

Introduction to using DBI and Oracle

Introduction

DBI is a Perl module, well documented in both online sources and the "Programming the Perl DBI" book by Alligator Descartes, Alistair Carty, Tim Bunce and Linda Mui (O'Reilly, ISBN:1565926994) but still, I can see questions about using DBI with Oracle on Oracle USENET groups. That motivated me to write this article. Oracle has many features and data types, while using DBI with each particular feature is not always entirely trivial. The purpose of this article is to cover many of those uses from Perl practitioner's point of view. Here is what you can find in this article:

- Connecting to Oracle instance and executing elementary SQL.
- Bind calls, in various forms.
- Executing PL/SQL procedures.
- Working with ref cursors and LOB data types.
- Discussion of the various supported and unsupported features of DBI/DBD::Oracle.

The sole purpose of this article is to serve as an introductory reading for those who have never used the two together. Therefore, some knowledge of both Oracle and Perl is assumed, and although I will not try to show off my obfuscation abilities, this article still assumes that the reader has read "Learning Perl" and has some experience with Oracle RDBMS. This article will not cover basic features of neither Perl nor Oracle. It will also not cover the installation of Perl, Oracle, DBI or DBD::Oracle. Furthermore, it makes no claims of completeness. This is simply an introduction to using Oracle with Perl DBI.

This article was created on Red Hat Fedora Core 4, with Oracle 10.2.0.2 and Perl 5.8.6. The decision not to cover the installation was made because the installation is different for each operating system, while I'd like to concentrate on the common features, the features that can be used across the whole range of supported systems. Describing installation on Unix or Linux and omitting Windows or VMS would open me for the accusations of being OS biased. As I really am biased toward one type of OS, I wanted to hide that fact and cover just the common parts.

In writing this article I was relying mostly on the experience, scripts and sins of the past and online documentation. The most accurate and up to date online DBI documentation can always be found on CPAN (<http://search.cpan.org>). In particular, the syntax of all DBI methods can be found on the DBI page. I will occasionally quote DBI documentation, where appropriate and will always warn the reader when I'm doing so. Quotes from the online documentation will also be marked by the different font (smaller and italicized). The online documentation for the DBI and DBD::Oracle can be found at:

<http://search.cpan.org/~timb/DBI-1.50/DBI.pm>

<http://search.cpan.org/~timb/DBD-Oracle-1.16/Oracle.pm>

This is, of course, the documentation for the versions used in this article. This documentation is likely to change with versions. If your DBI and DBD::Oracle versions are different from what was used to produce this article and the example scripts within it, you should check the documentation for your favorite version.

So, let's start with the article. In order to execute the examples from the article the reader will have to have access to an Oracle database and also access to Perl interpreter with DBI and DBD::Oracle installed. Finding Perl isn't a problem. Verifying that all necessary modules are installed is also very easy. So, how do we tell that DBI and DBD::Oracle are installed and how do we tell which versions are installed? The answer is really simple:

```
$ perl -e 'use DBI; print $DBI::VERSION, "\n";'
1.50
$ perl -e 'use DBD::Oracle; print $DBD::Oracle::VERSION, "\n";'
1.17
```

Therefore, we have DBI 1.50 and and DBD::Oracle 1.17. At the moment, these are the latest and the greatest versions. Here is the Oracle version:

```
$ sqlplus scott/tiger

SQL*Plus: Release 10.2.0.2.0 - Production on Sun Apr 16 23:09:35 2006

Copyright (c) 1982, 2005, Oracle. All Rights Reserved.

Connected to:
Oracle Database 10g Enterprise Edition Release 10.2.0.2.0 - Production
With the Partitioning, OLAP and Data Mining options
SQL>
```

This username and password (SCOTT/TIGER) will be used throughout this article, as well as the accompanying EMP and DEPT tables which will be used for almost all of the examples. EMP and DEPT tables do not have LOB columns, so I will have to create my own infrastructure for dealing with the LOB data types.

Connecting to Oracle

DBI uses the following syntax to connect to an Oracle instance:

```
my $db=DBI->connect("dbi:Oracle:local", "scott", "tiger");
```

The string on the beginning contains the following elements, separated by colons: the string "dbi", driver type, in this case "Oracle" and the database name, also known as TNS descriptor. In the line above, the database name was "local". The first argument, therefore, defines the database to connect to. The 2nd and 3rd arguments are, of course, username and password. Of course, everyone who has ever worked with databases knows that connection errors happen from time to time, so it is prudent to check for errors:

```
my $dbh = DBI->connect( "dbi:Oracle:$db", $username, $passwd ) ||  
die( $DBI::errstr . "\n" );
```

In case of connection error, database handle \$dbh is not created and the error string "errstr" comes from the DBI class itself. If, on the other hand, the connection attempt is successful and the database handle is created, there are several properties that can be very helpful with further programming:

- **AutoCommit**: when set to 1, DBI issues a commit after each successful SQL statement. This is very dangerous and on by default. Setting it to 0 is a good idea. This handle property is not Oracle specific, it is available for any database.
- **RaiseError**: When turned on (it is off by default) it sends an exception to your script and terminates it. It is a good idea to kill the script if a non-handled Oracle exception happens. This handle property is not Oracle specific, it is available for any database.
- **ora_check_sql**: Oracle has a performance enhancing trick called "deferred parse". When used, it decreases the number of the needed database calls, by bundling "parse" and "execute" phases of the SQL execution. Unfortunately, this feature is not turned on by default. To turn it on, you have to set ora_check_sql to 0. This is Oracle specific.
- **RowCacheSize**: this instructs the driver to create a local pre-fetch cache and defines its size.
- Both DBI database and statement handles always have **errstr** and **err** member variables which contain error message and code respectively. In case of a failed connection attempt, when database handle is not created, errstr and err can be used in class context, as \$DBI::errstr and \$DBI::err. That is used in the examples throughout the article.
- This list is by no means complete, I mentioned only the properties most frequently used in the scripts. For the complete list, please consult the online DBI documentation. In further text, I'll introduce two more handle properties, used to deal with LOB data types.

These properties are keys of an associative array, and here is how they're set:

```
$dbh->{AutoCommit} = 0;  
$dbh->{RaiseError} = 1;  
$dbh->{ora_check_sql} = 0;  
$dbh->{RowCacheSize} = 16;
```

So, this turns off auto-commit, instructs DBI to terminate the script if SQL error is incurred, instructs Oracle driver to use deferred parse and wait with parsing the SQL until an execution is attempted for the first time. It also creates a local pre-fetch cache with place for 16 rows. Unfortunately, DBD::Oracle does not support array interface, so the effects of this local cache are not as great as one would expect. As a matter of fact, this lack of support for Oracle array interface is the feature that severely limits the use of Perl with Oracle and makes it unsuitable for large data loads or massive transactions. Features that are also not supported are TAF (Transparent Application Failover) and direct loads. In other words, Perl with DBI and DBD::Oracle is not an industry strength tool for a production environment.

