

An Introduction to DBD::Oracle

A step-by-step guide to the Database-independent interface for Perl (DBI) and the Oracle database driver for the DBI module DBD::Oracle

Author: John Scoles <scoles@pythian.com>
Version: 1.0.0
Date: 2007-03-30
Copyright: [Creative Commons Attribution-ShareAlike 2.5 License](#)

Contents

- 1 Preface
- 2 Oracle Database XE
- 3 DBI and DBD::Oracle
- 4 Using DBI and DBD::Oracle
 - 4.1 Understanding DBI
 - 4.2 Handles
 - 4.2.1 Driver Handles
 - 4.2.2 Database Handles
 - 4.2.3 Statement Handles
 - 4.3 Connecting and Disconnecting to 10XE
 - 4.3.1 Data Source Name (dsn)
 - 4.3.2 Disconnecting from Oracle
 - 4.4 Error Handling
 - 4.5 Sample Queries
 - 4.5.1 Queries That Return No Rows
 - 4.5.2 Binding Parameters to Statements
 - 4.5.3 Using Simple Placeholders
 - 4.5.4 Complex Placeholders
 - 4.5.5 Fetching Data
 - 4.5.6 Atomic and Batch Fetching
 - 4.5.6.1 Atomic Fetching
 - 4.5.6.2 Batch Fetching
 - 4.6 DBI Transactions
 - 4.6.1 PL/SQL Procedures and Functions
 - 4.6.2 PL/SQL CURSORS

[4.7 Large Objects \(CLOBs and BLOBs\)](#)

[4.8 XML](#)

[5 Summary](#)

1 Preface

This document assumes a working installation of:

- Oracle XE
- Perl 5.8
- Perl::DBI
- DBI::Oracle

2 Oracle Database XE

Oracle Database XE is an entry-level, small footprint starter database. It is free to download, develop, and deploy, and you can freely distribute it with your applications. There are no database licensing costs.

Oracle Database XE is built using the same code base as Oracle Database 10g Release 2 products (Standard Edition and Enterprise Edition), and is available on 32-bit Windows platforms and Linux platforms. It is a good choice for developers working with:

- Perl
- PHP
- Java
- .NET
- and other applications that require a database

It is suitable also for:

- DBAs who need a free starter database for training and deployment
- Independent Software Vendors (ISVs) who want to embed an Oracle database in their application product
- educational institutions and students who need a free starter Oracle database

Database XE includes the following programming interfaces:

- SQL, PL/SQL
- Java, C, and PHP
- Windows .NET

- Oracle Application Express
- C++, ODBC, OLE DB

It has the following limitations:

- It may only use up to 4GB of user data.
- It provides only a single database instance.
- It may be deployed only on a single CPU.
- 1GB RAM used

Oracle Database XE has a browser-based management interface, Oracle Application Express. Support is provided through an OTN discussion forum via peers and product experts.

3 DBI and DBD::Oracle

DBI has been the standard database access module for the Perl for over 10 years. It is used by many thousands of programmers supporting a large number of databases.

DBI is an interface that defines a set of standard methods, variables, and conventions that provide a consistent database interface independent of the actual database being used. It provides a layer of “glue” between an application and one or more database driver modules. In the case of this tutorial, the module is DBD::Oracle.

4 Using DBI and DBD::Oracle

4.1 Understanding DBI

Since its inception by Tim Bunce in 1994, DBI (DataBase Interface) has been the *de facto* standard for interfacing databases with the Perl language. Thanks to the designers’ foresight, DBI is database-independent, with methods, variables, and conventions that provide a consistent programming interface, no matter what database management system is being used.

As an API, DBI is just an interface to a database driver--in our case, DBD::Oracle--so you can think of DBI as the glue that links the Perl language to a specific database driver, as seen in the diagram below.

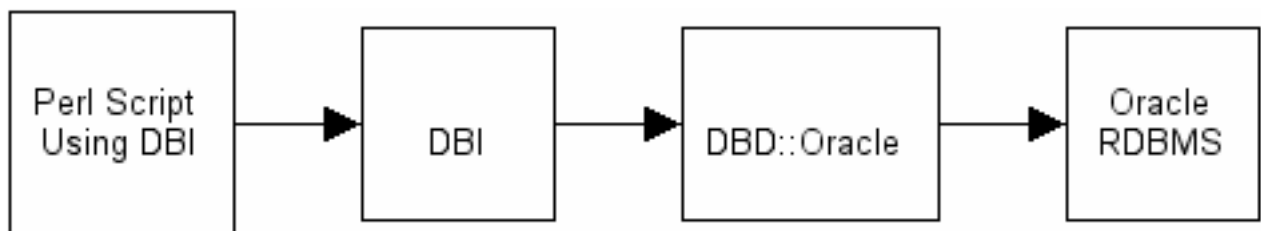


Figure 1: DBD::Oracle Diagram

4.2 Handles

The objects that DBI creates and which DBD::Oracle uses to interact with Oracle are commonly known as **handles**. There are three handles used in DBI:

1. **driver handles**, which can create
2. **database handles**, which can in turn create
3. **statement handles**.

