



Perl for DBAs

By Jared Still

By now, you have no doubt at least heard of Perl. It has been around for quite a few years, having been first developed in the 1980's, and came into widespread use in the 1990's.

Many people's exposure to Perl has been limited to its use as a Web scripting language, where it gained a great deal of popularity. Perl is a great language for Web use, but its functionality extends far beyond what is needed for use with Web applications.

There is a standard list of tools that most DBAs have at their disposal and know how to use. On Unix platforms this list of tools might include the Korn and Bourne shells, as well as the Bash shell on Linux systems, sed, grep, awk, tr, cut, paste, bc and numerous others. This list may be somewhat limited if you are using a Win32 platform, though there are sources for Unix utilities on Win32 platforms. See the list of resources at the end of the article.

This toolset is of course rounded out by the tools that come with the Oracle database: SQL*Plus, svrmgrl, or even sqldb. This list will vary depending upon the version of Oracle that you are using.

Though there may seem to be quite enough tools for anyone to manage databases, my goal is to convince you make room in your toolbox for one more.

A Little History

From its inception, Perl was designed for manipulating data. It was originally created by Larry Wall to take the best features of C, awk, sed, sh and other Unix utilities, and combine them into one powerful language.

Though originally designed with systems administration work in mind, Perl's innate ability to work with data made it a natural for database use, and so Oraperl was created by Kevin Stock in 1990.

Oraperl allowed connections to Oracle databases to be made directly from within Perl scripts, and direct manipulation of the data in the database via Perl. This was quite a refinement of the redirection that was frequently used in shell scripts when DBAs needed to access data outside of the database as shown in Example 1.

```
#!/bin/ksh

export ORACLE_SID=ts01
export ORAENV_ASK=NO
unset SQLPATH

. oraenv $ORACLE_SID

sqlplus -s <<EOF
system/manager

set feed off term off pause off head off
spool users.txt

select distinct s.username username
from v\session s
where s.username is not null
/
EOF

while read user
do
    if [ ! -z $user ]; then
        echo "$user: Please log out!" | mailx -s "please log
out of database!" $user@yourdomain.com
    fi
done < users.txt
```

Example 1

Oraperl was a library that had to be built and linked into the Perl binary. Since this required rebuilding and re-linking the Perl binaries, the process could be quite an undertaking on some platforms.

This method also required that you have a separate Perl binary for connecting to different databases. If you had both Sybase and Oracle databases, you needed to have a different binary for each database.

With the advent of Perl version 5 in 1994, a new modular Perl architecture made it possible to create modules for Perl that did not require re-linking the Perl binaries.

At this time Tim Bunce introduced the DBI, or DataBase Interface Module. The Perl DBI provided a consistent interface for writing database applications, regardless of the database. Each database would need a corresponding DBD, or DataBase Driver module, written to DBI specification. The DBD::Oracle module was also made available at this time.

Since then, there have been DBD modules created for most popular relational databases including Oracle, Sybase, DB2, Teradata, mysql, PostgreSQL and several others. There are also DBD drivers for ODBC and JDBC.

Another advantage Perl has is its ability to create several simultaneous connections to different databases. This makes it simple to move data between disparate databases without using database links or Oracle Heterogeneous Services.

Getting Perl

Installing Perl on your system is beyond the scope of this article. For the purposes of this article, it will be assumed that it is already installed, which is usually the case on most Unix systems. Should you need to install it on your system, see the Resources section at the end of this article.

Although Perl may not be installed by default on Win32 Systems, it is easily acquired and installed by visiting the Web site for ActiveState Perl. The binaries are already compiled and ready to go. See the Resources section at the end of this article for details.

