "uda"
end
```

```
function getState()
    state={}
    state[1] = { "", timestamp }
    state[2] = { "", integer   }
    return state
end

function getArgs()
    args={}
    args[1] = { "click_time", timestamp }
    args[2] = { "timeout",    integer   }
    return args
end

function getResult()
    return integer
end

function getOptions()
    options={}
    options[OPT_AGGTYPE] = "analytic"
    return options
end
```

UDTF Examples

Example #1

This example unpivots sales data that is stored in a row containing sales by quarter into four rows of data, one for each quarter.

```
-- Usage Example:
-- select * from table(unpivot_sales(2009,100,200,300,400));

function processRow(y,q1,q2,q3,q4)
    sls = {}
    sls[1] = { y, 1, q1 }
    sls[2] = { y, 2, q2 }
    sls[3] = { y, 3, q3 }
    sls[4] = { y, 4, q4 }
    return sls
end

function getType()
    return "UDTF"
end

function getName()
    return "unpivot_sales"
end

function getArgs()
    args={}
    args[1] = { "year",     "integer" }
```

```
    args[2] = { "q1sales",  "double"  }
    args[3] = { "q2sales",  "double"  }
    args[4] = { "q3sales",  "double"  }
    args[5] = { "q4sales",  "double"  }
    return args
end

function getShape()
    columns={}
    columns[1] = { "year",    "integer" }
    columns[2] = { "quarter", "integer" }
    columns[3] = { "sales",   "double"  }
    return columns
end
```

Example #2

The second UDTF example utilizes the outputRow() and outputFinalRow() methods to output rows instead of relying only on the processRow() method. This UDTF performs the same task as the first example, plus it also outputs one final row that contains the total value of all sales processed by the UDTF.

```
-- Usage Example:
-- select * from table with final(unpivot_final(2010,100,200,300,400));

total=0
function processRow(y,q1,q2,q3,q4)
    sls = {}
    sls[1] = { y, 1, q1 }
    sls[2] = { y, 2, q2 }
    sls[3] = { y, 3, q3 }
    sls[4] = { y, 4, q4 }
    total = q1 + q2 + q3 + q4
    return null
end

function outputRow(rownum)
    if sls[rownum] == null then return null end
    return sls[rownum]
end

function outputFinalRow(rownum)
    if rownum > 1 then return null end
    return { null, null, total }
end

function getType()
    return "UDTF"
end

function getName()
    return "unpivot_final"
end

function getArgs()
    return {{ "year",     "integer"},
```

```
            { "q1sales",  "double" },
            { "q2sales",  "double" },
            { "q3sales",  "double" },
            { "q4sales",  "double" }}
end

function getShape()
    return {{ "year",    "integer" },
            { "quarter", "integer" },
            { "sales",   "double"  }}
end
```

VARARGS Examples

Example #1

This VARARGS UDF takes from one to 64 arguments (the Netezza UDX implementation is currently limited to a maximum of 64 arguments), adds them together, and returns the result. The initialize() method is used to verify that all of the arguments being passed into the UDF are numbers. The global variable ARGCOUNT indicates the number of arguments that have been passed to the VARARGS UDX.

```
function initialize()
    for i,type in pairs(ARGTYPE) do
        if type != TYPE_NUMBER then
            error("All arguments must be numbers!",0)
        end
    end
end

function evaluate( ... )
    local x=0
    for i=1,ARGCOUNT do
        x = x + select(i, ...)
    end

    return x
end

function getType()
    return "udf"
end

function getName()
    return "varadd"
end

function getArgs()
    return varargs
end

function getResult()
    return double
end
```

UDF Output Sizer Examples

Example #1

This code creates a UDF that takes two strings of any length and returns the result of concatenating the strings together. The calculateSize() method is used to determine the size of the result. The calculateSize() method will always be called when the result type of a UDF is defined as varchar(any) or numeric(any). UDA and UDTF do not support the calculateSize() method.

```
function evaluate(s1, s2 )
      return s1 || s2
end

function calculateSize( args )
      return args[1].length + args[2].length
end

function getType()
      return "UDF"
end

function getName()
      return "concat"
end

function getArgs()
    return {{ "", varchar("any") },
            { "", varchar("any") }}
end

function getResult()
    return varchar(any)
end
```

UDTF Output Shaper Examples

Example #1

This simple UDTF splits a input string into multiple output strings based on the a delimiter string. The number of columns and the column names returned by this UDTF are determined at runtime based on the third parameter, which must be a constant value. To activate the runtime call to the calculateShape() method the getShape() method of the UDTF must return the ANYSHAPE constant.