4.2.1 Driver Handles

Driver handles are not normally referenced by programmers as they are loaded when DBI is initialized. The great thing about DBI is that the individual drivers are completely encapsulated within a handle. So it is possible and sometimes very useful to have more than one driver handle loaded at the same time. For example, if you need to transfer data from an Informix database into an Oracle database, it is very easy to write a single Perl script that connects to both and passes data between them.

4.2.2 Database Handles

These handles are an encapsulation of a single database connection and are always the children of the driver handle that created them. As with their parents, you can spawn any number of different connections to a single driver handle. In the example above, you could use two connections (database handles) for both Oracle and Informix DBs to account for an even more complicated data structure. Database handles are also thread-safe, so an individual handle will not leak into other handles. Database handles are by convention called `$dbh` in Perl scripts.

4.2.3 Statement Handles

Statement handles are the real meat of the DBI system, and as the name implies, they encapsulate single SQL statements to be executed on the database. Like the other handles, all statement handles are children of the database handle that created them, and like their parents, data will not leak out of them and into other handles. Statement handles are conventionally called `$sth` in Perl scripts.

4.3 Connecting and Disconnecting to 10XE

One of the great advantages of the Oracle platform is the myriad ways the DBA can connect to it. However, this is usually a bone of contention with many users, as we have all seen `ORA-12514 TNS:listener` once in a while.

To connect to Oracle with DBD, use DBI's `connect()` method:

```
$dbh = DBI->connect(dsn, user name, user password, {Attributes});
```

4.3.1 Data Source Name (dsn)

Like all database APIs, DBI needs to be told how to connect to a database, where to find the database, and how to log into the database. In this tutorial, we are going to use the DBD::Oracle driver to tell DBI how to connect. DBI has its own simple syntax for specifying the DB Driver. It is `dbi`, followed

by a colon :, then name of the driver, `Oracle`, followed by another colon :. So our data source will start with `dbi:Oracle:`. It is important to remember that this value is case-sensitive.

As we are using a local install of the 10XE database, it is best to keep the connection as simple as possible. So the connection call will look like this:

```
$dbh = DBI->connect(dbi:Oracle:, 'system', 'your_password');
```

In this case, DBI uses `DBD::Oracle` to connect to the database installed on the local computer using the userid “system” and the password “system”. DBI also lets you connect in any number of other ways, depending on how you have set up your Oracle listener. For example:

```
$dbh = DBI->connect(dbi:Oracle:XE, 'system', 'your_password');
```

will also work (assuming you have not changed the default value in the default `tnsnames.ora` file).

```
$dbh = DBI->connect(dbi:Oracle:, 'system@XE', 'your_password');
```

Will also work, and even

```
$dbh = DBI->connect(dbi:Oracle: 'system@localhost/XE', 'your_password');
```

At connection time, you may also set specific attributes for that connection, including some that are specific to particular DBD drivers. These values can control how errors are handled, and how data is committed. The `DBD::Oracle` driver supports all of the standard attributes and has no custom attributes itself. For a complete list of the attributes, see the `DBD::Oracle` Perl doc.

4.3.2 Disconnecting from Oracle

Though DBI does not require you to explicitly disconnect a database connection before your program ends, it is always good practice to do so. DBI has two ways to disconnect. The first uses the database handle.

```
$dbh->disconnect();
```

Or you could use the `DESTROY()` method:

```
$dbh->DESTROY();
```

The first is the more common practice. It is also good form to disconnect with the statement handle.

```
$sth->finish();
```

in tandem with `disconnect()` as this will end any processes that may be ongoing in the statement handle. This will stop DBI from throwing warning errors if a statement is still running when you disconnect from Oracle.

4.4 Error Handling

One of DBI’s most useful features is that it reports all errors to a single variable on the database handle.

```
$dbh->errstr
```

Once you are connected to Oracle, the vast majority of errors will either be some sort of SQL error, in which case `errstr` will show you the error message that Oracle generated and even give you the ORA number, if any. The next most common error DBI will throw is some form of syntax error in combination with the SQL you are calling.

DBI gives you two options for error trapping. The default is called `PrintError`. When this is set to *on*, all errors are sent as warnings to the `errstr` value. The other mode is `RaiseError`, and when this is set to *on*, DBI raises exceptions rather than warnings. Both can be active at the same time, but `PrintError` will run first. You can set the values of these when connecting to Oracle as part of the attribute parameters as below.

```
$dbh = DBI->connect(dbi:Oracle:, 'system@XE',
                  'system', {RaiseError => 1, PrintError =>0});
```

In the case above `RaiseError` is *on* and `PrintError` is *off* for the connection.