Getting DBI and DBD::Oracle

Installing DBI and DBD::Oracle on Win32 systems is a very simple process. There are two sources for pre-built versions of the DBI and DBD::Oracle modules. The first is from ActiveState. To install these modules, simply open a COMMAND window, and use the ActiveState Perl Package manager as shown in Example 2. Be sure to set the HTTP_PROXY environment variable if needed at your site.

```
C:> set HTTP_PROXY=http://myproxy.mydomain.com:8080
C:> ppm
PPM interactive shell...
PPM> install DBI
...
PPM> install DBD::Oracle
...
PPM> install DBD::ODBC
...
PPM> quit
```

Example 2

Another source for DBI and DBD::Oracle is from the www.xmlproj.com site. These prebuilt modules provided by Ilya Sterin are the preferred source since they are usually more up-to-date than those found at ActiveState, and are built with later versions of the Oracle libraries. The DBD::Oracle module at ActiveState is built with Oracle 7 libraries at the time this was written. See Example 3 for details.

```
C:> set HTTP_PROXY=http://myproxy.mydomain.com:8080
C:> ppm
PPM interactive shell...
PPM> install http://www.xmlproj.com/PPM/DBI.ppd
...
PPM> install http://www.xmlproj.com/PPM/DBD-Oracle.ppd
...
PPM> install http://www.xmlproj.com/PPM/DBD-ODBC.ppd
...
PPM> quit
```

Example 3

Assuming that Oracle is already installed, the Perl DBI, DBD::Oracle and DBD::ODBC modules are installed and ready to go.

On Unix systems this requires a bit more work, but it is still not too difficult. You will probably need to be logged in as “root” to do this, but check with your system administrator to be sure.

First download the source for DBI and DBD::Oracle. Both can be found at <http://search.cpan.org>. This needs to be done on a server that already has Perl 5 and Oracle installed.

Now decompress the source archive and untar it into a working directory:

```
gunzip -c DBI-1.30.tar.gz | tar xvf -
```

Create the makefile, run the make and install:

```
perl Makefile.PL
make test
make install
```

The steps for installing the DBD::Oracle module are similar. You will need to set an environment variable for the test phase to succeed. The ORACLE_USERID variable needs to be a valid username/password combination.

```
gunzip -c DBI-1.30.tar.gz | tar xvf -
export ORACLE_USERID=system/manager
perl Makefile.PL
make test
make install
```

Using Perl

By now, you are no doubt asking yourself, “This is all well and good, but what can I do with it?” I am glad you asked, because we are going to consider a few real life situations where Perl turned out to be the best tool for the job.

I was recently asked to help one of my employer’s financial analysts provide some data for external use. The catch was that the data needed to be provided in MS Excel workbooks. This would not normally be a problem if only a few hundred rows needed to be provided. Dump the data to a comma delimited text file, load into Excel and be done with it, right?

Wrong. That method works for small data sets, but is somewhat unwieldy when the data to be dumped is more than one million rows worth. There had to be a better way.

A few minutes search at the Comprehensive Perl Archive Network or CPAN, were rewarded with several modules that might make this task easier.

The DBI Template Script

Before delving into these details though, first examine a barebones Perl script that uses the DBI and DBD::Oracle modules. We will briefly outline the function of different parts of the script that set up the environment, make the database connection and prepare the SQL, and then spend a little more time on the details of the working section.

Do not worry about completely understanding all of the Perl syntax; it is not important at this point. What is important is seeing what Perl can do for you.

```
1: #!/usr/bin/perl -w
2:
3: use warnings;
4: use strict;
5: use DBI;
6: use Getopt::Long;
7:
```

Lines 1-7: These load the required Perl Modules.

continued on page 21

Lines 8-38 set up the command line interface via the Getopt::Long module, retrieve command line arguments and cause the script to exit with an error if the expected arguments are not available.

```
8: my %optctl = ();
9:
10: Getopt::Long::GetOptions(
11:   \%optctl,
12:   "database=s",
13:   "username=s",
14:   "password=s",
15:   "sysdba!",
16:   "sysoper!",
17:   "z","h","help");
18:
19: my($db, $username, $password, $connectionMode);
20:
21: $connectionMode = 0;
22: if ( $optctl{sysoper} ) { $connectionMode = 4 }
23: if ( $optctl{sysdba} ) { $connectionMode = 2 }
24:
25: if ( ! defined($optctl{database}) ) {
26:   Usage();
27:   die "database required\n";
28: }
29: $db=$optctl{database};
30:
31: if ( ! defined($optctl{username}) ) {
32:   Usage();
33:   die "username required\n";
34: }
35:
36: $username=$optctl{username};
37: $password = $optctl{password};
38:
```