Executing SQL

Now that we have a database handle properly created and configured, we can use it to execute SQL commands. Oracle SQL executes in several phases:

Parse phase: SQL statement is checked for syntactical validity and all objects are checked to see whether they exist or not and whether the user has appropriate access privileges or not. It is during this phase that the optimizer is invoked, statistics examined and the optimal access path determined. This parsing operation can be excruciatingly expensive and is best avoided, if at all possible.

Bind phase: placeholders in SQL statements are “connected” to the program variables. During this process the address of program variables is “made known” to oracle, so that it can read or write values from it. The same SQL command can be executed over and over again, without the need for re-parsing for various values of the program variable.

Execute phase: In this phase, the SQL statement in question is executed and the appropriate program areas within the program are updated to reflect that. Oracle can postpone parsing until the statement is actually executed, therefore decreasing the number of calls to oracle and the number of round trips over the network, necessary to reach the database.

Define phase: define phase exists only for queries. During this phase we define variables to receive output. This phase is not really necessary with Perl DBI as DBI calls also create variables to receive data.

Fetch phase: during the fetch phase, data is retrieved from an Oracle cursor and stored into program variables. Fetch is not bi-directional, it can be read sequentially and closed when no longer needed.

These phases of execution correspond to DBI calls. In particular, “parse” phase corresponds to the prepare DBI call. Now is the right time to demonstrate the fabled “deferred parse:

Example 1.

```
#!/usr/bin/perl -w
use strict;
use DBI;
my $db = DBI->connect( "dbi:Oracle:Local", "scott", "tiger" )
    || die( $DBI::errstr . "\n" );
$db->{AutoCommit} = 0;
$db->{RaiseError} = 1;
$db->{ora_check_sql} = 0;
$db->{RowCacheSize} = 16;
my $SEL = "invalid SQL statement";
my $sth = $db->prepare($SEL);
print "If you see this, parse phase succeeded without a problem.\n";
$sth->execute();
print "If you see this, execute phase succeeded without a problem.\n";
END {
    $db->disconnect if defined($db);
}
```

The result of this script is the following:

```
bash-3.00$ /tmp/ttt
If you see this, parse phase succeeded without a problem.
DBD::Oracle::st execute failed: ORA-00900: invalid SQL statement (DBD ERROR: OCISStmtExecute) [for Statement "invalid SQL
statement"] at /tmp/ttt line 13.
DBD::Oracle::st execute failed: ORA-00900: invalid SQL statement (DBD ERROR: OCISStmtExecute) [for Statement "invalid SQL
statement"] at /tmp/ttt line 13.
```

As you can see, parse succeeded without a problem, although the SQL statement in the variable \$SEL was clearly invalid. There is one more thing in this script that needs to be explained, namely the END block. The END block gets executed when the script exits, regardless of the way it exits. So , let’s comment it out and see what happens:

```
bash-3.00$ /tmp/ttt
If you see this, parse phase succeeded without a problem.
DBD::Oracle::st execute failed: ORA-00900: invalid SQL statement (DBD ERROR: OCISStmtExecute) [for Statement "invalid SQL
statement"] at /tmp/ttt line 13.
DBD::Oracle::st execute failed: ORA-00900: invalid SQL statement (DBD ERROR: OCISStmtExecute) [for Statement "invalid SQL
statement"] at /tmp/ttt line 13.
Issuing rollback() for database handle being DESTROY'd without explicit disconnect().
bash-3.00$
```

The last message was issued by the DBI, because of exiting without disconnect. To avoid that, it is a good practice to include an END block like in the example 1. End block can also be used to issue rollback after a failed transaction. To summarize the example 1, a successful “connect” call established a database handle \$db. Database handle has a prepare method, used to create a statement handle, \$sth. Statement handle has many methods. Execute, bind and fetch are all methods of a statement handle.

Now, let’s replace the invalid SQL with a valid one, namely “select * from emp”. Our little script will need more then cosmetic changes. The script will now look like this:

Example 2.

```
#!/usr/bin/perl -w
use strict;
use DBI;
my $db = DBI->connect( "dbi:Oracle:Local", "scott", "tiger" )
    || die( $DBI::errstr . "\n" );
$db->{AutoCommit} = 0;
$db->{RaiseError} = 1;
$db->{ora_check_sql} = 0;
$db->{RowCacheSize} = 16;
my $SEL = "SELECT * FROM EMP";
my $sth = $db->prepare($SEL);
$sth->execute();

while ( my @row = $sth->fetchrow_array() ) {
    foreach (@row) {
        $_ = "\t" if !defined($_);
        print "$_\t";
    }
    print "\n";
}
```

```
END {  
  $db->disconnect if defined($db);  
}
```

The output looks exactly as expected:

```
bash-3.00$ ./ttt  
7369 SMITH CLERK 7902 17-DEC-80 800 20  
7499 ALLEN SALESMAN 7698 20-FEB-81 1600 300 30  
7521 WARD SALESMAN 7698 22-FEB-81 1250 500 30  
7566 JONES MANAGER 7839 02-APR-81 2975 20  
7654 MARTIN SALESMAN 7698 28-SEP-81 1250 1400 30  
7698 BLAKE MANAGER 7839 01-MAY-81 2850 30  
7782 CLARK MANAGER 7839 09-JUN-81 2450 10  
7788 SCOTT ANALYST 7566 09-DEC-82 3000 20  
7839 KING PRESIDENT 17-NOV-81 5000 10  
7844 TURNER SALESMAN 7698 08-SEP-81 1500 0 30  
7876 ADAMS CLERK 7788 12-JAN-83 1100 20  
7900 JAMES CLERK 7698 03-DEC-81 950 30  
7902 FORD ANALYST 7566 03-DEC-81 3000 20  
7934 MILLER CLERK 7782 23-JAN-82 1300 10  
bash-3.00$
```

So, now we not only have execute, we also have fetch, implemented by `fetchrow_array` method of the statement handle. The array `@row` is created on the fly, for each row, by the `fetchrow_array` method. Therefore, we do not have to do “define” as we would have had, had we been writing an OCI program instead of a Perl script.