The simplest way to debug DBI statements is to tack Perl's `or die` syntax on the end of your DBI method call, as below.

```
$dbh = DBI->connect(dbi:Oracle:, 'system@XE', 'system')
      or die $dbh->errstr;
```

This will stop the execution of your program at the point where the error occurred, and print out the value in `errstr`.

4.5 Sample Queries

Connecting, disconnecting, and error trapping is all well and good, but what we really want to do is perform some SQL against our database, using Perl. To start, we have to include the DBI code.

```
use DBI;
use strict;
```

Next, of course, we need to define our variables.

```
use vars qw($dbh $sth);
```

We will be using the standard `$dbh` for our database handle, and `$sth` for our statement handle. Now let's connect to the DB using 10XE's default HR account by creating a database handle object with DBI's `connect()` method.

```
$dbh = DBI->connect('dbi:Oracle:XE', 'hr', 'hr');
```

Let's create a statement handle for our SQL using the `prepare()` method on our database handle `$dbh`.

```
$sth = $dbh->prepare('Select city from locations');
```

Next we call the `execute()` method on the statement handle to fire our SQL,

```
$sth->execute();
```

then fetch the data using the statement handle's `fetchrow()` method and then print it out.

```
while (my ($city) = $sth->fetchrow())
{
    print " $city \n";
}
```

Finally `finish()` with the statement handle and `disconnect()` the database handle.

```
$sth->finish();
$dbh->disconnect();
```

The results should look like this:

```
$ perl test.pl
Beijing
Bern
Bombay
Geneva
Hiroshima
London
Mexico City
Munich
Oxford
Roma
Sao Paulo
Seattle
Singapore
South Brunswick
South San Francisco
Southlake
Stretford
Sydney
Tokyo
Toronto
Utrecht
Venice
Whitehorse
```

Here is the whole program.

```
#!/usr/bin/perl

use DBI;
use strict;
use vars qw($dbh $sth);

# get a database handle
$dbh = DBI->connect('dbi:Oracle:', 'hr@localhost/XE', 'hr');

# get a statement handle
$sth=$dbh->prepare('SELECT city FROM locations');

# execute the statement handle
$sth->execute();

# loop through the results
while (my ($r_city) = $sth->fetchrow())
{
    print " $r_city \n";
}
```

```

}

$sth->finish();
$dbh->disconnect();

```

The above example illustrates the standard approach to getting data from DBI:

Prepare ⇒ Execute ⇒-> Fetch

This will be familiar to any Oracle programmer as it has been standard practice for many years.

4.5.1 Queries That Return No Rows

Many SQL statements do not return data, and DBI handles those as well. One could code such a statement like this:

```

$sth = $dbh->prepare("UPDATE locations SET city = 'Ottawa'
                    WHERE location_id = 1800");
$sth->execute();

```

However, this is a waste of resources, since you are creating a statement handle for a query that returns no rows. So the database handle has a built-in shortcut for just such a case. This is the `do()` method. The code below shows how it is used.

```

$rows_affected = $dbh->do("UPDATE locations SET city = 'Ottawa'
                        WHERE location_id = 1800");
print "Rows Affected = $rows_affected\n";

```

As an added bonus, with `DBD::Oracle`, the `do()` method will return the number of rows affected by the query or `-1` if an error occurred.

4.5.2 Binding Parameters to Statements

We know now that DBI can run hard-coded SQL statements. But to be truly useful it must be able to bind variables to placeholders.

With the following SQL,

```

SELECT city FROM locations WHERE location_id = 1800

```

we would like to replace the `1800` with a placeholder. We could just use Perl for this, which would look something like this:

```

my $loc_id = 1800;
$sql = "SELECT city FROM locations WHERE location_id = $loc_id";

```

This is called in-line binding. It can work quite well for simple SQL, but what if we get an SQL statement such as this?

```

SELECT city FROM locations WHERE location_id = 'IT';

```

Almost identical statements, but we have to add nasty single quotes to make the statement work. So our Perl will look like this:

```

my $country_id = 'IT';
$sql = "SELECT city FROM locations
      WHERE location_id = '$country_id'";

```

This leaves us vulnerable to potential bugs. For example, if `country_id` happened to contain a single quote, you would have to write a whole procedure in Perl to account for it. Of course you could write all of your SQL in-line as long as you are careful with ' and other characters that SQL does not like. DBI even has a method on the database handle called `quote()`, to fix most of these problems for you. So you may never encounter a problem.