Lines 39- 47 make the connection to the database, and line 49 causes the script to exit if the connection failed.

```
39: my $dbh = DBI->connect(
40:   'dbi:Oracle:' . $db,
41:   $username, $password,
42:   {
43:     RaiseError => 1,
44:     AutoCommit => 0,
45:     ora_session_mode => $connectionMode
46:   }
47: );
48:
49: die "Connect to $db failed \n" unless $dbh;
```

Line 51 sets the number of rows to be retrieved from the database in each call, much like SET ARRAYSIZE in SQL*PLUS.

```
50:
51: $dbh->{RowCacheSize} = 100;
52:
```

Lines 53-57 set up the SQL, prepare the SQL and request the database to execute it. If you have ever used the DBMS_SQL package, this may look somewhat familiar.

```
53: my $sql=q(select * from dual);
54:
55: my $sth = $dbh->prepare($sql);
56:
57: $sth->execute;
58:
```

Lines 59-61 are the working portions of this code. As each row is retrieved, it is placed into an array of name array in line 59, and the zeroth element of the array is printed in line 60. This section does not do much now, but we will add to it later.

```
59: while( my @array = $sth->fetchrow_array ) {
60:   print "$array[0]\n";
61: }
62:
```

Line 63 disconnects from the database. Since there is no more code to execute below this section, the Perl will exit the script.

```
63: $dbh->disconnect;
64:
```

Lines 65-70 contain a usage() subroutine that is called when the help option is present on the command line, or if the expected command line arguments are not present.

```
65: sub Usage {
66:   print "\n";
67:   print "usage: DBI_template.pl\n";
68:   print "   DBI_template.pl -database dv07 -username scott -password
69:   tiger [-sysdba || -sysoper]\n";
70: }
```

Running this script should prove rather uneventful, since all it does is connect to a database and print the output from SELECT * FROM DUAL.

Creating Excel Workbooks in One Easy Step

Now let us get back to dumping a few million rows of data to MS Excel.

To make this example easy, we will work with the well known SCOTT/TIGER account and the EMP table, which only contains a few rows. The concepts will be the same, only the data is different.

After searching the CPAN Web site, the module that appeared to fit my needs was Spreadsheet::WriteExcel. A little experimentation confirmed this, and only a few modifications were required to turn the DBI template script into a script that would create multiple 60,000 line Excel workbooks.

These lines were added at the top of the script:

```
use Spreadsheet::WriteExcel;
use Spreadsheet::WriteExcel::Big;
use constant LINES_PER_BOOK => 60001;
```

The Spreadsheet::WriteExcel::Big module was required to create workbooks over seven megabytes in size, which would be true in this case.

The LINES_PER_BOOK constant was set to track the maximum number of rows that would be created in each workbook.

The SQL statement at line 53 was changed to “select * from scott.emp.”

At the very end of the script, a short function was added to create a new file name for each successive workbook:

```
{
my $workBookNumber = 0;
sub newWorkBookName {
    return "C:/temp/emp_dump_" . ++$workBookNumber . ".xls";
}
}
```