There several more methods for fetching data from an Oracle cursor:

- `fetchrow_arrayref`
- `fetchrow_hashref`
- `fetchall_arrayref`,
- `fetchall_hashref`

These methods differ by what do they return (hash or array reference) and how many rows do they return (one or all). Methods returning hash reference are convenient when we want to pick column value by column name, not its sequence number. For methods that return reference to hash, retrieval works like this:

```
$row=$sth->fetchrow_hashref();  
$ename=$row->{ENAME};
```

Methods that fetch all rows are convenient when the underlying query returns relatively few rows that are frequently referenced within a script.

All those methods fetch relational NULL as “undef” value, which is a problem if we have “use strict” in effect. That is the reason for having the line which reads:

```
$_ = "\t" if !defined($_);
```

in the script. If that line is commented out, we get an ugly output, laced with warning messages, which looks like this:

```
bash-3.00$ ./ttt
Use of uninitialized value in concatenation (.) or string at line 17
7369 SMITH CLERK 7902 17-DEC-80 800
7499 ALLEN SALESMAN 7698 20-FEB-81 1600
7521 WARD SALESMAN 7698 22-FEB-81 1250
Use of uninitialized value in concatenation (.) or string at line 17
7566 JONES MANAGER 7839 02-APR-81 2975
7654 MARTIN SALESMAN 7698 28-SEP-81 1250
Use of uninitialized value in concatenation (.) or string at line 17
7698 BLAKE MANAGER 7839 01-MAY-81 2850
Use of uninitialized value in concatenation (.) or string at line 17
7782 CLARK MANAGER 7839 09-JUN-81 2450
Use of uninitialized value in concatenation (.) or string at line 17
7788 SCOTT ANALYST 7566 09-DEC-82 3000
Use of uninitialized value in concatenation (.) or string at line 17
Use of uninitialized value in concatenation (.) or string at line 17
7839 KING PRESIDENT 17-NOV-81 5000
7844 TURNER SALESMAN 7698 08-SEP-81 1500
Use of uninitialized value in concatenation (.) or string at line 17
7876 ADAMS CLERK 7788 12-JAN-83 1100
Use of uninitialized value in concatenation (.) or string at line 17
7900 JAMES CLERK 7698 03-DEC-81 950
Use of uninitialized value in concatenation (.) or string at line 17
7902 FORD ANALYST 7566 03-DEC-81 3000
Use of uninitialized value in concatenation (.) or string at line 17
7934 MILLER CLERK 7782 23-JAN-82 1300
bash-3.00$
```

The strict checking, enforced by the “use strict” pragma, will flag out any NULL values encountered by our script. That must be handled if we don’t want a warning message after each NULL value.

This was a very simple script, executing and retrieving results of the simplest possible query from the SCOTT.EMP table. The additional question one can immediately ask is the following:

I don’t know much about the EMP table and SCOTT demo schema. How many columns are returned, what are their names, types and sizes? How many rows were returned by the query? Of course, one solution is to scold the unfortunate wannabe programmer and tell him to go and read the fine manual, but RTFM is not always considered a polite and civilized solution, so I developed a DBI solution, presented in an extended version of the example 2:

Example 2a.

```
#!/usr/bin/perl -w
use strict;
use DBI;
my $db = DBI->connect( "dbi:Oracle:Local", "scott", "tiger" )
    || die( $DBI::errstr . "\n" );
$db->{AutoCommit} = 0;
$db->{RaiseError} = 1;
$db->{ora_check_sql} = 0;
$db->{RowCacheSize} = 16;
my $SEL = "SELECT * FROM EMP";
```

```
my $sth = $db->prepare($SEL);
$sth->execute();
my $nf = $sth->{NUM_OF_FIELDS};
print "This statement returns $nf fields\n";

for ( my $i = 0; $i < $nf; $i++ ) {
    my $name = $sth->{NAME}[$i];
    my $type = $sth->{TYPE}[$i];
    my $prec = $sth->{PRECISION}[$i];
    my $scl = $sth->{SCALE}[$i];
    my $tn=$db->type_info($type)->{TYPE_NAME};
    print
        "Field number $i: name $name of type $tn with precision $prec,$scl\n";
}
while ( my @row = $sth->fetchrow_array() ) {
    foreach (@row) {
        $_ = "\t" if !defined($_);
        print "$_\t";
    }
    print "\n";
}
print "\n\nThis query returned " . $sth->rows . " rows.\n";

END {
    $db->disconnect if defined($db);
}
```

Of course, the most interesting thing here is the output of this script, which looks like this:

```
This statement returns 8 fields
Field number 0: name EMPNO of type DECIMAL with precision 4,0
Field number 1: name ENAME of type VARCHAR2 with precision 10,0
Field number 2: name JOB of type VARCHAR2 with precision 9,0
Field number 3: name MGR of type DECIMAL with precision 4,0
Field number 4: name HIREDATE of type DATE with precision 75,0
Field number 5: name SAL of type DECIMAL with precision 7,2
Field number 6: name COMM of type DECIMAL with precision 7,2
Field number 7: name DEPTNO of type DECIMAL with precision 2,0
7369 SMITH CLERK 7902 17-DEC-80 800 20
7499 ALLEN SALESMAN 7698 20-FEB-81 1600 300 30
7521 WARD SALESMAN 7698 22-FEB-81 1250 500 30
7566 JONES MANAGER 7839 02-APR-81 2975 20
7654 MARTIN SALESMAN 7698 28-SEP-81 1250 1400 30
7698 BLAKE MANAGER 7839 01-MAY-81 2850 30
7782 CLARK MANAGER 7839 09-JUN-81 2450 10
7788 SCOTT ANALYST 7566 09-DEC-82 3000 20
7839 KING PRESIDENT 17-NOV-81 5000 10
7844 TURNERSALESMAN 7698 08-SEP-81 1500 0 30
7876 ADAMS CLERK 7788 12-JAN-83 1100 20
7900 JAMES CLERK 7698 03-DEC-81 950 30
7902 FORD ANALYST 7566 03-DEC-81 3000 20
7934 MILLER CLERK 7782 23-JAN-82 1300 10

This query returned 14 rows.
```

So, what is the answer to our question? Column name, column type, column precision and scale, as well as the number of columns returned are all attributes of the statement handle. Those attributes are defined when the statement is parsed. In our script, we bundled the parse phase with the execute phase, which means that those attributes are not defined until we execute the handle, which is a colloquial expression for “invoke the handle’s execute method”. If we have a valid statement handle \$sth, that handle automatically has the following elements:

- \$sth->{NUMBER_OF_FIELDS} is an integer field telling us how many columns does the query return, if the statement is a query.
- \$sth->{NAME} is an array, with \$sth->{NUMBER_OF_FIELDS} elements, each element being the column name of the corresponding returned column.
- \$sth->{TYPE} is also an array which contains column types. Column types are integers, because they're consistent across databases. Supported types depend on the version of the DBI and can be translated into type names by using the type_info() method as shown in the example 2a.
- \$sth->{PRECISION} and \$sth->{SCALE} are both arrays containing the information about the precision and the scale of the returned column. Note that the precision for VARCHAR2 column is its size. Its scale, of course is 0. The value of scale traditionally defines the number of decimal places in the number field.
- Last but not least is the rows method. Invoking \$sth->rows() will give you the number of rows fetched so far or the number of affected rows. This method is completely analogous to the PL/SQL ROWCOUNT attribute and is used in exactly the same way.
- There are, of course, other elements which can be located and studied on the online documentation page.