A couple caveats when using in-line binding. The first pertains to how Oracle optimizes SQL. As we know, when a query is run in Oracle, the optimizer analyzes and then saves a plan for the query. With in-line coded SQL the optimizer will not reuse execution plans in the same way that it can reuse execution plans where placeholders are used. So in the long run your DB will work faster when you use placeholders.

The second warning: when you use in-line binding, you may also be opening your system to [SQL injection attacks](#).

4.5.3 Using Simple Placeholders

In DBI, the most simple placeholder is the `?` character. So, if we make the following change to our first SQL like this:

```
$sql = 'SELECT city FROM locations WHERE location_id = ?';
```

the `?` will act as our placeholder, and all you have to do to make it work is use the database handle's `prepare()` method to get a statement handle.

```
$sth = $dbh->prepare('SELECT city FROM locations
                    WHERE location_id = ?');
```

Alternatively:

```
$sql = 'SELECT city FROM locations WHERE location_id = ?';
$sth = $dbh->prepare($sql);
```

You can then pass the parameter to the statement handle with its `execute()` method.

```
$sth->execute(1800);
```

So our code would look like this:

```
#!/usr/bin/perl

use DBI;
use strict;
use vars qw($dbh $sth $sql);

my $loc_id = 'IT';
$dbh = DBI->connect('dbi:Oracle:', 'hr@localhost/XE', 'hr');
$sql = 'SELECT city FROM locations WHERE location_id = ?';
$sth = $dbh->prepare($sql);
$sth->execute($loc_id);
```

```

while (my ($r_city) = $sth->fetchrow())
{
    print " $r_city \n";
}

$sth->finish();
$dbh->disconnect();

```

If the `$loc_id` variable contained a `'` character, as in `IN'T`, DBI would automatically fix this for us when you execute the SQL. So no worries about single quotes - DBI takes care of them, and you know the same execution plan will be reused no matter what value you enter for `country_id`.

4.5.4 Complex Placeholders

What about the case where you have SQL like this?

```

SELECT city FROM locations
WHERE
    (city = 'Rome' or city = 'New York')
    AND country_id <> 'IT'
    AND state_province <> 'New York'

```

This could have the parameters like the following.

```

SELECT city FROM locations
WHERE
    (city = ? or city = ?')
    AND country_id <> ?
    AND state_province <> ?

```

With a usage like this, DBI allows you to use the `execute()` statement as:

```
$sth->execute('Rome', 'New York', 'IT', 'New York');
```

DBI will substitute the parameters one after another in the same order as it finds `?` characters. So the `execute()` above would return a set of records.

Given the same SQL as above, and this execute statement:

```
$sth->execute('Rome', 'New York', 'New York', 'IT');
```

a different set of records would be returned, as this code corresponds to this SQL:

```

SELECT city FROM locations
WHERE
    (city = 'Rome' or city = 'New York')
    AND country_id <> 'New York'
    AND state_province <> 'IT'

```

There is nothing wrong with working with just the `execute()` statement and `?` placeholders, but sooner or later you will be forced to mix in-line SQL and parameter SQL, or have an SQL with so many parameters that the SQL statement will be unreadable and confusing.

A statement like this comes to mind.

```

$sql = 'INSERT INTO table_1
      values(field_1, field_2, field_3, field_4, month)
      select field_1, field_2, field_3, field_4,
      add_months(sysdate, ?) FROM locations
      WHERE
        (city = $city_1 or city = ?)
        AND country_id <> ?
        AND state_province <> ?';

$sth = $dbh->prepare($sql);
$sth->execute($var_1, $var_2, $var_3, $var_4);

```

Happily, the statement handle has a the `bind_param()` method so we can write code that strongly binds one parameter to a placeholder. So, given this code:

```

$sql = 'SELECT city FROM locations
      WHERE
        (city = ? or city = ?)
        AND country_id = ?
        AND state_province <> ?';

$sth = $dbh->prepare($sql);

```

We can use the `bind_param()` like this:

```

$sth->bind_param(1, "Rome");
$sth->bind_param(2, "New York");
$sth->bind_param(3, "US");
$sth->bind_param(4, "New York");

```

This will make your code much easier to read. DBD::Oracle also allows you to use the colon `:` placeholder as well. This placeholder lets you use named placeholders, which can make your code very readable. The above code could be written something like this:

```

$sql = 'SELECT city FROM locations
      WHERE
        (city = :p_city_1 or city = :p_city_2)
        AND country_id = :p_country_id
        AND state_province <> p_state_province';

$sth = $dbh->prepare($sql);

$sth->bind_param(":p_city_1", "Rome");
$sth->bind_param(":p_city_2", "New York");
$sth->bind_param(":p_country_id", "US");
$sth->bind_param(":p_state_province", "New York");

```

In the case above, I could have used any string or numeric value after the `:` as long as it matches the value used in the `bind_param()` method. The use of a prefix such as `p_` makes it easier to read the code.

