User’s Guide To The RWP*Load Simulator

Version 2.1

Table of Contents

What is the RWP*Load Simulator....3
What is the RWP*Load Simulator not?....4
Installation....5
- Installation in an Instant Client environment....6
- Creating the rwloadsim repository....6
- De-install....6
- Upgrade from earlier releases....6

Working with RWP*Load Simulator....7
- A few simple examples....7
- Variable and procedure declaration and execution....7
- Interact with the database....8
- Providing default values....10
- Simulating think time....11
- Using multiple execution threads....12
- Using a session pool; specifying execution time....13

RWP*Load Simulator Syntax....16
- Rwloadsim....16
- Simple executable statements....16
- Compound statement....20
- Scalar declaration....22
- SQL declaration....23
- Declaration....26
- Procedure declaration....26
- Argument list....27
- Function declaration....28
- Database declaration....29
- Random string declaration....31
- Random procedure declaration....32
- File declaration....32
- Expression....32
- Identifier and their scope....35
- Direct execution vs. compilation....37
- Thread execution....37
- Selecting a database using the at clause....42

Error handling....44
Saving runtime statistics....44

Overview of execution time gathering....45
Results tables....45
Rwlrun....45
Runres....46
Histogram....46
Persec....47
Statistics with stacked at-clauses....47
RWP*Load Simulator options....49
File contents....52
Providing positional arguments....52
Using rwloadsim as an interpreter....53
Startup file....54
Creating tags file for vi....54
Directives....54
$iformat:%<format>....55
$dformat:%<format>....55
$include:"<filename>"....55
$randseed:<hexdigits>....55
$startseconds:<double> $clockstart:<double>....56
$debugon:<hexdigits> $debugoff:<hexdigits>....56
$mute:<integer>....56
$oraerror:stop....56
$oraerror:continue....56
$bindoffset:0....56
$maxcode:N $maxident:N....56
$statistics:basic $statistics:histograms $statistics:all....56
$setaction:on $setaction:reset $setaction:off....56
$queue:on $queue:off....57
$errortime:on $errortime:off....57
Hints and tips....57
Multi-process execution....58
A complete example....59
Execution statistics....61
A real simulation....63
A multi-process run....65
Using multiple hosts....67
Interrupting rwloadsim....68
Queuing system simulations....68
Using LOB data....69
Demonstration library....69
awr.rwl....69
ovid2.rwl....69
sqlid2file.rwl....69
A word about the rwl language....70
Keywords and predefined variables....70
List of current ordinary rwl keywords....71
List of ordinary rwl keywords, etc reserved for future use....71
List of predefined variables....71
List of predefined functions....71
List of predefined procedures....71

List of keywords and punctuation causing scanning for SQL text to begin............................71
List of keywords and punctuation causing scanning for PL/SQL text to begin.......................71
List of directives...71
List of punctuation..71
Known issues..72
Debug bits..72
Credits..72

What is the RWP*Load Simulator

The RWP*Load Simulator – or rwloadsim – is a programming and simulation tool that can be used for several purposes involving execution of SQL statements. One of these is reflected in its name and is to simulate database load. However, it really does much more than that, and it is best thought of as a tool that fills in the space between SQL*Plus, OCI and scripting done with e.g. the Unix command shell. As it is a command line tool, it is well suited for various types of batch or scripting environments, including but surely not limited to testing, triage, and load simulation.

RWP*Load Simulator includes a small programming language used to specify and generate the load, and it therefore has elements from general application programming environments. The programming language has some resemblance with PL/SQL (declaring SQL statements and some operations on cursors) with awk (free intermixing of data types), and typical programming languages (procedures, functions, if/then/else logic, for loops, expressions). It also has rudimentary printing without formatting possibilities.

As a load simulator, it can be used to execute specified SQL statements in a certain order and fashion specifying different types of random input, with variable run duration and/or execution counts or frequency. At the same time, statistics about execution of the individual SQL statements and/or procedures (combination of SQL statements) are collected and saved in a database schema for later analysis, printing, etc. All this can be done with different connection methods such as dedicated connections or a session pool.

As just one example, rwloadsim can simulate a simple order entry workload, where each order gets an order number from a sequence, has a random, erlang (k=2) distributed number of order lines, and e.g. has a status code that gets one of a set of random values with a certain distribution such as 80% open, 15% on hold and 5% closed.

RWP*Load Simulator includes a programming language, and it therefore has elements from general application programming environments. The programming language has some resemblance with PL/SQL (you can declare SQL statements and do many typical operations on cursors) with awk (data types such as strings or numbers can be freely mixed), and several typical programming languages (procedures, functions, if/then/else logic, for loops, expressions). It also also includes features for printing or writing to files in a fashion that is very suitable for many kinds of scripting.

The way SQL is being processed is similar to how OCI does it with bind and/or define variables, so you can also think of rwloadsim as a way to execute OCI without having to write a C program.

As a command line tool (the executable is rwloadsim), RWP*Load Simulator reads one or more

RWL files as input, parses and executes these. It is generally multi-threaded, which particularly is useful for simulating load, where threads can represent workers, with execution statistics potentially being saved to a database schema.

The following very simple example shows some of the basic features of rwloadsim. Consider a file, emp.rwl, with this contents:

```
database scott username "scott" password "tiger" default;
# Tell how to connect to the database

integer empno, deptno:=10, numemps:=0;
# Declare some variables, and possibly initialize them
string ename;

sql selemps # Declare a SQL statement
  select empno, ename from emp where deptno=:1;
  define 1 empno, 2 ename; # As it is a query, define the select list elements
  bind 1 deptno; # Bind the single placeholder to a variable
  array 10; # Set an array size
end;

for selemps loop # Execute a cursor loop
  println empno, ename; # print something to stdout
  numemps := numemps + 1; # count the number of rows
end loop;

if numemps=0 then # If there were no rows, print a message
  printline "No employees in department", deptno;
end if;
```

If you execute rwloadsim with this file as argument, you will get

```
rwloadsim emp.rwl

RWP*Load Simulator Release 2.0.1.30 Beta on Mon Aug 13 02:13:03 2018

Connected scott to:
Oracle Database 12c Enterprise Edition Release 12.2.0.1.0 - 64bit Production

7782 CLARK
7839 KING
7934 MILLER
```

If you wanted to do execute the same, although specifying a different value of deptno, you may get

```
rwloadsim -i deptno:=42 emp.rwl

RWP*Load Simulator Release 2.0.1.30 Beta on Mon Aug 13 02:14:32 2018

Connected scott to:
Oracle Database 12c Enterprise Edition Release 12.2.0.1.0 - 64bit Production

No employees in department 42
```

What is the RWP*Load Simulator not?

The RWP*Load Simulator is not an application programming tool, and it should not be used as such or as a general purpose programming environment. If you attempt using it beyond its design purpose, you will quickly find that several important things are missing. Some examples are a complete lack of GUI, very limited ability to take input, only simple printing and output, several programming capabilities found in almost any other programming language are missing, only few simple data types are supported. That said, it does support a modular programming approach that is suited for the purpose of simulating load.

Installation

Rwploadsim is currently only available on 64 bit Linux, and it requires an Oracle Client release 11.2 or newer. The download is one gzip compressed tar file, which on purpose is created such that it can be un-tar'ed at the top of your ORACLE_HOME, although that is absolutely not necessary. It contains the following files and directories:

bin/rwloadsim	A helper shell script that will inspect LD_LIBRARY_PATH to see which of the three executables to actually call.
bin/rwloadsim11 bin/rwloadsim12 bin/rwloadsim18	The actual RWP*Load Simulator executables compiled for execution in an 11.2, 12 or 18 environment.
rwl/demo	The samples and demonstrations discussed in this document
rwl/admin/vim.tar	A file that will provide vim with rwl syntax
rwl/admin/create_user.sql	A SQL script to create the rwloadsim repository schema
rwl/admin/rwloadsim.sql	A SQL script to create the rwloadsim repository
rwl/admin/rwlviews.sql	Create various various view in the repository schema
rwl/admin/rwldrop.sql	A script to drop the repository; can be used if re-create is necessary
rwl/doc	This document including the full rwl input file syntax

After download, you should either create a directory where you can put the contents of the .tgz file, or you can put it directly into your ORACLE_HOME. To expand the file, go to your chosen directory and type (replace with the actual release):

```
tar -zxvf rwl.2.0.8.beta.tgz
```

If you do this into your existing ORACLE_HOME (release 11.2 or newer), nothing else is necessary, although you may manually need to do it multiple nodes of a RAC installation. If you unpack the contents of the .tgz file somewhere else, you should copy the executables, rwloadsim{11,12,18} and the shell script rwloadsim, from the bin directory to a directory in your PATH or simply put the bin directory of your install into your PATH.

It is necessary that your LD_LIBRARY_PATH includes $ORACLE_HOME/lib of an Oracle release 11.2.0.4 or newer; a client install (e.g. instant client) is sufficient.

If you want vim to understand the syntax of rwloadsim input file with suffix .rwl, you should untar the file rwl/admin/vim.tar into your home directory, which will provide the two files .vim/ftdetect/rwl.vim and .vim/syntax/rwl.vim that respectively identifies files with suffix .rwl as rwloadsim input files and defines the syntax of these for vim.

If you installed into ORACLE_HOME, It is strongly recommended that you copy the contents of the rwl/demo directory to a work directory of your own before running the various tests and demonstrations discussed here.

Installation in an Instant Client environment

If you want RWP*Load Simulator in a client environment that is using Instant Client, you are suggested following these steps

- Unpack the contents of the tar file directly into the top of your Instant Client directory
- After un-packing, move the files in rwl/bin to the Instant Client top level directory

If you are following this approach, your existing environment variables such as PATH and LD_LIBRARY_PATH will allow execution of rwloadsim.

Creating the rwloadsim repository

You should subsequently create the schema for the RWP*Load Simulator repository by doing these two steps

- From the rwl/admin directory, modify the create_user.sql script as needed and run it using sqlplus logged in as a DBA.
- From the same directory, log in to sqlplus as the new user (rwloadsim by default) and execute the rwloadsim.sql and rwlviews.sql scripts.

If you are planning to use rwloadsim for several different projects, it is suggested that you only create one repository schema, as the repository is prepared to use in multiple projects.

De-install

To de-install from an ORACLE_HOME, execute:

```
rm -f bin/rwloadsim bin/rwloadsim11 bin/rwloadsim12 bin/rwloadsim18
rm -rf rwl
```

To de-install from a place of your own, simply remove the executables bin/rwloadsim* and the rwl directory.

Upgrade from earlier releases

If you upgrade from an earlier release of rwloadsim, you don't need to de-install before upgrading. If the newer release includes changes to the repository, upgrade scripts will be provided.

Working with RWP*Load Simulator

The RWP*load Simulator (rwloadsim) has a relatively simple command interface and is primarily targeted at scripting environments, where sqlplus together with things like shell, sed, awk and other Unix tools doesn't allow sufficient control of how to execute SQL statements or a mix of concurrently executing SQL statements. Rwloadsim has several ways to specify random values for SQL or PL/SQL with bind variables, and it can control things like frequency of execution, mix of "transactions", several connection mechanisms including session pooling, while it also may gather runtime execution statistics such as throughput and histograms of execution times. All such statistics are stored in database tables in the repository schema to allow for reporting. The rwloadsim language is also sufficient to allow automatic generation of awr reports.

Some of the rwloadsim behavior is controlled via options to the executable, and the actual rwl program is specified in text files that are read by rwloadsim; several options can be set globally using startup files. The text files are written in the rwloadsim language "rwl" (pronounced "rawl") and has typical programming language elements such as declarations and executable code. One or more of these files are given as input to rwloadsim; they are read in sequence, and are parsed and executed. There is no separate "compile" and "execute" step as in real programming languages, declarations of things like variables and procedures are stored immediately and code is executed immediately. It is also possible to include rwl files within others similar to using the @ clause in SQL*Plus or #include in C.

A few simple examples

The samples shown here will gradually introduce some of the important features of rwloadsim. All samples are available in the download file in the rwl/demo subdirectory; it is recommended to copy these files to a personal directory before execution.

Variable and procedure declaration and execution

The first example in sample1.rwl does not interact with the database, and it shows how to declare variables and procedures, has examples of expressions, and shows how to execute a procedure.

```
integer a, b; # declare two integer variables

# declare a procedure that takes two arguments
procedure add(integer v1, integer v2)
  integer c; # a local variable
  c := v1+v2; # assign sum of arguments to c
  if c>100 then # sometimes print
    println "c is larger than 100", c;
  end;
end;

procedure setab()
  a := uniform(0,50); # give a a random value between 0 and 50
  b := uniform(50,150);
end;

integer i; # declarations can come anywhere

procedure runten()
```

```
  for i := 1 .. 10 loop # do something ten times
    setab();
    add(a,b);
  end;
end;

runten(); # execute the procedure

printline a,b,a-b; # print the values of a and b
```

You can simply execute

```
rwloadsim sample1.rwl

RWP*Load Simulator Release 2.0.1.30 Beta on Mon Aug 13 02:21:30 2018

c is larger than 100 117
c is larger than 100 159
c is larger than 100 158
c is larger than 100 127
c is larger than 100 192
5 83 -78
```

The sample shows some of the important concepts of rwloadsim:

- Variables are declared and used quite similarly to how it is done in other programming languages; the available data types are integer, double and string(N) with a specified maximum length.
- Procedures, which are the main constructs of rwloadsim, can take arguments and contain a list of statements
- Variables are either public, private or local to a procedure. Private variables will be discussed later.
- Expressions such as a+b, a-b, uniform(0,50) can be used in assignments and other places
- Procedures can call other procedures
- Programming constructs like if/then/end and for loops are available.
- The language is free format with semicolon as terminator
- Comments span from the # symbol until end of line

Interact with the database

The main purpose of rwloadsim is to execute SQL statements against an Oracle database, so let us show a small example of how this can be done. As the same database credentials are likely to be used for many different cases, let us first create a file that contains just a database credential. You can execute the testuser.sql file (not shown here) from sqlplus logged in as a DBA, which will create a user named "rwltest". Assume this is done, you can then declare a database in rwloadsim using:

```
# create a default database connection called rwltest
database rwltest username "rwltest" password "rwltest" default;
```

The keyword database starts the declaration (just like the keyword integer starts a declaration of an integer), and the actual username and password are entered as stings. The keyword default marks this database as default, which means it will be used when no database has been explicitly named. The above is availble as the file rwltest.rwl; you can modify it if necessary. If you need to have a connect string you could e.g. use something like:

```
# create a defalt database connection called rwltest
database rwltest connect "//host/service:dedicated"
username "rwltest" password "rwltest" default;
```

Let us next see how SQL statements are declared and executed. First use sqlplus to create a table like this in the above rwltest schema:

```
create table verysimple
( a number
, b varchar2(30)
)
```

Then look at the file simplesinsert.rwl which contains:

```
# declare some variables that
# are used to bind in the insert statement
double a;
string(30) b;

# declare a SQL statement that does the insert
sql sqlinsert
  insert into verysimple
  ( a, b )
  values
  ( :1, :2 )
  /
  # bind the two place holders to the variables
  bind 1 a;
  bind 2 b;
  array 5; # set a bind-array size
end;

integer max := 12;

# declare a procedure that inserts some rows
procedure doinsert()
  integer i;
  for i := 1.. max loop
    # assign values to the two bind variables
    a := erlang2(1);
    # the next line shows that strings and
    # integers can be concatenated
    b := " row number "||i;
    # the bind array is used implicitly
    sqlinsert;
  end ;
  # this will also flush the bind array
  commit;
end;

# actually execute the procedure
doinsert();
```

You can now execute this:

```
rwloadsim rwltest.rwl simpleinsert.rwl
```

There will be no output shown, except the banner as there are no print statements in your input files, but you can use sqlplus to actually see the result of the insert, where the first few lines might be:

```
SQL >select * from verysimple;

         A B
---------- ------------------------------
2.18538601  row number 1
1.81184842  row number 2
2.07463075  row number 3
```

This example shows some important concepts of rwloadsim:

- Rwloadsim processes multiple input files, one after another, which e.g. can used to have modules for different purposes. Here, it is just used to separate the declaration of the database connection from what actually gets executed.
- Just as you can declare scalar variables, you can declare more complex things, such as databases, sql statements, procedures, etc. Note that wloadsim only has one namespace for public identifiers, so all public names (which e.g. excludes arguments to procedures and names of local variables) must be unique. You could e.g. not give a SQL statement and a procedure the same name.
- SQL statement syntax is similar to that of sqlplus and can be terminated by a line with just a single "/"; a "." could also have been used, just as a line ending with ";" (except when using PL/SQL). To allow for nice indentation of source files, the “/” or “.” may have white-space in front of it.
- For SQL statements with placeholders (bind variables), you need to specify which of your declared variables should be bound to which placeholders. In the example, this is done as bind-by-position; bind-by-name is also possible.
- As this is an insert operation, using the array interface is highly recommended. When specified, the array is implicitly created and flushed as needed; in the case above with an array size of five and a total of 12 rows, there will be three actual flushes, the first two with the array filled (the array insert is actually happening at every fifth to sqlinsert), the last with only two elements in the array, and which actually takes place at commit.
- The assignment to the variable "b", which is of type string(30) shows that numbers (here the value of the integer variable "i") are implicitly converted to a string as needed. The opposite is also the case – if you use a string value in an expression, an implicit conversion to a number (integer or double) will be done. These implicit conversions never result in errors, similar to the behavior in awk.

Providing default values

When you execute the above, it will insert 12 rows into the database table as the variable "max" is initialized with the value 12. What if you wanted to insert some other number of rows? You could obviously change the file and rerun, but this is not practical to do, and would be hard to do as a script. However, because the variable named "max" is initialized in the file, you can overwrite that initialization value when calling rwloadsim. If you e.g. run

```
rwloadsim -i max:=100 rwltest.rwl simpleinsert.rwl
```

100 rows will be inserted into the table. The -i option (which can be repeated) is used to initialize an integer variable; there is also a -d option for double variables.

Simulating think time

We have so far just looked at busy loop without think time and with a certain number of executions. Such things can be handy for filling tables, but what if you wanted run a simulation taking 10 minutes with some average think time between each execution? Doing such things is exactly what rwloadsim is created to do. The file simpleinsert2.rwl is a slightly modified version of simpleinsert.rwl, and is used to show that and also shows how procedures can take arguments:

```
# declare some variables that
# are used to bind in the insert statement
double a;
string(30) b;

# declare a SQL statement that does the insert
sql sqlinsert:
  insert into verysimple
  ( a, b )
  values
  ( :1, :2 );
  # bind the two place holders to the variables
  bind 1 a;
  bind 2 b;
  array 5; # set a bind-array size
end;

# the next variable will be summed from threads
integer threads sum totalrows:=0;

# declare a procedure that inserts some rows
procedure doinsert(integer rows)
  integer i;
  tottalrows := totalrows + rows;
  for i := 1.. rows loop
    # assign values to the two bind variables
    a := erlang2(1);
    # the next line shows that strings and
    # integers can be concatenated
    b := " row number "||i;
    # the bind array is used implicitly
    sqlinsert;
  end ;
  # this will also flush the bind array
  commit;
end;
```

There are a few important changes:

- The number of rows to insert is provided as an argument to the procedure doinsert();
- There is a variable "totalrows", which is declared with the threads sum option; it will shortly be described what the purpose of this is
- There is no doinsert() at the end of the file, so no actual execution of the procedure takes place

Due to the missing execution, if you run

```
rwloadsim rwltest.rwl simpleinsert2.rwl
```

nothing will actually be inserted into the database. If you had included a line like

```
doinsert(5);
```

running it would have inserted 5 lines into the table. The file runsimple.rwl has this contents:

```
procedure someinserts()
  integer rr;
  loop wait 0.5 stop 10;
    rr := uniform(1,10);
    doinsert(rr);
  end;
end;

someinserts();

printline "inserted", totalrows;
```

If you now execute rwloadsim with all three input files, you may see something like:

```
rwloadsim rwltest.rwl simpleinsert2.rwl runsimple.rwl

RWP*Load Simulator Release 1.2.0.3 Beta on Fri Jul 20 02:12:48 2018

Connected rwltest to:
Oracle Database 12c Enterprise Edition Release 12.2.0.1.0 - 64bit Production

inserted 101
```

The loop .. end construct in the file runsimple.rwl is called a *control loop* and it shows one of the most important core features of rwloadsim, namely the possibility to execute something that simulates what end users may do. In the example shown above, there is a wait time specified at 0.5 (seconds) and a stop time specified at 10 (seconds). The implication is that the loop will execute 10s and after each execution of the loop, there will be a wait of 0.5s. Assuming the actual time taken to execute the statements of the loop is negligible compared to the 0.5s wait time, the loop will therefore execute (approximately) 20 times.