```
function processRow(str,delim,columns)
    return { split(str,delim) }
end

function calculateShape(arg)
    local maxcols = arg[3].value

    local cols = {}
```

```
    for i=1,maxcols do
        cols[i] = { "S"||i, "varchar", 255 }
    end

    return cols
end

function getType()
    return "udtf"
end

function getName()
    return "nzlua_split"
end

function getArgs()
    return {{"", varchar(any)  },
            {"", char(1)       },
            {"", integer       }}
end

function getShape()
    return ANYSHAPE
end
```

SPUPad Example

This example uses the nzlua table function, which is automatically installed in the database by the nzLua installation script. In addition to the code itself, three small tables are created by this SQL script to properly demonstrate the capabilities of nzlua in combination with the SPUPad features.

```
drop table elt_source;
drop table elt_good;
drop table elt_bad;

create table elt_source (str varchar(1000)) distribute on random;
create table elt_good (id integer, dt date, text varchar(255)) distribute
on random;
create table elt_bad (str varchar(1000), mesg varchar(255)) distribute on
random;

insert into elt_source values ('1|20100101|aaaaaa');
insert into elt_source values ('2|2010-01-02|bbbb');
insert into elt_source values ('3|20100103|cccccc');
insert into elt_source values ('4|20100104|ddd');
insert into elt_source values ('x|20100130|eeeeeeeeee');
insert into elt_source values ('6|20100110|ffffff|fffff');
insert into elt_source values ('7|20100132|ggggg');
insert into elt_source values ('9|20101011|hhhhhhhhhhhhhhhhhhhhh');

begin;

\echo
**************************************************************************
\echo **** insert good rows into ELT_GOOD, bad rows get saved in SPUPad
\echo
```

```
************************************************************************
insert into elt_good (id,dt,text)
select id, dt, text
from elt_source, table with final(nzlua('
function initialize()
    errors={}
end

function saveError(str,mesg)
    push(errors, { str, mesg })
end

function processRow(str)
    t = split(str,"|")

    if #t != 3 then
        saveError(str, "Invalid number of columns!" )
        return null
    end

    id = tonumber(t[1])
    if id == null then
        saveError( str, "column 1: Invalid id = " || t[1])
        return null
    end

    ok,dt = pcall(to_date, t[2], "YYYYMMDD")
    if not ok then
        saveError( str, "column 2: Invalid date = " || t[2])
        return null
    end

    text = t[3]

    return { id, dt, text }
end

function outputFinalRow(rownum)
    if rownum > 1 then return null end
    saveTable("errors", errors)
    return null
end

function getShape()
    columns={}
    columns[1] = { "id", integer }
    columns[2] = { "dt", date }
    columns[3] = { "text", varchar(255) }
    return columns
end'
,str)) tf;


\echo
************************************************************************
\echo **** Extract bad rows from SPUPad and insert them into ELT_BAD
table
```

```
\echo
**********************************************************************
insert into elt_bad (str,mesg)
select str, mesg
from _v_dual_dslice, table(nzlua('
function initialize()
    errors=restoreTable("errors")
end

function processRow()
    if errors == null or #errors == 0 then return null end
    return errors
end

function getShape()
    columns={}
    columns[1] = { "str",  varchar(255) }
    columns[2] = { "mesg", varchar(255) }
    return columns
end',
dsid)) tf;

commit;

\echo **** GOOD DATA (ELT_GOOD table)
select * from elt_good;

\echo **** BAD DATA (ELT_BAD table)
select * from elt_bad;
```

nzLua Code Library Example

nzLua provides a way to create reusable nzLua code libraries. An nzLua code library uses a .nzll filename extension rather than the standard .nzl extension. The code can then be dynamically loaded by a UDX by using the require function at the beginning of the nzLua program.

Example #1

This example defines the library. The file must be named testlib.nzll to function correctly with the second example, which utilizes this library.

```
function testcalc(x,y,z)
    return (x+y) * z
end
```

Example #2

This example uses the testcalc function that was defined in the first example.