Another feature of the `:` placeholder is that it can be used to substitute the same value a number of times within your SQL. Given this SQL,

```

SELECT to_char(add_months(sysdate, 1), 'Month yyyy'),
       add_months(sysdate, 1),
       sysdate
FROM dual

```

we could use the `bind_param()` method like

```

$sql = "SELECT to_char(add_months(sysdate, :p_add_months), 'Month yyyy'),
              add_months(sysdate, :p_add_months),
              sysdate
FROM dual";

$stmt->bind_param(":p_add_months", $months_to_add);

```

As such, we need only make one call to the `bind_param()` method to make two substitutions.

4.5.5 Fetching Data

So far we have seen the `fetchrow()` method used to get at our data -- or “record set”, to use the proper RDBMS term -- that has been returned from Oracle. This is of course familiar to all Oracle programmers as it is similar to a cursor, in that we iterate through the returned records sequentially, processing each as we move along until none are left.

The `fetchrow()` method works by getting the current row from the record set and binding the values from the record set into specific Perl variables. So given this code:

```

#!/usr/bin/perl

use DBI;
use strict;
use vars qw($dbh $sth $sql);

my $city = 'Seattle';
$dbh = DBI->connect('dbi:Oracle:', 'hr@localhost/XE', 'hr');
$sql = 'SELECT city, state_province, country_id FROM locations
       WHERE city = :p_city';
$stmt = $dbh->prepare($sql);
$stmt->bind_param(':p_city', $city);
$stmt->execute();

my ($r_city, $r_state) = $sth->fetchrow();

print "My city is $r_city.\n";
print "My state is $r_state.\n";

$stmt->finish();

```

you would get this result:

```

$ perl test_4.pl
My city is Seattle.
My state is Washington.

```

Note how in our SQL statement we were also selecting `country_id` but we did not use it when we fetched the data from the result set. If I changed the code to this:

```
my ($r_city, $r_state, $r_country) = $sth->fetchrow();
```

we could then also print out the country value.

Using `fetchrow()` quickly gets very tedious when you are returning a large number of fields from a database. Fortunately, the statement handle has the `fetchrow_array()` method which returns the row as an array. It can work the same way as `fetchrow()` where we name the values to be fetched individually.

```
my ($r_city, $r_state, $r_country) = $sth->fetchrow_array();
```

Or we could fetch all the data into a Perl array.

```
my (@a_row) = $sth->fetchrow_array();
```

The code to print this would look like this:

```
print "$a_row[0] \n";
print "$a_row[1]\n";
print "$a_row[2]\n";
```

(Note that this method always returns a 0--based array.)

As with `fetchrow()`, you can iterate with `fetchrow_array()` with a simple Perl while loop. An example of this is in the code below.

```
#!/usr/bin/perl

use DBI;
use strict;
use vars qw($dbh $sth $sql);

my $country = 'US';
$dbh = DBI->connect('dbi:Oracle:', 'hr@localhost/XE', 'hr');
$sql= 'SELECT city, state_province, country_id FROM locations
      WHERE country_id = :p_country';
$sth = $dbh->prepare($sql);
$sth->bind_param(':p_country', $country);
$sth->execute();

while (my (@a_row) = $sth->fetchrow_array())
{
    print "My city is $a_row[0], my state is $a_row[1], ";
    print "my country is $a_row[2].\n";
}

$sth->finish();
```

This would output a result like this:

```
$ perl test_5.pl
My city is Southlake, my state is Texas, my country is US.
My city is South San Francisco, my state is California, my country is US.
My city is South Brunswick, my state is New Jersey, my country is US.
My city is Seattle, my state is Washington, my country is US.
```

The statement handle also has a number of other methods that fetch single rows, namely `fetchrow_arrayref()` and `fetchrow_hashref()`, both of which will be discussed in later in this tutorial.

4.5.6 Atomic and Batch Fetching

One-way row-fetching is suitable for most of the work you may do with DBI, but occasionally you will need to move back and forth through data, or you may need to cut down the number of steps involved in a process. For those requirements, DBI provides two alternative fetching modes: **atomic** and **batch**.

4.5.6.1 Atomic Fetching

With atomic fetching, you can compress the data-fetching cycle into just a single method. There are two database handle methods that can be used in this way:

1. `selectrow_array()`, and
2. `selectrow_arrayref()`

They work in the same manner as the statement handle methods, however they require no **prepare** and **execute** statements.