Using multiple execution threads

Simulating a load with just a single thread of execution is in most case far from sufficient. What if you wanted to simulate ten concurrent users (or application server threads) each concurrently running the above? You could start rwloadsim ten times in the background, simulating ten end users. But what if you want your simulation to use a session pool rather than having ten individual dedicated connections to the database? Ability to do this is another very important feature of rwloadsim.

We are now using a slightly modified version of the last file above, runsimple2.rwl:

```
procedure someinserts()
  integer rr;
  loop wait 0.5 stop 10;
    rr := uniform(1,10);
    doinsert(rr);
  end;
end;

run
  threads 10
    someinserts();
  end;
end;

printline "inserted", totalrows;
```

The procedure is declared in the same way, but in stead of just calling the procedure, we use the run/threads/end construct which is similar to starting things in the background using & in the shell.

We can now execute:

```
rwloadsim rwltest.rwl simpleinsert2.rwl runsimple2.rwl

RWP*Load Simulator Release 1.2.0.3 Beta on Fri Jul 20 02:34:05 2018

Connected rwltest to:
Oracle Database 12c Enterprise Edition Release 12.2.0.1.0 - 64bit Production

inserted 1150
```

What happens at the run command in the third file is the following:

- Ten threads will be started and each thread will make a dedicated connection to the database
- Each thread will execute the named procedure
- As there are ten threads each running a loop for ten seconds with 0.5s wait, doinsert() will be executed about 200 times, each with a random number of rows as argument.
- Note that each thread has its own copy of the variables such as "a", "rr", etc; these variables are destroyed when the threads terminate.
- Each thread also has its own copy of the "totalrows" variable, but after execution of threads, the actual value of the contents from each thread is added to the variable in the main thread; this behavior is what "threads sum" in simpleinsert2.rwl does.
- Due to this addition of the individual "totalrows" variables, the grand total can be printed after the threads have finished.

Using a session pool; specifying execution time

In the example above, we were using the existing database called "rwltest", which is a dedicated connection. This implies each worker thread (the ten threads above) all will acquire their own dedicated connection as well. You would normally want to use a session pool in stead. In rwloadsim this can be achieved by a slightly different database declaration. Take a look at rwltest2.rwl with the following contents. If you need a connect string, you must add it as described previously.

```
# Use a dedicated connection as default:
database rwltest username "rwltest" password "rwltest" default;
# And declare another database as pooled:
database rwlpool username "rwltest" password "rwltest" sessionpool 1..4;
```

When giving this file as input to rwloadsim, you declare two databases; one is the same we have used above, the second one named "rwlpool" really is a session pool with a variable poolsize between 1 and 4. Note that this second database does not have the default keyword, so it must be explicitly named when you want to use it. Now, finally take a look at runsimple3.rwl with this contents:

```
integer exectime := 60; # default 1 min execution time
integer numthreads := 10; # default 10 threads

procedure someinserts()
  integer rr;
  loop wait erlang2(0.02) stop exectime;
    rr := uniform(1,10);
    doinsert(rr);
  end;
end;


run
  threads numthreads at rwlpool
    someinserts();
  end;
end;

printline "inserted", totalrows;
```

Execution will run for just over a minute and may now show this:

```
rwloadsim rwltest2.rwl simpleinsert2.rwl runsimple3.rwl

RWP*Load Simulator Release 1.2.0.3 Beta on Fri Jul 20 02:47:59 2018

Connected rwltest to:
Oracle Database 12c Enterprise Edition Release 12.2.0.1.0 - 64bit Production

Created rwlpool as session pool (1..4) to:
Oracle Database 12c Enterprise Edition Release 12.2.0.1.0 - 64bit Production

inserted 154622
```

Some comments about this:

- There are now two database connections, one being a session pool; the version strings are shown from both.

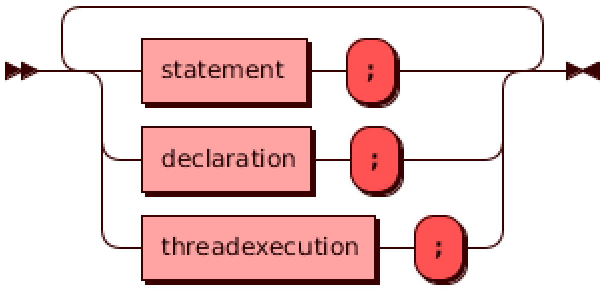
- If you run a command like "top" in another window, you will see rwloadsim and four dedicated connections being active; these four are the four connections from the session pool.
- The ten worker threads in rwloadsim will not have dedicated connections, but will in stead acquire a session from the pool just before each execute of the procedure "doinsert()", and release the session back to the pool immediatedly after the call.
- Effectively, this means the wait time (which is erlang (k=2) distributed with a mean of 0.02s) simulates user think time, during which no session is held.
- Each worker thread will run until a certain stop condition is met – in this case the stop condition is stop exectime or stop after 60s.

RWP*Load Simulator Syntax

The rwloadsim program reads one or more input files with a suggested suffix of .rwl. The files must be written in the rwloadsim language, "rwl", as the various samples show. The full syntax diagram is at rwlsyntax.xhtml which is included in the distribution. While all syntax is explained below, only some syntax elements are shown, so it is suggested that you open a browser window with the complete syntax while reading this chapter. If your browser does not support xhtml files with embedded svg, the full syntax is available at rwlsyntax.html using plain html with image files.

Rwloadsim

An rwloadsim program consists of a sequence of entries, which are all terminated by a semicolon. Effectively, the semicolon is a terminator rather than a separator, and there is e.g. a semicolon before the keyword "else" in an if/then/else construct. You can include comments in your program in the same fashion is in e.g. the standard UNIX shell where comments are initiated by # and terminate at the end of a line. There are three types of entries in an rwloadsim program: statements (which include simple declarations), declarations, and thread executions.



Each file opened by rwloadsim, directly on command line or via a $include directive (see the chapter on directives), must contain one or more complete rwloadsim programs.

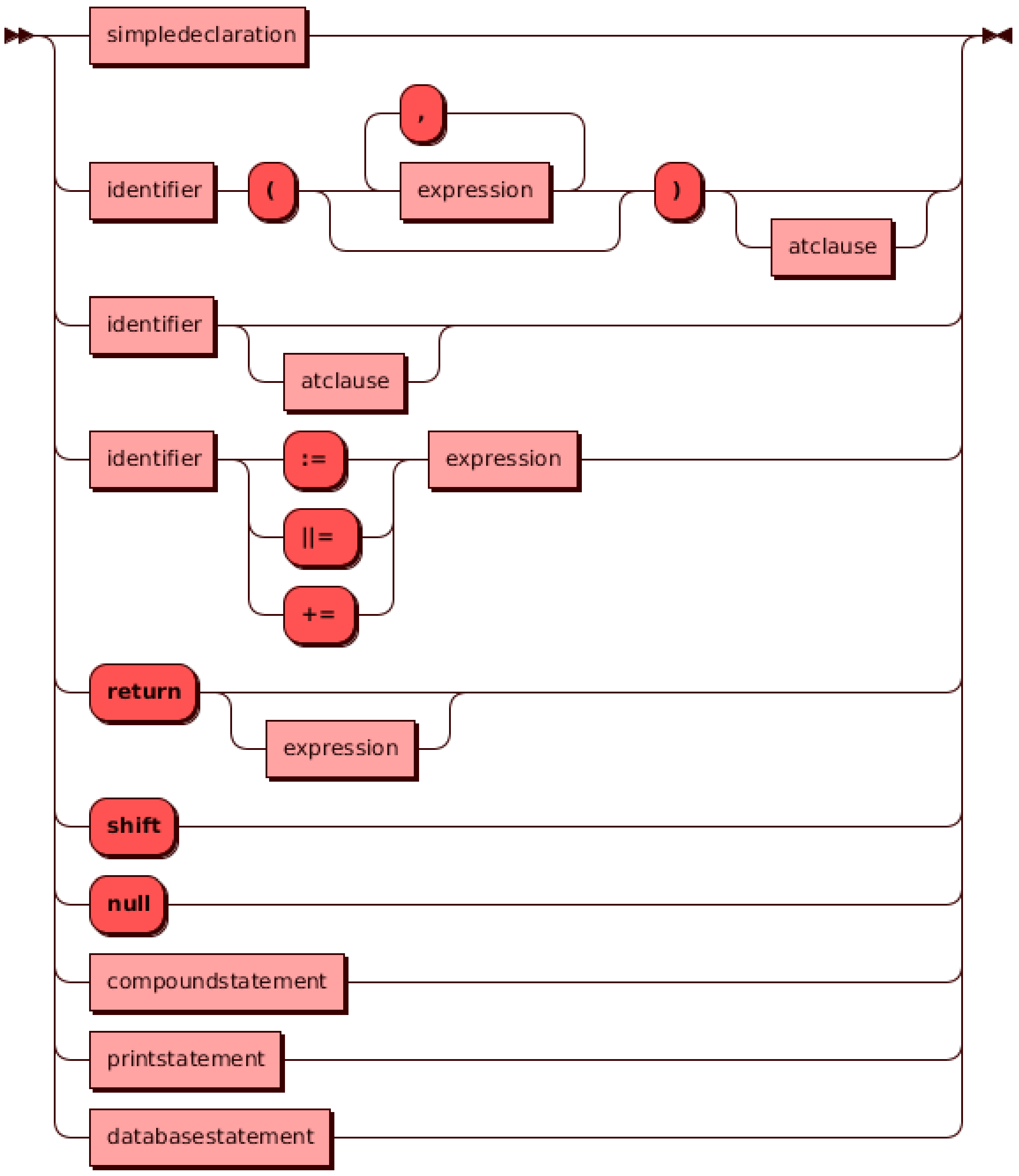
Statements are the most important building blocks of an rwloadsim program, and they include actual executable code as well as declarations of simple variables and SQL. Note that when statements are found directly at the "top level" of the input file, they are executed immediately as they are parsed. Statements that are part of a procedure, function or compound statement are compiled for subsequent execution; they are characterized by having a body that is a list of statements.

Declarations are more complex declarations that are only available at the top level of an rwloadsim program.

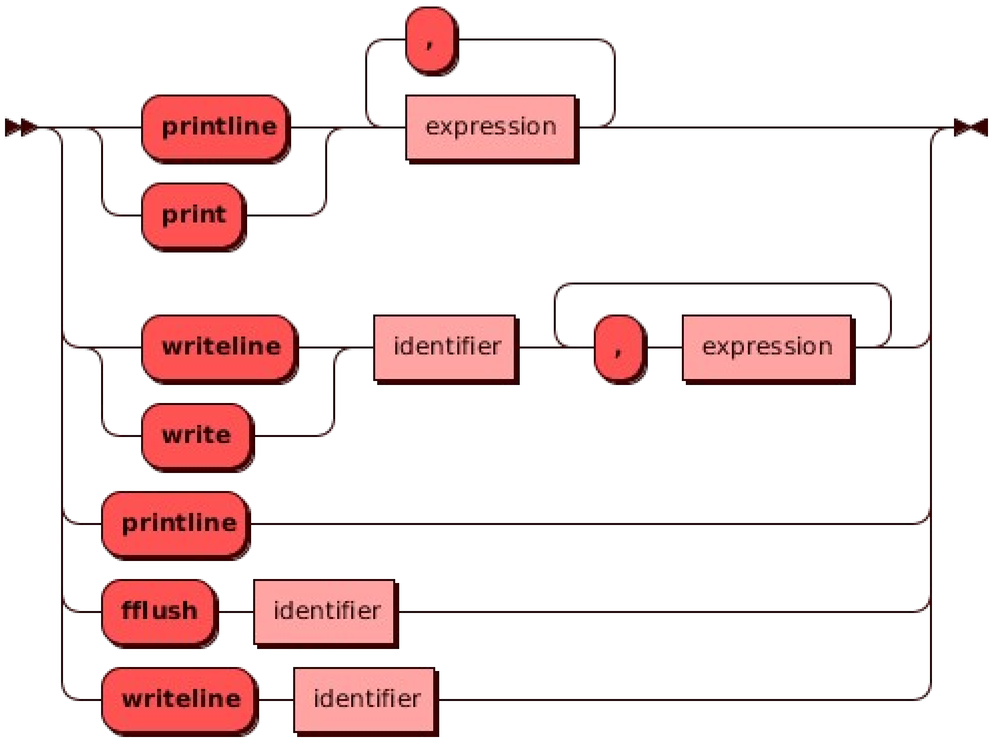
A threadexecution is also only available at the top level, and handles everything on starting and stopping threads.

Simple executable statements

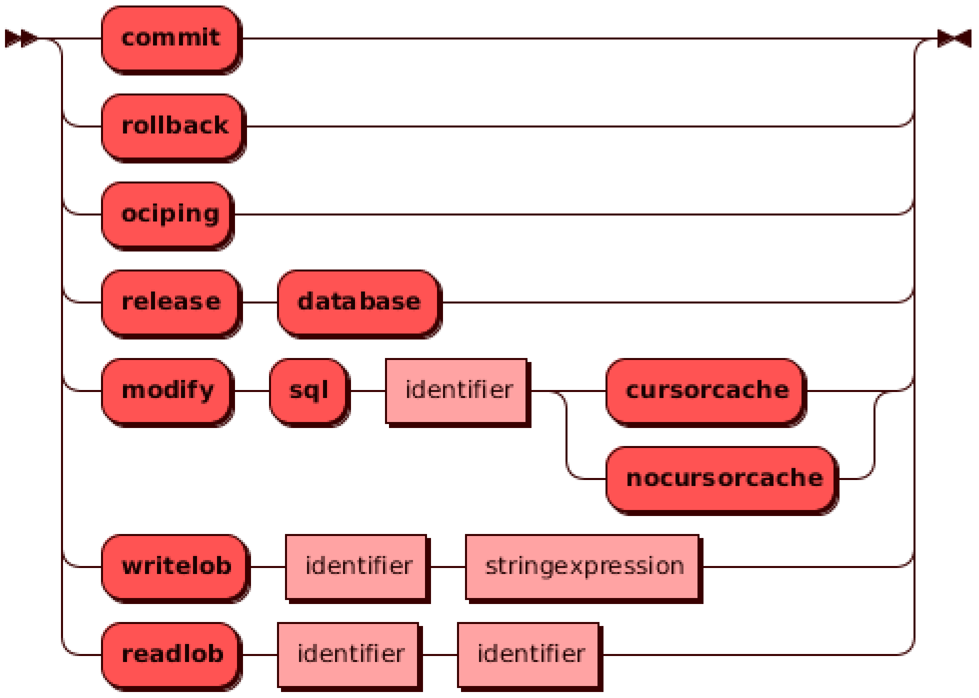
All simple statements are shown on the next pages.


simpledeclaration
identifier
(
,
expression
)
atclause
identifier
atclause
identifier
:=
||=
+=
expression
return
expression
shift
null
compoundstatement
printstatement
databasestatement

A printstatement is one of the following:



A databasestatement is one of these:



Simple declaration	Declaration of simple types or SQL – see below for detail.
Named sql or subroutine	Procedures can call other procedures or can execute a declared SQL statement, by simply naming the procedure (with a potentially empty argument list) or SQL statement. If no database is named the default database or the database specified somewhere else (e.g. as a thread specification) will be used.
Assignment	The identifier named will be assigned the value of the expression after the `:=` symbol. Note that multiple assignments as one statement, such as `a:=b:=0`; are not allowed. The `||=` operator appends the expression to the end of the named string variable, and the += operator adds the expression to the named double or integer variable. No other assignment operators known from the C language exist.
return	Returns execution from a procedure; returns an expression from a a function.
commit/ rollback	The current transaction (if any) will be commited or rolled back. If SQL statement with array DML has been executed, the contents of the array will be flushed before commit or rollback
ociping	Call OCIPing against the current database; can e.g. be used to measure latency.
release database	When using a session pool, the next release of the session to the pool will also disconnect the session from the database.
modify sql	Turn cursor (statement) caching on or off for the named statement. If this is executed in a procedure called from the main thread (direct execute of a procedure), caching will be turned on/off for all subsequent executes of that SQL statement. If this is executed in a running thread, caching will only be modified for that SQL statement in that thread.
writelob	Write data to a LOB by specifying the LOB variable and the (string) expression that will be writting to the LOB
readlob	Read data from a LOB (the first identifier) into a string variable (the second identifier)
print/ printline	Prints a list of expressions (with a single blank between each) to stdout, printline terminates with a newline, print does not.
write/ writeline	Similar except output is to the file identified, which must be open.
fflush	Flushes a file. Useful for stdout in pipelines, etc.
shift	Shift positional arguments left and reduce $# by one. Effectively copies $2 to $1,

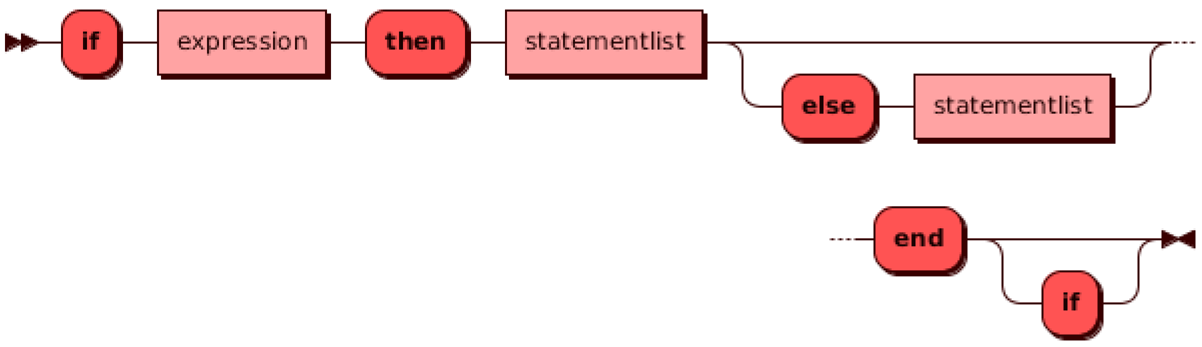
	$3 to $2, etc. without actually reducing the number of variables.

Compound statement

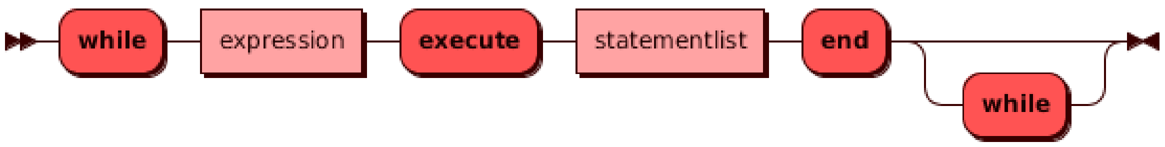
Compound statements includes several types of loops and if/then/else constructs. They all begin with some keyword and terminate with the keyword “end” optionally followed by the initial keyword, which can be used to clarify rwl source code. When these are found at the top level (i.e. outside procedures or functions) while rwloadsim reads an input file, they are first compiled as a block before actual execution. All compound statements can be nested and/or called recursively, except the control loop.