Generally speaking, statement handles are Oracle cursors, disguised as Perl objects. The whole philosophy of database handles and statement handles is the same as the philosophy of database connections and cursors: to open a cursor, one has to connect to a database. In the OO world of the modern programming languages, objects have “methods” and “properties” or “members”, so those are the terms that we must use when talking about DB handles and statement handles.

Now we know how to execute a simple SQL statement and get the basic information about it. For statements like “ALTER SESSION SET NLS_DATE_FORMAT='DD-MON-RR'”, there is also a shortcut:

```
$db=DBI->connect(“dbi:Oracle:Local”,“scott”,“tiger”);  
$db->do(“ALTER SESSION SET NLS_DATE_FORMAT='DD-MON-RR'”);
```

No prepare is necessary, no statement handle is available, the “do” method is a database handle method which fuses prepare and execute methods in a single step. That is convenient for the alter session commands or DDL commands. The next thing to learn is how to make a program variable “known” to Oracle.


```
$ename,          $job,          $dept,          $sal
.
END {
  $db->disconnect if defined($db);
}
```

This script produces the following output:

```
bash-3.00$ ./ttt
Enter EMPNO:7934
Employee Name   Job           Department    Salary
-----
MILLER         CLERK        ACCOUNTING    1300
```

In case that a non-existing employee number is entered, no output is produced. So, here we have a script that reads a value from the standard input, and needs that value to complete the SQL execution. The placeholder used here is a simple question mark: “?” :“WHERE empno=?”. That is a DBI specific trick, later emulated in other languages and frameworks, like PEAR::DB. The trick is that DBI uses so called positional bind, in which each question mark is a placeholder, replaced by its sequence number. It can’t be simpler then that, but there is a problem:

If our SQL statement was something like:

```
WHERE empno = ? and ?<10000
```

2 different placeholders would be needed. If a condition includes the same placeholder appearing in several places in the same SQL, positional binds can not be used, we have to resort to the named binds. Named binds differ from positional binds in such a way that placeholders are not simple question marks but are named instead. As a matter of fact, our first example was a part of a named bind sequence. The statement below

```
SELECT e.ename,e.job,d.dname,e.sal
FROM emp e, dept d
WHERE e.deptno=d.deptno AND
      e.empno=:EMPNO
```

will lead to `$sth->bind_param(":EMPNO", $empno)`; Other than that, positional binds are identical to the named ones. I also find named binds easier to follow and less confusing than a forest of question marks that I have to count by using my fingers in order to really understand the underlying SQL. I have only 10 fingers, which is somewhat limiting when it comes to really complex SQL statements. I also adopted the naming convention which mandates naming placeholders the same as the variables they’re being bound to, only in capital letters, just like in the code snippet above.

So far, Oracle has just been reading script variables and using them in SQL. What happens if Oracle needs to write something into a script variable? In other words, what happens if the placeholder is in place of the input/output

parameter in PL/SQL procedure? It turns out that we need somewhat more complex bind. In order to do that, let's take a look at the following PL/SQL script:

```
CREATE OR REPLACE PROCEDURE DBITEST(emp_no in number, d_name out varchar2)
AS
BEGIN
SELECT D.DNAME INTO D_NAME
FROM EMP E, DEPT D
WHERE E.EMPNO=EMP_NO AND
      D.DEPTNO=E.DEPTNO;
EXCEPTION
  WHEN NO_DATA_FOUND THEN
    D_NAME:='NO SUCH EMPLOYEE';
END;
/
```

In a Perl script, we would have to call it, by using a snippet like this:

```
my $SCR="begin
      dbitest(:EMPNO,:DNAME);
end;";
```

Unfortunately, the bind method we've studied so far cannot help us here. In other words, using calls like `$sth->bind_param(":ENAME",$ename)` and `$sth->bind_param(":DNAME", $dname)` would produce nothing but a nasty error message. Let's modify our example 3 and demonstrate:

Example 3a (Doesn't work)

```
#!/usr/bin/perl -w
use strict;
my $dname;
use DBI;
my $db = DBI->connect( "dbi:Oracle:Local", "SCOTT", "TIGER" )
  || die( $DBI::errstr . "\n" );
$db->{AutoCommit} = 0;
$db->{RaiseError} = 1;
$db->{ora_check_sql} = 0;
$db->{RowCacheSize} = 16;
my $SEL = "begin
          dbitest(:EMPNO,:DNAME);
          end;";

my $sth = $db->prepare($SEL);

print "Enter EMPNO:";
my $empno = <STDIN>;
chomp($empno);

$sth->bind_param( ":EMPNO", $empno );
$sth->bind_param( ":DNAME", $dname);
$sth->execute();
print "Department name: $dname\n";
```

```
END {  
  $db->disconnect if defined($db);  
}
```

The execution produces the following:

```
bash-3.00$ ./tst  
Enter EMPNO:7934  
DBD::Oracle::st execute failed: ORA-06502: PL/SQL: numeric or value error: character string buffer too small  
ORA-06512: at "SCOTT.DBITEST", line 4  
ORA-06512: at line 2 (DBD ERROR: OCISstmtExecute) [for Statement "begin  
  dbitest(:EMPNO,:DNAME);  
  end;" with ParamValues: :empno='7934', :dname=undef] at ./tst line 23, <STDIN> line 1.  
DBD::Oracle::st execute failed: ORA-06502: PL/SQL: numeric or value error: character string buffer too small  
ORA-06512: at "SCOTT.DBITEST", line 4  
ORA-06512: at line 2 (DBD ERROR: OCISstmtExecute) [for Statement "begin  
  dbitest(:EMPNO,:DNAME);  
  end;" with ParamValues: :empno='7934', :dname=undef] at ./tst line 23, <STDIN> line 1.  
bash-3.00$
```

That is not nice. So, how do we fix that? DBI has another type of bind method, used for PL/SQL procedures, LOB parameters or ref cursors. That method is called `bind_param_inout`. There are two important differences between `bind_param` and `bind_param_inout`:

- `Bind_param` takes a program variable as an argument while `bind_param_inout` takes a REFERENCE to program variable as an argument. In other words, we're passing a pointer to variable to Oracle, not the variable itself.
- Specifying the maximum length of the bind variable is mandatory for `bind_param_inout`.