Atomic Fetching is most useful when you have to get a single row of data, such as the name of a country for a given country code, as in the example below.

```
#!/usr/bin/perl

use DBI;
use strict;
use vars qw($dbh $ $country);

$country = "US";
$dbh = DBI->connect('dbi:Oracle:', 'hr@localhost/XE', 'hr');

my ($r_country_id, $r_country_name) =
    $dbh->selectrow_array("SELECT country_id, country_name
                          FROM countries
                          WHERE country_id = '$country'");

print "My country_id is $r_country_id,
      my country name is $r_country_name.\n";
```

4.5.6.2 Batch Fetching

Many times when working with data, it is desirable to have all of the rows returned at once. This is especially so in the Perl world, which has so many nifty little tools to manipulate arrays in all sorts of different ways. DBI has both a method on the statement handle and one on the database handle that can utilize batch fetching.

The database handle method, `selectall_arrayref()`, like its atomic cousin `selectrow_arrayref()`, is most useful when you want to get the data from static SQL more than once in a program.

For example, a record set of `country_id()` and the name of the country associated with the id, can be used to populate a select box and then be used in the same program to look up the name of a country. So instead of preparing, executing, and iterating through an entire record set each time we need it, needlessly sucking up Oracle resources, you can just use a statement such as this:

```
#!/usr/bin/perl
```

```

use DBI;
use strict;
use vars qw($dbh $sth );

$dbh = DBI->connect('dbi:Oracle:', 'hr@localhost/XE', 'hr');

my $countries =
    $dbh->selectall_arrayref("SELECT country_id, country_name
                            FROM countries
                            ORDER BY country_name");

foreach my $row (@$countries)
{
    print "My country_id is $row->[0], ";
    print "my country name is $row->[1].\n";
}

foreach my $row (@$countries)
{
    print "My country's name is $row->[1], ";
    print "my country's Id is $row->[0].\n";
}

```

where we use only one DBI method call and just reuse the returned arrayref.

The statement handle's batch feed method, `fetchall_arrayref()` works in the same manner as its other fetch handle methods, in that it needs the prepare and execute stages, but it is flexible enough to work in one of three different ways.

Without arguments, it simply returns a reference to an array of all the data in the same way `selectall_arrayref()` does. You could rework the code example for `selectall_arrayref()` like this:

```

$sth = $dbh->prepare("SELECT country_id, country_name
                    FROM countries
                    ORDER BY country_name");

$sth->execute();
my $countries = $sth->fetchall_arrayref();

```

and the results would be the same.

The next way you can invoke `fetchall_arrayref()` is to tell the method to return only certain columns to the reference array. Given this SQL:

```

SELECT Employee_Id, First_Name,
       Last_Name, Email, Phone_Number,
       Hire_Date, Job_Id, Salary, Commission_Pct,
       Manager_Id, Department_Id
FROM employees

```

where we want the first, third, second, and the fifth column, we can invoke `fetchall_arrayref()` like this:

```

my $employees = $sth->fetchall_arrayref([0,2,1,5]);

```

Our final code will look like this:

```

$sth = $dbh->prepare("SELECT Employee_Id ,First_Name,
                    Last_Name, Email, Phone_Number,
                    Hire_Date, Job_Id, Salary, Commission_Pct,
                    Manager_Id, Department_Id
                    FROM employees");
$sth->execute();
my $employees = $sth->fetchall_arrayref([0,2,1,5]);

foreach my $row (@$employees)
{
    print "My employee id is $row->[0], my name is $row->[1], ";
    print "$row->[2] and I started on $row->[3].\n";
}

```

We can use even Perl array indices specifying a range of columns like this [0 ... 6] (columns one to six) or even negative indices like this [-2, -1] (last two columns).

Since all of us like to write code that is easy to read and understand, the `fetchall_arrayref()` method also lets us use the actual column names rather than confusing indices, by mapping the SQL column names to an anonymous hash (`employee_id=>1`). So for our example above, we could substitute the following:

```

my $employees = $sth->fetchall_arrayref({
    employee_id=>1,
    hire_date=>1,
    first_name=>1,
    last_name=>1,});

foreach my $row (@$employees)
{
    print "My employee id is $row->{employee_id}, ";
    print "my name is $row->{last_name}, $row->{first_name} ";
    print "and I started on $row->{hire_date}.\n";
}

```

and get the same results. As you can see, this form of data return is very convenient, as it does not care in which order the mapping is done or how it is pulled out of the reference array. You need only make a call for the appropriate key.

Two important points to remember when working with hash key values are:

1. SQL column names are always treated in lower case, so the case in the query is ignored.
2. If your SQL statement has columns with the same name, your returned hash reference will have only a single value for that key.

4.6 DBI Transactions

In a perfect world, we would never need our old friends `commit` and `rollback`. But hard disks die, the power goes out, and fingers click the wrong buttons; so DBI and DBD::Oracle support the age-old ACID module for transactions.