The following compound statements exist:

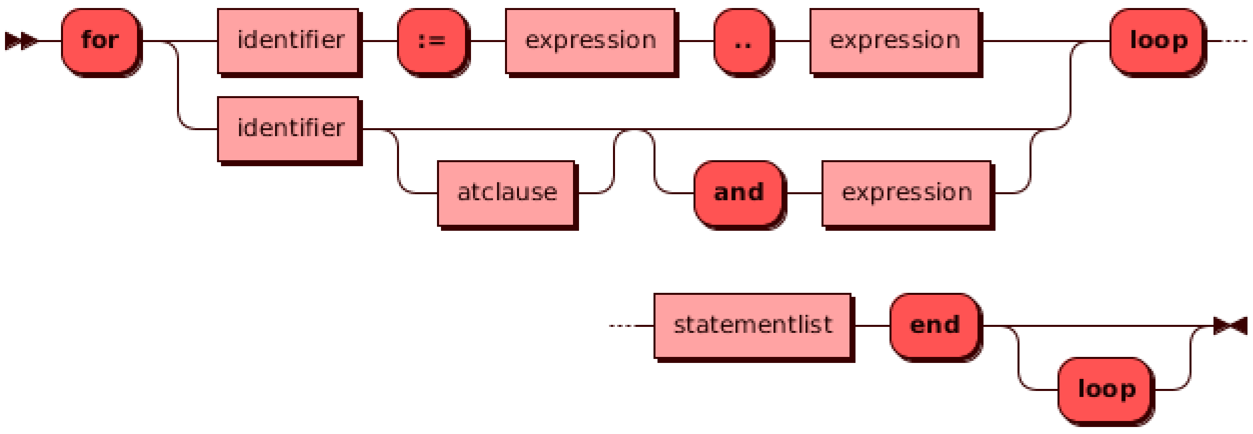
If statement:



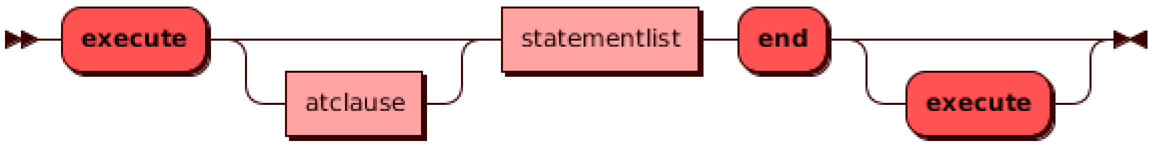
While statement:



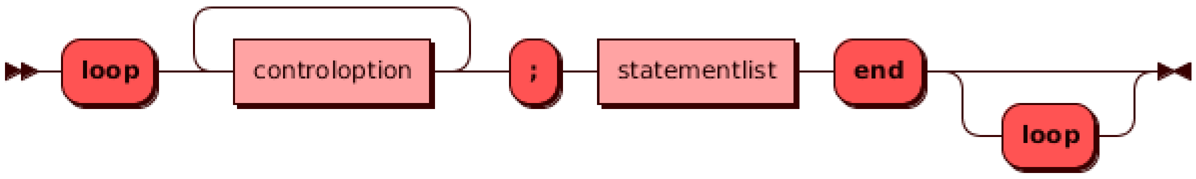
For loops:



Execution block:



Control loop:



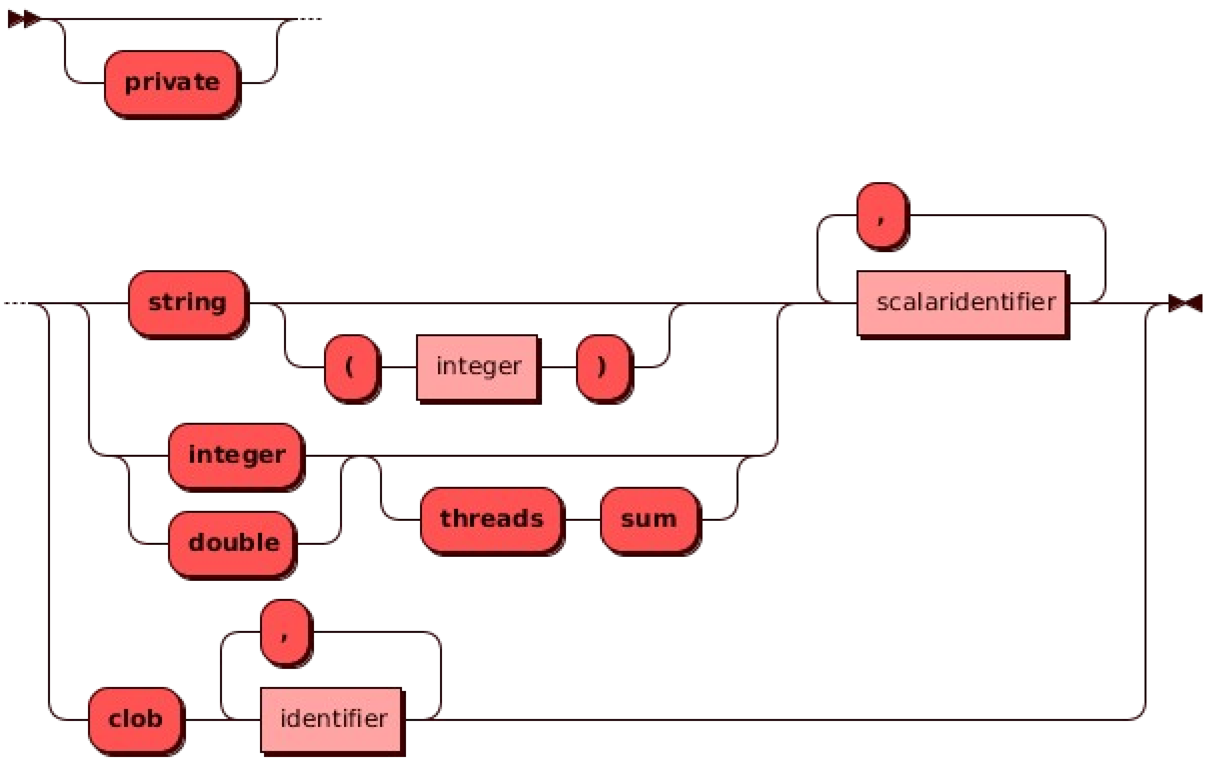
if/then/else/end	The expression is evaluated and if it non-zero, the statements between “then” and “end” or “else” are executed. Otherwise, the statements between “else” and “end” are executed. The second part is optional. If the expression is null, a warning is shown and the statement list after “then” will not be executed.
while/execute/ end	Execute the statements between the “execute” and “end” keywords while the expression is true (non-zero).
for loop (with scalar variable)	The named identifier loops through the values of the expressions between := and .. and .. and “loop” respectively and the statement list between “loop” and “end” is executed once with each value. If the former of these values is larger than the latter, no statements are executed.
for cursor loop	The identifier must refer to a SQL statement that is a query, and the statement list between “loop” and “end” will be executed once for each row fetched from the query. If no rows are returned, no execution will take place. If you have “and” and an expression after the name (and possibly at clause) the fetch will be canceled when the expression is false. Note that due to array fetching, some rows may have been fetched from the database but not yet processed when the condition after “and” becomes false.
execute/end (execution block)	The list of statements between “execute” and “end” are executed as a block. If any SQL is being executed, it will be executed against the default database or the database explicitly name after the “at” keyword. This is similar to begin/end in other programming languages. Effectively, there is only need for execution blocks when used with an at clause.

loop/end (control loop)	The control loop is the primary mechanism used to simulate load. It is initialized by the “loop” keyword and defines a typically repeated execution of the statements in the statement list. The start/stop timing and/or count of executions is controlled via the control options, which are described in detail below under threadexecution. Note that any attempt at nesting control loops will cause a runtime error.

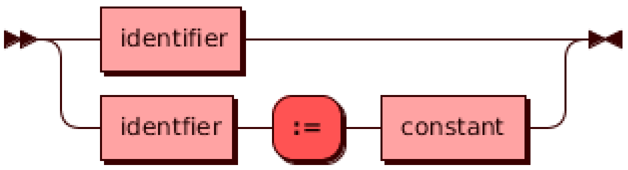
Scalar declaration

Scalars (which include SQL) can be declared globally or locally inside a procedure. There is only one namespace for global declarations so all identifiers must be unique within all the files provided to a single execution of rwloadsim. Most declarations can be made private which implies they are only available in the rwl input file in which they are declared.

The syntax for simple scalars is:



A scalaridentifier is an identifier potentially followed by initialization:



A scalar declaration is used to declare and potentially initialze a simple scalar variable. There are several types of variables:

string	A string is used to store character data, just like the varchar2 database type. If a length is not provided, the default is 128. Note that in rwloadsim, a string is never considered null, although an empty string will be considered null by the database. The length is in bytes, irrespective of the client character set being used. When the -A argument is used, there will be string variables named $1, $2, etc that get their values from the command line.
integer	An integer variable is a 64-bit signed whole number. It can be null.
double	A double variable is a 64-bit floating point number. It can be null.
clob	The clob datatype closely matches an OCILobLocator

All simple variables can be initialized at declaration time, and all variables declared outside functions and procedures are global by default. When worker threads are started, variables are initialized to their values from the main thread, except for variables with the threads sum attribute, which are initialized as zero in worker threads, and summed to the value in the main thread after worker threads finish.

Variables declared with the private keyword are only available in the rwl source file where they are declared. Variables declared inside procedures and functions are local to that procedure or function.

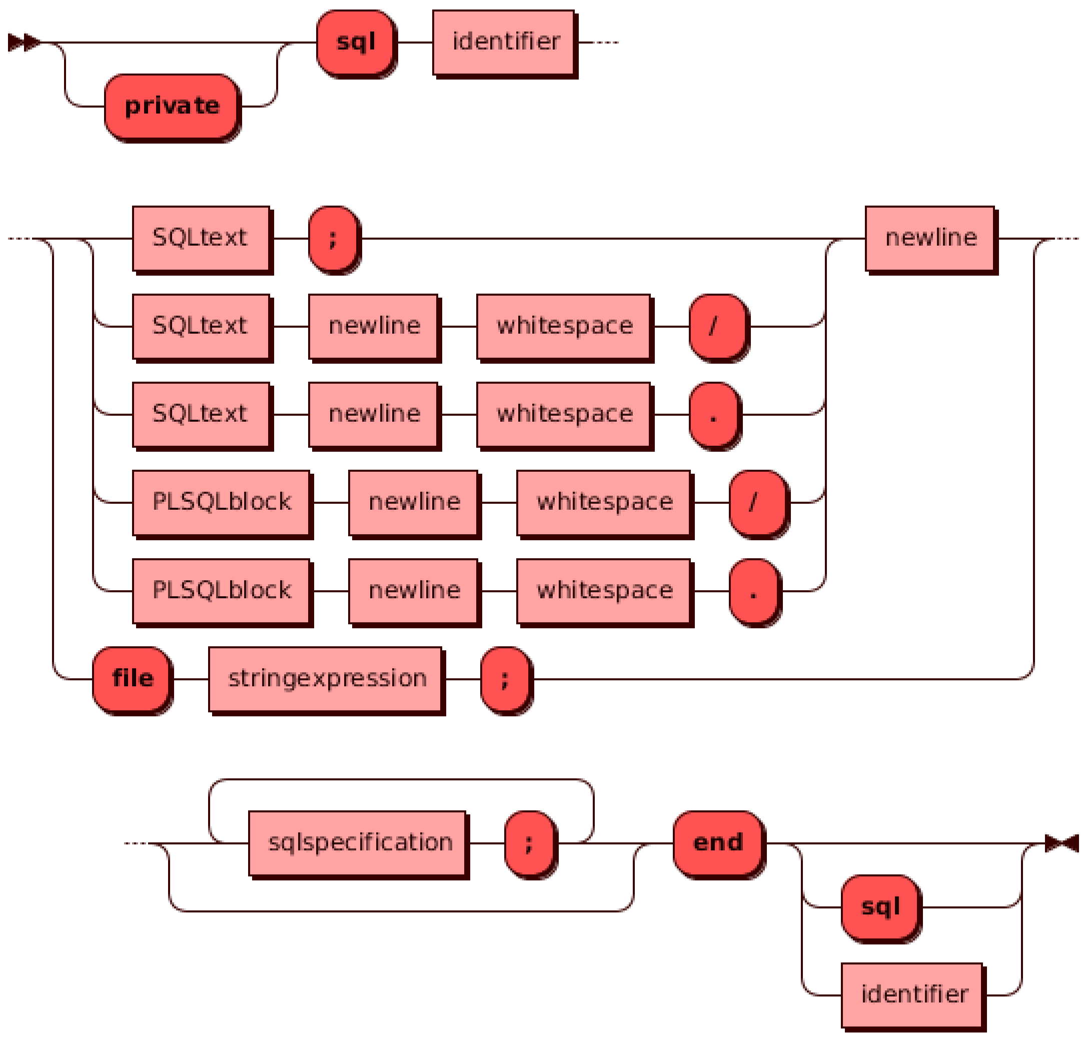
Some example variable declarations:

```
integer a := 27; b;
# declare two integers, the first will be initialiazed

double threads sum total=:0;
# declare a double, which is initialized and
# which will be summed when workers finish

string(30) filename := "output.txt";
# declare a string and assign it the value "output.txt"
```

SQL declaration



SQL statements (queries, dml, ddl or PL/SQL blocks) are declared with the sql keyword, a name, the actual SQL text or a file name containing it and possibly a list of specifications including bind and define variables; the keyword end (optionally followed by the name or the keyword sql) terminates the declaration. Note that no parsing of the actual SQL text is being done by rwloadsim, everything is taken verbatim between identifier and the end-marker or read from the specified file. In case when the SQL really is a block of PL/SQL, the end-marker is either a / or a . on a separate line of input, ordinary SQL can also have a line with ; as the last character as the end-marker. This is like sqlplus, except that empty lines do not serve as terminator in rwloadsim. The end-marker / or . can have preceding white-space e.g. as indentation. Note that in order to be recognized as SQL text, one of the following keywords must be found:

```
select, insert, update, delete, merge, truncate, create, drop, alter, /*
```

In order to be recognized as PL/SQL, one of the following keywords must be found:

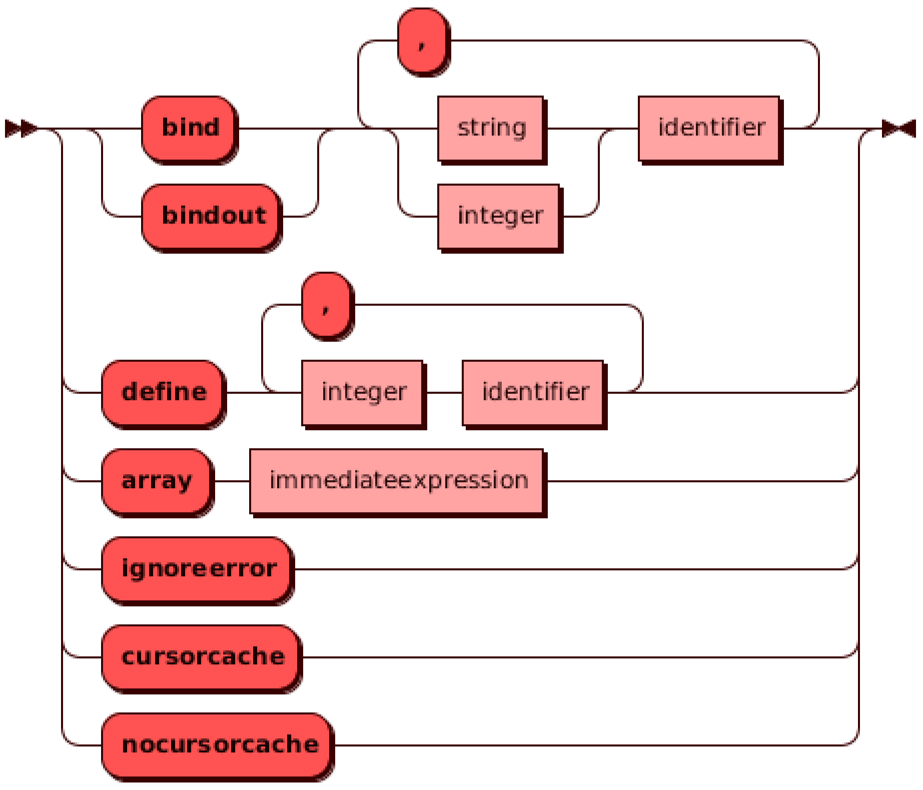
```
begin, declare, call, --
```

If you e.g. need to execute "grant select on abc to def" as a SQL statement, you need to prefix it with a /* */ style comment as the keyword "grant" is not recognized as the beginning of a SQL statement. While rwloadsim is generally case-sensitive, the keywords to initialize SQL or PL/SQL are not.

Note that rwloadsim really doesn't "know" anything about SQL or PL/SQL: The two lists of keywords lists simply imply rwloadsim will start scanning until the appropriate end-marker is found. Anything found between either of the shown keywords and the end-marker is given as the stmttext argument to OCIStmtPrepare2().

If you use the keyword “file” followed by a string expression (typically just a string constant), the file is opened and its contents is read as the complete text of the SQL statement. You should not include any terminators in the file, and the file does e.g. not need to be terminated by a newline. The only interpretation of the file done is the lookup for any of the recognized P/SQL keywords, and if found, rwloadsim assumes the contents of the file is PL/SQL. Note that the file is read at compile time, which means the mechanism cannot be used as a way for rwloadsim to handle dynamically (runtime) generated SQL. It is suggested to use the suffix .rws for such files containing SQL text. You can use this possibility if you have very long SQL text, that you prefer not including directly in your rwloadsim source file.

For anything except ddl, you will normally need bind and/or define, etc, which is done via one or more sqlpecifications with this syntax:



These are used in the following way:

define	In a query, each select list element will be stored in a declared variable of type integer, double, string or clob. For each, you specify the position as an integer and then the name of the variable. If you don't specify define variables for all select list elements, a warning will be given at execution, but your rwloadsim program will continue.

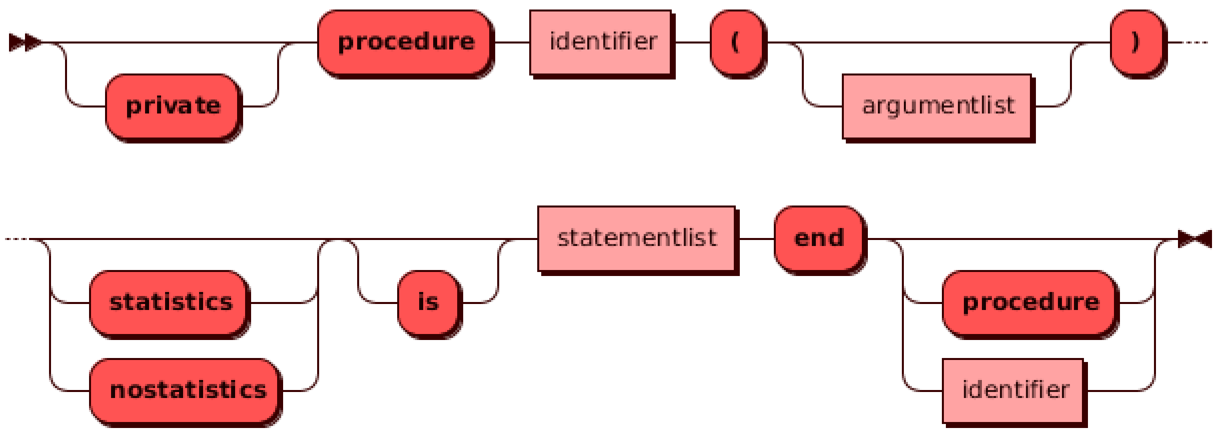
bind	For queries, dml and PL/SQL, the bind keyword is used to associate variables with placehodlers in the SQL or PL/SQL text. This can be done by position (e.g. when placeholders are named :1, :2, etc) or by name when placeholders have actual names (such as ":abc", ":def"). These binds are used for input only.
bindout	For dml having the returning clause or for PL/SQL, the bindout keyword is used to bind placeholders to variables that will retrieve values. At present, this cannot be used with the array interface.
array	For queries, this sets the prefetch buffer used by OCI to one less than the value. For dml (without the returning clause), this implictly creates an array of the specified size. For DML, when the declared sql variable is being executed, values will be copied from the ordinary variables to entries in this array; the array gets flushed and the actual SQL execution takes place when the array is full or when a commit is executed; a rollback will cause the array to be emptied. Effectively, array processing is completely transparent to the user of rwloadsim.
ignoreerror	Normally, if SQL execution causes any Oracle error, this will be printed. With this specification, error printing will not occur; this is only supported for SQL not using arrays. It can e.g. be used to avoid printing ORA-00001 errors during DML or to allow single row queries to return no rows. See the predefined variables oraerror and oraerrortext.
nocursorcache	By default, SQL statements will be cached in the cursor (statement) cache; specify nocursorcache to prevent this.