Lines 59-62 were replaced with:

```
59: my $workbook =
    Spreadsheet::WriteExcel::Big->new(newWorkBookName());
60: die "unable to create workbook - $!\n" unless $workbook;
61: $workbook->set_tempdir('C:/temp');
62: my $worksheet = $workbook->addworksheet();
63:
64: my $colNames = $sth->{NAME_uc};
65:
66: my $rowCount=0;
67: my $lineCount=0;
68: $worksheet->write_row($lineCount,0,$colNames);
69:
70: while( my @empData = $sth->fetchrow_array ) {
71:
72:     print "." unless $rowCount++%1000;
73:
74:     if ( ++$lineCount >= LINES_PER_BOOK ) {
75:         $workbook->close;
76:         my $workBookName = newWorkBookName();
77:         $workbook =
            Spreadsheet::WriteExcel::Big->new($workBookName);
78:         die "unable to create workbook - $!\n" unless $workbook;
79:         $worksheet = $workbook->addworksheet();
80:         $lineCount=0;
81:         $worksheet->write_row($lineCount,0,$colNames);
82:         $lineCount=1;
83:         print "\nNew Workbook: $workBookName\n";
84:     }
85:     $worksheet->write_row($lineCount,0,\@empData);
86: }
87: $workbook->close;
```

These few lines are responsible for dumping an arbitrarily large SQL query into an arbitrarily large number of Excel workbooks. Though this example uses the SCOTT.EMP table and will only create a single workbook with a few rows, it will happily keep churning out workbooks for millions of rows, sequentially numbering each as it goes.

This was good news to the analyst who thought she would have to manually import this data into a great many spreadsheets.

As written for this article, this script is set up to run on a Win32 machine. It could just as easily have been run from a Unix database server, and in fact, that is how the script originally worked. The Excel workbooks were created directly on a NetApp filer mounted on a Linux server via SAMBA. This allowed the script to be run by a machine with a great deal more power than my desktop machine, while eliminating any ftp copies that might be necessary to make the files accessible to the analyst.

Monitoring Database Connectivity

One of a DBAs many responsibilities is ensuring that databases are up and accepting connections. The only sure way to test database connectivity is by making a connection from a remote machine. This ensures that the listener is up, the database is up, and that it is accepting connections.

There are a number of possible methods for doing this, but many of them have one fatal flaw. When a database connection hangs without returning an error message, the monitor does not move on to the next database to be checked.

A simple example of this is shown in Example 4. This script will work fine until a connection hangs, and then it will fail to move on to the next database that needs to be checked. There could be other databases that have problems as well, but this script will not inform you under those conditions.

```
#!/usr/bin/ksh
export ORACLE_SID=ts01
. oraenv $ORACLE_SID

while :
do
    for db in ts01 ts02 ts03
    do
        rm -f /tmp/connect.txt
        sqlplus scott/tiger@$db <<EOF
set head off term off echo off pause off pagesize 0
spool /tmp/connect.txt
select 'DB OK' from dual;
EOF
        [ grep 'DB OK' /tmp/connect.txt 2>/dev/null ] || {
            echo "db $db is down | mailx -s "database $db is down" \
                dba@somewhere.com
        }
    done
done
```

Example 4

This is clearly a job for Perl. By using the alarm() function, we can prevent hanging connections from also hanging the connectivity monitor.

This is best implemented on Unix, Unix-like systems (Linux) or others that supply the alarm() or POSIX SIGALRM functionality. ActiveState Perl does not currently include this.

This simplified Perl script assumes that the Oracle environment is already set up (ORACLE_HOME specifically), and that the databases to be checked all have a SCOTT/TIGER account. Example 5 has been kept simple for clarity. One new module has been used: Mail::Sendmail at line 6. This may be installed using the same methods previously detailed for installing DBI and DBD::Oracle.