To get transactions to work on DBI, you first have to set the `AutoCommit` attribute of our database handle to *off*. One normally does this when creating the handle like this:

```
$dbh = DBI->connect('dbi:Oracle:', 'hr@localhost/XE',
                  'hr', {AutoCommit => 0});
```

We can now use the database handle's `rollback` and `commit` methods much as you would in any PSQL program.

```
UPDATE employees SET salary = ? WHERE last_name = ?
```

To use transactions with the foregoing SQL, you would create the handle like this:

```
$dbh = DBI->connect('dbi:Oracle:', 'hr@localhost/XE',
                  'hr', {AutoCommit => 0});
```

You could then write Perl like the following.

```
$last_name = "Cambrault";
$salary = 200000;
$sql = "UPDATE employees SET salary = :p_salary
      WHERE last_name = :p_last_name";
$sth = $dbh->prepare($sql);
$sth->bind_param(':p_salary', $salary);
$sth->bind_param(':p_last_name', $last_name);
$rows_affected = $sth->execute();

if ($rows_affected > 1)
{
    $dbh->rollback();
    print "There are $rows_affected employees with ";
    print "the last name $last_name. Transaction canceled!\n";
}
else
{
    $dbh->commit();
    print "$last_name's salary is now $salary\n";
}
```

4.6.1 PL/SQL Procedures and Functions

Like all good database APIs, DBI allows you to call PL/SQL programs that you might have written in the past. DBI allows you not only to call them, but also get the data retrieved by them.

Calling a procedure is quite straight-forward -- you just wrap it between the `BEGIN` and `END` PL/SQL keywords.

```
$sth = $dbh->prepare("BEGIN fire_employee(:p_employee_id); END;");
$sth->bind_param(':p_employee_id', 101);
$sth->execute();
```

In this case, there is only one input parameter in the procedure, so we need only bind our value to it.

When there is a value returned by the procedure, you will have to the statement handle's `bind_param_inout()` method to get the returned value. Let us say our procedure has one input param, (`employee_id`), and one output param, (`hire_date`). We would call the stored procedure like this:

```

$in_employee_id = 101;
my $hire_date;
$stmt = $dbh->prepare("BEGIN get_startdate(:p_employee_id, :p_start_date);
                        END;");

$stmt->bind_param(':p_employee_id', 101);
$stmt->bind_param_inout("", \$hire_date, "SQL_NUMERIC");
$stmt->execute();

```

When this code runs, the `$hire_date` variable will automatically be updated to the value returned by the procedure, because we bound it using the `bind_param_inout()` method.

Calling a PL/SQL function with DBI is similar to calling it in PL/SQL -- a placeholder is used to store the value returned from the function, and then the `bind_param_inout()` method is used get that value into your Perl program.

So the Perl code to get the value for a PL/SQL function that returns the `employee_id` for the top seller would look like this:

```

my $big_seller;
$stmt = $dbh->prepare("BEGIN :p_big_seller := get_top_seller(); END;");
$stmt->bind_param_inout("", \$big_seller, "SQL_NUMERIC");
$stmt->execute();
print "The id of the top seller is $big_seller.";

```

You might have noticed that in both of these examples, `SQL_NUMERIC` was added to each of the `bind_param_inout()` calls. This is because Perl is a loosely-typed language, so we have to explicitly tell Perl -- and the `DBD::Oracle` driver as well -- what type of variable is being passed to Oracle from Perl, and from Oracle and back into Perl.

4.6.2 PL/SQL CURSORS

`DBD::Oracle` is able to use PL/SQL cursors -- both those that have been generated from functions and those generated directly . The key thing to remember when using cursors is that they have to be bound to a specific `DBD::Oracle` datatype constant called an `ORA_RSET`. This constant and the other Oracle-specific ones have to be explicitly imported into your program by the following command.

```

use DBD::Oracle qw(:ora_types);

```

With the constants loaded, you can then pass `ORA_RSET` datatype when binding parameters. Alternatively you could just enter the numeric value as well, but this is very poor programming style.

To get a cursor into a statement handle you would start with this code:

```

use DBI;
use DBD::Oracle qw(:ora_types);
use strict;

```

Then, of course, come our variables and connection.

```

use vars qw($dbh $sth1 $sth2);
$dbh = DBI->connect('dbi:Oracle:', 'hr@localhost/XE', 'hr');

```

Then we create a statement handle with SQL that will return a cursor.

```

$sth1 = $dbh->prepare(q{
    BEGIN OPEN :p_cursor FOR
        SELECT first_name, last_name
        FROM employees ORDER BY last_name;
    END;
});

```

Next, bind :p_cursor to variable reference \ \$sth2 and set its size to 0. Then tell DBD::Oracle that it is a ORA_RSET or cursor.

```

$sth1->bind_param_inout(":cursor", \ $sth2, 0, { ora_type => ORA_RSET });