SQL statements can be declared publicly, privately or locally inside a procedure or function. The name of the former is in the same single, global namespace as all other global declarations.

Declaration

The non-scalar declarations give a name to complex objects like databases, files, procedures and functions; these can only be declared at the top level. The namespace for complex declarations is shared with the namespace for scalar declarations.

Procedure declaration



A procedure has a header and a list of executable statements followed by the end keyword. Procedures can take arguments that must be declared between parenthesis; these are local in the procedure. Procedures without arguments must include the empty parenthesis. Local variables of type integer, double, string or SQL can also be declared. Procedures are the most important building blocks of an rwloadsim program as they e.g. control how SQL statements are being executed, sets values for parameters to SQL statements (bind values), gets return values from queries, etc. Your rwloadsim program needs to be written such that procedures, potentially calling other procedures together with SQL statements, simulates the "business transactions" that your program is simulating. If you simulate an order entry system, you may e.g. have procedures simulating "create an order", "get all orders from some customer", "modify an order", "get total order value", etc.

Procedures calling SQL will by default have runtime statistics gathered and saved in the rwloadsim repository. You can prevent that from happening by providing the nostatistics keyword, this can e.g. be useful for procedures that deal with awr reporting. Similarly, you can make sure a procedure behaves as if it were calling SQL, which means it will acquire a session from a pool upon start and release it at the end, and have statistics gathered, if you provide the statistics keyword.

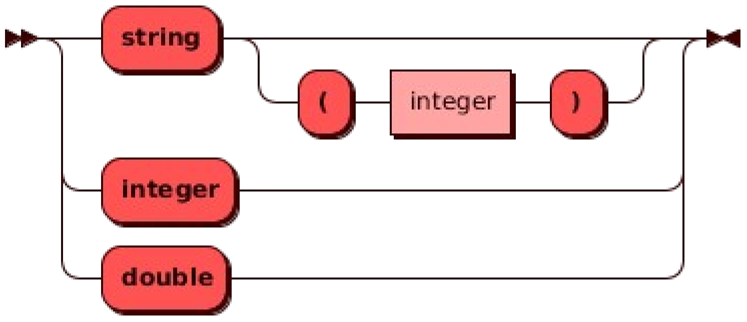
Procedures can be called recursively, although there is a fixed, maximum call depth.

If a procedure (or function) is declared with the private keyword, it is only available in the rwl source file, where it is declared.

The end keyword completing the procedure declaration can optionally by followed by the name of the procedure or by the procedure keyword.

Argument list

Procedures (and functions) can optionally take arguments of the simple data types:



Note that all arguments and all local variables declared inside a procedure or function at the same level; the only distinction is between global and local variable.

An sample procedure declaration is the following:

```
procedure isbig(double x)
  if x>1000.0 then
    println x||” is larger than one thousand”;
  else
    println x;
  end if;
end isbig;
```

Due to the scope rules of local variables, if you e.g. attempt something like

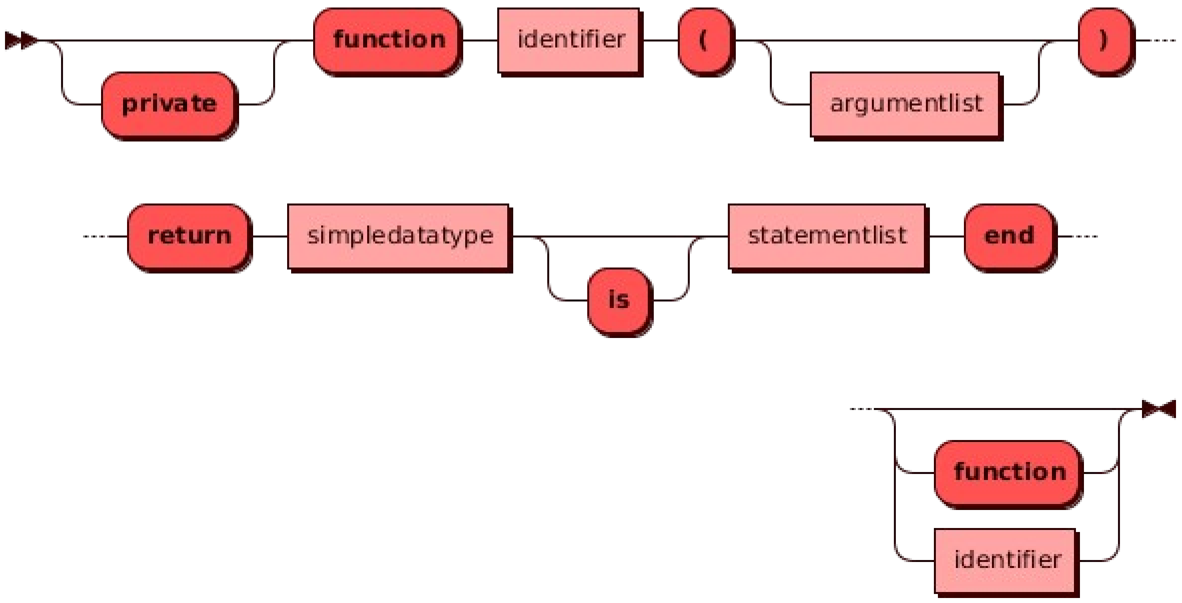
```
procedure trynestedlocal(integer x)
  if x>0 then
    integer y := 3*x; # y is local to the procedure
  else
    integer y := -3*x; # so you cannot redeclare y here
  end;
end;
```

you well get an error about re-declaration of the local variable y.

Function declaration

A function is similar to a procedure, but it has a return value of a specified type, and it is used in expressions. Just like procedures, it has a list of executable statements; there must be at least one return statement. Function arguments are declared between parenthesis and they are local to the function; the same is true about variables declared inside the function. Functions can call themselves recursively within the limits of the maximum allowed call depth. Functions do not get runtime statistics. Functions can also be declared private and just as procedures, you can have either the name or the keyword function after the terminating end.

The syntax for a function declaration is:



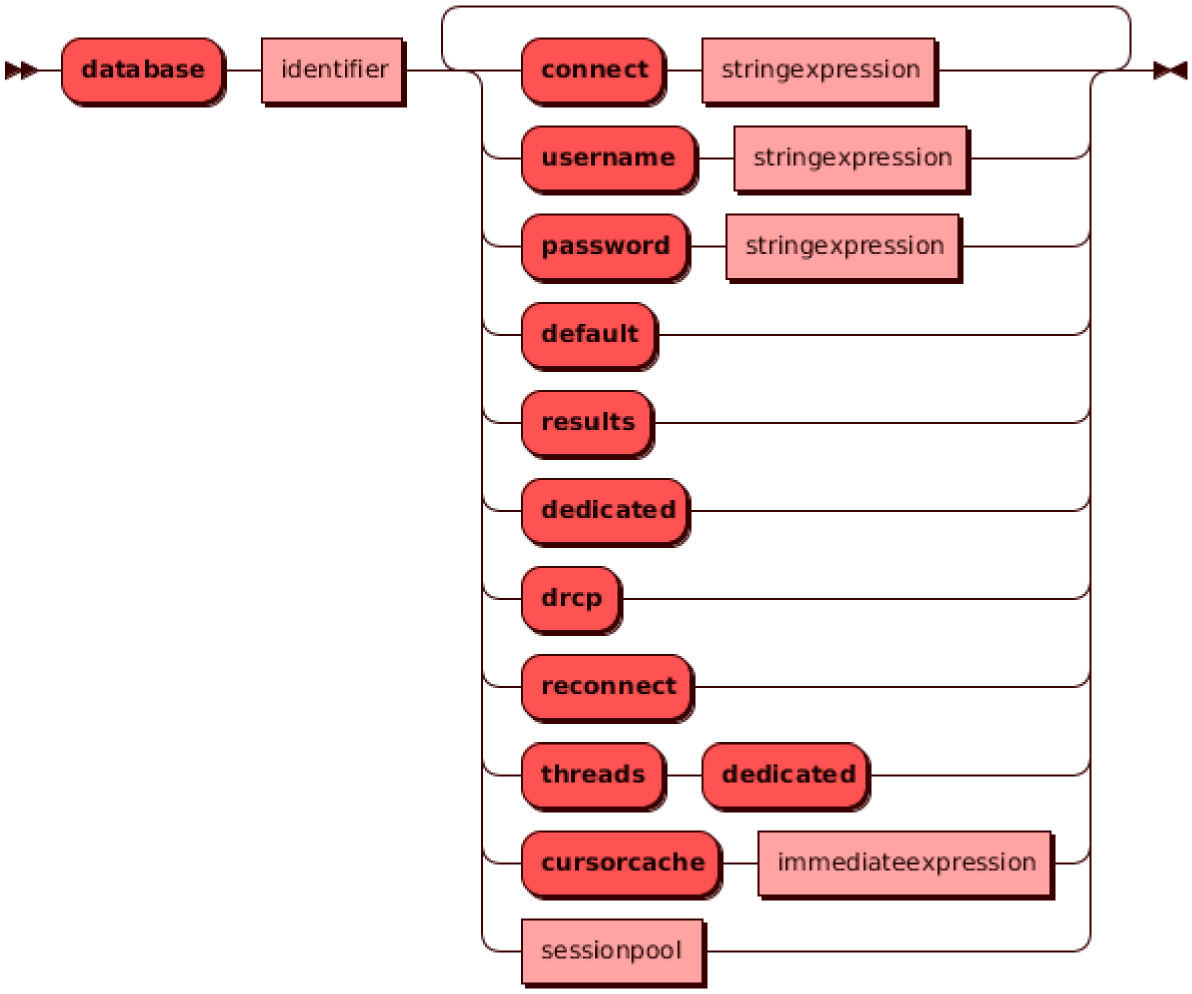
An example of a function declaration is the following:

```
function add(integer a, integer b)
return integer
is
  return a+b;
end function;
```

Database declaration

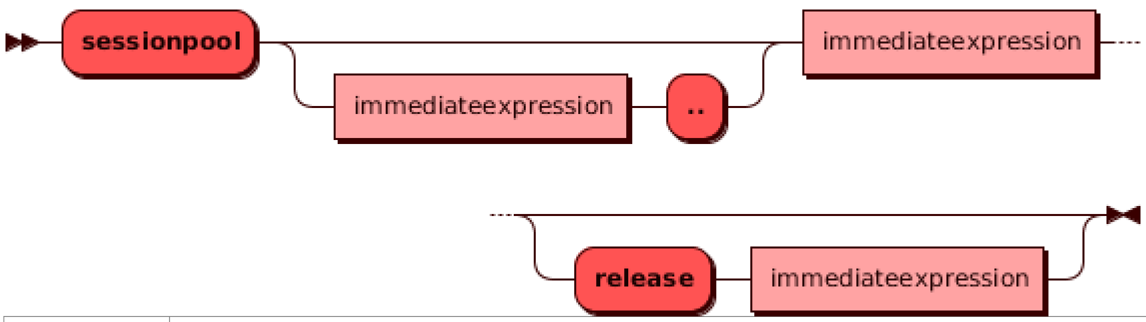
A named database declaration tells rwloadsim a database and a schema to connect to (username, password and connect string) and tells how the connection is done (e.g. dedicated or using a session pool). You can declare as many databases as needed, and you can mark one database as being the "default" database, which is used when no explicit database is named. You can similarly mark one database as the "results" database, which is used by rwloadsim stores runtime results.

The syntax for a database declaration is:



the syntax for a session pool is:

The available database specifications are:



username	The database username to connect to; this is a required specification
password	The password of the database account; if not provided, rwloadsim will prompt for it.
connect	Connect string to the database; this is optional
default	Mark this database as the default database; in simple cases, a default database will be fine, otherwise it is recommended only to use explicitly named databases.
results	Mark this database as the results database; a results database is required for rwload to save any runtime statistics.
dedicated	The connection will be established once during startup and kept until rwloadsim finishes. Worker threads will similarly create their own connection, which is kept throughout the execution of the thread.
sessionpool	Create a session pool with the specified number of entries; if two values are specified with `..`, a dynamic pool with those values as minimum and maximum poolsize will be created. Note that due to bug 26568177, rwloadsim will always set the mininum and maximum values to differ by at least 1 when running in an Oracle release 12 environment or earlier.
release	Session pool with different minimum and maximum poolsize will disconnect sessions from the database after some time. By default this is 60 seconds; this time can be set differently using the release option.
cursorcache	Set the size of the cursor (statement) cache, default is 20. Note that the cursor cache is using the OCI implementation for a statement cache.
drcp	Create a session pool using database side pooling with DRCP. You also need to specify a connect string including ":pooled" such as "//machine/database:pooled".
reconnect	Create a connection that is only held while actual processing takes place; a complete database logon/logoff will be done for every execution. You should only consider using this when dealing with long-running SQL.

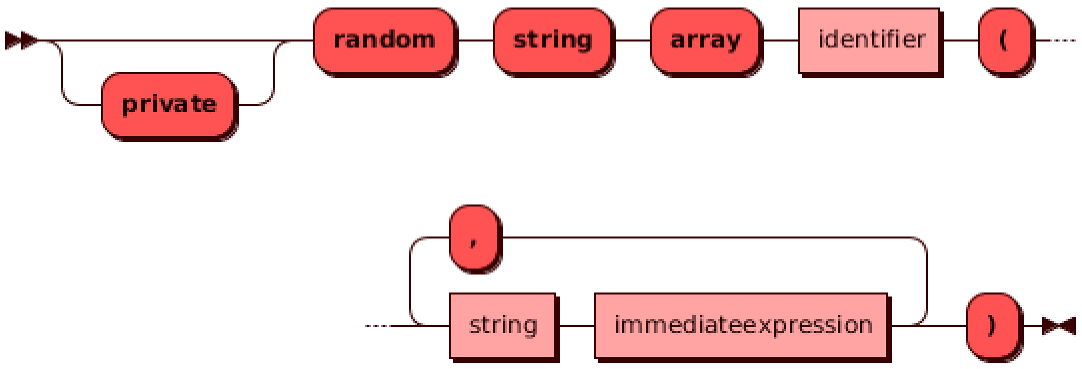
threads dedicated	Use the reconnect method for direct execution in main, while threads will get a dedicated connection. This is useful to save database connections in multi-process runs with only few threads in each process, as no connection is held in main.

For databases that include some type of pooling (sessionpool, reconnect or drcp), session acquire and release takes place at the beginning and end of the first procedure containing SQL, or the first procedure with an explicit statistics attribute. If “release database” has been executed before or while a session is acquired from a session pool, the subsequent release of that session to the pool will additionally also disconnect the (released) session from the database. This typically implies a new database connection will be created when some procedure later acquires a session.

The RWP*Load Simulator uses several methods to prevent database connection storms from happening. A connection storm is characterized by an attempt to create many database connections almost simultaneously. If you are using variable sized session pools, are using the reconnect method or extensively use “release database”, the risk of running into connection storms will be considerably higher than with any other connection methods.

Databases cannot be declared private or local in a procedure or function.

Random string declaration



A random string array is a variable that will return a different value each time it is used in an expression. It contains a fixed number of entries, each having a weight and a string constant. Weights cannot be negative, their sum must be positive, and rwloadsim will scale them to probabilities in the range [0;1]. Here is a short example:

```
random string array maybe ( "yes" 20, "no" 70, "maybe" 10);
println maybe, maybe, maybe, maybe, maybe, maybe, maybe;
```

This declares the variable "maybe" as a random string array with three entries. Whenever "maybe" is used in an expression, one of the three entries will be returned, and each entry will be returned with a certain probability. In this case, the weights are given as 20, 70 and 10, and because the sum of these is 100, these values effectively are percentages. A few executions of this might give:

```
no no maybe no no no no
no yes no no no no no
no no yes maybe no yes yes
```

The primary use of this is to generate data for data loading; see the file ovid2.rwl in the demo directory as an example. Note that you cannot use a random string array as a bind parameter.

Random procedure declaration

Random procedure arrays are similar to random string array. When a named random procedure is being executed, one of the procedures in the array will be executed with a certain probability. The primary use of this is to simulate a workload that is a mix of certain business transactions. Take this example:

```
procedure one(integer x) ... end;
procedure two(integer y) ... end;
procedure three(integer z) ... end;
procedure four(integer x) ... end;
random procedure array doit ( one 10, two 25, three 5, four 60 );
```

As the sum of the four values is 100, the values are effectively percentages. The net result is that whenever your rwloadsim program executes "doit(value)", the actual execution will with the specified probability be one of those four. As with random strings, rwloadsim will scale the weights to probabilities in the range [0;1].

Although you don't actually declare the arguments in a random procedure array declaration, they do take arguments just like a normal procedure, and all procedures used in its definition must take the same arguments (argument *names* can be different.)

File declaration

A file represents an actual operating system file; it can be opened for write (truncate) or for append, and it can be written to. Files are opened by assigning a string expression to it, the expression contains the file name; files are closed by assigning null to them. At present, you cannot use a file for reading. If the first two characters of the string assigned to a file are ">>" a file with the name starting at the third character will be opened for append. If the first character of the string assigned to a file is "|", the rest of the string will be created as a pipeline via the popen(3) system call. Note that files cannot be used as procedure/function arguments and cannot be declared locally.

A simple example of using a file is:

```
file hello;
hello := “hello.txt”;
writeline hello, “Hello, World”;
hello := null;
```

which will create a file named “hello.txt” and write the text to it.

Expression

Expressions are similar to expressions in other programming languages and follow standard operator precedence rules. They can be used in several different context such as assignments, as booleans in if/then, etc. When evaluating expressions, the three simple scalar types (integer, double, string) can be intermixed freely, however all expressions have a "dominant" type depending on context. If the expression is used as assignment, the type of the variable being assigned to will be the dominant type. In non-assignment cases (such as print), a single double constant or variable in the expression will cause double to be the dominant type.

If an integer needs to be converted to a string, the printf format "%ld" will be used as it is internally represented as a 64-bit long, for double, the printf format will be "%.2f"; the buffer used for both of these holds at least 30 characters. You can use directives to change these formats. When a string needs to be converted to an integer or double, the behavior is the same as using the standard C functions atol() or atof() respectively; effectively this means no error is returned when a string contains non-numeric data.

Boolean expressions as such do not exist, but all boolean operators return either 0 or 1; in if/then constructs and anywhere else a boolean is expected, 0 means false, any other value means true.

Most operators are dyadic, there are two prefix monadic operators, - and ! respectively taking the negative or the opposite boolean of an expression, and there are two triadic operators between/and which behaves like it does in the SQL language, and ? : which behaves as in C. There are two postfix monadic operators is null and is not null that behave as in SQL.

The three operators and, or and ? : are using short-circuit evaluation. As an example

```
printline 0 ? func1() : a and func2();
```

will not call func1() and func2() will only be called if a is non-zero.