Bind using the value of the variable, like `bind_param` are also called binds by value, as the value of the variable is made available to Oracle. Binds using `bind_param_inout` are also known as binds by reference or binds by address as reference (address) of a Perl variable is made known to Oracle.

So, let's now modify our example 3a to use bind by reference:

Example 3b

```
#!/usr/bin/perl -w  
use strict;  
my $dname;  
use DBI;  
my $db = DBI->connect( "dbi:Oracle:TESTDB01", "mgdba", "qwerty" )  
  || die( $DBI::errstr . "\n" );  
$db->{AutoCommit} = 0;  
$db->{RaiseError} = 1;  
$db->{ora_check_sql} = 0;  
$db->{RowCacheSize} = 16;  
my $SEL = "begin  
  dbitest(:EMPNO,:DNAME);  
  end;"
```

```
my $sth = $db->prepare($SEL);

print "Enter EMPNO:";
my $empno = <STDIN>;
chomp($empno);

$sth->bind_param_inout( ":EMPNO", \$empno, 20);
$sth->bind_param_inout( ":DNAME", \$dname, 20 );
$sth->execute();
print "Department name: $dname\n";

END {
    $db->disconnect if defined($db);
}
```

The execution now proceeds in much nicer fashion:

```
bash-3.00$ ./ttt
Enter EMPNO:7934
Department name: ACCOUNTING
```

The only difference is the use of `bind_param_inout` instead of `bind_param` in the “b” version. Can the “inout” version of the call be used for simple querying as in the example 3? Yes, it can, but it is simpler to use binds by value then binds by address. Interestingly enough, both binds by value and binds by address (binds by reference) support positional binds, with “?” for placeholders. Binds by reference also have to be used with `RETURNING INTO` clause as a part of an `INSERT` statement.

So far, this article has covered connecting to Oracle, simple fetches, binding variables to placeholders and executing PL/SQL scripts. To really spice it up, this article will also cover working with LOB data types and with REF cursors.

LOB Types and REF Cursors

REF cursors are an extremely handy way of returning sets of records. Before proceeding with Perl scripting, let’s first create PL/SQL code that will help us demonstrate the concept:

```
CREATE OR REPLACE PACKAGE dbi AS
    type refcsr IS ref CURSOR;
    PROCEDURE test(dno IN NUMBER,  csr OUT refcsr);
END;
/

CREATE OR REPLACE PACKAGE BODY dbi AS
    PROCEDURE test(dno IN NUMBER,  csr OUT refcsr) IS
    BEGIN

    OPEN csr FOR
```

```
SELECT ename,  
       job,  
       sal,  
       hiredate  
FROM emp  
WHERE deptno = dno;  
END;  
END;  
/
```

Let's say we want to execute the following PL/SQL code in our Perl script:

```
BEGIN  
    DBI.TEST(:DNO,:CSR);  
END;
```

The key to success lies in understanding what exactly is our REF cursor variable. REF Cursor variables are Oracle's way of returning a statement handle. I strongly believe that an example is worth a thousand words, so here it is:

Example 4

```
#!/usr/bin/perl -w  
use strict;  
use DBI;  
  
# Note inclusion of DBD::Oracle data types. Without it, we cannot  
# work with REF Cursor variables.  
  
use DBD::Oracle qw(:ora_types);  
  
my $db = DBI->connect( "dbi:Oracle:Local", "scott", "tiger" )  
    || die( $DBI::errstr . "\n" );  
$db->{AutoCommit} = 0;  
$db->{RaiseError} = 1;  
$db->{ora_check_sql} = 0;  
$db->{RowCacheSize} = 16;  
  
my $dno=10;  
my $csr;  
my ($ename, $job, $sal, $hiredate);  
  
my $SEL = "begin  
           dbi.test(:DNO,:CSR);  
           end;";  
  
my $sth = $db->prepare($SEL);  
  
$sth->bind_param_inout( ":DNO", \$dno, 20);  
# Now comes the important part....  
$sth->bind_param_inout( ":CSR", \$csr, 0, { ora_type => ORA_RSET } );  
$sth->execute();
```

```
print "\nNow, we have a valid handle...\n\n";
# Observe that here we are fetching from csr, a handle which was
# prepared and executed by the DBI.TEST procedure.
while ( ($ename, $job, $sal, $hiredate)=$csr->fetchrow_array() {
    write;
}
no strict;
format STDOUT_TOP =
Employee      Job      Hiredate      Salary
-----
.

format STDOUT =
@<<<<<<<<<< @<<<<<<< @<<<<<<<<< @<<<<<<<
$ename,      $job,    $hiredate,    $sal
.

END {
    $db->disconnect if defined($db);
}
```

When executed, this code will produce the following output:

```
bash-3.00$ ./ttt
Now, we have a valid handle...
Employee      Job      Hiredate      Salary
-----
CLARK         MANAGER   09-JUN-81     2450
KING          PRESIDEN  17-NOV-81     5000
MILLER        CLERK     23-JAN-82     1300
```

So, what can we see in this example? First, we had to explicitly include DBD::Oracle, with the type definitions that are not normally exported by the module. Second, we had to do an “inout” bind, with size set to 0 and with a hash reference which tells DBI the oracle type of the variable being bound. Third, REF Cursor variables are, in essence, statement handles. When a PL/SQL procedure returns REF Cursor variable, it is just the same as if the handle was produced by using \$db->prepare.