```

We then just fire the execute() method of the first statement handle, which will load the returned cursor into a second statement handle that DBD::Oracle will create. Then we can then simply iterate through the second handle.

```

$sth1->execute();

while (my @row = $sth2->fetchrow_array)
{
    print "My name is $row[0] $row[1].\n";
}

```

Here is all the code.

```

#!/usr/bin/perl

use DBI;
use DBD::Oracle qw(:ora_types);
use strict;
use vars qw($dbh $sth1 $sth2);

$dbh = DBI->connect('dbi:Oracle:', 'hr@localhost/XE', 'hr');

$sth1 = $dbh->prepare(q{
    BEGIN OPEN :cursor FOR
        SELECT first_name, last_name
        FROM employees ORDER BY last_name;
    END;
});

$sth1->bind_param_inout(":cursor", \ $sth2, 0, { ora_type => ORA_RSET } );
$sth1->execute();

while (my @row = $sth2->fetchrow_array)
{
    print "My name is $row[0] $row[1].\n";
}

```

This code will also work if you were calling a function or procedure that returns a cursor, provided you use the Oracle cursor type SYS_REFCURSOR.

4.7 Large Objects (CLOBs and BLOBs)

Retrieving, updating, and inserting CLOB and BLOB data with DBI when using DBD::Oracle is straight-forward: all you have to do is use the statement handle's `bind_param()` method to tell DBD::Oracle what sort of LOB to expect. So your bind call would look like this:

```
$sth->bind_param(:p_big_desc, $big_desc, {ora_type=>ORA_CLOB});
```

Or if you are dealing with a BLOB:

```
$sth->bind_param(:p_pic, $a_pic, {ora_type=>ORA_BLOB});
```

The only caveat when using CLOBs and BLOBs is this: when doing an insert or update and you have more than one CLOB or BLOB field, you will have to tell DBD::Oracle which field corresponds to which datatype. To do this, you simply add the `ora_field=>` attribute to the bind call, with the corresponding table field name for the bind.

```
$sth->bind_param(:p_caption, $caption,
                {ora_type=>ORA_CLOB, ora_field=>'caption'}
            );
$sth->bind_param(:p_picture, $a_pic,
                {ora_type=>ORA_BLOB,
                 ora_field=>'picture'}
            );
$sth->bind_param(:p_comment, $coms,
                {ora_type=>ORA_CLOB, ora_field=>'comment'}
            );
```

When working with LOBs, is it also very important to tell DBD::Oracle what is the maximum size of the LOB that is to be returned. If you do not, DBD::Oracle will truncate the LOB and abort the fetch. You have to set a few attributes on the database handle, the first being `LongTruncOk`. You will want to set this to false, so that any fetch of a LOB variable that is too big will cause the fetch to fail.

```
$dbh->{LongTruncOk} = 1;
```

You also have to tell DBD::Oracle what the maximum size is for the LOB with the `LongReadLen` attribute. Set this value to the maximum size in bytes you want your LOBs to be. If you want to set a two megabyte limit for returned LOBs, you would set the attribute like this:

```
$dbh->{LongReadLen} = 2*1024*1024;
```

4.8 XML

Oracle's general guideline is to let the database manage data and to transfer the minimum amount of data across the network, so the best practice when using DBD::Oracle and XML is to let Oracle process the data into XML format, retrieve it as a CLOB and then process it with one of Perl's many XML modules.

For example, the PL/SQL package `DBMS_XMLGEN` returns the XML as a CLOB column, which DBD::Oracle can easily handle. With this SQL:

```
$sql = "SELECT dbms_xmlgen.getxml('
        SELECT first_name
```

```
        FROM employees
        WHERE department_id = 30') xml
FROM dual";
```

the code to get it would be a simple as:

```
$dbh = DBI->connect('dbi:Oracle:', 'hr@localhost/XE', 'hr');
$dbh->{LongTruncOk} = 0;
$dbh->{LongReadLen} = 2*1024*1024; # 2MB limit for displaying LOG
$sql = "SELECT dbms_xmlgen.getxml('
        SELECT last_name, first_name
        FROM employees
        ') xml
FROM dual";

$sth = $dbh->prepare($sql);
$sth->execute();
my ($r_xml) = $sth->fetchrow();
# let Perl play with $r_xml
```

5 Summary

In this document, we have introduced you to DBD::Oracle's most common usages. We will address more advanced aspects in a later document, but for most developers, the information herein will cover the majority of DBD::Oracle's applications.

IF you have any comments about this document, or wish to contribute any changes or corrections, please contact its maintainer, John Scoles <scoles@pythian.com>.

Generated on: 2007-03-30.