The following operators exist (in order of precedence):

is {not} null	Check for (not) null	Monadic and postfix
- ! not	Arithmetic negative, boolean not	Monadic
* / %	Multiplication, division, and integer remainder	Dyadic
+ -	Addition, subtraction	Dyadic
< <= > >= between/and	Inequality comparisons	Dyadic (between/and triadic)
= != <>	Equality and two synonyms for non-equality	Dyadic
and	Boolean and	Dyadic
or	Boolean or	Dyadic
? :	Conditional expression	Triadic
||	String concatenation	Dyadic

Expressions can include a user declared function or one of the predefined functions:

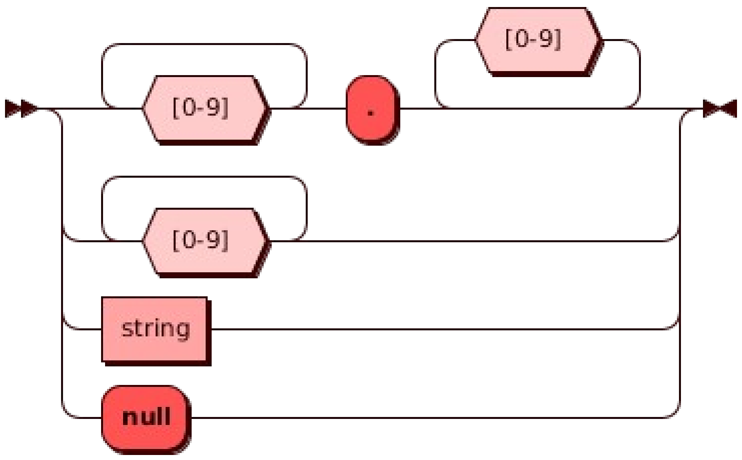
uniform(a1,a2)	Return a random number with uniform distribution in the range a1..a2. If both a1 and a2 are integer, the return value is also integer and the range of values returned will be [a1;a2]. If at least one of the arguments is a double, the return value will be double, and the range of values returned will be in the interval [a1;a2[
erlang(a)	Return a random erlang distributed value with k=1 (a.k.a. expontial distribution) and average a. This will throw a warning if used as in integer.
erlang2(a)	The same, except k=2, i.e. simulating random arrival rates. See https://en.wikipedia.org/wiki/Erlang_distribution for details
erlangk(k,a)	Return a random erlang distributed value with a specified value of k and average a. k must be at least 1 and only the integer part is used.
sqrt(a)	Returns the square root of its argument
lengthb(s)	Returns the number of bytes in its string agument.

substrb(s,p)
 	
substrb(s,p,l)
	Returns a substring of its first argument, starting at byte position p and with length l measured in bytes. Semantics is like the same functions in SQL.

Functions with empty argument lists are called as part of expressions using ().

Note that whenever the word "expression" is found in the syntax diagrams, this effectively refers to an expression that is compiled but not evaluated until at execution time. The syntactically identical "immediateexpression" and "stringexpression" are evaluated immediately during parsing of the rwload input file; the difference between the two is that the former indicates the integer or double value is used, the latter that the string representation is used.

Constants are integers, doubles, strings or `null`:



Strings are delimited by double quotes and they can span newline in which case the newline becomes part of the string. Within a string \" represents a double quote, \\ represents a single backslash, \n represents newline; no other escapes are possible. Within strings, all characters in your terminals character set are allowed; they are disallowed anywhere else.

Some examples of expressions are:

```
integer a:=2, b:=3;
integer c;

c := (a = b+1) + 10;
# + has higher precedence than =, so b+1 is calculated first as 4,
# then because 4=2 is false, the value of the parenthesis is 0,
# adding 10 gives 10, which will be assigned to c.

printline c + (a=b-1)*(a!=b);
# - has higher precedence than =, so the first parenthesis is evaluated
# as 2=2 returning 1, the second parenthesis is also true, hence 1,
# so the product is 1, adding the value of c which is 10, gives 11
# so this will be printed

printline 10.0/(!c+a*b);
# as c is 10, !c is false or 0, adding the product of a and b (2 and 3)
# the result in the parenthesis is 6.  As there is a double constant, the whole
# expression is evaluated as double, and the result of 10.0/6.0 will be printed

printline 5 between 0 and 10 and 2;
# between/and has higher precedence than and
# so 5 between 0 and 10 evaluates first giving 1
# subsequently, 1 and 2 evaluates to 1

printline 5 between (0 and 10) and 2;
# the parenthesis evaluates to to 0
# and subsequently 5 between 0 and 2 evaluates to 0

double d;
d := 1 + "2.9"; # will assign 3.9 to d as the dominant type is double
a := 1 + "2.9"; # will assign 3 to the integer as the
# implicit conversion from string to integer stops at "."
```

Identifier and their scope

Identifiers are used to name variables, procedures, etc, and all identifiers are in the same, single name space. Identifier names are case sensitive, and they consist of a letter in the range a until z followed by one or more letters, numbers or the underscore character; positional arguments are named $1, $2, etc. and $# is the count of positional arguments. There are three scopes of identifiers, in this priority:

- Local identifiers (which includes arguments) are available to the function or procedure where they are declared.
- Private identifiers are available within the rwl source file where they are declared with the private keyword.
- Public (global) identifiers are available in all rwl source files provided to rwloadsim.

If equally named identifiers are found, the one with highest priority is in use. As an example, a local identifier will be used if a private identifier of same name is also found. Within each name scope, all identifiers must be different. You can e.g. not declare a procedure argument and a local SQL statement in that procedure with the same name. Rwloadsim will return an error or a warning if there is a conflict. Please also note that scope of local variables is the whole procedure/function and not e.g. any compound statements.

There are a number predefined identifiers – both variables and procedures – that can be used in user code, and all these are public. Predefined variables are all read-only seen from the programmer perspective.

The following predefined variables are available:

threadnumber	An integer variable that is assigned a unique number in each worker thread; numbering starts at 1. It can be used for e.g. debugging, printing, etc.
runnumber	An integer variable that uniquely identfies each run, it is maintained as a database sequence in the results database, it is valid in the main thread and in worker threads if a results database is provided.
loopnumber	An integer variable that is used internally in control loops. It is initialized to 1 when a control loop starts and is incremented by 1 for every execute in the block; it can e.g. be used for debug printing as needed.
everyuntil	A double variable that is used in control loop. For every loop, it is calculated as the expected timestamp to start the next loop.
runseconds	A double variable (syntactically it is a function without arguments), which returns the timestamp with microsecond resolution since the common start time of all control loops in threads. It is available in both the main and worker threads, and the value can be negative.
usrseconds	A double variable that contains the user time in seconds from the last call to the getrusage() procedure.
sysseconds	A double variable that contains the system time in seconds from the last call to the getrusage() procedure
oraerror	An integer variable that contains the Oracle error number of the most recently executed SQL statement. As an example, if executing a single row query that returns no rows, its contents will be 1403.
oraerrortext	A string variable that contains the Oracle error text of the most recently executed SQL statement
stderr	A file variable that can be used in write/writeline statements to write output to stderr.
stdout	A file variable that refers to stdout. Mostly useful with fflush when using the output from rwloadsim in a pipeline.

$1, $2, etc	String variables that are made available as positional arguments when e.g. the -A option is used.
$#	Integer variable containing the number of positional arguments. Reduced by 1 for every call to shift.

The following pre-defined procedures are available:

getrusage()	A call to the getrusage procedure will execute getrusage() with RUSAGE_SELF and as a result, will put the user and system time as seconds into the two variables usrseconds and sysseconds. Note that irrespective of where getrusage is called, whether in a thread or in the main program, the values returned will be the total from all threads executing. Also note that the resolution is the same as the resolution in getrusage(), which typically is 1.0 ms on Linux.
wait(double t)	The wait procedure suspends execution for the number of seconds specified as the double argument. The resolution is O/S dependent, typically 1ms or shorter. Note that if a database session is held, it will *not* be released during the wait.

Direct execution vs. compilation

The RWP*Load Simulator includes parsing and execution just like any ordinary programming language, but these two phases are handled implicitly. When a procedure or a function is declared, the statements that it contains, which includes both actually executable code (such as assignments) and declarations of simple scalars and SQL, are compiled and saved internally (in memory) for later execution. At the top level, statements can also be executed directly during parsing of your input files. If you e.g. provide this as input to rwloadsim:

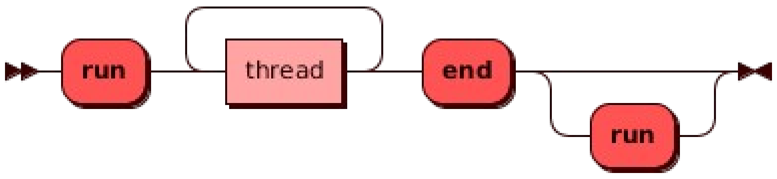
```
integer a;
a := 10;
if a<10 then
  println "as is less than ten";
end if;
```

the two first statements, the declaration of the integer variable a and the assignment to it are directly executed as they are parsed. The if/then/end code, however, is compiled into an anonymous procedure on the fly, and it is subsequently executed. Internally this snippet includes both compilation and execution, while user sees it as one. An implication of this is that if input to rwloadsim has an error, e.g. a syntax error, everything until that error will be compiled and potentially executed.

You can use the -e option to rwloadsim if you only want to compile to see any potential syntax errors.

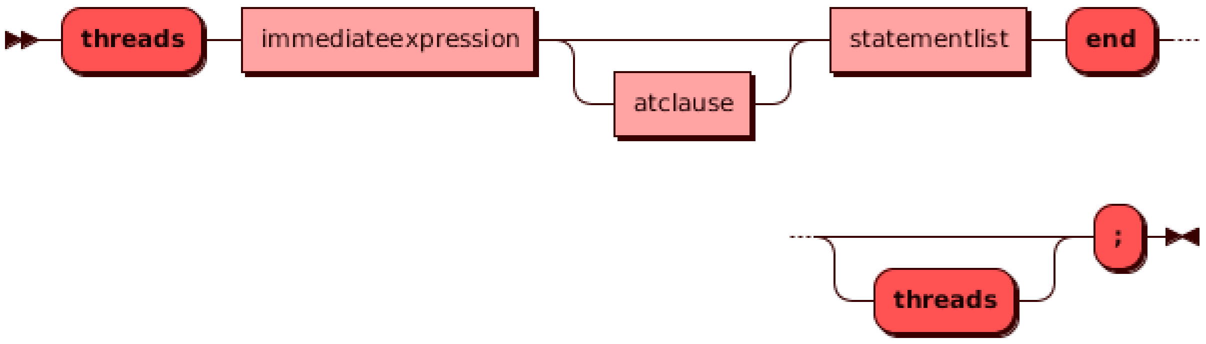
Thread execution

A thread execution or "a run" is the primary mechanism to simulate database load, and it depends



heavily of using control loops. Each thread executes a list of statements very much like a structured statement or direct execution of a declared procedure or function executes a list of statements. You typically have multiple threads doing the same thing, which would simulate a number of workers in a real application environment. The complete set of threads are described between the keywords run and end and each of potentially multiple identical threads are described between the threads and end keywords. The control loop has a mechanism is included to make sure all threads start at approximately the same time. Each rwloadsim execution is identified by a number, which is retrieved from a database sequence from the database marked with the results keyword, that number is available as the predefined variable runnumber and is stored in various results tables in the column runnumber.

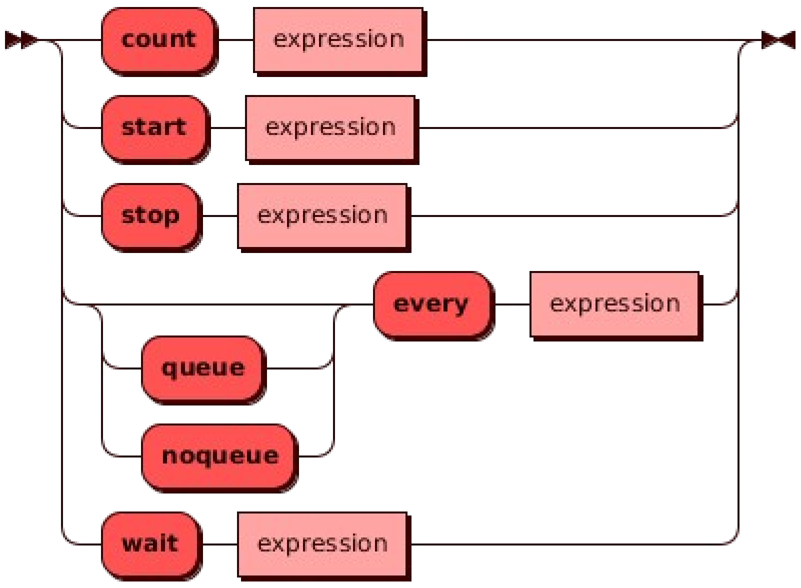
Between the run and end keywords, you specify one or more threads, multiple threads are used to simulate concurrent execution of different things. If your workload is simple and only consists of many concurrent executes of the same thing, only one thread specification is needed. During runtime, each of these threads actually execute as one or more worker threads, each executing the same code. With the addition of the count of threads, the syntax and semantics of a thread is almost identical to the syntax and semantics of an execution block as it contains a list of statements that are executed. The execution will either be against the default database or against an explicitly named database. The syntax for thread is:



and the two options are:

immediateexpression	This should evaluate to a non-negative integer, and the value will be the number of actual worker threads started. They will all perform the same work. A value of zero can be used to keep the code in your rwl file while not actually executing anything. If the threads are supposed to simulate some actual workload, it should somehow include a control loop, either as one of the statements in the list, or in a procedure being called.
optional atclause	All database work done by the worker threads started will using the same database. The database to be used can either be explicitly named after the at keyword, or it will be the database declared with default. If the chosen database has the dedicated attribute, logon and logoff will be done once at start respectively finish of each worker thread. If the database has the sessionpool or drcp attribute, a session will be acquired and released as needed.

When a thread runs, the statements will execute in sequence, and if a control loop is executed (either directly as one of the statements between threads and end or indirectly in a procedure being called) that control loop will control execution via one or more of the control options:



start	Start execution at this timestamp, the default is 0s and negative values are allowed in which case the thread will start before the normal thread start time as explained below.
stop	Stop execution at this timestamp; either count or stop must be provided. Note that the stop time is calculated when the loop *starts* and *not* recalculated during loop execution. This could be considered a bug and the behavior may change in a future release.
count	Execute the loop that may times; either count or stop must be provided.
wait	Wait this many seconds after each execution; the default is not to wait, and you cannot specify both wait and every.
every	Attempt execution every so often. This is different from a wait time, as every sets the time between each start of the procedure execution, so every simulates an expected arrival rate in a queuing system. Before an execution, the point in time to start the next execution will be calculated, either relatively to start time of the current loop or relatively to the original common start time. If a planned start time is surpassed, the execution will start immediately. To simulate queuing systems, use every with an argument of erlang2(1/x), where x is the expected arrival rate per second. The keyword can be prefixed with either “queue” or “noqueue” which will enable or disable queuing simulation using a backlog. Either of these overwrites the setting done via the -Q or -N options to rwloadsim.

All control loops (in all threads) have their timings coordinated at (approximately) the same time, and unless the start timestamp is explicitly set to non-zero, this means starting at the same time. The actual timestamp with reference to this common start time is returned by the predefined variable runseconds. In order to ensure reasonable ramp-up to e.g. start database connections in session pools, or to actually start operating system threads, rwloadsim has a built-in delay from the time it is called until this 0s timestamp when control loops really start processing; the default is 5s. If you use the value of runseconds before this common control loop start time, it will have a negative value. Multiple control loops can very well be executed (by the same thread) after each other, but they *cannot* be nested, neither directly nor indirectly. There is no requirement to use a control loop, but if you do not, execution will start as soon as the thread is created. To execute something just once at a controlled time, you therefore need to use a control loop with count set to one.

When a pooled database is in use, a session will be acquired from the pool before each execute of a procedure (or function) doing database work, and it will be returned to the pool when the procedure (or function) exits.

Some examples will show how this works:

```
procedure abc() ... end;
run
  threads 10
```

```
    loop stop 60; abc(); end;
  end threads;
end run;
```

Start ten worker threads, each having a control loop that, starting at (approximately) the same time, doing nothing but calling the procedure abc() in a busy loop and stop after 60s. Assuming the procedure abc() contains SQL and assuming a default database has been declared with a sessionpool, each call to abc() will acquire a session from the pool upon entry and release it upon exit.

```
procedure abc() ... end;
procedure def() ... end;
procedure xyz() ... end;

random procedure array doit (abc 20, def 80);
run
  threads 10 at mydb
    loop every erlang(0.1) stop 300; doit(); end;
  end;
  threads 1
    loop start 10 count 2; xyz(); end;
  end;
end;
```

Start 10 worker threads that will start start at (approximately) the same time, each will simulate a random arrival rate of 10 per second, in 20% of the cases will execute "abc", and 80% of the cases will execute "def", stop after 5 minutes. Assuming both abc() and def() execute SQL, they will acquire and release sessions from the named database, mydb. Start another single thread, that with a delay of 10 seconds will execute the procedure "xyz" twice.

```
integer exectime := 120;
integer onetwothree := 2;

run
  threads 10
    loop start threadnumber*0.1 every 1 stop exectime; something(); end;
  end;
  threads onetwothree # start threads that will execute three different things
    loop count 10; one() end;
    loop start 30 count 5; two() end;
    loop start 50 count 1; three();
  end;
  threads 1 at system # start 1 thread at a named database
    wait 10-runseconds;
    begawr();
    wait exectime-runseconds-10;
    endawr();
  end;
end;
```

This example shows several useful features:

- The first ten worker threads will ramp up in their start, as each control loop has a starttime which is 0.1s times the actual thread number.
- The first ten threads will execute the procedure "something" once every second, and it will

stop after 120s (the value of the exectime variable). If it contains SQL, the procedure something() will acquire and release database sessions.

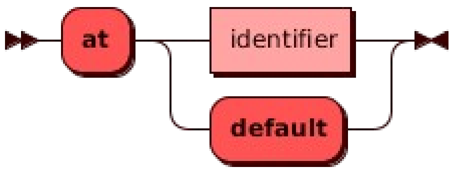
- The next two threads will first execute "one" ten times, then 30s after the starttime execute "two" five times, and then 50s after the starttime execute "three" once. Note that all start (and stop) times are relative to the initial starttime, i.e. the (approximately) common starttime for all threads.
- If any start time is passed, start will be immediate. If – as an example – in this case each execution of "two" takes 3s, the total time to execute it ten times is 30s, which means the execution in the third threadstatement of "three" will be delayed.
- The last single thread will be using the named, non-default database, 10s after the initial starttime the procedure "begawr()" will be executed and 10s before the finish of the ten first threads the procedure "endawr()" will be executed.

The overall rwloadsim program continues when all threads have finished. Although it is technically possible, you are strongly recommended to not have more that one run command.

If a procedure is long-running, it may potentially finish after the stop time of a thread. In such a case, the procedure will *not* be interrupted, so actual thread finish time may be *after* the stop time.

Selecting a database using the at clause

The at clause, which is found several places in the syntax is used to specify which of the declared databases is used for execution:



An important aspect of this to understand when the connect and disconnect takes place. In the typical cases that are described in this document, the choice of database is determined at *compile* time before actual execution of code takes place. Most simulation runs will have one run command often with multiple threads that each at some point in time execute a control loop. Whenever a procedure with some SQL in it (normally inside a control loop) is executed, it will ensure there is a database session upon start, and potentially release it upon exit. If the database is dedicated the database logon and logoff is already done (at the start/end of rwloadsim or a thread), so ensuring a database is effectively a no-op. In the other cases, an actual database session will be acquired, either through a complete logon, or from a session pool or DRCP pool. This mechanism is an important part of rwloadsim, and it is also what is behind the statistics gathering, where both the time to get a database and time to actually perform database work is registered.

This mechanism applies in the following cases of the use of an at clause *outside* a declared procedure or function, i.e. *directly* at the “top level” of your rwloadsim code:

`procedurecall() at database;`
 	
`sqlstatement at database;`
 	
`execute at database; … end;`
	When any of these is found directly in your main program, a database session will be acquired, the code will be executed and the session will be released. In case the named database is marked dedicated, a database session will already exist for the entire duration of execution of rwloadsim.

`run`
`  threads N at`
`  database;`
`  …`
`  end;`
`end;`
	When threads are started and an explicit database is named, the entire thread will use that database. If the database is declared dedicated (or thread dedicated), an actual logon will be performed when the thread starts and logoff will be performed when it terminates. Otherwise (i.e. with any pooling), whenever the first procedure executing SQL (or with statistics attribute) is called, that procedure will acquire and release a database session at start and exit.

Note that any rollback or commit executed is *always* against the database session that is in use as a result of this mechanism; this implies two phase commit it *not* supported.