What is the most frequent use of REF Cursor variables? I’m a database administrator and I’ve never had to use them, I only needed to support them when the real programmers were using them. The most frequent situations involving use of the REF Cursor variables were migration projects where the source database used result sets as a return from a stored procedure. Databases that are known for encouraging such development style are Sybase and SQL Server. Any migration of a program which runs against SQL Server or Sybase to a Perl script that runs against Oracle database is likely to need REF Cursor variables. Of course, this part of the article would not be complete without explanation of the “real programmers” reference. For those among you who want to become real programmers, here are some definitions:

<http://www.toodarkpark.org/textfile.cgi/computers/humor/real-programmers>

I can only wish you luck with your further career endeavors. Real programmers aside, let’s return to DBI and

Oracle. REF Cursor is a special column type that warrants inclusion of DBD::Oracle directly, in order to gain access to the definition of types. The same is true for LOB data types. LOB is an abbreviation for “Large Objects”. LOB types are used when we need to store large documents, images or other types of media files into the database. Good thing about the LOB data types is that it is mostly handled automatically by the DBI itself. There are two separate cases which need handling: inserting LOB data and selecting LOB data. To demonstrate loading LOB data into the database, we'll need some database infrastructure:

```
SQL> create table dbi_lob (
  2 id number(10,0) constraint dbi_lob_pk primary key,
  3 name varchar2(128) not null,
  4 data blob);

Table created.

SQL> create sequence dbi_lob_seq start with 1 nocycle nomaxvalue;

Sequence created.

SQL> create trigger dbi_log_pk_trg
  2 before insert on dbi_lob
  3 for each row
  4 begin
  5 select dbi_lob_seq.nextval into :new.id
  6 from dual;
  7 end;
  8 /

Trigger created.
```

Unfortunately, normal Oracle demo tables, belonging to SCOTT do not contain LOB columns. No problems, we have just created everything needed for such a demo. The next script will read the file name from the command line argument and read the whole file into a memory buffer. That memory buffer will then be inserted as a BLOB into the database.

Example 5

```
#!/usr/bin/perl -w
use strict;
use DBI;
use DBD::Oracle qw(:ora_types);
use Getopt::Long;
my ( $user, $passwd, $db ) = ( "scott", "tiger", "local" );
my ( $file, $id );
my $stat = GetOptions( "u|user=s" => \$user,
                      "p|password=s" => \$passwd,
                      "d|database=s" => \$db,
                      "f|file=s" => \$file,
                      "h|help|?" => \&usage
);
if ( !defined($file) or !$stat ) { usage(); }
my $dbh = db_connect( $user, $passwd, $db );
```

```
my $INS = "INSERT INTO DBI_LOB(NAME,DATA) VALUES (:FL,:BUFF)
          RETURNING ID INTO :ID";
my $sth = $dbh->prepare($INS);
$sth->bind_param( ":FL", $file );
$sth->bind_param_inout( ":ID", \$id, 20 );
my $buff = gobble($file);
$sth->bind_param( ":BUFF", $buff, { ora_type => ORA_BLOB } );

$sth->execute();

print "BLOB ID=$id inserted\n";
$dbh->commit();

END {
    $dbh->disconnect() if defined($dbh);
}

sub db_connect {
    my ( $username, $passwd, $db ) = ( @_, $ENV{"TWO_TASK"} );
    my $dbh = DBI->connect( "dbi:Oracle:$db", $username, $passwd )
        || die( $DBI::errstr . "\n" );
    $dbh->{AutoCommit} = 0;
    $dbh->{RaiseError} = 1;
    $dbh->{ora_check_sql} = 0;
    $dbh->{RowCacheSize} = 16;
    return ($dbh);
}

sub usage {
    print qq(
        USAGE:$0
        -u <username> -p <password > -d <database>
        -f <file name>

        -----
        This script loads file defined by -f argument into database, as a
        BLOBS. This script was written as an educational tool and is free
        to use and modify as needed.
    );
    exit(0);
}

sub gobble {
    my $file = shift;
    local $/ = undef;
    if ( !defined($file) ) { usage(); }
    open( FL, "<", $file ) or die "Cannot open file $file for reading:$!\n";
    my $buff = <FL>;
    close FL;
    return ($buff);
}
```

The execution of this script produces very unspectacular, if expected results:

```
$. /load_lobs -f adodb480.tgz
BLOB ID=1 inserted
$. /load_lobs -f php-5.1.2.tar.bz2
BLOB ID=2 inserted
```

So, lets check our work:

```
$ ls -l adodb480.tgz php-5.1.2.tar.bz2
-rw-r--r-- 1 500 6319905 Mar 30 21:04 php-5.1.2.tar.bz2
-rw-r--r-- 1 500 452518 Mar 30 21:04 adodb480.tgz

SQL> column name format a25
SQL> select name,length(data) as filesize from dbi_lob;

NAME                FILESIZE
-----
adodb480.tgz        452518
php-5.1.2.tar.bz2  6319905
```

The number of bytes in the database is equal to the number of bytes produced by the “ls -l” command, which means that both files are successfully loaded into my little home database.

This script inserts binary files into our little DBI_LOB table. The important things are the following:

Bind of the large memory buffer, containing file content (“\$buff”) was done by value, rather than by reference. That way, I didn't have to specify length of the data, just the type was sufficient. BLOB variables aren't much different from VARCHAR2 or NUMBER variables, they're just bigger and the only thing that needs to be done is to specify the type, to warn Oracle that something big is coming its way.

RETURNING INTO clause saves us trip to over the network, but the target needs to be bound by reference, using “bind_param_inout” method.

I advocated the naming convention about naming placeholders and variables and violated it two scripts later. Why did I do that? The answer is very simple: placeholder :FILE is reserved and cannot be used. The truth of the matter is that I've hit the error below:

```
DBD::Oracle::st execute failed: ORA-01745: invalid host/bind variable name (DBD ERROR: error possibly near <*> indicator at char 48 in
'INSERT INTO DBI_LOB(ID,NAME,DATA) VALUES (NULL,;<*>file,:buff) RETURNING ID INTO :id')
```

I distinctly remember using the bind variable :FILE earlier in my career, but there was no other recourse here except changing the placeholder name. Things like that depend upon the version of DBI, the version of DBD::Oracle and the version of Oracle RDBMS used. Obviously, there are still things to discover about this particular combination. This error forced me to break my own rule. Such is life.

That concludes the section about inserting LOB variables into database. How do we get them out of the database? Database is not a black hole, what comes in, has to come out sooner or later.

Few lines earlier, I said that LOB variables can be thought of as big VARCHAR2 variables. That especially applies to selecting them from the database. We can use normal binds, LOB variable is treated as any other variable would be. There are, however, some things that need careful handling. It is time to learn about some other properties of the database handles. Here they are, taken directly from the DBI online documentation:

LongReadLen (unsigned integer, inherited)

The LongReadLen attribute may be used to control the maximum length of 'long' type fields (LONG, BLOB, CLOB, MEMO, etc.) which the driver will read from the database automatically when it fetches each row of data.

The LongReadLen attribute only relates to fetching and reading long values; it is not involved in inserting or updating them.

A value of 0 means not to automatically fetch any long data. Drivers may return undef or an empty string for long fields when LongReadLen is 0.

The default is typically 0 (zero) bytes but may vary between drivers. Applications fetching long fields should set this value to slightly larger than the longest long field value to be fetched.

Some databases return some long types encoded as pairs of hex digits. For these types, LongReadLen relates to the underlying data length and not the doubled-up length of the encoded string.