continued on page 23

```

1: #!/usr/bin/perl -w
2:
3: use warnings;
4: use strict;
5: use DBI;
6: use Mail::Sendmail qw(sendmail);
7:
8: my $timeOut = 60; # wait 60 seconds for connection
9: my $interval = 300; # 5 minutes between connectivity checks
10:
11: my @databases = ('ts01','ts02','ts03');
12: my ($username, $password) = ('scott','tiger');
13:
14: while(1) { # loop forever
15:     foreach my $db ( @databases ) {
16:         print "checking $db\n";
17:         my $dbh="";
18:         eval {
19:             # set alarm to timeout current operation
20:             local $SIG{ALRM} = sub {die "connection timeout\n"};
21:             alarm $timeOut;
22:
23:             $dbh = DBI->connect(
24:                 'dbi:Oracle:' . $db,
25:                 $username, $password,
26:                 {RaiseError => 1}
27:             );
28:         };
29:
30:         alarm 0; # reset the alarm
31:
32:         if ($dbh) { # success
33:             print "Connection succeeded for $db\n";
34:             $dbh->disconnect;
35:         } else { # failure
36:             print "Error connecting to $db\n";
37:             my %mail = (
38:                 To => 'thedba@somewhere.com',
39:                 From => 'oracle@somewhere.com',
40:                 Subject => "Database $db is down!",
41:             );
42:             unless (sendmail %mail) { print "Error sending mail:
$Mail::Sendmail::error \n" }
43:         }
44:     }
45:
46:     sleep $interval; # wait for next db check
47: }

```

Example 5

The lines of particular interest here are 18-28 and 32-44:

- Line 18 enters an 'eval' block. This is code for Perl to evaluate and execute at runtime, much like EXECUTE IMMEDIATE in PL/SQL.
- Lines 20 and 21 set up the timeout functionality. If the connection to the database at line 23 is not made before 60 seconds have passed as specified by the \$timeOut variable, the alarm will cause the eval block to be exited.
- Line 32 checks to see if the connection attempt was successful. If not, the \$dbh variable will be an empty string as assigned in line 17. In that case lines 36-42 are responsible for sending an email notifying the DBA that the database is down.
- Line 46 causes the script to wait \$interval seconds before doing it all again.

This simple script serves to illustrate why many DBAs are finding a place in their toolbox for Perl. I hope this brief introduction will provide incentive for you to learn more.

Resources

Recommended books to learn Perl:

Learning Perl, 3rd Edition

Randal L. Schwartz, Tom Phoenix

O'Reilly 2001

0-596-00132-0

<http://www.oreilly.com/catalog/lperl3/>

Learning Perl on Win32 Systems

Randal L. Schwartz, Erik Olson, Tom Christiansen

O'Reilly, 1st Edition August 1997

1-56592-324-3

<http://www.oreilly.com/catalog/lperlwin/>

Programming the Perl DBI

Alligator Descartes, Tim Bunce

O'Reilly 2000

1-56592-699-4

<http://www.oreilly.com/catalog/perldb/>

Perl for Oracle DBAs

Andy Duncan, Jared Still

O'Reilly 2002

0-596-00210-6

<http://www.oreilly.com/catalog/oracleperl/>

Perl Source, tutorials, and more

<http://www.perl.com/>

Perl Mailing Lists

<http://lists.perl.org/>

Facts About Perl

http://www.perl.org/press/fast_facts.html

Perl History

<http://www.perl.org/press/history.html>

Perl binaries for Win32

<http://www.activestate.com/Products/ActivePerl/>

Cygwin – Unix tools for Win32

<http://sources.redhat.com/cygwin/>

Source for DBD modules

<http://www.cpan.org/modules/by-module/DBD/>

Ilya Sterin's prebuilt modules for DBI and DBD::Oracle

<http://xmlproj.com/PPM/>

O'Reilly's Perl books

<http://perl.oreilly.com/>

Template script

http://www.cybcon.com/~jkstill/util/templates/dbi_template.html



About the Author

Jared Still has been working with Oracle since 1994, beginning with version 7.0.13 of the database. He quickly became immersed in all things Oracle, and has been employed as an Oracle DBA ever since, joining the ranks of Oracle Certified Professionals in 1997. Perl became an integral part of his toolkit when it became the tool of choice to rapidly prototype and create complex reports from custom applications. Jared can be reached at Jared.Still@radisys.com.