If you are using the at clause at any other place (typically inside a declared procedure), the behavior is somewhat different. In such cases, any existing database session will be “stacked”, and a new database session will be acquired from a session pool as a completely new database connection for the duration of the SQL, procedure call or execution block. The session will be released or a disconnect will be done after finishing the SQL, procedure call or execution block. As sessions are acquired and released, only databases that have real client side pooling (session pool or reconnect) can be used. Note that only queries can be executed using this approach.

If you use `at default` *inside* a procedure, the effect is to use the database that was chosen at compile time for the top level procedure or thread being executed; this is *not* necessarily the database that has the default attribute. Take this example:

```
procedure copytodest()
  sql selsource select … end;
  sql insdest insert … end;

  for selsource at sourcedb loop
    insdest at default; # destdb would imply a new session!
  end loop;
  commit; # always against the effective database in use
end;
copytodest() at destdb;
```

When the procedure copytodest() is being called at the last line above, the named database (destdb) is being used during the call. Inside the procedure, a cursor loop is being executed selecting rows from the named database (sourcedb), and for each row returned, the SQL insdest is executed against the actual database session used by the procedure. This is the same database later used when commit is executed. Effectively, rows are copied from one database to another. Any array interface – for both the query and the insert – will be used.

Note that two-phase commit is not supported.

The exact semantics of these “stacked at-clauses” may change in a later release.

Error handling

Syntax errors during parsing or executions errors will be printed to stderr with an RWL-nnn error number, file name, line number and error text, much like ORA-nnnnn errors. Syntax errors during parse will prevent later execution. There are three categories of errors: warnings, errors and critical errors, plus a special RWL-600 internal error which is used by rwloadsim itself to report abnormal situations. If you receive an RWL-600 error, please report it providing as much evidence as possible. If executing database calls results in some ORA-nnnnn error, that will also be reported and – if available – the error location in the SQL text is also identified. A few examples are shown by this rwl program (here shown with line numbers)

```
 1  integer a, b, c; b := 3;
 2  procedure ifnull()
 3    if a then
 4      printline "a is not zero";
 5    end;
 6  end;
 7  execute ifnull();
 8  c := ; # wrong syntax
 9  c := 0;
10  printline b/c;
```

If you save the above (taking out the line numbers) in a file called errors.rwl, and execute it, you will see following output:

```
RWL-064: warning at [errors.rwl;3]<-[errors.rwl;7]: executing if with null
argument - false assumed
RWL-008: error at [errors.rwl;8]: expected valid expression
RWL-022: error at [errors.rwl;10]: attempted division by zero
3
```

The RWL-064 error tells that at line 7 in errors.rwl, you called a procedure which then at line 3 executed an if statement with a null argument. The RWL-008 error is a syntax error found at line 8, where the right side of the assignment is missing. Finally, when executing line 10, a division by zero was attempted. Note that in all cases, execution actually continues, even in the division by zero case.

The full list of errors will be added to this document at a later time.

Saving runtime statistics

As a primary purpose of the RWL*Load Simuator is to simulate load, it is important to be able to gather execution statistics such as average execution times, execution counts, etc. If you e.g. run a simulation, simulating an order entry system, you may want to see the average and maximum time taken to execute e.g. an order entry, an order query, etc. Rwloadsim can save up to three different types of statistics.

Overview of execution time gathering

Your simulation is likely to include one or typically more procedures that each simulate a certain part of an application. Assume you have two procedures named "insorder()" and "selorder()" respectively, that each are implemented by executing a set of SQL statements. You may further have a random procedure array, "dosomething()" that calls either of these two with a certain probability. The run command may then do little more than start a number of threads, all running for two minutes, and all calling "dosomething" every so often using a control loop as in:

```
run
  threads 20 at mypool
    loop every 0.1s stop 120;
      dosomething();
    end loop;
  end threads;
end run;
```

This will start 20 threads that all run for 120s and all call either "insorder" or "selorder" every 0.1s.

Whenever a procedure that contains SQL is executed, the procedure will be timed by rwloadsim, and the control loop will therefore effectively perform these steps:

```
loop
   sleep
   get a session from the session pool
   execute the statement list in the procedure
   release the session back to the session pool
until end condition is met
```

The sleep time depends on the every or wait option of the control loop, and while in the sleep, the thread does not have a database session. In each loop, rwloadsim will save three timestamps: t1) Just before getting a session, t2) Just before executing the procedure, t3) just before releasing the session. The difference between t2 and t1 is the time waiting for an available session, and the difference between t3 and t2 is the time actually spent doing database work. These two differences will be summed, and will together with the total execution count be saved in a database table. Additionally, an execution time histogram, and a count of executions per second can be saved.

Results tables

The following four tables are used to save execution time results. They are created when running the rwloadsim.sql file in the rwloadsim repository schema.

Rwlrun

```
create table rwlrun
( runnumber number not null
, key varchar2(30)
, komment varchar2(100)
, rdate date not null
, hostname varchar2(64)
, constraint rwlrun_pk primary key(runnumber)
)
```

Every time you execute rwloadsim, a sequence number (also created in rwloadsim.sql) will be retrieved, and this will become the primary key of the rwlrun table in the "runnumber" column; the same number is available as an internal variable. The "key" and "komment" columns are not used by rwloadsim, but values can be provided at the command line. The purpose of the key column is to allow grouping of similar runs. The execution time (the common control loop start time as a date) will be saved in the "rdate" column. The "hostname" column will be filled with the hostname of the system where rwloadsim is running (with the -P option). The purpose of the "hostname" column is similar to the purpose of the "key" and "komment" column to allow for various aggregates of results or simply to distinguish results when one repository is used for several environments. If “key” is not provided, the current timestamp will be used as a default; the default for “komment” is null.

Runres

A run (everything between the `run` and the `end` keywords) will typically consist of execution of several different procedures that each simulate a specific operation such as "order entry", "query customer", etc. Statistics (time waiting for a session and for actual execution) are gathered for each of these individually, and the overall results are stored in the table "runres". Its primary key contains the runnumber, the name of the procedure being timed (“vname”), and a process number (“procno”). In single process mode (see multiprocess below), “procno” is always 0. The “wtime” column is the sum of all wait times for a session, “etime” is the sum of all execution times, “ecount” is the number of executions, and “tcount” is the number of threads that were executing.

```
create table runres
( runnumber number not null
, procno   number not null
, vname    varchar2(30) not null
, wtime    number(*,6)
, etime    number(*,6)
, ecount   number
, tcount   number
, constraint runres_pk primary key(runnumber, procno, vname)
)
```

Histogram

If you also gather histograms, the following table will be filled

```
create table histogram
( runnumber number not null
, procno   number not null
, vname    varchar2(30) not null
, buckno   number not null
, bucktim  number virtual -- effectively 2^(buckno-19)
, bcount   number
, ttime    number (*,6)
, constraint histogram_pk primary key(runnumber, procno, vname, buckno)
)
```

The effective total execution time (the sum of the wait for session time and the actual execution time) is grouped into histogram buckets much like the wait event histograms of an awr report. Each bucket is twice as long as the previous one, and the end value of the range in a bucket is $2^{(buckno-19)}$, so the range for a bucket is [$2^{(buckno-20)}$;$2^{(buckno-19)}$[. The primary key of the table consists of the same three columns as in runres plus the “buckno” column. The “bcount” column contains the number of executions in that bucket, and the “ttime” column contains the total time for all executions in that bucket.

The histogram table can be used for analysis such as execution time fractiles, etc. The rwlviews.sql file creates such two views.

Persec

In cases where the simulation processes one or a few types of short-running procedures, an often wanted result is a graph over some run period, showing the throughput every second. This information can be calculated by rwloadsim and it will be stored in this table:

```
create table persec
( runnumber number not null
, procno    number not null
, vname     varchar2(30) not null
, second    number not null
, scount    number
, constraint persec_pk primary key(runnumber, procno, vname, second)
)
```

The primary key consists of the same three columns as in the runres and history tables, plus the column "second", which is simply the number of seconds since start. The "scount" column stores the number of executions of that particular procedure within the interval [second-1;second[.

By default, the per second statistics are not saved until the run completes, which means you cannot use queries against the repository database to show e.g. a progress display. If you use the -Z (--flush-stop) and possibly -U (--flush-every) options flushing of the per second statistics will take place continuously at regular intervals. Using the first of these options implies per second statistics will be flushed every second until the clock has reached the (whole) number of seconds provided by the option. The second of these options specifies a different (whole) number of seconds between flushes. Note that flushing per second statistics has an *overhead* as the internal counters need to be protected by a mutex. Also note, that due to timing issues, it is not guaranteed that the regular flush will be absolutely correct; the correct statistics will eventually by flushed and are available after completion of rwloadsim.

Statistics with stacked at-clauses

If you are using the at-clause inside a procedure to perform some queries against a database differently from the database chosen at the top level, statistics gathering may be impacted. If you e.g. do something like:

```
database db1 … sessionpool … ;
database db2 … sessionpool … ;
procedure top()
  sql sql1 … end;
  sql sql2 … end;
  …
  sql1;
  sql2 at db2;
…
end;

run
  threads 2 at db1
    loop every 0.1 stop 60; top(); end loop;
  end threads;
end run;
```

the control loop in the two threads will call top() every 0.1s and each call with be timed for the time taken to acquire a session from db1 and to execute the code in the procedure. However, as the call to sql2 *includes* an actual session acquire/release from db2, the time registered for executing top() will not only be the time taken by the actual SQL statements (such as sql1), but also the time taken for the full processing of sql2 *including* session acquire/release for db2.

RWP*Load Simulator options

The RWP*Load Simulator is one executable, that is executed in the following way:

```
rwloadsim [options] inputfile ...
```

Options can be specified in the classic getopt(3) format with a hyphen and a single option letter potentially followed by the option value or using the GNU extension long options with double hypens. All options are shown in the table below

Short option letter	**Long option name**	**Option argument when applicable**	**Description**
-h	--help		Print short help
-v	--version		Make the banner text include the client version
-q	--quiet		Be quiet, no connect messages, and some warnings muted
-s	--statistics		Gather and save execution statistics, this requires a results database
-ss	--histograms		Also gather execution time histograms
-sss	--persecond		Also gather per second execution counts
-Z	--flush-stop	N	Flush the per second statistics every second until N seconds have passed. You would normally want to set N to the actual stop time of your longest running thread.
-U	--flush-every	N	If the previous option is in use, flush every N seconds in stead of every 1 seconds.
-k	--key	key	If saving statistics, save this key value as well; can be used to group runs
-K	--comment	comment	If saving statistics, save this comment as well; use is fully user dependent.
-c	--clockstart --startseconds	N.N	Sets the control loop start time to this many seconds after program start

Short option letter	**Long option name**	**Option argument when applicable**	**Description**
-P	--prepare	file	Prepare a multi-process execution; this writes a multi-process argument to the named file; this requires the -s option and a results database.
-R	--multirun	file	Execute a multi-process run by reading the contents of the file created by the prepare execution. All concurrently running rwloadsim processes must use the same file.
-M	--multirun-value	mparg	Alternative way to execute a multi-process run, by providing the *contents* of the file created by the prepare run. The contents will always be a single string without blanks, so it can easily be used in shell scripts using ssh.
-p	--procno	N	The value of procno stored in the results tables. Default is 0 for single process runs, process id for multi process runs
-i	--integer	intspec	Change a default value for some integer, intspec must be variablename:=integer
-d	--double	dblspec	The same for a double variable
-D	--debug	X	Set debug bits, mostly used for program debugging; the argument is a sequence of hexadecimal digits optionally preceded by 0x or 0X. Bit 0x8 is potentially useful to ordinary users; it will make parse errors generated by bison(1) more verbose.
-a	--array-size	N	Set the default array size for cursor for loops (fetch loops)
-C	--codesize	N	Set the size of the code array
-I	--namecount	N	Set the size of the identifier array
-l	--default-database	u/p@c	Create a default database with the specified username, password and connect string. @ and connect string is optional. It will be a dedicated database unless the next option(s) are also used.

Short option letter	Long option name	Option argument when applicable	Description
-X	--default-max-pool	N	When a default database is created using -l option, it will be created as a session pool of size 1..N
-Y	--defaul-min-pool	N	Also set the minimum pool size
-w	--nowarn-deprecated		Do not warn about use of deprecated features
-e	--compile-only		Do not execute functions, procedures, threads and database calls (except connections). This can be used to ensure input files parses correctly.
-A	--argument-count	N	The last N arguments (after all options) are positional and will be made available as string variables named $1, $2, etc. and the total count will be available as $#.
-F	--file-count		The first N arguments (after all options) are rwl input file names, the rest will be positional arguments. This option cannot be used with -A.
-x code	--execute-code	code	Execute 'code' before reading the first file. The code can be any statement or declaration including the terminating ; or it can be a directive. If the code is a declaration with an initialization assignment, it can be used to overwrite an initialization value in an input file.
-E	--event-notify		Setup event-notification. The only actual effect of the option is to print output to stdout when events such as “service down” is received. OCISessionPool already handles these in OCI by default and the OCI_EVENTS flag is always specified during initialization.
-S -SS	--set-action --set-action-reset		Set the action (v$session.action) to the name of the procedure whenever that procedure acquires a session or starts using a dedicated connection. With -SS (or –set-action-reset) a reset of the action is done whenever the session is no longer in use; doing so adds an extra otherwise not necessary roundtrip to the database.
-Q	--queue		Simulate a real queue with the “every” control loop option. Note that this may be the default in a future release.

Short option letter	Long option name	Option argument when applicable	Description
-N	--no-queue		Traditional behavior for “every” in control loop. Currently a no-op as it is the default.
-W	--errortime		Augment all execution time errors with the value of the runseconds variable.
-T	--vi-tags	filename	Create a vi tags file from all rwl files being processed; typically do this together with --compile-only

After processing all options, the files given as arguments will be opened and processed in sequence. If you put the tradtional `--` marker between the options and the list of files, you can replace a file argument by a single argument containing '-x code' causing the same behavior as a -x option. As an example, the following:

```
rwloadsim -q -- '-x printline "hello, world";'
```

will do nothing but printing the text `hello, world`.

Note that the complete text, including both -x and the code to be executed must be contained within one single shell argument to rwloadsim.

File contents

Each file provided to rwloadsim (including implicitly using -x) must be “complete”, and you can e.g. not have a procedure header in one file and the body in a second one. So each file must contain a complete rwloadsim statement, declaration or thread execution including the terminating ;.

Providing positional arguments

You can provide positional arguments similar to how it is done in shell by using either the -A or the -F option or by separating file arguments from positional arguments by ; (which must be escaped by the shell) or -- . The -A option value is the count of positional arguments and it implies that many arguments at the *end* of the rwloadsim command line are positional rather than rwl input files to be read. Alternatively, the -F option specifies how many arguments (after all options) are taken as rwl input files; the rest will be positional. The third possibility where you don’t need to specify the count of either files or arguments, is to separate the list of files from the list of arguments by a single ; which must be escaped by the shell. You can also use -- (two hyphens) in stead of the single ; , but you then need to have *two* occurrences of of -- on your command line, as getopt(3C) uses the first of these to mean the end of options. You can use the method that is best suited for your scripting; they effectively serve the same purpose.

Positional arguments are made available to your rwl program as string variables named $1, $2, etc. and their count is available as $#. If you e.g. have an input file called dollar.rwl with this contents

```
printline $1, $2;
```

you can execute something like:

```
rwloadsim -A 2 dollar.rwl hello world
```

```
RWP*Load Simulator Release 2.0.7.13 Development on Mon Oct 22 03:12:07 2018
```

```
hello world
```

Note that due to the implicit conversion from strings to integer or double, you can similarly use the positional argument variables in numerical expressions.

The shift statement works by shifting arguments left ($2 overwrites $1, etc) and at the same time subtracting one from $#. Note that the shift statement does not change the value of the highest numbered argument. As an example, if you save this small rwloadsim program in a file called echo.rwl:

```
while $# execute
  if $# = 1 then
    printline $1;
  else
    print $1||" ";
  end if;
  shift;
end while;
```

you can e.g. do:

```
rwloadsim -q -F1 echo.rwl hello world of simulation
hello world of simulation
```

If you prefer using the ; marker to end the list of files, you could alternatively do :

```
rwloadsim -q echo.rwl ‘;’ hello world of simulation
hello world of simulation
```

Note that the variables $#, $1, $2 are global like any other predefined variables and that threads therefore get their own copies of these variables. Positional arguments beyond the count return an empty string.

Using rwloadsim as an interpreter

You can use rwloadsim as in interpreter using the usual #!/path/to/executable syntax as the first lines of a script file. Due to limitations in the execve() system call, only one argument can be provided to rwloadsim using this approach, and you would often want that argument to be -F 1 making the rest of the arguments available as positional argument to your rwloadsim program. As an example, if your actual rwoadsim exeuctable is found in /opt/oracle/bin/rwloadsim, and you create a file called echo.rwl with the following contents, put it in your PATH, and give it execute permission:

```
#!/opt/oracle/bin/rwloadsim -qF1

# This is an emplementation of the "echo" command
# using rwloadsim
```

```
while $# execute
  if $# = 1 then
    printline $1;
  else
    print $1||" ";
  end if;
  shift;
end while;
```

you have effectively implemented the echo command using rwloadsim, and it can used like this:

```
$ echo.rwl hello world
hello world
```

Note that the single argument found in echo.rwl contain both -q to not display the banner and -F1 to make rwloadsim use all subsequent arguments as positional arguments made available as $1, etc.

Startup file

During startup, one startup file may be read. If the file pointed to by the environment variable RWLOADSIMRC is readable it will be read during startup, otherwise if $HOME/.rwloadsim.rwl exists, it will be read. The startup file can only contain comments and directives (see next chapter) and is e.g. useful if you want to globally change $iformat, etc.

Creating tags file for vi

You can create a vi tags file with the specified name using the -T option, which you typically should use together with the -e option. Tags file creation is still experimental and it does e.g. not deal with cases, where you have run commands to start threads in multiple rwl files. The first run command found will be used to create a tag named "run", so you can use it with vi by typing 'vi -t run'. There will also be a tag named runxx, which refers to the run command in the file named xx.rwl.

If you have multiple simulations and/or multiple files with run commands in the same directory, it may be beneficial to create different tags files, tags1, tags2, etc and subsequently merge these using a command line like:

```
env LC_ALL=C sort -u tags1 tags2 … > tags
```

Directives

The behavior of rwloadsim can be modified by directives, which are embedded in the input files; some are only available in the startup file. They can be found anywhere in the input and are acted upon immediatedly; they are not actually parsed by the rwl language parser. Directives have the format:

$<directivename>:<directivevalue>

without any white-space. The following directives exist:

$iformat:%<format>

This directive sets the printf format used whenever an integer is converted to a string; the defalt value is "%ld". A warning will be displayed if the conversion isn’t valid for a (small) set of verification values. Some useful values may be:

```
$iformat:%08ld
# string representation of integers will have up to 8 leading zeros


$iformat:%20ld # string representation of integers will have size 20.
```

The size of the internal buffer used to store the string representation of an integer is at least 30 characters.

Note that in the following code snippet:

```
integer x := 123; $iformat:%08ld printline x;
```

the output will be '123' and not '00000123'. The reason is that the string representation of the integer x is generated when the value 123 is assigned to it, which is *before* the $iformat directive. If you want the directive to be effective in this case, you need to include it *before* the assignment:

```
$iformat:%08d integer x := 123; printline x;
```

$dformat:%<format>

This is similar to $iformat, except it is used when the string representation of double variables is needed. The default dformat is %.2f, and the same minimum 30 characters buffers is used.

$include:"<filename>"

Includes the named file (the name cannot contain the character ") very similar to what #include does in C or @ does in sqlplus. The primary use is to replace a long list of file names provided as arguments to rwloadsim by a single file having multiple $include directives. $include files can themselves contain a $include directive for multiple levels of inclusion. You can only use $include when you are not parsing a statement, declaration or thread execution and the file included must similarly contain complete rwl syntax. The double quote characters are part of the $include directive.

A possibility to specify a list of directories to search for the files may be provided in a later version of rwloadsim.