Changing the value of LongReadLen for a statement handle after it has been prepare'd will typically have no effect, so it's common to set LongReadLen on the \$dbh before calling prepare.

For most drivers the value used here has a direct effect on the memory used by the statement handle while it's active, so don't be too generous. If you can't be sure what value to use you could execute an extra select statement to determine the longest value.

LongTruncOk (boolean, inherited)

The LongTruncOk attribute may be used to control the effect of fetching a long field value which has been truncated (typically because it's longer than the value of the LongReadLen attribute).

By default, LongTruncOk is false and so fetching a long value that needs to be truncated will cause the fetch to fail. (Applications should always be sure to check for errors after a fetch loop in case an error, such as a divide by zero or long field truncation, caused the fetch to terminate prematurely.)

If a fetch fails due to a long field truncation when LongTruncOk is false, many drivers will allow you to continue fetching further rows.

These attributes are typically set for the database handle, immediately after the connection is established. This is the reason for having separated connecting to database in a separate subroutine in the last script. The example which will fetch LOB data from the database will use the same skeleton, command line arguments, usage and everything else. It will only modify subroutine db_connect and the SQL executed within the script. Cut and paste between scripts works really well with Perl and that is one more reason for loving it.

So, in the example 6, LongTruncOk will be set to false and LongReadLen will be set to 5 MB The first file (ID=1) is around 400k in length and it will not make any problems. The other file (ID=2) is, however, more then 6 MB long and will help us to demonstrate what happens when the length of the LOB is larger then LongReadLen. Without further ado, here it is:

Example 6

```
#!/usr/bin/perl -w
use strict;
use DBI;
use Getopt::Long;
```

```
my ( $user, $passwd, $db ) = ( "scott", "tiger", "local" );
my ( $id, $blob, $file ) = (1);
my $stat = GetOptions( "u | user=s"    => \$user,
                      "p | password=s" => \$passwd,
                      "d | database=s" => \$db,
                      "i | id=s"      => \$id,
                      "h | help!"     => \&usage
);
if ( !$stat ) { usage(); }
my $dbh = db_connect( $user, $passwd, $db );
my $SEL = "SELECT name,data FROM dbi_lob WHERE id=:ID";
my $sth = $dbh->prepare($SEL);
$sth->bind_param( ":ID", $id );
$sth->execute();
($file,$blob)=$sth->fetchrow_array();
spew($file,$blob);
print "File $file written successfully. Len=",length($blob),"\\n";

END {
    $dbh->disconnect() if defined($dbh);
}

sub db_connect {
    my ( $username, $passwd, $db ) = ( @_, $ENV{"TWO_TASK"} );
    my $dbh = DBI->connect( "dbi:Oracle:$db", $username, $passwd )
        || die( $DBI::errstr . "\\n" );
    $dbh->{AutoCommit} = 0;
    $dbh->{RaiseError} = 1;
    $dbh->{ora_check_sql} = 0;
    $dbh->{RowCacheSize} = 16;
    $dbh->{LongReadLen} = 5242880;
    $dbh->{LongTruncOk} = 0;
    return ( $dbh );
}

sub usage {
    print qq(
        USAGE:$0
        -u <username> -p <password > -d <database>
        -i id

        -----
        This script unloads blob with id defined by -i parameter from the
        database. This script was written as an educational tool and is
        free to use and modify as needed.
    );
    exit(0);
}

sub spew {
    my $file = shift;
    my $buff= shift;
    if ( !defined($file) || (ref($buff) ne 'SCALAR') ) { usage(); }
    open( FL, ">", $file ) or die "Cannot open file $file for writing:#!\\n";
    printf FL ("%s", $$buff);
    close FL;
}
}
```

Unloading of adodb480.tgz goes without a problem:

```
$ rm -i adodb480.tgz
rm: remove regular file `adodb480.tgz'? y

$ ./unload_lob -i 1
File adodb480.tgz written successfully. Len=452518

$ tar ztvf adodb480.tgz | head -5
-rw-rw-rw- 0/0      12458 2006-03-10 03:58:55 adodb/adodb-active-record.inc.php
-rw-rw-rw- 0/0      8304 2006-03-10 03:58:54 adodb/adodb-csvlib.inc.php
-rw-rw-rw- 0/0     21041 2006-03-10 03:58:55 adodb/adodb-datadict.inc.php
-rw-rw-rw- 0/0      8541 2006-03-10 03:58:56 adodb/adodb-error.inc.php
-rw-rw-rw- 0/0      2748 2006-03-10 03:58:56 adodb/adodb-errorhandler.inc.php
```

Not only is the file size correct, tar and gunzip utilities also work correctly. Now, let's try with the `id=2`, the source code for PHP 5.1.2, which exceeds 5 MB, the value that we've set our `LongReadLen` to:

```
$ ./unload_lob -i 2
DBD::Oracle::st fetchrow_array failed: ERROR fetching field 3 of 2. LOB value truncated from 6319905 to 5242880. DBI attribute LongReadLen too small and/or LongTruncOk not set [for Statement "SELECT name,data FROM dbi_lob WHERE id=:ID" with ParamValues: :id='2'] at ./unload_lob line 19.
DBD::Oracle::st fetchrow_array failed: ERROR fetching field 3 of 2. LOB value truncated from 6319905 to 5242880. DBI attribute LongReadLen too small and/or LongTruncOk not set [for Statement "SELECT name,data FROM dbi_lob WHERE id=:ID" with ParamValues: :id='2'] at ./unload_lob line 19.
```

DBI killed our script because `RaisError` was set and `LongTruncOk` was not set. This is a desired outcome. If `LongTruncOk` was set, the script would have produced a shortened version of the original file which would be useless. Unfortunately, `LongReadLen` can not be set to `-1` or any other catch-22 value, to prevent the handler from kicking in. If you're unloading LOB columns from database into a Perl script, the script must have enough memory space available to accommodate the largest of the LOB columns. In all other respects, LOB columns are just like the other `VARCHAR2` or `NUMBER` columns, just larger. In this case, size does matter.

This is not the end of the story. We can work with LOB columns on piece by piece base, using either `DBMS_LOB` database package or specific abilities for `DBD::Oracle`, as described in the online documentation. This is, however, an advanced use of BLOB fields, which has no place in an introductory document like this. Advanced LOB handling capabilities of `DBD::Oracle` are beyond the intended scope of this document.