```
require "testlib"

function evaluate(a,b,c)
    return testcalc(a,b,c)
end
```

```
function getName()
    return "libtest"
end

function getType()
    return "udf"
end

function getArgs()
    args={}
    args[1] = { "a", double }
    args[2] = { "b", double }
    args[3] = { "c", double }
    return args
end

function getResult()
    return double
end
```

APPENDIX B
Notices and Trademarks

Notices

This information was developed for products and services offered in the U.S.A. IBM may not offer the products, services, or features discussed in this document in other countries. Consult your local IBM representative for information on the products and services currently available in your area. Any reference to an IBM product, program, or service is not intended to state or imply that only that IBM product, program, or service may be used. Any functionally equivalent product, program, or service that does not infringe any IBM intellectual property right may be used instead. However, it is the user's responsibility to evaluate and verify the operation of any non-IBM product, program, or service.

IBM may have patents or pending patent applications covering subject matter described in this document. The furnishing of this document does not grant you any license to these patents. You can send license inquiries, in writing, to:

IBM Director of Licensing
IBM Corporation
North Castle Drive
Armonk, NY 10504-1785 U.S.A.

For license inquiries regarding double-byte character set (DBCS) information, contact the IBM Intellectual Property Department in your country or send inquiries, in writing, to:

Intellectual Property Licensing
Legal and Intellectual Property Law
IBM Japan Ltd.
1623-14, Shimotsuruma, Yamato-shi
Kanagawa 242-8502 Japan

The following paragraph does not apply to the United Kingdom or any other country where such

provisions are inconsistent with local law: INTERNATIONAL BUSINESS MACHINES CORPORATION PROVIDES THIS PUBLICATION "AS IS" WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESS OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF NON-INFRINGEMENT, MERCHANTABILITY OR FITNESS FOR A PARTICULAR PURPOSE. Some states do not allow disclaimer of express or implied warranties in certain transactions, therefore, this statement may not apply to you.

This information could include technical inaccuracies or typographical errors. Changes are periodically made to the information herein; these changes will be incorporated in new editions of the publication. IBM may make improvements and/or changes in the product(s) and/or the program(s) described in this publication at any time without notice.

Any references in this information to non-IBM Web sites are provided for convenience only and do not in any manner serve as an endorsement of those Web sites. The materials at those Web sites are not part of the materials for this IBM product and use of those Web sites is at your own risk.

IBM may use or distribute any of the information you supply in any way it believes appropriate without incurring any obligation to you.

Licensees of this program who wish to have information about it for the purpose of enabling: (i) the exchange of information between independently created programs and other programs (including this one) and (ii) the mutual use of the information which has been exchanged, should contact:
IBM Corporation
26 Forest Street
Marlborough, MA 01752 U.S.A.

Such information may be available, subject to appropriate terms and conditions, including in some cases, payment of a fee.

The licensed program described in this document and all licensed material available for it are provided by IBM under terms of the IBM Customer Agreement, IBM International Program License Agreement or any equivalent agreement between us.

Any performance data contained herein was determined in a controlled environment. Therefore, the results obtained in other operating environments may vary significantly. Some measurements may have been made on development-level systems and there is no guarantee that these measurements will be the same on generally available systems. Furthermore, some measurements may have been estimated through extrapolation. Actual results may vary. Users of this document should verify the applicable data for their specific environment.

Information concerning non-IBM products was obtained from the suppliers of those products, their published announcements or other publicly available sources. IBM has not tested those products and cannot confirm the accuracy of performance, compatibility or any other claims related to non-IBM products. Questions on the capabilities of non-IBM products should be addressed to the suppliers of those products.

All statements regarding IBM's future direction or intent are subject to change or withdrawal without notice, and represent goals and objectives only. This information is for planning purposes only. The information herein is subject to change before the products described become available.

This information contains examples of data and reports used in daily business operations. To illustrate them as completely as possible, the examples include the names of individuals, companies,

brands, and products. All of these names are fictitious and any similarity to the names and addresses used by an actual business enterprise is entirely coincidental.

COPYRIGHT LICENSE:

This information contains sample application programs in source language, which illustrate programming techniques on various operating platforms. You may copy, modify, and distribute these sample programs in any form without payment to IBM, for the purposes of developing, using, marketing or distributing application programs conforming to the application programming interface for the operating platform for which the sample programs are written. These examples have not been thoroughly tested under all conditions. IBM, therefore, cannot guarantee or imply reliability, serviceability, or function of these programs. The sample programs are provided "AS IS", without warranty of any kind. IBM shall not be liable for any damages arising out of your use of the sample programs.

Each copy or any portion of these sample programs or any derivative work, must include a copyright notice as follows:

© (your company name) (year). Portions of this code are derived from IBM Corp. Sample Programs. © Copyright IBM Corp. (enter the year or years). All rights reserved.

NOTICES FOR LUA 5.1.4

Copyright © 1994–2012 Lua.org, PUC-Rio.

Permission is hereby granted, free of charge, to any person obtaining a copy of this software and associated documentation files (the "Software"), to deal in the Software without restriction, including without limitation the rights to use, copy, modify, merge, publish, distribute, sublicense, and/or sell copies of the Software, and to permit persons to whom the Software is furnished to do so, subject to the following conditions:

The above copyright notice and this permission notice shall be included in all copies or substantial portions of the Software.

THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM, OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE SOFTWARE.

END OF NOTICES FOR LUA 5.1.4

Trademarks

IBM, the IBM logo, ibm.com and Netezza are trademarks or registered trademarks of International Business Machines Corporation in the United States, other countries, or both. If these and other IBM trademarked terms are marked on their first occurrence in this information with a trademark symbol

(® or ™), these symbols indicate U.S. registered or common law trademarks owned by IBM at the time this information was published. Such trademarks may also be registered or common law trademarks in other countries. A current list of IBM trademarks is available on the Web at “Copyright and trademark information” at ibm.com/legal/copytrade.shtml.

The following terms are trademarks or registered trademarks of other companies:

Adobe is a registered trademark of Adobe Systems Incorporated in the United States, and/or other countries.

Linux is a registered trademark of Linus Torvalds in the United States, other countries, or both.

Microsoft, Windows, Windows NT, and the Windows logo are trademarks of Microsoft Corporation in the United States, other countries, or both.

NEC is a registered trademark of NEC Corporation.

UNIX is a registered trademark of The Open Group in the United States and other countries.

Java and all Java-based trademarks are trademarks of Sun Microsystems, Inc. in the United States, other countries, or both.

Red Hat is a trademark or registered trademark of Red Hat, Inc. in the United States and/or other countries.

D-CC, D-C++, Diab+, FastJ, pSOS+, SingleStep, Tornado, VxWorks, Wind River, and the Wind River logo are trademarks, registered trademarks, or service marks of Wind River Systems, Inc. Tornado patent pending.

APC and the APC logo are trademarks or registered trademarks of American Power Conversion Corporation.

Other company, product or service names may be trademarks or service marks of others.

Regulatory and Compliance

Regulatory Notices

Install the NPS system in a restricted-access location. Ensure that only those trained to operate or service the equipment have physical access to it. Install each AC power outlet near the NPS rack that plugs into it, and keep it freely accessible. Provide approved circuit breakers on all power sources.

Product may be powered by redundant power sources. Disconnect ALL power sources before servicing. High leakage current. Earth connection essential before connecting supply. Courant de fuite élevé. Raccordement à la terre indispensable avant le raccordement au réseau.

Homologation Statement

This product may not be certified in your country for connection by any means whatsoever to interfaces of public telecommunications networks. Further certification may be required by law prior to making any such connection. Contact an IBM representative or reseller for any questions.

FCC - Industry Canada Statement

This equipment has been tested and found to comply with the limits for a Class A digital device, pursuant to part 15 of the FCC rules. These limits are designed to provide reasonable protection against harmful interference when the equipment is operated in a commercial environment. This equipment generates, uses, and can radiate radio-frequency energy and, if not installed and used in accordance with the instruction manual, may cause harmful interference to radio communications. Operation of this equipment in a residential area is likely to cause harmful interference, in which case users will be required to correct the interference at their own expense.

This Class A digital apparatus meets all requirements of the Canadian Interference-Causing Equipment Regulations.

Cet appareil numérique de la classe A respecte toutes les exigences du Règlement sur le matériel brouilleur du Canada.

CE Statement (Europe)

This product complies with the European Low Voltage Directive 73/23/EEC and EMC Directive 89/336/EEC as amended by European Directive 93/68/EEC.

Warning: This is a class A product. In a domestic environment this product may cause radio interference in which case the user may be required to take adequate measures.

VCCI Statement

この装置は、情報処理装置等電波障害自主規制協議会 (VCCI) の基準に基づくクラス A 情報技術装置です。この装置を家庭環境で使用すると電波妨害を引き起越すことがあります。この場合には使用者が適切な対策を講ずるう要求されることがあります。

Index

H

I

J

L

M

N

O

P

R

S

T

U

V

W

X