$randseed:<hexdigits>

The random number generator is normally provided a seed from /dev/urandom, such that different executions of rwloadsim generates different random values. If you want repeatable results, you can use this directive; the argument is any sequence of hexadecial digits(0-9, a-f, A-F); up to twelve of these are used as the unsigned short entries in the xsubi[] array used by erand48(3) and nrand48(3). The random numbers in threads will also be repeatable as they are generated by a thread-specific permutation of bits in the xsubi[] array. You can optionally put 0x or 0X in front of the up to twelve hexadecimal digits.

$startseconds:<double> $clockstart:<double>

These directive serves the same purpose as the -c option and it sets the common start time as a number of seconds after starting rwloadsim. The directive must be used before the results databsase is declared. The two directives have identical behavior.

$debugon:<hexdigits> $debugoff:<hexdigits>

The debugon directive has the same effect as using the -D option to the rwloadsim executable; debugoff reverses it. Both of these take a string of hexadecimal digits as argument; 0x or 0X in front of them is optional.

$mute:<integer>

This directive will mute the rwl error with the number provided.

$oraerror:stop

Tell rwloadsim to stop as soon as SQL execution gives any ORA- error. Note that doing so can imply more errors during the actual program termination.

$oraerror:continue

Tell rwloadsim to continue when ORA- errors are found during SQL execution. This is the default.

$bindoffset:0

Change bind by position offset to start at zero; this is primarily useful if you are running with event 10046 tracing where bind value dumping starts at #0. With this directive set, the bind position actually used with the OCIBindByPos call is one larger than the position set at `bind` or `bindout`. The effect of the directive can be reversed by using `$bindoffset:1`. Note that this directive has no impact on the numbering of select list elements used with `define`.

$maxcode:N $maxident:N

These set the maximum size of the code and maximum number of identifiers respectively. They serve the same purpose as the -I and -C flags and can only be used in startup files.

$statistics:basic $statistics:histograms $statistics:all

These have the same purpose as the -s, -ss, and -sss options respectively. They need to be used before the repository database is declared.

$setaction:on $setaction:reset $setaction:off

The $setaction directives control setting the action (in v$session.action). When on, the action is set to the procedure name when a database session is acquired from a pool or a dedicated connection is being used; the action name is set using the OCI_ATTR_ACTION attribute of the session and is therefore sent to the database during the first coming actual roundtrip. When reset is specified, the action name is reset when the session is released or a dedicated connection no longer is in use; using reset implies an extra otherwise unnecessary roundtrip to the database.

$queue:on $queue:off

These have the same effect as the -Q and -N options; at present the default is that queuing simulation using a backlog is off. This may change in a future release.

$errortime:on $errortime:off

The first of these has the same effect as the -W option, i.e. to augment all execution time errors with a timestamp measured as the number of seconds since clock start. It is primarily used as a debugging tool if the exact time of errors is needed. The second directive turns this off.

Hints and tips

Your simulation should consist of a set of rwl input files potentially with inclusion of other files via the $include directive, which allows for modular development. You could e.g. have files that contain database declarations, files that contain SQL and procedure declarations, and files that contain the actual simulation. Although possible, you are advised to only run one simulation (that is one thread execution using the run keyword)

It is well known, that connection storms can cause severe trouble and rwloadsim has been created to avoid them if at all possible. If you are using session pools, you are strongly advised to only specify the maximum pool size, which will make OCI create all the connections when the pool is created. If you are using dedicated connections, rwloadsim will create one for each thread, and this will be done serially before the threads actually start running.

The -c option (control loop start time) is important with respect to this. Rwloadsim is created such that all control loops (typically one per thread) actually start working at (approximately) the same time, which will be some seconds after start of rwloadsim, and this time is what the -c option specifies. The default is (arbitrarily) chosen at 5s, which should allow moderately sized session pools and moderate counts of threads with dedicated connections to complete connection ramp up before actually starting. If you have large session pools or large number of threads with dedicated connections, you are strongly advised to increase the control loop start time. You can similarly decrease it if you have very low counts. If the start time is not sufficient, you will both risk connection storms, and there is a risk that control loops will start too late. See the description of the start, wait and every options to control loops.

Note that all start times are approximate, and that they can vary as the Operating System isn't a real-time Operating System. The actual implementation of the start times is via the clock_nanosleep() system call with a first argument of CLOCK_REALTIME and a second argument of TIMER_ABSTIME; see the notes of the clock_nanosleep(2) manual page.

The control loop start time is saved in the results database as in the rdate column of the rwlrun table.

Multi-process execution

In normal cases, in particular on small to medium sized hardware, only one simulation will run at a time, that is only one rwloadsim process. But for large hardware configurations, you may want to have multiple (identical or different) simulations running concurrently. As an example, consider a shell script doing:

```
procs=10; p=1;
while test $p -le $procs
do
  rwloadsim -sss -k test1 -K "comment to test1" ... &
  p=`expr $i + p`
done
wait
```

which will start 10 rwloadsim processes in the background and wait for these to finish. This will all work as expected, however the 10 executions will be individual, and their start times of actual simulations will not be coordinated. Also, for the purpose of statistics gathering, the 10 rwloadsim executions will be individual, and results will be saved with 10 different runnumbers.

Rwloadsim has been designed to allow a multi-process simulation to behave as one. This is done as a two-step process; the first step does little more than get a sequence number from the repository, which will become the runnumber, and the second step (which is the real simulation) uses that sequence number and it also attempts making sure control loops in all threads in each start at (approximately) the same time. This is controlled via the -P and -M/-R options. The -P option tells rwloadsim to prepare a multi-process run, which it does by writing a value that contains the runnumber and clock start time information to a file. The -R option reads the contents of the file such that multiple individual processes coordinate runnumber and start time. The -M option needs the contents of the file created via the -P option but does otherwise behave like -M. You can use the option that best suits your scripting. In order to do a multi-process run, you need to change the shell script slightly:

```
procs=10; i=1;
rwloadsim -P preparefile.txt -k test1 -K "comment to test1" resultsdb.rwl

while test $i -le $procs
do
  rwloadsim -R preparefile.txt -sss ... &
  i=`expr $i + 1`
done
wait
```

Alternatively using the -M option, your shell script may look like this:

```
procs=10; i=1;
rwloadsim -P Moption.txt -k test1 -K "comment to test1" resultsdb.rwl
Moption=`cat Moption.txt`

while test $i -le $procs
do
  rwloadsim -M $Moption -sss ... &
  i=`expr $i + 1`
done
wait
```

The first execution of rwloadsim should only have a single rwl input file, namely one that contains a database declaration for the repository database. (If other files are given as input, they *will* be executed, but you are advised against this.)

All subsequent executions of rwloadsim will get the runnumber and their control loop start time from the -R or -M option produced be the previous run. When these finish, they will insert rows into the results tables (per the -s option), and these rows will have the process id stored in the procno column rather than the default 0. Subsequent queries against the results tables can use aggregations to aggregate results, such that all executions are considered as one single run. When installing rwloadsim, there will be aggregation views created named runres_a, histogram_a and persec_a, that do not have the procno column, but instead have a pcount column being the count of processes. If you prefer to control the procno value rather than having it as the process id, the -p option can be used.

In multi-process runs, you will often want a higher control block start time than then default of 5s after process start; it should not only cover the ramp-up of threads and session pools in individual processes, but should also cover the ramp-up of the multiple processes by the shell. You may also need to take time into account taken between the rwloadsim prepare call (with the -P option) and the actual start time of the processes in the background with the -M/-R option. In multi-process runs, the complete ramp-up time is provided via the -c option to the *prepare* call. If you e.g. want 15 seconds total ramp-up time, use a call like:

```
rwloadsim -c 15 -P preparefile.txt -q resultsdb.rwl
```

The generated value in preparefile.txt will ensure all control loops in threads in all rwloadsim processes start at (approximately) 15s after the prepare call. As with single-process execution, if the clock start time is passed when a thread actually starts a control loop, start will be delayed and immediate. If too long time has passes until a process is started in the background, a warning about a negative clock start time is shown. An implication of this is that the -P/-M/-R options should *only* be used in scripts and not when using rwloadsim interactively; if there is a (too) long time between the call to rwloadsim with the -P option, and the calls with the -M/-R option, timing will not be as expected. Also, although actually possible, you should not have any real runs in the input files to the call with -P; only a declaration of the results database is needed. The actual contents of the file created by run with the -P option is a short text string in the format NNN:SSSSSSSS.MMM, where NNN is the runnumber, SSSSSSSS.MMM is a double number representing the number of seconds (with millisecond resolution) since an epoch, which will correspond to the control loop start time in the threads of all rwloadsim processes using the same R/M option. This interpretation may change in the future, so it should *not* be relied upon.

There is no difference between using the -R and the -M options and you can use the one that fits your scripting model best.

A complete example

The demo directory contains a more complete example that shows some of the potential of the RWP*Load Simulator such as:

- Modular approach with files containing declarations, test executions and actual simulation runs
- Examples with different "business processes" being simulated, and how they can be mixed
- Show how awr reports can be generated
- Use of multi-process runs
- Sample statistics usage from the rwloadsim results tables

The following files are available:

demouser.sql	SQL script to create an rwldemo user; it may need modification for tablespace allocation
demotables.sql	SQL script to create an extemely simple order system with headers and lines.
demouser.rwl	Declaration of a rwl database that accesses the rwldemo user.
insertdemo.rwl	Declaration of SQL and procedures that will insert orders into the system
querydemo.rwl	Declaration of two types of queries simulating respectively a simple order query, and a more complex query
massinsert.rwl	A simulation that really is a mass insert of orders
testquery.rwl	A small test of the querydemo.rwl procedures
awr.rwl	Declarations of everything necessary to create awr snapshots and report, this includes database, SQL statements and procedures
runsimulation.rwl	An actual simulation which executes a mix of the three "business" procedures, insert, simple query, complex query, and at the same time creates an awr of the run

If you have installed rwloadsim (untar'ed the downloaded file) directory into an ORACLE_HOME, it is recommended that you copy the contents of the rwl/demo directory to a work directory of your own before running these demonstrations.

To run the demonstration, you should initially create the rwloadsim repository as descibed previously. Next, create the rwldemo user using the demouser.sql script (after potential modification for your environment), and load the tables into the new schema using demotables.sql. You are now ready to add some data, and you can e.g. add 3000 orders by executing (you will need to modify demouser.rwl if you e.g. need a connect string):

```
rwloadsim demouser.rwl insertdemo.rwl massinsert.rwl
```

You should see something like:

```
RWP*Load Simulator Release 1.2.0.3 Beta on Fri Jul 20 04:51:24 2018
```

```
Created demouser as session pool (4..5) to:
Oracle Database 12c Enterprise Edition Release 12.2.0.1.0 - 64bit Production

created 1000 orders with 5519 order lines in thread 2
created 1000 orders with 5424 order lines in thread 1
created 1000 orders with 5567 order lines in thread 3
created 3000 orders with 16510 order lines in total
```

You can use sqlplus to actually query the rwl_demo_ord or rwl_demo_lin tables of your rwldemo schema. Now load some more data by executing:

```
rwloadsim -i runcount:=10000 demouser.rwl insertdemo.rwl massinsert.rwl
```

You should inspect the two files insertdemo.rwl and massinsert.rwl to see what actually happens. You will e.g. see that massinsert.rwl starts a number of threads (3 by default) that each will execute a control loop inserting orders in a busy loop.

Next, take a look at querydemo.rwl and testquery.rwl and execute:

```
rwloadsim demouser.rwl querydemo.rwl testquery.rwl
```

The testquery.rwl starts two threads that execute the simple query and one thread that executes the complex query. An output may be:

```
RWP*Load Simulator Release 1.2.0.3 Beta on Fri Jul 20 04:56:55 2018

Created demouser as session pool (4..5) to:
Oracle Database 12c Enterprise Edition Release 12.2.0.1.0 - 64bit Production

selected 1 orders with 2 order lines in total
sumetotal 43839.42 in thread 3
selected 10 orders with 61 order lines in thread 2
selected 10 orders with 54 order lines in thread 1
selected 21 orders with 117 order lines in total
sumetotal 56494.25 in totall
```

Also try

```
rwloadsim -i showres:=2 demouser.rwl querydemo.rwl testquery.rwl
```

and see the difference; this shows how you can prepare your rwloadsim input files such that they can output some debug if needed.

Execution statistics

Let us next go on to see how execution results are stored in the rwloadsim repository. Repeat the above, but also give the -sss option and rwloadsim.rwl file as input (as before, you may need to modify password and add a connect string in this file):

```
rwloadsim -sss rwloadsim.rwl demouser.rwl querydemo.rwl testquery.rwl
```

In addition to output like the above, it will also show a runnumber at the end. This number is the primary key of the rwlrun table and the foreign key of all results tables. Now logon via sqlplus to the rwloadsim user, and (here with runnumber 144 as example), run these queries:

```
SQL >select * from runres where runnumber=144;

 RUNNUMBER     PROCNO VNAME                WTIME      ETIME     ECOUNT     TCOUNT
---------- ---------- --------------- ---------- ---------- ---------- ----------
       144          0 qcomplex           .000961      .0776          4          1
       144          0 selorder           .002127    .016213         14          2
```

This example output shows that there were two different procedures being executed, qcomplex and selorder, the wtime and etime columns show the total time waited to get a session, respectively to execute the procedure, ecount shows the execution count, and tcount shows the number of threads where this procedure was executed.

The total time for execution (that is session wait time plus actual execution time) is grouped into buckets and the results are saved in the histogram table as this query shows:

```
SQL >select * from histogram where runnumber=144
  2  order by vname, buckno;

 RUNNUMBER     PROCNO VNAME               BUCKNO    BUCKTIM     BCOUNT      TTIME
---------- ---------- --------------- ---------- ---------- ---------- ----------
       144          0 qcomplex                12   .0078125          1    .005058
       144          0 qcomplex                13    .015625          1    .011962
       144          0 qcomplex                14     .03125          1    .018795
       144          0 qcomplex                15      .0625          1    .042746
       144          0 selorder                10 .001953125         12    .014195
       144          0 selorder                11  .00390625          2    .004145
```

You can for example see that there were 12 executions of "selorder" that took less that 1.95ms and 2 that took between 1.95ms and 3.9ms. Similarly, there were 1 execution in each of the execution time intervals ending at 7.8ms, 15.6ms, 31.2ms and 62.5ms respectively for the "qcomplex" procedure. The TTIME column shows the total execution time for all executes in that particular execution time bucket. The BUCKTIM column is the high end limit for the range of the bucket, the low end limit is BUCKTIM/2 (which is the high end of the preceeding bucket).

Finally take a look at the persec table, which shows the number of execute in each 1s interval since the start of the simulation:

```
SQL >select * from persec where runnumber=144
  2  order by vname, second
  3  /

 RUNNUMBER     PROCNO VNAME                              SECOND     SCOUNT
---------- ---------- ------------------------------ ---------- ----------
       144          0 qcomplex                                1          1
       144          0 qcomplex                                2          1
       144          0 qcomplex                                3          1
       144          0 qcomplex                                4          1
       144          0 selorder                                1          0
       144          0 selorder                                2          2
       144          0 selorder                                3          4
       144          0 selorder                                4          4
       144          0 selorder                                5          4
```

The four entries for the qcomplex procedure show that there was one execution in the each of the first four seconds. For the selorder procedure, there were no executes in the first second, two in the second, and four in each of the next three. If you take a look at testquery.rwl, you will see that these counts are expected, as both the execution of selorder and of qcomplex have a fixed arrival rate specified with every 0.5 and every 1 respectively. For the two threads executing selorder, you will also see that the start time is specified as 1*threadnumber, so the actual work will start with a delay, which is visible in the last column above for selorder.

A real simulation

A real simulation would execute things much faster than every second, it would run for a longer time, and it would also need to create an awr report. First take a look at awr.rwl and potentially modify the username and/or password for the account that can execute the dbms_workload_repository package; you may also need to add a connect string. Next, take a look at runsimulation.rwl which combines everything. The core of it contains:

```
run
  # Start a number of real worker threads
  threads thrcount # start 20 threads
  at demouser      # Using this database
    loop
      every erlang2(0.05)  # simulate arrival rate of 20 per second
      stop totaltime;  # this many times
      doeither(); # executing this
    end;
  end;

  # Use one thread to gather and make awr
  threads !!doawr # make sure 0 or 1 threads start
  at awruser
    # begin 5 seconds into run
    wait 5;
    beginawr();
    # end 5 seconds before finish
    wait totaltime-runseconds-5;
    makeawr();
  end;
end;
```

There are two groups of threads, one doing the actual simulation and one simply taking an awr. The first group only has a control loop, the number of threads and the stop time are provided as variables (that have default values which can be changed on the rwloadsim command line), and it executes a procedure called "doeither". This is defined as a random procedure array, so it will effectively call either of the three procedures, insorder, selorder, qcomplex with certain probability.

The second group has has a sequence of simple statements, it effectively calls a procedure called beginawr() after 5 seconds, which effectively gathers a snapshot in the workload repository. It then waits until 5 seconds before endtime, and calls makeawr(), which gathers a second snapshot and actually writes the awr file (by default in text format; see the awr.rwl file).

Now execute:

```
rwloadsim -sss awr.rwl rwloadsim.rwl demouser.rwl insertdemo.rwl querydemo.rwl
runsimulation.rwl
```

it will run for about a minute and upon completion show something like:

```
created 1475 orders with 8127 order lines in total
selected 3814 orders with 20803 order lines in total
sumetotal 11711940.24 in total
```

```
runnumber: 146
```

Your rwloadsim repository will now contain data from this run, and you can e.g. run queries like (you need to specify your actual runnumber shown):

```
SQL >select vname, wtime/ecount avgw, etime/ecount avge
  2  , (wtime+etime)/ecount avgt
  3  from runres where runnumber=146;

VNAME                                AVGW       AVGE       AVGT
------------------------------ ---------- ---------- ----------
insorder                       .000041883 .001258232 .001300115
qcomplex                       .000038146 .015732283 .015770429
selorder                       .000054529 .000891503 .000946032



SQL >select * from histogram where runnumber=146
  2   and vname = 'selorder' order by buckno;

 RUNNUMBER     PROCNO VNAME               BUCKNO    BUCKTIM     BCOUNT      TTIME
---------- ---------- --------------- ---------- ---------- ---------- ----------
       146          0 selorder                 8 .000488281         15    .006649
       146          0 selorder                 9 .000976563       3091    2.32041
       146          0 selorder                10 .001953125        669    .741815
       146          0 selorder                11  .00390625         11    .027817
       146          0 selorder                12   .0078125         22    .137362
       146          0 selorder                13    .015625         20    .206681
       146          0 selorder                14     .03125          0          0
       146          0 selorder                15      .0625          3    .183512
```

The first query shows the average wait time to get a session, the average execution time for the database work, and the average total time for the three procedures. The second query shows the histogram of execution times for the "selorder" query; most executions (3091) were in the (rounded) interval [0.5ms;1ms], 15 were even faster and 3 were in the (rounded) interval [31ms;62ms]. You can run similar queries for the other procedures, and you can query the "persec" table to see how many executions there were of each in each of the 60 seconds total elapsed time. There are also two views named fractiles and percentiles respectively, and you can e.g. execute:

```
SQL >select * from fractiles where runnumber=146
  2  and vname='selorder' order by bucktim;

 RUNNUMBER VNAME                             BUCKTIM     BCOUNT   FRACTILE
---------- ------------------------------ ---------- ---------- ----------
       146 selorder                       .000488281         15 .391542678
       146 selorder                       .000976563       3091 81.0754372
       146 selorder                       .001953125        669 98.5382407
       146 selorder                        .00390625         11  98.825372
       146 selorder                         .0078125         22 99.3996346
       146 selorder                          .015625         20 99.9216915
       146 selorder                           .03125          0 99.9216915
       146 selorder                            .0625          3        100
```

to show the fractile information in the histogram. As an example, this results tells that approximately 81% of the executions of selorder were faster than (about) 1ms. Similarly, you can do

```
SQL >select vname, pct90, pct95, pct98 from percentiles
  2  where runnumber=146;

VNAME                               PCT90      PCT95      PCT98
```

```
---------------------------- ---------- ---------- ----------
insorder                     .001854784  .00192695 .003092448
qcomplex                     .040267857   .0540625 .062339286
selorder                     .001475646 .001755258 .001923025
```

showing the 90%, 95% and 98% percentiles of execution times for the three procedures.