Oracle Array Interface

What is Oracle array interface and what does it do for us? Oracle array interface is a part of Oracle API which allows executing oracle calls with array arguments, instead of the scalar ones. If a Perl script contains array @ARR, the array interface would enable us to insert it into the database with a single call, not having to call the “execute” method for every element of the array. That, of course, can save a lot of time and round trips over the network. The primary purpose of the array interface is to make data loads and large reports much faster. DBI has long had array interface but, prior to the version 1.18a of DBD::Oracle, it didn't use Oracle's native array interface. DBD::Oracle 1.18a was published in July 2006 and it is a very promising step. So, how do we use the array interface? An example is worth a thousand words:

```
#!/usr/bin/perl -w
use strict;
use DBI;
use Data::Dumper;
my ( @empno, @ename, @job, @mgr, @hiredate );
my $SEL = qq(select empno,ename,job,mgr,hiredate
    from emp);
my $INS = qq(insert into emp_test(empno,ename,job,mgr,hiredate)
    values(?,?,?,?));
my $SESS =
    "alter session set events='10046 trace name context forever , level 12'";
my $dbh = db_connect( "scott", "tiger" );
$dbh->do($SESS);
my $sel = $dbh->prepare($SEL);
my $ins = $dbh->prepare($INS);
$sel->execute();

while ( my @row = $sel->fetchrow_array() ) {
    push @empno, $row[0];
    push @ename, $row[1];
    push @job, $row[2];
    push @mgr, $row[3];
    push @hiredate, $row[4];
}
$ins->bind_param_array( 1, \@empno );
$ins->bind_param_array( 2, \@ename );
$ins->bind_param_array( 3, \@job );
$ins->bind_param_array( 4, \@mgr );
$ins->bind_param_array( 5, \@hiredate );
$ins->execute_array( { ArrayTupleStatus => \my @tuple_status } );

# print Dumper(@tuple_status);
$dbh->commit();

END {
    $dbh->disconnect if defined($dbh);
}

sub db_connect {
    my ( $username, $passwd, $db ) = ( @_, $ENV{"TWO_TASK"} );
    my $dbh = DBI->connect( "dbi:Oracle:$db", $username, $passwd )
```

```
    || die( $DBI::errstr . "\n" );
$dbh->{AutoCommit} = 0;
$dbh->{RaiseError} = 1;
$dbh->{ora_check_sql} = 0;
$dbh->{ora_array_chunk_size} = 16;
return ($dbh);
}
```

There are two parts to the array interface: select and update. Select part is activated by setting the `ora_array_chunk_size` attribute on the statement handle level or, as in the above script, on the database handle level. Examining the trace shows that there has been only one fetch, fetching all 14 rows. The logic of the program didn't need any change or modifications. It just works. The value of the `ora_array_chunk_size` attribute determines the number of rows that will be fetched at once, into an internal buffer. Further "fetchrow_array" calls will first retrieve the contents of the internal buffer, before going to the database again, to fetch the next batch.

Insert was a different story. Insert doesn't support binding to the named variables yet, so the positional "?" placeholders had to be used. Arrays had to be bound to the placeholders using `bind_param_array` call, and then executed using the `execute_array` call. The trace file looks like this:

```
select empno,ename,job,mgr,hiredate
      from emp

call  count    cpu  elapsed    disk  query  current  rows
-----
Parse   1   0.01   0.04     0     0     0     0
Execute 1   0.00   0.00     0     0     0     0
Fetch  1   0.00   0.01     2     3     0    14
-----
total   3   0.01   0.06     2     3     0    14

Misses in library cache during parse: 1
Optimizer mode: ALL_ROWS
Parsing user id: 50

Rows   Row Source Operation
-----
    14  TABLE ACCESS FULL EMP (cr=3 pr=2 pw=0 time=15918 us)

*****

insert into emp_test(empno,ename,job,mgr,hiredate)
      values(:p1,:p2,:p3,:p4,:p5)

call  count    cpu  elapsed    disk  query  current  rows
-----
Parse   1   0.00   0.00     0     0     0     0
Execute 1   0.02   0.17     2     8     5    14
Fetch   0   0.00   0.00     0     0     0     0
-----
total   2   0.02   0.17     2     8     5    14

Parsing user id: 50
```

The important lines are marked by the underlining and the bold font. They show a single fetch of 14 rows and a single insert of the 14 rows. Prior to the DBD::Oracle version 1.18a, the trace file would have shown 14 fetches of a single row and 14 executions of a single row insert. If your version of DBD::Oracle is earlier than 1.18a, you can execute the example script above before and after the upgrade to the version 1.18a and see the difference for yourself.

Question of Life, Universe & Everything

This is the ending part of this, hopefully useful, document in which the author rides into sunset, after few remarks and wisecracks. To achieve the true Hollywood happy end, I must list and comment on the unsupported features and the most appropriate use of Perl and Oracle combination. Without further ado, here are the unsupported features that I consider important:

- Transparent application failover (TAF).
- Direct load/unload

Transparent failover capabilities are very important. Assembling RAC database is no longer excruciatingly expensive. Nowadays, it can be done by anyone willing to invest in 2 PC boxes with at least 1GB per PC, FireWire or multi-host SCSI adapter, corresponding external storage device, two additional network interfaces (preferably GigaBit Ethernet) and a dedicated network switch. That price is well within reach of an ordinary DBA like me, without having to give up food and gas for six months. As a result, commercial RAC configurations abound these days. That makes TAF more important than ever. What is TAF? Short answer to this question is that it is a mechanism that makes it possible for an application to continue without reconnecting to the database after the instance to which the application was connected fails. TAF capabilities have several levels. The lowest level is that the session content is preserved, while the highest level implies moving the session transparently to another instance, without losing the current transaction. In other words, the user wouldn't even notice that the instance to which he was connected has just failed. Unfortunately, very few interfaces which weren't written by Oracle Corp. support TAF. The authors and maintainers of DBD::Oracle are putting in the best effort to provide the Perl community with those valuable missing features. Recent addition of the support for the array interface is a giant step in the right direction. That makes it possible to parse and load multi-megabyte files into an Oracle database, without the performance penalty imposed by the DBI. In other words, DBD::Oracle can load with the best of them.

Direct load/unload capabilities are very important for the loads of the extremely large files, tens of gigabytes in size. Fortunately, there are relatively few of those to load and they're almost always loaded using SQL*Loader, without the need for additional parsing. Nevertheless, it would be a nice thing to have. Nice, but not necessarily urgent.

What is the most appropriate use of Oracle and Perl combination? Perl is extremely useful wherever parsing capabilities are needed. Its unique regular expression capabilities make it possible to quickly parse a file and load it to the database or to create an intelligent report very quickly. Perl is a general programming language, supported

by an incredible wealth of modules and tools on CPAN. Perl is a better screwdriver and it can be used with Oracle in the very much the same way as anywhere else.

Finally, what is the answer to the question of life, universe and everything? 42, of course. You should have already known that!