A multi-process run

The final step in this complete example is to show a multi-process run. It requires two steps: A prepare step with the -P option producing a file that will be read by the -R option, and an actual execution step with many processes having the same file as -R option. The shell script runmany.sh does exactly this. It initially does:

```
rwloadsim -q -P Moption.txt rwloadsim.rwl
```

and then in a loop starts a number of processes in the background similar to

```
rwloadsim -R Moption.txt -d totaltime:=300 -sss awr.rwl rwloadsim.rwl
demouser.rwl \
insertdemo.rwl querydemo.rwl runsimulation.rwl &
```

and finally waits for the background processes to finish. Some details are left out here, as an example, it is necessary to ensure awr is only generated by one of the processes. When executing, you need to be able to create about 50 sessions in your database. If you type

```
./runmany.sh
```

it will run for about 5 minutes and upon finishing, you will see something like this (repeated five times for each process):

```
created 7299 orders with 40083 order lines in total
selected 19202 orders with 105399 order lines in total
sumetotal 55399747.91 in total
runnumber: 147
```

An awr file will have been created (find its name by doing ls -ltr), and an extract from the SQL ordered by CPU might be:

```
SQL ordered by CPU Time                    DB/Inst: C2/c2  Snaps: 18963-18964

    CPU                   CPU per           Elapsed
  Time (s)  Executions    Exec (s) %Total   Time (s)   %CPU    %IO    SQL Id
---------- ------------ ---------- ------ ---------- ------ ------ -------------
      35.2       14,265       0.00   21.4       82.9   42.4     .3 61ht2hkmagbn5
Module: rwloadsim@slc09nzr (TNS V1-V3)
select /*+use_nl(l o) use_nl(r l) index(o pk_ord)*/ o.ordno , o.b , o.c , o.
pl , l.linno , l.e , l.pl , r.g , sum(l.e) over (partition by o.ordno) sume
 , count(l.e) over (partition by o.ordno) cntl , sum(length(translate(l.pl,' a
bcdefghijklmnopqrstuvwxyz','-'))) over (order by o.ordno) sltp from rwl_demo_or

       4.0       92,074       0.00    2.4        8.2   48.6    2.7 855gyybkhcnc9
Module: rwloadsim@slc09nzr (TNS V1-V3)
select l.ordno , l.linno , l.e, l.pl , l.refno , r.g, r.pl from rwl_demo_lin l j
oin rwl_demo_ref r on l.refno = r.refno where l.ordno = :1 order by l.linno
```

Let us now finally see a few queries against the rwloadsim repository. Because this was a multi process run, the procno column of the various tables will now have a non-zero value, and there will be a total of five different values as seen here:

```
SQL >select * from runres where runnumber=147
  2  and vname = 'qcomplex';

 RUNNUMBER     PROCNO VNAME                               WTIME      ETIME     ECOUNT
---------- ---------- ------------------------------ ---------- ---------- ----------
       147       9179 qcomplex                           .25899   57.71752       2933
       147       9182 qcomplex                          .290062  58.146928       2980
       147       9176 qcomplex                          .107874  57.712318       2999
       147       9172 qcomplex                          .097843  58.014568       2937
       147       9174 qcomplex                          .097493  56.297332       2906
```

You would very likely need to aggregate this and for that purpose, each of the three tables, runres, histogram and persec, have aggregate views where the procno column is replaced by a pcount column; the view are named as the base table with an appended _a. As an example:

```
SQL >select * from runres_a where runnumber=147;

 RUNNUMBER     PCOUNT VNAME                               WTIME      ETIME     ECOUNT
---------- ---------- ------------------------------ ---------- ---------- ----------
       147          5 qcomplex                          .852262 287.888666      14755
       147          5 selorder                         3.802066  90.211869      95625
       147          5 insorder                          1.24865 132.845251      36669
```

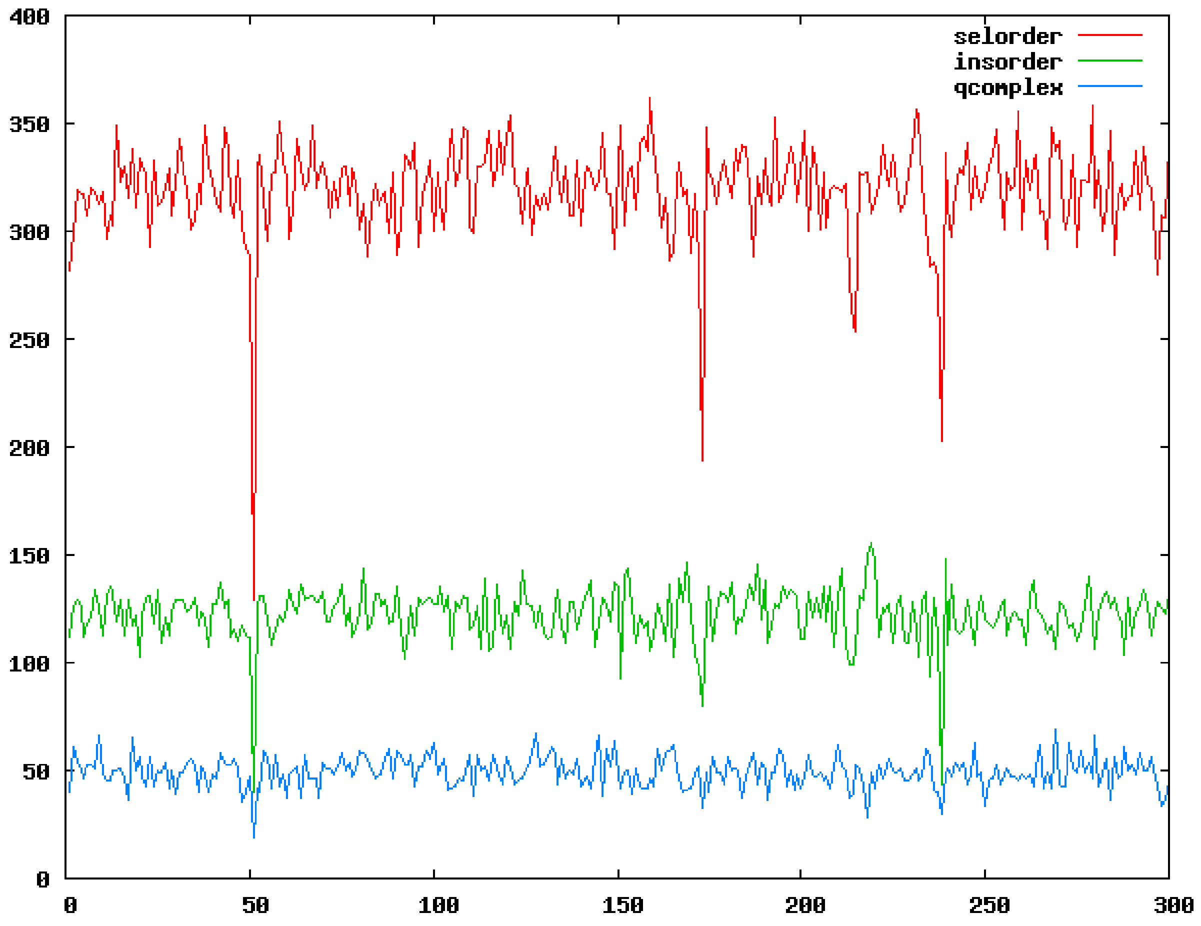
An example of a more complex query using the pivot clause to generate per second execution counts for each of the three procedures is available in persec_q.sql. An example (with just the few first and few last result rows) is:

```
SQL >set echo on
SQL >@persec_q
SQL >select * from
  2  (select vname, second, scount from persec_a where runnumber = (select max(runnumber)
from rwlrun))
  3  pivot
  4  (sum(scount) for vname in ('selorder' as selorder, 'insorder' as insorder,
'qcomplex' as qcomplex))
  5  order by second
  6  /

    SECOND   SELORDER   INSORDER   QCOMPLEX
---------- ---------- ---------- ----------
         1        282        112         40
         2        299        126         61
         3        319        129         54
         4        317        125         50
         5        317        112         46
         6        307        118         53
         7        320        121         53
...
       297        280        128         43

    SECOND   SELORDER   INSORDER   QCOMPLEX
---------- ---------- ---------- ----------
       298        307        125         34
       299        306        123         36
       300        341        132         45
```

If the output from this is saved to a spool file, you could subsequently use gnuplot or some other tool with graphing possibilities to produce a graph like:



Using multiple hosts

The option pair -P followed by -R or -M is used to make multiple processes have a common start time for all control loops. As each of these processes are individual and as there is no inter process communication between them, you can just as well arrange (e.g. via ssh) that these processes actually execute on different hosts. Such a configuration can e.g. be used if there is a limitation to how much can be executed from a single client system.

For this to work well, you need to ensure that the timestamp on the multiple hosts are in sync, e.g. by using an NTP server. The -P option that prepares a multi-process run does little more than calculating the timestamp at which the individual processes will start; hence an offset in time synchronization between multiple hosts, will directly impact the control loop start time.

The -c option value used when preparing such a multi-host, multi-process run should take any overhead of ssh into account.

The text string output to the -P file argument is guaranteed to have no white-space.

Interrupting rwloadsim

If rwloadsim is being interrupted with SIGINT by a user hitting ctrl-c, a graceful termination is attempted. If there is outstanding SQL at the time, such SQL is likely to receive the ORA-01013 error and receiving this error sets an internal flag, which will make rwloadsim terminate. The same flag is also set if rwloadsim itself catches the interrupt generated by hitting ctrl-c. When the flag is set, all worker threads will terminate as quickly as possible, which may imply rolling back outstanding transactions; this specific behavior is however not guaranteed. It is the responsibility of the user to make sure any clean-up – if necessary – is done. Note that with many worker threads and/or many processes in a multi-process run, it is normal to see a sequence of RWL and/or ORA errors if ctrl-c is hit, but execution will normally terminate shortly after hitting ctrl-c.

If rwloadsim should not stop after hitting ctrl-c, you can hit ctrl-c multiple times, which will cause it to terminate using SIGTERM.

In rare cases, rwloadsim will *not* terminate, even after receiving multiple ctrl-c interrupts. In such cases, you can attempt using ctrl-\ (generating SIGQUIT), or use the kill (or killall) command to terminate the rwloadsim processes.

Note that if the directive $oraerror:stop is in effect, *any* ORA error (not only ORA-01013) will cause the internal termination flag to be set, which means that any ORA error in any thread will lead to process termination after a short time.

Queuing system simulations

It is mentioned above, that using every erlang2(1/x) simulates a queuing system with an Erlang distributed arrival rate of *x* per second. Although rwloadsim does *not* actually have a queue, one will be simulated if you put the keyword “queue” in front of “every” or for all control loops if you supply the -Q option. Without this, the “every” option of control loops calculates when the *next* loop should start based on the start of the current loop. If the actual execution time of the current loop (including acquiring session from a session pool) is longer than this time, the next loop will simply start immediately as the calculated start time is passed. If you use the -Q option or put “queue” in front of “every”, the expected execution start time is calculated with respect to the initial common start time. As a simple example, if you specify every 1, each execution will start exactly 1 second after the start of the previous without -Q, but if an execution is longer than 1 second, the next (and subsequently all following) executions will be delayed. With the -Q option or “queue” keyword, the starttime of execution N in the loop will be N (after the common start time), which may imply a number of consecutive executions will follow immediately after each other in the case where the first few take longer than 1s. Using erlang2(1/x) when *1/x* is much larger than the typical time per execution, there is little difference with or without the -Q option, but if 1/x is close to or even smaller than the typical time per execution, the total workload using -Q will be higher than without -Q. If you use the -Q option, you can revert to the traditional behavior without backlog for an individual control loop by using “noqueue” before “every”.

Note that some future version of rwloadsim may have -Q as the default.

Using LOB data

The RWP*Load Simulator has support for reading and writing CLOB from/to a database. You can declare variables of type clob which can be used for bind or define of SQL statements, and which can be used with writelob and readlob statements. A clob variable is actually handled like an OCILobLocator. The writelob statement takes an arbitrary expression (as a string) and writes it to the CLOB, and the readlob statement reads a CLOB from the database into a string variable. At present, only complete CLOB's can be written or read; there is no support for piece-wise reading/writing and no support for NCLOB or BLOB. A clob variable is initialized as empty by setting the OCI_ATTR_LOBEMPTY attribute.

Demonstration library

The rwl/demo directory contains the various samples discussed above. Additionally, there are a few rwl files, that may be generally useful.

awr.rwl

This file contains the necessary declarations to generate awr reports. You will need to modify username, password and possibly connect string to point to a user in your database that allows calling the dbms_workload_repository package. You will probably also want to modify the file name of the output file.

ovid2.rwl

This file contains a large random string array with words in the latin language; it can be used to generated nicely looking gibberish. Please follow the comments in the file. To test is, you can execute:

```
rwloadsim ovid2.rwl ovid2test.rwl
```

which will print 16 lines with 1 until 16 words of random gibberish.

sqlid2file.rwl

This file contains a procedure that – given a sqlid – will lookup the SQL text in either gv$sql or dba_hist_sqltext and create a file named by the sqlid plus a suffix of .rws containing the SQL text. It requires an rwl file containing declaration of a database connecting as system. If such a declaration is found in a file named system.rwl, an example execution is:

```
$ rwloadsim system.rwl sqlid2file.rwl -- '-x sqlid2file("gnzsgs00h57s5") at system;'

RWP*Load Simulator Release 2.0.1.18 Beta on Mon Aug  6 06:09:43 2018

Connected system to:
Oracle Database 12c Enterprise Edition Release 12.2.0.1.0 - 64bit Production

text of sqlid gnzsgs00h57s5 found in gv$sql written to gnzsgs00h57s5.rws
```

After this, the file gnzsgs00h57s5.rws will contain the SQL with that particular sqlid. This sqlid is actually one of the SQL statements found in sqlid2file.rwl itself.

A word about the rwl language

The rwl (pronounced "rawl") language is a procedural language with usual procedural elements such a procedures, expressions, if/then/else constructs, functions with arguments in addition to calls of SQL statements, cursor and counter loops, etc. Some of these have declarative elements, such as the control loop, threads with attributes such number of threads to run, size of session pool, bind variables for SQL statements etc. This combination is reflected in the syntax for rwl. As an example, a procedure is a list of statements it needs to execute, which can include loops, conditional execution, call of SQL statements, etc. However, procedures also have an attribute, namely if its execution statistics should be saved or not.

Similarly, a run consists of one or more threads, each having one or two attributes (number of threads doing the same, and potentially the database they connect to), and a body being similar to the statement list of a procedure, where execution of one statement will follow the previous. The important control loop has attributes like when to start, when to stop or how many times to execute, potentially the frequency or the wait time between each.

It is also worth mentioning the use of ; as a terminator and of end. In order for the parser (which is written in bison) to deal well with the rwl syntax, clear separation between syntax elements is needed (that is the ;) and nice encapsulation of potentially long sequences of such elements must be achieved, which is what end does. Procedures are lists of statements and a statement may itself recursively be a statement list encapsulated with e.g. loop and end. Therefore procedures are encapsulated by procedure and end. Similarly, SQL declarations are complex as they require a list of bind/define etc, each of these also being terminated by ;. So SQL declarations also require encapsulation with sql and end. A complete simulation run may include threads doing different things, so a run has the run/end encapsulation, and likewise, each thread has a body of statements, each terminated by ;, so each thread has the thread/end encapsulation.

The input file scanner is generated by flex, with the exception of the scan for blocks of SQL or PL/SQL text, which is read verbatim. Therefore, the scanner would happily scan an input like:

```
integer select 1 from dual;
```

as two tokens, the keyword integer and the SQL text select 1 from dual, and the parser will subsequently return an error.

Another effect of using flex is that keywords *never* can be used as identifier. You can e.g. not do integer dedicated:=1; as dedicated as a keyword, albeit only used during database declarations.

Keywords and predefined variables

All keywords, predefined functions, procedures etc are reserved words in the rwl language, and they cannot be used as identifier names. All ordinary keywords are in lower case and rwl is generally case dependent. There are also a number of predefined variables that are read-only. In addition to ordinary keywords, a subset of SQL and PL/SQL keywords are used in rwl to initiate scanning for complete SQL or PL/SQL text as described above. Additionally, there are a number of directives that are handled during lexical scanning but as such, aren't part of the rwl language. Finally, some punctuation characters are used.

List of current ordinary rwl keywords

```
all and array assign at between bind bindout clob
commit concat connect count cursorcache database
dedicated default define double drcp else end
every execute file fflush for function
ignoreerror if integer is isnull loop modify
nocursorcache noqueue nostatistics not null ociping or
password print printline printvar procedure queue raw
random readlob reconnect results return rollback run
shift sessionpool sql start statistics stop
string sum then threads username wait while
write writeline writelob
```

List of ordinary rwl keywords, etc reserved for future use

```
atan2 blob break cos exp leak length log nclob raw shardkey sin substr
```

List of predefined variables

```
everyuntil loopnumber runnumber runseconds threadnumber usrseconds sysseconds
oraerror oraerrortext stderr
```

Note that runseconds syntactically is a keyword, but it is semantically like a variable.

List of predefined functions

```
erlang erlang2 lengthb sqrt substrb uniform
```

List of predefined procedures

```
getrusage wait
```

List of keywords and punctuation causing scanning for SQL text to begin

```
alter create drop insert merge update select truncate /*
```

As en exception to the general rule, these keywords are case insensitive.

List of keywords and punctuation causing scanning for PL/SQL text to begin

```
call declare begin --
```

As en exception to the general rule, these keywords are case insensitive.

List of directives

```
$clockstart $dformat $iformat $include $mute $maxmesg $oraerror $randseed
$bindoffset $debuon $debugoff $maxcode $maxident $queue $startseconds
$statistics $setaction
```

List of punctuation

```
, ; ( )
:= += .. ||= ||
+ - * / % : ?
```

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

Known issues

- Oracle bug 26024282 implies a quite visible overhead when starting many threads, so it is recommended to keep the thread count moderate (say less than 20-50 depending on hardware) and use multi-process execution.
- Oracle bugs 26568177/22707432 imply sessions pool cannot be created with identical values for minimum and maximum size for Oracle version 12 and earlier.
- Stacked at-clauses are somewhat obscure and are likely to change semantics in a later release.
- Although you can use any character set including multi byte character sets for strings, there is no actual national character support. The string function (substrb and lengthb) treat strings as a sequence of bytes (with a null terminator), and may therefore break multi-byte characters. There are no current plans to change this behavior.
- The value of the options `stop` and `count` provided to a control loop are calculated when the loop is *entered*. You can therefore not cause a control loop to terminate earlier than planned by changing either of these value.

Debug bits

The following debug-bits (in hex) which can be set using $debugon directive or -D option may be generally useful

1	Turn on certain experimental and unsupported features.
8	Make the parser more verbose upon parsing errors by printing yyerror. Using this bit can help understanding of syntax errors.
20	Debug code execution.
100	Debug expression evaluation.
200	Debug database calls.

Other debug bits exists; they should not be used by ordinary users.

Credits

Thank you:

- https://en.wikipedia.org/wiki/Agner_Krarup_Erlang for his work on the mathematics on queuing in telephony systems. Without him, those functions would not have been called erlang() and erlang2() respectively. He was a great Dane.

- Gunther Rademacher for making it easy to create nicely looking syntax diagrams at http://www.bottlecaps.de/rr/ui
- Jutta Degener et al for making Yacc grammar for C available at http://www.lysator.liu.se/c/ANSI-C-grammar-y.html, which is where the expressions of rwloadsim come from.
- Google for creating the algorithm that made it reasonably easy for me to understand some of the weird details of flex and bison and lots of other stuff
- About six billion people for not making "yylex" mean anything else than what it does in yacc&lex. That surely makes google searches more efficient.